



**Международная Академия Ноосферы**  
**Балтийское отделение**

**В.З. Аладьев, В.А. Ваганов, Д.С. Гринь**

**Избранные системные задачи в  
 программной среде Mathematica**

**Херсон – 2013**

ББК 22.18я7

З 153

ISBN 978-966-289-012-9

З 153

**Избранные системные задачи в программной среде Mathematica:**

В.З. Аладьев, В.А. Ваганов, Д.С. Гринь. – Х.: Олди-Плюс, 2013. – 556 с.

Системы компьютерной математики находят все более широкое применение в целом ряде областей как естественных, так и экономико-социальных. Эти системы являются достаточно важным инструментарием для ученых, преподавателей, исследователей и инженеров, хорошо сочетая символьные методы с продвинутыми вычислительными методами. Одним из лидеров среди средств этого класса несомненно является пакет *Mathematica*. Естественно, дать полный анализ этому средству в отведенных книгой рамках просто нереально. И здесь акцентируется внимание лишь на одном аспекте – *модульном* программировании, поддерживаемом средой пакета. Данный аспект имеет особую значимость не только для решения прикладных задач, но и, прежде всего, он довольно важен в создании собственных средств, расширяющих часто используемые стандартные средства пакета и/или устраняющих его недостатки, или дополняющих пакет новыми средствами. Программные средства, представленные в книге, содержат целый ряд достаточно полезных и эффективных приемов *процедурно-функционального* программирования в системе *Mathematica*, расширяя программную среду системы и позволяя, порой, существенно эффективнее и проще программировать задачи самого различного назначения, прежде всего, носящие системный характер. Книга снабжена нашим пакетом *AVZ\_Package\_1* с *Freeware* лицензией в файлах форматов *{nb, n, cdf, mx}*, содержащим более 580 процедур, функций и глобальных переменных.

Настоящая книга рассчитана на широкий круг пользователей систем компьютерной математики, преподавателей и студентов университетов по курсам вычислительной техники и информатики, математики и других естественно-научных дисциплин.

ББК 22.18я7

ISBN 978-966-289-012-9

© Аладьев В.З., Ваганов В.А., Гринь Д.С., 2013

## Содержание

Предисловие	5
Глава 1. Краткий экскурс в историю систем компьютерной математики	15
Глава 2. Общие сведения по встроенному языку программирования системы компьютерной математики <i>Mathematica</i> ( <i>Math-языку</i> )	24
Глава 3. Средства работы в режиме текущего документа системы <i>Mathematica</i>	36
Глава 4. Дополнительные средства обработки выражений в программной среде системы компьютерной математики <i>Mathematica</i>	51
Глава 5. Дополнительные средства обработки символов и строчных структур в программной среде системы компьютерной математики <i>Mathematica</i>	88
Глава 6. Дополнительные средства обработки последовательностей и списков в программной среде системы <i>Mathematica</i>	117
Глава 7. Дополнительные средства, расширяющие стандартные функции системы <i>Mathematica</i> либо ее программную среду в целом	145
Глава 8. Задачи процедурного программирования в среде <i>Mathematica</i>	170
8.1. Определение процедур в программной среде системы <i>Mathematica</i>	170
8.2. Определение функций и чистых функций пользователя в среде <i>Mathematica</i>	184
8.3. Средства тестирования процедур и функций в среде системы <i>Mathematica</i>	196
8.4. Заголовки процедур и функций в среде системы <i>Mathematica</i>	216
8.5. Формальные аргументы процедур и функций, и средства манипулирования ими в программной среде системы <i>Mathematica</i>	235
8.6. Локальные переменные процедурных и блочных объектов, и средства манипулирования с ними в программной среде системы <i>Mathematica</i>	254
8.7. Глобальные переменные процедурных и блочных объектов, и средства манипулирования с ними в программной среде системы <i>Mathematica</i>	269
8.8. Атрибуты, опции и значения по умолчанию для аргументов пользовательских блоков, функций и модулей, дополнительные средства манипулирования с ними в программной среде системы <i>Mathematica</i>	280
8.9. Некоторые дополнительные средства для работы с блоками, функциями и модулями в программной среде системы <i>Mathematica</i>	295
Глава 9. Средства ввода/вывода программной среды системы <i>Mathematica</i>	337
9.1. Средства системы <i>Mathematica</i> для работы с внутренними файлами	337
9.2. Средства системы <i>Mathematica</i> для работы с внешними файлами данных	346
Глава 10. Организация работы с пакетами в среде системы <i>Mathematica</i>	397
10.1. Понятие контекста и его использование в программной среде <i>Mathematica</i>	398

10.2. Определение пакетов и их использование в программной среде <i>Mathematica</i>	402
10.3. Дополнительные средства работы с пакетами в программной среде системы компьютерной математики <i>Mathematica</i>	421
Глава 11. Организация программного обеспечения пользователя в среде системы компьютерной математики <i>Mathematica</i>	478
11.1. Организация программного обеспечения пользователя в программной среде системы компьютерной математики <i>Mathematica</i>	480
11.2. Программирование больших систем в программной среде системы компьютерной математики <i>Mathematica</i>	508
11.3. Система <i>Mathematica</i> или <i>Maple</i> – субъективная точка зрения	515
Литература	533
Monographs, textbooks and books on Computer Mathematics Systems <i>Maple</i> and <i>Mathematica</i> , prepared and published by members of the Baltic Branch of International Academy of Noosphere during 1995 – 2013	539
Перечень пользовательских процедур и функций для системы <i>Mathematica</i> , рассматриваемых и упоминаемых в настоящей книге; некоторые полезные рекомендации по использованию средств, представленных в данной книге	541
(Для удобства читателя данный перечень наряду с именами средств содержит начальные страницы, на которых приведены их исходные коды, описания и примеры использования, тогда как сами их программные реализации находятся в пакете, прилагаемом к данной книге. Средства классифицированы по своему основному функциональному назначению, однако использованная классификация в определенном отношении довольно условна, ибо во многих случаях средства являются достаточно широкого спектра применимости, но даже такая классификация позволяет лучше ориентироваться в таком обилии средств)	
Summary	554

*Mathematica 2, 5 ÷ 9 – торговые марки Wolfram Research Inc.*

*Maple V, 6 ÷ 15 – торговые марки MapleSoft Inc.*

## Предисловие

Системы компьютерной математики (СКМ) находят все более широкое применение в целом ряде областей как естественных, так и экономико-социальных наук таких как: химия, математика, физика, информатика, техника, технологии, образование и т.д. Такие системы как *Mathematica*, *Maple*, *REDUCE*, *MuPAD*, *Derive*, *Magma*, *Axiom*, *GAP*, *Maxima*, *MathPiper* и другие все более востребованы для преподавания [5,20,53,60,82] математически ориентированных дисциплин, в различных научных исследованиях и технологиях. Эти системы являются основными инструментами для преподавателей, ученых, исследователей, и инженеров. Исследования на основе СКМ-технологии, как правило, хорошо сочетают алгебраические методы с продвинутыми вычислительными методами. В этом смысле СКМ – междисциплинарная область между информатикой и математикой, в которой исследования сосредотачиваются на разработке алгоритмов для алгебраических (символьных) и численных вычислений и обработки данных, и на создании языков программирования и программной среды для реализации подобного типа алгоритмов и базирующихся на них задач различного назначения.

Решение прикладных пользовательских задач в той или другой области приложений поддерживают пакеты прикладных программ (ППП или просто пакеты) специального, узко-специального либо общего назначения. Классификация и характеристика данного класса программных средств (ПС) может быть найдена в наших предыдущих книгах [1-3,36]. Естественно, квалифицированный пользователь, владеющий в достаточной мере одним из эффективных языков программирования (например, *Basic*, *C*, *Fortran*, *PL/I*, *Pascal*, *Lisp*, *Prolog* и др.), в целом ряде случаев для решения своих задач может самостоятельно написать и отладить отдельную программу или комплекс программ, позволяющих реализовать на ПК алгоритм его задачи. Причем, в ряде случаев такой подход может быть более эффективным, чем использование для этих целей готовых ПС, т.к. разработчик ПС одновременно хорошо владеет спецификой решаемой задачи и условиями ее эксплуатации. Однако, такой подход требует, как правило, серьезных трудозатрат и при нынешнем обилии различного типа и назначения средств для ПК (особенно широко распространенных IBM-совместимых) в значительной мере становится нецелесообразным. Вместе с тем, развитые ППП снабжаются собственным встроенным языком программирования того или иного уровня сложности, позволяющим в среде пакета программировать целые задачи или их отдельные фрагменты, которые может быть нецелесообразно, неэффективно, а в ряде случаев и невозможно, реализовывать стандартными средствами пакета. Настоящая книга посвящена классу ПС (названному системами компьютерной математики), которые, в первую очередь, предназначены для решения задач математического характера, и, прежде всего, лидерам в этом классе системам *Mathematica* фирмы *Wolfram Research Inc.* и *Maple* фирмы *MathSoft Inc.* Более того, второй системе уделено лишь косвенное внимание в плане сравнения систем по некоторым отдельным моментам, тогда как довольно развернутый их сравнительный анализ можно найти в наших книгах [97,98]. Настоящая книга является естественным продолжением серии книг по вычислительной технике и программному обеспечению: ЕС ЭВМ (IBM 360/370), СМ ЭВМ (PDP-11), ПК ИСКРА 226 (WANG 2200-MVP), ИСКРА

1030 (IBM PC/XT), а также средствам для *IBM-совместимых ПК* [36]. При этом, большое внимание уделялось как опыту работы с описываемым средством, так и особенностям его применения, а также вытекающим из этого рекомендациям для пользователя. По мере возможности, предлагались наиболее эффективные технологии *применения* этих средств для решения тех либо иных прикладных пользовательских задач. Достаточно детальная характеристика данной серии книг может быть найдена, в частности, в [36] и в настоящей книге не рассматривается.

В настоящее время *ПС*, ориентированные на решение *математических* задач (где под *математической* понимается любая задача, чей алгоритм может быть описан в терминах того или иного раздела математики), весьма обширны и в определенной степени могут быть условно дифференцированы на 5 уровней: (1) встроенные средства различной степени развития той либо иной системы программирования; (2) специальные языки программирования; (3) узко-специальные, (4) специальные, (5) общие *ППП*. При этом, пятый уровень ранее представляли *три* основных математических пакета *MathCAD*, *Reduce* и *MatLab*. В указанной литературе приведены довольно подробное описание этих популярных пакетов; особенности их эксплуатации и использования в решении различного типа математических задач. При этом, на основе *всесторонних* апробации и адаптации для отечественные *ПК* пакетов *MathCAD* и *REDUCE* в наших книгах [1-7] были представлены достаточно детальный анализ особых и ошибочных ситуаций, рекомендации по использованию и предложения по дальнейшему развитию данных пакетов. Немало из указанных аспектов, доведенных до сведения разработчиков этих пакетов, было затем учтено при создании последующих версий пакетов. Естественно, данная рубрикация программных средств во многом носит субъективный характер, носящий следы опыта нашей работы с *ПС*, в той или иной мере ориентированными на программирование математических задач как в *числовом*, так и в *символьном* видах.

Наконец, *современное* развитие компьютерных технологий, ориентированных, прежде всего, на создание интегрированных пакетов *multimedia*-технологии вскоре привело к появлению нового уровня математических пакетов, из которых наиболее известными являются системы *Maple* и *Mathematica* соответственно фирм *MapleSoft Inc.* и *Wolfram Research Inc.* Эти системы, превосходя по целому ряду важных показателей упомянутые средства 5-го уровня, вместе с тем, наследуют целый ряд их стандартов, как пионеров-эталонов *ПС* такого типа, что достаточно легко прослеживается при более детальных их рассмотрении, использовании при решении различных задач и апробации.

Наш опыт эксплуатации данных *СКМ* оказал существенное влияние на апробацию и применение систем *Mathematica* и *Maple*, сравнительный аспект программирования в среде которых в различных контекстах был рассмотрен в наших книгах [97-99]. Здесь вполне уместно заострить внимание на достаточно существенном *терминологическом* аспекте, а именно. Рассматриваемое в данной книге программное средство является ярким представителем класса т.н. *систем компьютерной алгебры (СКА, или CAS в англо-язычном сокращении)*, ориентированных, в первую очередь, на *алгебраическое* решение математически ориентированных задач на компьютере. Между тем, все последующее развитие систем данного класса и, прежде всего, его лидеров (*Mathematica*, *Maple*) не только позволяет выполнять весьма сложные численные вычисления с произвольной

точностью, графически представлять результаты вычислений и т.д., но и с постоянно расширяющимся кругом разделов математики, охватываемых *решающими* средствами систем, что вполне резонно позволяет говорить уже о более развитом классе систем – *систем компьютерной математики (СКМ, или CMS в англо-язычной аббревиатуре)*.

Наряду с отмеченным, такие системы зачастую называются нами *пакетами* лишь по той причине, что созданные в их среде программные продукты нуждаются для своего выполнения наличия той среды, в которой они были запрограммированы, и не могут выполняться независимо в операционной среде, например, *Ms DOS*; т.е. они требуют наличия инсталлированной специальной среды (*Mathematica, Maple* и др.), довольно существенно снижая мобильность созданных в их средах программных продуктов. В книгах [1–3] такое понимание подобных средств обсуждается более детально. Далее с учетом сказанного понятия *СКА* и *СКМ* будем полагать тождественными там, где это не вызывает каких-либо недоразумений или двусмысленностей. Тут же отметим, что, например, специализированные *СКА*, вообще говоря, *СКМ* не являются. Сказанное в полной мере относится также и к использованию понятий «система» и «пакет».

Система *Mathematica* наряду с упомянутой системой *Maple* является одной из *СКМ*, наиболее известных и пользующихся популярностью систем. *Mathematica* содержит большое число функций для обеспечения как *символьных* преобразований, так и для *численных* вычислений. Наряду с этим, система поддерживает работу с графикой 2 и 3 измерений, импорт и экспорт изображений и звука и целый ряд других. *Mathematica* обеспечивает возможность решения следующих основных классов задач, а именно:

- интегрирование и дифференцирование функций, решение систем *полиномиальных* и *тригонометрических* уравнений и неравенств, *рекуррентных* уравнений, решение *дифференциальных* уравнений и уравнений в частных производных, ряды Тейлора, упрощение выражений, вычисление пределов, нахождение конечных и бесконечных сумм и произведений, а также целый ряд других задач в *символьном* виде;
- решение систем уравнений, вычисление пределов, *полиномиальная* интерполяция функций, нахождение сумм и произведений, вычисление значений элементарных и специальных функций с произвольной степенью точности, преобразование Лапласа, интегрирование и дифференцирование, решение *дифференциальных* уравнений и уравнений в частных производных, а также целый ряд других в *численном* виде;
- решение задач из *линейной алгебры*, теории чисел и других разделов математики;
- задачи *графического представления информации* (*построение графиков функций по их формулам, параметрических кривых и поверхностей, построение геометрических фигур, импорт и экспорт графики во многих растровых и векторных форматах*), включая также анимацию графических объектов различных типов и размерности *2D* и *3D*;
- начиная с последних версий, система поддерживает *распределенные* вычисления, для чего используется пакет *Parallel Computing Toolkit* (*пакет поддерживает довольно широкий спектр многопроцессорных систем, кластеров на базе CCS и grid-сетей*); планировщик для *оптимизации* решения конкретных задач, автоматическое восстановление после сбоя и перезапуск остановленных процессов, средства *распараллеливания* встроенного языка системы, тестирование и эмуляция *параллельных приложений* и др. При этом, отдельно разработчики поставляют интегрированную систему *Mathematica Personal Grid Edition*,

сочетающую в себе систему *Mathematica* и пакет *Parallel Computing Toolkit*;

– система поддерживает *встроенный* процедурно-функциональный язык, обеспечивая пользователя достаточно эффективным средством программирования своих задач.

При этом, *Mathematica* обладает даже большей функциональной *избыточностью*, чем *Maple*. Средства поддержки работы со *звук*ом и простейшими *клеточными* автоматами в *Mathematica*, *Socket* в *Maple* и ряд других вполне можно отнести к «*архитектурным*» *излишества*м систем. Между тем, система *Mathematica* обеспечивает высокую скорость численных вычислений. А вот к недостаткам *Mathematica* довольно часто относится несколько необычный язык программирования, что на наш взгляд не совсем верно. И именно в этом отношении настоящая книга попытается прояснить ситуацию. Также отметим, что обе системы не являются *универсальными* с общепринятой точки зрения системами программирования, давая возможность пользователю создавать средства, выполнимые только в среде самих систем (*т.е. в полной мере невозможно создание exe- и com-файлов*), что существенно сужает область выполнимости данных средств, требуя наличия самих систем, для целого ряда наших пользователей достаточно дорогих.

Система *Mathematica* на сегодня является многофункциональным средством, которое включает большое число возможностей для решения довольно широкого круга задач. Естественно, привести *полный* анализ данным средствам в отведенных книгой рамках просто невозможно. И здесь акцентируется внимание лишь на *одном* аспекте системы – возможностях ее программной среды при решении специальных задач *системного* и *массового* характера. Данный аспект имеет особую значимость не только для решения прикладных задач, но, прежде всего, он довольно важен при создании программных средств, расширяющих часто используемые средства системы и/или устраняющих их недостатки, или дополняющих систему *новыми* средствами. Именно в этом контексте представляют интерес возможности системы по созданию такого типа процедур или функций. Итак, программирование в среде системы – *многоаспектная* тема и в ней мы акцентируем внимание только на вопросах разработки процедур/ функций, которые представляют основные программные объекты как для наиболее частоиспользуемых средств пользователя, так и для средств, расширяющих и улучшающих стандартные средства системы, в программной среде *самой* системы, т.е. реализованные средствами *встроенного* языка системы (*Math-языка*). Именно в этом контексте и можно в довольно полной мере оценить *программную среду* системы *Mathematica*, не взирая на некоторые субъективные моменты, прежде всего, пользовательские предпочтения и привычки. Естественно, эти моменты играют достаточно существенную роль для пользователя, имеющего определенный опыт работы с языками процедурного типа, тогда как для начинающего это не стоит столь остро по причине отсутствия такого опыта.

Уместно здесь вкратце остановиться и на популярности систем *Mathematica* и *Maple*. Среди отечественных пользователей *СКМ* наибольшую популярность имеет система *Maple*; в качестве некоторого косвенного подтверждения степени популярности двух систем в *СНГ* может служить посещаемость форумов по таким системам на наиболее известном русскоязычном образовательном математическом сайте [www.exponenta.ru](http://www.exponenta.ru). Сегодня на данном сайте форум *Maple* имеет 2638 тем и 12702 сообщений, тогда как форум *Mathematica* лишь 912 тем и 2985 сообщений. Естественно, к такой статистике



следует относиться достаточно осторожно, ибо она не совсем верно отражает степень популярности обеих систем и вот почему. Прежде всего, данная статистика связана с образовательным процессом в университетах, которые используют именно *Maple* по двум основным причинам, а именно: (1) *Maple* более лоялен к его нелегальному использованию и (2) как для преподавательского состава университетов, так и (прежде всего) студентов он оказался более простым в освоении, прежде всего тем, что его язык синтаксически более близок к известным императивным языкам программирования, в частности, к *Pascal*. А как известно, в общем случае императивные языки несколько проще в освоении, чем функциональные, хотя и здесь не так все однозначно. Более того, основываясь на опыте проведения мастерс-классов по системам *Maple* и *Mathematica*, слушателями которых являлись люди, имевшие вполне определенные навыки как по языкам программирования, так и по различным приложениям, с уровнем подготовки физико-математического профиля не ниже третьего курса университета, а нередко и остепененные, у нас сформировалось мнение, что для достаточно хорошего освоения системы *Maple* требуется меньше усилий, чем для такого же уровня освоения системы *Mathematica*. И этому имеется ряд объяснений, часть из которых приведена в [98-101].

Может быть, именно по этой причине, несмотря на двадцатилетний возраст системы *Mathematica*, ее достаточно мощные вычислительные возможности при возможности установки на такие операционные платформы, как *Windows, Linux, Mac OS X, Ubuntu* популярность данной СКМ на постсоветском пространстве относительно невелика. В беседах, проведенных нами на целом ряде мастер-классов по СКМ, многие слушатели считают систему *Mathematica* достаточно сложной, а в функциональном отношении избыточной. И если с первым, в определенной степени можно согласиться, то вторым система *Maple* также злоупотребляет, пусть и не в такой степени. Об этом несколько детальнее можно найти в наших книгах [98-101]. Нами несколько раз просматривался целый ряд тем и сообщений на вышеуказанном сайте в форумах по обоим системам, картина сложилась следующая. На обоих форумах, в массе своей, обитают студенты, пытающиеся решить свои задачки чужими силами, либо постеры, ищущие некоего самоутверждения способом решения задач студенческого характера (хотя, возможно, имеются и некоторые исключения), и если *Maple* им как-то удалось освоить на довольно низком уровне, более сложная для восприятия вторая система для освоения требует и несколько больших усилий. Тогда как достаточно квалифицированные пользователи СКМ *Mathematica* и *Maple*, как правило, не посещают форумов такого уровня. Итак, вышеупомянутая статистика с определенной долей достоверности вполне может быть отнесена к студенческой среде, причем, далеко не самой лучшей (в целом ряде случаев проблему таких посетителей форумов составляет не столько та или иная система, сколько даже суть задачи, предложенной им для решения в среде системы), хотя она и представляет существенный пласт пользователей СКМ, правда, во многом лишь на время учебы по курсам, завязанным на указанные СКМ. Замечено, во многих случаях вопрошающие на довольно низком уровне владеют постановочной стороной вопроса, а отвечающие не очень сведущи в программной среде данных систем. Более того, на ряде и других форумов по СКМ ситуация хоть и выглядит несколько лучше, однако и статистика на их основе представляется нам весьма ненадежной для анализов подобного рода.

Настоящая книга состоит из 11 глав и затрагивает достаточно обширный материал по программной среде системы *Mathematica* в контексте его возможностей в *процедурно-функциональном* программировании. Между тем, назначение данной книги оставило в стороне вопросы, представляющие интерес, прежде всего, для читателей, которые в своей деятельности находятся на первых этапах освоения системы *Mathematica*. Для читателей такого рода рекомендуется обратиться к соответствующей литературе, чей список достаточно обширен, прежде всего, англоязычной. Рассматривается система и в русскоязычной литературе, однако именно англоязычная представляется нам более предпочтительной. В целом, с англоязычной литературой возможно ознакомиться на сайте [www.wolfram.com/books](http://www.wolfram.com/books) разработчика системы, довольно полезные источники можно найти в [3,6,7,51,58-60,98-102,106,109-140], включая весьма полезные ссылки на интернет-источники. Таким образом, данная книга представляет собой некий набор избранных системных задач, цель которых не только расширить либо сделать более эффективной программную среду *Mathematica*, но и оказать определенную помощь тем пользователям *Mathematica*, которые хотели бы перейти с уровня *пользователя* на уровень *программиста* или тем, кто при использовании *Mathematica* уже столкнулся с некоторыми ее ограничениями и хотят улучшить ее программную среду. Между тем, опытные программисты *Mathematica*, вероятно, также смогут найти для себя в нашей книге полезную информацию как прикладного характера, так и к размышлению.

Поэтому зачастую имеет место освещение лишь сути вопроса без обстоятельного его обсуждения, отдельных нюансов и рассмотрения смежных (*часто интересных и весьма важных самих по себе*) вопросов. Основные положения данной книги сопровождаются соответствующими иллюстративными примерами, проработку которых мы считаем необходимой для лучшего усвоения предлагаемого материала. Более того, системные средства, приведенные в данной книге, могут быть использованы в качестве довольно полезных средств при разработке собственных приложений в среде *Mathematica*. На наш взгляд, анализ исходных кодов средств, представленных в данной монографии и использующих как эффективные, так и нестандартные приемы программирования, наряду с вполне определенным практическим интересом позволит довольно глубоко освоить программную среду системы *Mathematica*. Для удобства их использования в данном качестве читателю предоставляется возможность свободной загрузки пакета *AVZ\_Package\_1* для системы *Mathematica* версий 5 ÷ 9, содержащего их [120]. В целом, содержание глав настоящей книги можно охарактеризовать следующим образом.

*Система компьютерной математики (Computer Mathematics System – CMS)* представляет собой тип программного средства, предназначенного, в основном, для оперирования математическими формулами. В данном контексте основная задача *CMS* сводится к автоматизации достаточно громоздких и сложных символьных преобразований. При этом, основное различие между *CMS* и традиционными компьютерными системами состоит в том, что первая имеет дело, в основном, с уравнениями в символьной, а не в числовой форме. И если как цели, так и возможности подобных систем изменяются в весьма широком диапазоне, все же основная их цель остается неизменной, а именно: манипулирование с символьными выражениями. И *первая* глава книги представляет краткий экскурс в историю систем компьютерной алгебры, в дальнейшем развитии

сформировавших и более развитые системы компьютерной математики (*СКМ*). Тогда как *вторая* глава книги на содержательном уровне представляет общие сведения по встроенному языку программирования *СКМ Mathematica (Math-языку)*.

Тогда как *третья* глава представляет базовые сведения по такой важной компоненте системы *Mathematica*, как документы текущего сеанса системы (*содержимое монитора, которое видит в процессе работы с системой пользователь*). Основное внимание уделено дополнительным нестандартным средствам работы с текущим документом. Данные средства, как и другие, рассматриваемые на протяжении настоящей книги, в полной мере отвечают основной цели предлагаемой книги, которую можно охарактеризовать следующими двумя основными направлениями, а именно:

- (1) представление целого ряда довольно полезных средств системного характера, как расширяющих, так и дополняющих стандартные средства системы *Mathematica*;
- (2) иллюстрация на их примере приемов и методов, довольно полезных в *процедурно-функциональном* программировании, наряду с целым рядом довольно существенных особенностей данной парадигмы программирования в условиях программной среды системы *Mathematica*.

Здесь вполне уместно отметить естественный механизм формирования собственного программного инструментария пользователя, работающего в некоторой *программной* среде. В процессе программирования того либо другого средства, или целого проекта вполне реальна ситуация, когда достаточно целесообразно написать дополнительные средства, отсутствующие среди стандартных, либо эффективнее, либо удобнее их. Во многих важных случаях применимость данных средств может иметь весьма массовый характер, позволяя формировать программный инструментарий довольно широкого спектра применимости. Именно во многом благодаря описанному механизму и были созданы наши довольно известная библиотека для *Maple* и 2 пакета для *Mathematica* *AVZ\_Package, AVZ\_Package\_1*, содержащие более 850 и 500,580 средств соответственно [45,90,120]. Все вышеперечисленные средства снабжены *FreeWare* лицензией и имеют открытый программный код. Именно подобный подход к разработке многих средств как в *Mathematica*, так и в *Maple* в значительной степени и способствовал появлению ряда системных средств из наших библиотеки и пакетов [45,90,120], когда разработка программных средств выявляла целесообразность определения в целях упрощения их реализации новых сопутствующих средств системного характера, часто являющихся довольно массовыми как в практическом, так и в системном программировании.

*Четвертая* глава представляет ряд достаточно полезных дополнительных средств по обработке выражений в программной среде *Mathematica*, в определенной степени ее расширяющих. Тогда как *пятая* глава книги представляет дополнительные средства для обработки символов и строчных структур в программной среде *Mathematica*; при этом, по тексту книги по мере целесообразности рассматриваются другие средства из данной группы, общее количество которых составляет 70. Среди таких средств можно отметить *StrStr, StrSymbParity, SubStrSymbolParity, SuffPref, ToString1* и др. Средства этой группы как расширяют стандартные средства, так и расширяют среду системы в целом, что весьма актуально в виду того, что символы и строки – базовые структуры.

Списки и списочные структуры также относятся к числу базовых структур системы и в данном контексте *шестая* глава представляет дополнительные средства обработки последовательностей и списков в программной среде системы *Mathematica*. При этом, по тексту книги по мере целесообразности рассматриваются другие средства из этой группы, общее количество которых составляет 67. Среди средств этой группы можно отметить такие, как *ElemOnLevels*, *ListAssign*, *ListListQ*, *ReduceList*, *SplitList* и другие. Средства этой группы как расширяют стандартные средства, так и расширяют среду системы в целом, что весьма актуально в виду того, что списки – базовые структуры.

В *седьмой* главе представлено наиболее значительное число дополнительных средств, расширяющих стандартные функции системы *Mathematica* и ее программную среду в целом. При этом, по тексту книги по мере целесообразности рассматриваются также другие средства из данной группы, общее количество которых составляет 163. Среди наиболее часто используемых при программировании средств нашего пакета можно отметить следующие процедуры и функции: *Attributes*, *Definition1* ÷ *Definition4*, *Df* ÷ *Df2*, *Diff*, *ExprOnLevels*, *GenRules*, *HowAct*, *Int*, *Integrate*, *Integral*, *Mapp*, *Map1* ÷ *Map17*, *MapInSitu*, *MemberQ1* ÷ *MemberQ4*, *PureDefinition*, *RepStandFunc*, *Replace1* ÷ *Replace4*, *RevRules*, *RhsLhs*, *VarExch*, *WhatObj* и целый ряд других. Средства данной группы как расширяют функциональные возможности стандартных средств *Mathematica*, так и довольно существенно расширяют программную среду системы в целом, обеспечивая более широкие возможности в программировании приложений различного уровня.

*Восьмая* глава представляет нестандартные средства *процедурного* программирования в среде *Mathematica*. Так как *процедурная* парадигма является одной из основных для программной среды системы *Mathematica*, представлению средств, ее расширяющих, в настоящей главе уделено особое внимание. Представлены нестандартные средства, обеспечивающие обработку следующих объектов типов *Module*, *Block*, *Function*, *Pure Function* (манипулирование с определениями таких объектов, средства их тестирования, их заголовки; формальные аргументы такого типа объектов и средства манипулирования ими; локальные и глобальные переменные объектов и средства манипулирования ими; значения по умолчанию для аргументов, атрибуты, опции, средства манипулирования ими и другие). При этом, по тексту книги по мере целесообразности рассматриваются также другие средства из этой группы, общее количество которых составляет 127. Среди наиболее часто используемых в программировании средств нашего пакета и других системных приложений можно отметить следующие средства: *Args*, *ArgsTypes*, *Arity*, *ArityBFM*, *BlockModQ*, *CallsInProc*, *CsProcsFuncs*, *FunctionQ*, *Globals*, *HeadPF*, *HeadingsQ*, *Locals* ÷ *Locals2*, *LocalsGlobals*, *ProcFuncBlQ*, *ProcQ*, *SubProcs* ÷ *SubProcs3*, *TestArgsTypes* и целый ряд других. Средства данной группы как расширяют функциональные возможности стандартных средств *Mathematica*, так и весьма существенно расширяют собственно саму программную среду системы в целом, обеспечивая более широкие возможности в программировании приложений различного уровня, включая системные.

Нестандартные средства ввода/вывода программной среды *Mathematica* представляет *девятая* глава настоящей книги. Представлены средства работы как с внутренними, так и внешними файлами данных, где под внутренними понимаются файлы данных, автоматически распознаваемые системой. Средства этой группы сконцентрированы,

в основном, именно в данной главе, используя средства из других групп не столь широко за исключением, пожалуй, процедуры *ReadFullFile*. Общее количество таких средств составляет 58. Среди наиболее часто используемых для программирования средств нашего пакета и других приложений различного назначения можно отметить следующие средства: *Attrib*, *Attribs*, *ClearRecycler*, *CloseAll*, *DelAllAttribs*, *DirQ*, *FileQ*, *EmptyFileQ*, *FileExistsQ1*, *FileOpenQ*, *FindFile1*, *FindSubDir*, *LoadExtProg*, *ReadFullFile*, *RenDirFile*, *SearchFile*, *StandPath* и другие. Средства данной группы как расширяют возможности системных средств *Mathematica*, так и весьма существенно расширяют собственно саму программную среду системы в целом, обеспечивая ее более широкие возможности в программировании приложений различного назначения и уровня.

Система *Mathematica* обеспечивает весьма развитые средства для расширения своих функциональных возможностей. Для очень многих видов вычислений стандартный набор средств, поставляемых с *Mathematica*, является вполне достаточным. Однако, приложения в целом ряде различных областей вполне могут выявить недостаточность стандартных средств, требуя создания новых средств для своих нужд. В таком случае *Mathematica* предоставляет возможность написания на ее языке специального пакета, содержащего определения требуемых средств. Именно в таком аспекте *девятая* глава и представляет ряд дополнительных средств работы с пакетами в программной среде *Mathematica*. Общее количество таких средств составляет 76. Среди наиболее часто используемых для программирования средств нашего пакета и других приложений различного назначения можно отметить следующие средства, а именно: *AddMxFile*, *CNames*, *ContentOfFile*, *ContextFromFile*, *DefFromPackage*, *ExtrOfMfile*, *LoadMyPackage*, *LoadPackage*, *MfileToMx*, *RemovePackage*, *UpdatePackage* и другие. Средства данной группы существенно расширяют программную среду системы в целом, обеспечивая ее более широкие возможности в работе с пользовательскими пакетами.

Наконец, *одиннадцатая* глава представляет ряд специальных средств в организации пользовательского программного обеспечения в среде системы *Mathematica* наряду с вопросами программирования достаточно больших систем. В заключение настоящей главы представлена наша субъективная точка зрения на предпочтительность систем *Mathematica* и *Maple* для решения тех либо иных задач пользователя.

Более того, изложение носит довольно выраженный фрагментарный характер, что не предполагает использования его в качестве начального освоения программной среды *Mathematica*. Основное его назначение сводится к представлению целого ряда наших программных средств, расширяющих программную среду *Mathematica* и устраняющих ограничения ряда ее системных средств. Поэтому данные средства и сопровождаются ассоциированной с ними справочной информацией с предоставлением необходимой и поясняющей дополнительной информации. Однако, целый ряд представленных в книге методов и приемов, использованных при программировании рассматриваемых средств, могут оказаться довольно полезными в разработке собственных приложений в *Mathematica*, позволяя рассматривать настоящую книгу также в качестве пособия по программированию в среде системы для достаточно опытного пользователя.

Настоящая книга представляет более 580 (точнее 582) средств, решающих различные системные задачи. Более того, значительное число этих средств в том либо ином виде

---

были ранее представлены в нашей предыдущей книге [101], они содержатся в пакете *AVZ\_Package* [90]. Между тем, это не простое дублирование, а результат различного рода модификаций в следующих основных направлениях, а именно:

- расширение функциональности предыдущих программных средств;
- существенное пополнение новыми средствами (*процедурами и функциями*);
- оптимизация средств, включая использование новых как наших, так и стандартных средств системы *Mathematica* версий **9.0.0.0 ÷ 9.0.1.0**;
- повышение робастности средства относительно возможных и особых, и ошибочных ситуаций, включая потенциальные из-за модификации стандартных средств.

Все эти средства дополнительно оформлены в виде отдельного пакета *AVZ\_Package\_1* [120], являющегося развитием нашего пакета *AVZ\_Package* [90]; более того, оба пакета распространяются по *FreeWare*-лицензии с открытым программным кодом. При этом, подавляющее большинство средств *1-го* включено в состав второго пакета. Между тем, значительная часть была перепрограммирована как в плане развития, так и с учетом новаций *9-й* версии системы. К сожалению, будучи достаточно пролонгированным в плане стабильности, и *Math*-язык в определенной степени подвержен обновлениям с появлением *новых* версий системы. Правда, для обычного утилитарного пользователя такие изменения пока, практически, незаметны и для них *Math*-язык остается весьма стабильным. Однако при разработке системных средств в рамках аксиоматики *Math*-языка его небольшие изменения функционального характера играют весьма важную роль. Именно по этой причине ряд средств *первого* пакета не вошли во *второй*. В то же время *2-й* пакет наполнен новыми средствами с исключением средств специального назначения. В целом же второй пакет наследует организацию первого и его основное функциональное назначение. Между тем, начиная с *Mathematica 9.0*, предпочтение следует отдавать пакету *AVZ\_Package\_1* [120]. Пакет содержит и средства, которые не приведены в книге. Открытость кода *AVZ\_Package\_1* позволяет как модифицировать содержащиеся в нем средства, так и программировать на их основе собственные либо использовать их компоненты в своих приложениях. На наш взгляд, задачи и средства их реализации в *Mathematica*, представленные в указанных пакетах, могут оказаться достаточно полезными при более глубоком освоении системы и в целом ряде случаев позволят достаточно существенно упрощать программирование в ней приложений, прежде всего, системных. При этом, методологические соображения, представленные в предыдущих книгах [98–101], сохраняют силу и относительно настоящей книги.

В процессе подготовки настоящей книги нами использовались лицензионные версии **7.0.0 ÷ 9.0.1.0 (License # L4704–9074) Mathematica**, предоставленные *Wolfram Research Inc.* Все вопросы, замечания и предложения по настоящей книге необходимо направлять в любой из адресов, указанных на нашем сайте *www.aladjev.narod.ru*. Все они будут приняты нами с благодарностью, весьма тщательно рассмотрены и на все будут даны исчерпывающие ответы, а при необходимости и соответствующие рекомендации.

Во многом выход данной книги обусловлен *спонсорской* помощью **Калле Солба (Kalle Solba)** – главного исполнительного директора фирмы *HydroScand AS (Tallinn, Estonia)*, за которую авторы книги выражают большую благодарность и признательность.

## Глава 1. Краткий экскурс в историю компьютерной математики

Система компьютерной алгебры (*Computer Algebra System – CAS*) представляет собой тип программного средства, предназначенного для манипулирования математическими формулами. Основная цель *CAS* состоит в автоматизации зачастую утомительных и в целом ряде случаев трудных алгебраических преобразований. Между тем, основное различие между *CAS* и традиционным вычислителем состоит в том, что первая имеет дело в основном с уравнениями в символьной, а не в числовой форме. И если целевые назначения и возможности этих систем изменяются в весьма широком диапазоне, все же основная их цель остается одинаковой, а именно: манипулирование символьными выражениями. При этом, *CAS* обеспечивают того либо иного уровня развитости язык программирования, позволяя пользователю создавать собственные процедуры. Более того, *CAS* не только существенно повлияли на методику преподавания математики и других математически-ориентированных дисциплин, но и обеспечили математиков и специалистов из ряда других областей гибким инструментальным средством. *CAS*, например, могут использоваться для факторизации полиномов, решения различных систем уравнений, включая дифференциальные, нахождения пределов, символьного дифференцирования и интегрирования выражений и так далее. Итак, *CAS* – система обеспечения символьной (*алгебраической*) математики, а ядро ее функциональности – манипулирование математическими выражениями в символьном виде.

История *CAS* восходит к началу 1970-х прошлого века и является непосредственным следствием прогресса исследований по *искусственному* интеллекту, однако на сегодня оба направления в значительной степени являются самостоятельными. На начальном этапе *CAS* были довольно простыми, обладая очень ограниченными возможностями, по символьному решению относительно узких классов задач. И лишь с таких *CAS* как *Reduce*, *Derive* и *Macsyma* начинается серьезный этап становления систем этого типа; более того, данные системы и сегодня все еще коммерчески доступны, тогда как *CAS Macsyma (Maxima)* все еще поддерживается и доступна бесплатно. Текущие лидеры в данной области – *Maple*, *Mathematica*, и в определенной мере также *MuPAD*. Данные системы, как правило, используются математиками, исследователями и инженерами, активно использующими математические методы, наряду с использованием данных систем в совершенствовании преподавания математически-ориентированных курсов. Наконец, целый ряд *CAS* сосредоточивается на определенной прикладной области и, как правило, они разрабатываются и поддерживаются в академической среде, являясь свободно распространяемыми. Такие *CAS* могут быть достаточно неэффективны для числовых вычислений по сравнению с системами, ориентированными на численные вычисления. Далее мы несколько детализируем историю становления самой *CAS*.

Корни возникновения *CAS*, как отдельного направления программного обеспечения, восходят к 60-м годам прошлого века и лежат в *двух* совершенно различных областях: теоретической физике и искусственном интеллекте. Действительно, первой работой в этом направлении можно считать разработку в 1963 г. Мартином Вельтманом (*Martin Veltman*) программы *Schoonschip*, ориентированной на символьное решение проблем и, прежде всего, в области физики высоких энергий. В свою очередь в MITRE в рамках

исследований по искусственному интеллекту на базе языка программирования *Lisp* в 1964 г. Карлом Энгельманом (*Carl Engelman*) была создана система *MATHLAB*, которая в дальнейшем использовалась на компьютерах *PDP-6* и *PDP-10*. И даже до настоящего времени ее можно использовать на *SIMH*-эмуляторах *PDP-10*. При этом, не следует ее отождествлять с одноименным пакетом *MATHLAB*, ориентированным на численные вычисления и созданном лишь в 1979 г. в университете Нью Мехико (*New Mexico*). Из первых же наиболее известных *CAS* общего назначения необходимо особо отметить систему *REDUCE*, во многом ориентированную на физическую область приложений.

Создание *REDUCE* относится к началу 60-х прошлого века и было инициировано Э.С. Хиерном (*A. Hearn*). С тех пор много специалистов из многих стран внесли свой вклад в ее развитие под его руководством. *CAS REDUCE* целиком написана на собственном диалекте языка *Lisp*, называемом стандартным *Lisp*; чей синтаксис алголо-подобен и назван *RLisp*. Именно он использован в качестве основы для пользовательского языка программирования в *CAS REDUCE*. Реализации *REDUCE* доступны на большинстве релизов *Unix*, *Linux*, *Windows Microsoft* или *Apple Macintosh*, используя стандартный мобильный *Lisp*. Уже первые версии *REDUCE* ориентированы на решение довольно несложных задач скалярной и векторно-матричной алгебры, при этом одновременно поддерживая произвольную точность числовых вычислений и базовые графические функции. Она допускает интерактивный режим для простых вычислений, но также предоставляет пользователю язык программирования, синтаксис которого довольно подобен другим современным языкам программирования. В целом, *REDUCE* можно рассматривать в качестве интерактивной программной системы, предназначенной, в первую очередь, для алгебраических вычислений общего характера, интересную для математиков, физиков, а также ученых и специалистов из других областей. В качестве ее основных возможностей следует отметить следующие, а именно:

- разложение и упорядочивание многочленов и рациональных функций;
- подстановки и сравнения шаблонов в весьма широком разнообразии форм;
- автоматическое и управляемое пользователем упрощение выражений;
- вычисления с символьными матрицами;
- поддержка произвольной точности арифметики целых и действительных чисел;
- возможность определения новых функций и расширения программного синтаксиса;
- символьные дифференцирование и интегрирование выражений;
- факторизация полиномов и полиномиальных выражений;
- решение разнообразных алгебраических уравнений;
- вывод выражений в разнообразных форматах;
- генерация оптимизированных числовых программ на основе символьного входа;
- вычисления с широким набором специальных функций и др.

Между тем, эта система подобно другим современным *CAS* весьма часто используется как алгебраический калькулятор для задач, достаточно легко выполнимых вручную. Однако, главная цель *REDUCE* состоит в решении задач, не выполнимых или трудно выполнимых вручную. Многие из подобных задач требуют достаточно существенных временных затрат (часы, а то и дни) на самых мощных компьютерах. Обеспечивают эту цель следующие важные характеристики системы *REDUCE* такие, как:



**Стабильность исходного кода.** Различные версии *REDUCE* использовались в течение более сорока лет. За этот период большим сообществом пользователей, прежде всего, из академической среды (*главным образом физиков и математиков*) была произведена большая работа по расширению и совершенствованию *CAS*, что обеспечило развитие ее в мощную систему, основные компоненты которой весьма надежны, устойчивы, а также достаточно эффективны во временном отношении.

**Обширная пользовательская среда.** Поддерживается как собственно возможностями *CAS*, так и большим числом специальных пакетов, ориентированных на различные, в первую очередь, *физические* приложения. Многие из них созданы многочисленными пользователями системы из различных областей вычислительной науки и техники.

**Полная доступность исходного кода.** Начиная с *первых* версий системы, было можно получить исходный код *REDUCE*, включая ее ядро. По этой причине *CAS REDUCE* не только прекрасный образовательный ресурс, но и хорошая основа для экспериментов в области компьютерной алгебры. Именно данные предпосылки и позволили многим пользователям эффективно изменять исходный код для своих собственных нужд. Так, в нашей книге [5] представлены некоторые расширения данной *CAS*. Тут же отметим, что наши книги [4-8] явились одними из самых первых отечественных книг по таким математическим пакетам как *MatCAD*, *REDUCE*, *Mathematica* и *Maple*; причем книги [5-6] вообще были *первыми* массовыми книгами по *MatCAD*, *REDUCE* и *Mathematica*.

**Гибкое обновление.** Важнейшим преимуществом открытости исходного кода является то, что относительно легко модифицировать *CAS* как с целью устранения возможных ошибок, так и ее развития с целью расширения применимости существующего кода к новым прикладным областям. Электронная почта и Интернет дают пользователям возможность оперативно получать такие модернизации, не дожидаясь формального выпуска очередного релиза системы. Именно данная открытость весьма импонирует многим серьезным специалистам и разработчикам в области *CAS* и выгодно отличает *REDUCE* от таких *CAS*, как *Maple* и в значительно большей степени, как *Mathematica*. Именно закрытость *Maple*, и в большей степени *Mathematica* не позволяют устранять силами большого пользовательского сообщества имеющиеся многочисленные огрехи в обоих пакетах, весьма часто недопустимые для такого уровня средств, тем более так агрессивно себя рекламирующих (*в первую очередь это относится к Mathematica*).

**Современные алгоритмы.** Еще одно важное преимущество открытых систем состоит в том, что возможно совместное участие разработчиков и пользователей по улучшению системы и используемых ею алгоритмов различного назначения. В результате, более просто поддерживать систему в актуальном состоянии при *лучших*, существующих на текущий момент алгоритмах, используемых вскоре после их разработки. В этой связи *REDUCE* отличается в значительно лучшую сторону от пакетов *Maple* и *Mathematica*.

**Алгебраический акцент.** *CAS REDUCE* позиционируется в качестве *специальной* части научной вычислительной среды, а не все охватывающей вычислительной средой, как это, например, пытается сделать *Mathematica*. Данная концепция предполагает, что *CAS REDUCE* ориентируется на эффективное символьное решение пользовательских задач, тогда как пользователь по мере своих нужд может дополнительно использовать и другие современные системы, специализирующиеся на численных, графических и

других вычислениях, а не зависеть лишь от одной системы, да и вряд ли одна даже и очень большая (зачастую громоздкая и требующая больших ресурсов) система обладает в равной степени эффективными алгоритмами. Немало «архитектурных» излишеств в *CAS Maple* и *Mathematica* с большой натяжкой можно отнести к *CAS*-проблематике. И здесь вполне уместно вспомнить изречение *К. Пруtkова* – «не объять необъятное».

**Мобильность.** Успешность *REDUCE* объясняется также доступностью на новых либо специальных компьютерах вскоре после их появления. В настоящее время *REDUCE* в полной мере доступен по существу на всех вычислительных платформах.

**Единообразие.** Даже при том условии, что *CAS REDUCE* поддерживается различными версиями языка *Lisp* на различных вычислительных платформах, большое внимание было уделено такому аспекту, как *единообразие* выполнения системы безотносительно различных ее реализаций и релизов.

**Гибкие предложения.** Для обеспечения различных требований пользователя *REDUCE* на сегодня доступна в широком диапазоне различных конфигураций.

**Стоимость.** Первая цена полных версий *REDUCE* для пользователя была умеренной и существенно не изменялась от платформы к платформе. Кроме того, поскольку все версии получены из одного и того же источника, они совместимы безотносительно от используемой платформы (от ПК до суперкомпьютера *Cray*). В настоящее время версия *Open Source CAS REDUCE* зарегистрирована на сайте *www.SourceForge.net*, с которого ее можно загружать для бесплатного использования и дальнейшего развития. Между тем, ранее стоимость *CAS REDUCE* составляла \$ 695.

Повышенное внимание, уделенное *REDUCE*, обуславливается тем обстоятельством, что она занимает выдающееся место в истории систем компьютерных алгебр. Она оказала большое влияние на дальнейшее развитие исследований в этом направлении так, что основные ее черты четко прослеживаются в последующих системах. И до настоящего времени *REDUCE* достаточно широко используется англоязычными пользователями, прежде всего из-за ее бесплатности и возможностей для такого доступного средства. В результате работы со многими достаточно серьезными *CAS* (*Axiom*, *Maxima*, *Maple* и *Mathematica*) именно *REDUCE* выделяется своей полезностью, надежностью наряду с универсальностью как для исследовательских целей, так и для обучения. Она весьма хорошо документирована, имеет открытый код, а ее программная среда достаточно удобна как для программирования задач пользователя, так и для расширения *CAS*. В среде отечественных пользователей, имеющих дело с исследованиями в физических областях, она до сих пор пользуется значительной популярностью. И действительно, в виду истории своего появления она ориентирована на физику и ее приложения, и в 80-е годы прошлого века система получила весьма широкое распространение в СССР. По нашему мнению система *REDUCE* со временем была значительно подзабыта, но даже относительно современных популярных *CAS* она все еще продолжает обладать рядом преимуществ. Так, в задачах, имеющих дело с *внешними* дифференциальными системами (дифференциальные формы, производные Ли и т.д.) пакеты *EXCALC*, *CRACK* и *EDS* из *REDUCE* превосходят соответствующие аналоги в *CAS Maple* и *Mathematica*. Несколько детальнее с системой можно ознакомиться, например, в книгах [5,63-72], а более детальную информацию можно получить в интернет по фразе «*CAS Reduce*».

Довольно важное место среди систем компьютерной алгебры занимает система *Axiom*. В отличие от остальных систем, представляющих собой пакеты программ, общение с которыми осуществляется на некотором входном языке, система *Axiom*, развившаяся из *SCRATCHPAD-II*, имеет дело с более привычными для математиков объектами. В частности, в ней ключевым понятием является понятие *категории*; по этой причине в ней можно рассматривать, например, категории *множеств*, *полугрупп*, *левых модулей*, *дифференциальных колец* и т.д. Система имеет высокий уровень универсальности, но требует для своей реализации мощных компьютеров, и раньше распространялась на коммерческой основе, поэтому использовалась лишь в ограниченном числе довольно мощных университетских и научных центров. К началу 21 века продажи резко упали и проект стал коммерчески нерентабельным, посему система *Axiom* была обеспечена лицензией *GPL*. Эта система с полным основанием может рассматриваться в качестве лидера среди *CAS* на европейском уровне аналогичных программных средств.

Система *SCRATCHPAD*, лежащая у истоков *Axiom*, была разработана для обеспечения диалогового символьного средства, ориентированного на пользователя-математика. Система обеспечивает пользователя языком общения, учитывающим лаконичность и стиль математической нотации наряду с предоставлением возможности расширения языка новыми понятиями. Более того, развитая библиотека системы включает также символьные средства, поддерживаемые такими системами как *REDUCE* и *MATHLAB*.

Для математика *Axiom* имеет целый ряд весьма привлекательных черт. Язык системы обладает весьма мощной и простой системой типов, ею поддерживаются *абстрактные* операции из области абстрактной алгебры и теории чисел. Более того, система *Axiom* имеет весьма *эффективный* алгоритм вычисления интегралов. Так, большинство *CAS*, получив на вход слишком сложный для них *интеграл*, или ничего не возвращают, или возвращают его *невывчисленным*. Тогда как *Axiom* возвращает интеграл *невывчисленным* лишь в случае, когда может доказать, что он не берется в элементарных функциях. К тому же, подобно *Maxima*, *Axiom* обладает достаточно высоким уровнем абстракции, предоставляемым пользователю. Обе системы поддерживают т.н. *Rule-преобразования*, позволяя выполнять преобразования не только над математическими выражениями, но также и над любыми формульными выражениями.

Из некоторых других популярных в данном направлении *CAS* можно отметить такие, как *muMATH*, *Derive* (базируется на *CAS muMATH*) и *Macsyma*; при этом, популярная бесплатная версия *Macsyma*, именуемая *Maxima*, довольно активно поддерживается и до настоящего времени. Вкратце остановимся и на этих популярных системах.

В 1980 г. *Soft Warehouse* – один из пионеров в разработке символьного математического программного обеспечения выпустил первый официальный релиз системы *muMATH* (последний релиз *MuMATH-83*), которая предназначалась для символьного решения на небольших компьютерах и простых калькуляторах относительно несложных задач. В дальнейшем *muMATH* была заменена более развитой системой *Derive*, написанной на языке программирования *muLisp* – одном из наиболее удачных диалектов языка *Lisp* [72-74]. Существует несколько реализаций: *muLisp81*, *muLisp83*, *muLisp85*, *muLisp87*. В то же время система *Derive* не получила сколько-нибудь широкого признания в среде специалистов, но относительно широко применяется в учебном процессе в школах.

В отличие от системы *REDUCE*, *Macsyma* разрабатывалась с самого начала в качестве коммерческого продукта в MIT в 1968 ÷ 1982 годах как часть известного проекта MAC. В системе более тщательно проработаны алгоритмические вопросы, эффективность ее существенно выше; в то же время, по причинам реализации на малом количестве вычислительных платформ и коммерческому распространению она довольно долго имела ограниченное распространение. Система *Maxima* выросла из *CAS Macsyma* и подобно большинству систем компьютерной алгебры реализована на языке *Lisp*. Она включает в себя *Algol*-подобный язык программирования и лучше всего подходит для обучения концепции программирования в системах компьютерной алгебры. *Maxima* доступна на бесплатной основе и имеет открытый исходный код. В настоящее время последней версией *Maxima* является 5.22 (2010). Несмотря на это, распространенность системы на отечественном рынке весьма ограничена и причина этого, на наш взгляд, состоит в том, что до появления *Maple* и *Mathematica* отечественные пользователи в массе своей применяли *REDUCE*, а с появлением двух последних предпочтение было отдано именно им, правда, с различной степенью популярности.

Отдельно здесь уместно также упомянуть известный математический пакет *MathCAD* [4,78-80], довольно широко используемый отечественными учеными, специалистами из различных областей и студентами естественно-научного профиля. Пакет ранних версий нельзя было относить к классу *CAS*, но, начиная уже с версии 3.0, этот пакет в определенной мере наследует черты современных *CAS*, чему успешно способствовал альянс *MathSoft Inc.* – разработчика пакета – с *Waterloo Maple Software* – разработчиком пакета *Maple*. Результатом явилась имплантация в пакет *MathCAD* ядра *Maple V* при довольно существенных ограничениях возможностей последнего. С другой стороны, появилась возможность использовать функции *Maple V* с вызовом их из *MathCAD*. По этой причине говорить о *MathCAD* как о полноценной *CAS* пока преждевременно. В этой связи хотелось бы также отметить, что разработчики *MathCAD 14* отказались от использования ядра *Maple V* и для обеспечения поддержки символьных вычислений обратились к более дешевому, но менее мощному пакету *MuPAD* фирмы *Mathworks*.

На сегодня самыми популярными коммерческими *CAS* общего назначения являются *Mathematica* и *Maple*, достаточно широко используемые в различных теоретических и прикладных областях [1-3,6-62]. Детальнее эти системы на сравнительном уровне рассматриваются в настоящей книге. При этом, бесплатно доступной альтернативой для указанных двух *CAS* выступает система *SAGE* (как внешний интерфейс к некоторым бесплатным *CAS*). Эта система уникальна тем, что может функционировать в качестве интегрирующей среды для целого ряда других компьютерных алгебраических систем, позволяя пользователю усиливать индивидуальные возможности различных пакетов программ. Система *SAGE* реализована на языке программирования *Python*, используя интерактивную *Python*-оболочку в качестве пользовательского интерфейса. Первый релиз системы появился в феврале 2005, текущим является релиз 4.6 (октябрь 2010). В качестве основной задачи разработчики *CAS SAGE* полагают создание эффективной бесплатной с открытым исходным кодом системы, альтернативной к *Magma*, *Matlab*, *Maple* и *Mathematica* [76,77]. Система *SAGE* все чаще именуется как *sagemath*, однако на сегодня данная система по популярности весьма далека от известных *CAS*.

Выше рассматривались CAS т.н. *общего* назначения (т.е. универсальные по назначению), тогда как наряду с ними существует немало систем специального назначения и даже CAS для калькуляторов. Специализированные системы отличаются несколько более высокой эффективностью, но область их применения более ограничена. Например, в качестве специальных можно отметить такие, как *GAP*, *CALEY*, *Magma* (алгебра, теория групп, теория чисел), *Fermat*, *CoCoA* (полиномиальные, матричные вычисления), *MathPiper*, *Algebrator* (ориентированы на задачи обучения), *SCHOONSHIP* (система для вычислений в физике высоких энергий), *Macaulay2* (алгебраическая геометрия, коммутативная алгебра), *TRIP* (механика небесных тел), *SINGULAR* (полиномы, кольца, идеалы), *VEGA* (операции с дискретными математическими структурами), *Cadabra* (тензорная алгебра, теория поля), *Mathomatic* (элементарная алгебра, комплексные числа, операции с полиномами), *PARI/GP* (теория чисел, эллиптические кривые, арифметика с произвольной точностью), *GiNaC* (б-ка языка C++), *Meditor* (б-ка символьных вычислений языка *Java* совместно с математическим редактором) и др., включая немногочисленные отечественные CAS. Особый интерес с точки зрения математика здесь имеет CAS *GAP* (*Groups, Algorithms and Programming*). *GAP* распространяется бесплатно и ориентирована на задачи абстрактной алгебры. На сегодня *GAP* наиболее мощная из систем такого рода, включая и коммерческие. В ней, например, в качестве типов данных выступают такие математические структуры, как «поле», «кольцо», «группа», а вычисление идеала банаховой алгебры поддерживается стандартной функцией системы. С более детальными обзорами современных систем компьютерной алгебры можно ознакомиться в целом ряде изданий и в интернете.

Наконец, имеется также целый ряд CAS, ориентированных на ресурсы портативных калькуляторов, среди которых можно отметить такие популярные, как *ZoomMath300*, *TI-Nspire CAS*, *TI-89 Titanium*, *Voyage 200* (для графических калькуляторов фирмы *Texas Instruments*), *Casio CFX-9970G*, *Casio ClassPad 330* (для калькуляторов фирмы *Casio*), *iCAS*, *i41CX+* (системы на основе *REDUCE* для *iPhone* и *iPad*), *HP 49 series* (для калькуляторов фирмы *Hewlett-Packard*) и др. Имеются отечественные разработки в этом направлении.

Более детальный статистический сравнительный анализ современных различных как по назначению, так и по поддерживаемым вычислительным платформам СКА можно найти в [81], однако некоторые его аспекты представляются нам довольно спорными. Между тем, анализ в целом представляет общую картину в этой области. В этой связи еще раз следует акцентировать внимание на факт становления более широкого, чем CAS класса программных средств – систем компьютерной математики (СКМ, или CMS в англо-язычном сокращении). Далее мы с учетом сказанного понятия СКА и СКМ будем полагать тождественными там, где это не вызывает каких-либо недоразумений либо двусмысленностей. В частности, специализированные СКА не являются СКМ. Более того, ниже зачастую для СКМ *Mathematica* будет использоваться термин «*пакет*» по той причине, что данное средство в общей классификации программных средств [1-3] соответствует именно данной группе. Рассмотрим несколько более детально историю создания и развития СКМ *Mathematica*, которая сама по себе довольно поучительна.

История создания *Mathematica* восходит к довольно простой программе символьного манипулирования (SMP), являющейся довольно простой CAS, созданной S. Wolfram и C. Cole в 1979, которая первоначально развивалась большой группой исследователей

под руководством *S. Wolfram*. При этом, сама программа *SMP* возникла под влиянием ранних *CAS Schoonschip* и *Macsyma*; так, *S. Wolfram* исследовал исходный код первой и являлся пользователем второй системы. Коммерческое распространение программы *SMP* было начато в 1981 *Computer Mathematics Corp.* (США), вошедшей в последующем в состав *Inference Corporation*, в рамках которой проводились последующие развитие и распространение программы *SMP* в период с 1983 по 1988. Имеются все основания рассматривать программу *SMP* в качестве точки отсчета в разработке *Mathematica*.

Начало интенсивного развития пакета *Mathematica* началось в рамках основанной в 1987 *Wolfram Research Inc.*, первая коммерческая версия которого увидела свет в 1988. В архитектурном отношении *Mathematica* по большому счету состоит из двух основных компонент, а именно – ядра и интерфейса с пользователем. Ядро пакета обеспечивает все вычисления, оно реализовано на *Mathematica*-языке и языке программирования *C*, тогда как интерфейс, разработанный *T. Gray*, обеспечивает развитый графический пользовательский интерфейс (GUI), поддерживающий создание и достаточно удобное редактирование текущего документа (*notebook*), содержащего как программный текст, так и обычный текст, графические объекты и др. При этом, и содержимое документа, и его форматирование можно выполнять как в программном, так и в интерактивном режимах. Интерфейс поддерживает наиболее принятые функции обработки текстов, однако он поддерживает лишь один уровень восстановления предыдущего состояния (*undo*) текущего документа, что вряд ли вызывает определенные затруднения. Интерфейс включает средства разработки типа отладчика и автоматической окраски синтаксиса вводимых выражений, что существенно упрощает ввод, позволяя по ходу ввода выражений корректировать их корректность. Пакет *Mathematica* использует по умолчанию стандартный интерфейс, допуская, при этом, другие интерфейсы такие, как *Wolfram Workbench* и *Eclipse* наряду с интерфейсом командной строки, который на компьютерах небольшой мощности является наиболее приемлемым.

С целью повышения производительности пакета версия *Mathematica 5.2 (2005)* была дополнена режимом автоматической мультиобработки для обеспечения вычислений на мульти-ядерных процессорах. Наряду с этим, эта версия содержала специальные библиотеки, оптимизирующие работу процессора, наряду с поддержкой *ClearSpeed* – специальным процессором, обеспечивающим весьма высокую производительность в вычислениях с плавающей точкой. Так, *CSX600* на сегодня является самым быстрым в мире 64-битным процессором с производительностью 25 гигафлоп, что почти в 10 раз быстрее стандартного персонального компьютера. При этом, такой процессор может работать и с традиционными 32-битными системами. С подобной целью в 2002 была создана надстройка *gridMathematica*, обеспечивающая параллельное программирование на однородных кластерах и мультипроцессорных системах, тогда как в 2008 в пакете была обеспечена технология параллельных вычислений, включая поддержку сеточной технологии типа *Windows HPC Server 2008*, *Microsoft Compute Cluster Server* и *Sun Grid*. С 2010 пакетом обеспечивается поддержка для *CUDA* и *OpenCL GPU*. Более того, версия *Mathematica 8.0* может генерировать *C*-код, который автоматически компилируется *Intel C++ Compiler* и *Visual Studio 2010*, обеспечивая скорость выполнения *Mathematica*-кода аналогично скорости кода, написанного на языке программирования *C*.

Протокол *MathLink* обеспечивает интерфейс не только между ядром и *GUI*, но также с рядом других приложений, расширяя функциональные возможности *Mathematica*. Свободно распространяются средства для обеспечения интерфейса написанных на *C* приложений с *Mathematica* через *MathLink*. Можно отметить интерфейсы с такими приложениями, как *Visual Basic*, *Python*, *Ms Excel*, *MATLAB*, *SINGULAR*, *MathModelica*, *R*, *OpenOffice.org Calc* и с рядом других. Некоторые из приложений позволяют проверять документы, подготовленные в среде *Mathematica*; так, бесплатная версия *Wolfram CDF Player* служит для выполнения *Mathematica*-программ, сохраненных в *cdf*-формате, а также для просмотра стандартных *Mathematica*-файлов без их выполнения. Именно по этой причине предоставляется возможность сохранения документа в *cdf*-формате.

Более того, за период 1988 ÷ 2012 была создана 21 версия пакета, из которых *первой* из доступных для массового пользователя выступает версия *Mathematica 1.0* (1988), тогда как текущей на сегодня является версия *Mathematica 9.0.1* (2013). При подготовке этой книги нами использовались, в основном, версии *Mathematica 8.0.0.0 ÷ 8.0.4.0* и *9.0.1.0*.

В заключение целесообразно кратко остановиться на истории разработки и развития отечественных *CAS*. Уже в самом начале создания программ символьной обработки и вычислений выяснилось, что для них требуются большие вычислительные ресурсы и достаточно квалифицированный штат разработчиков как в концептуальном, так и в программистском отношениях. Поэтому работы по *CAS* и проводились небольшими коллективами в крупных научных центрах (*Москва, Ленинград, Киев, Новосибирск и др.*). В этом направлении наиболее существенный прогресс был достигнут пионерскими работами по *CAS*, выполненными под руководством акад. *В.М. Глушкова* в институте кибернетики *АН УССР*. Именно в рамках данных работ в 1968 г. был реализован язык программирования *Аналитик*, являющийся дальнейшим развитием языка *АЛМИР-65* при сохранении полной с ним совместимости. Более того, отличительными чертами языка *Аналитик* являются абстрактные типы данных, аналитические преобразования, вычисления в произвольных алгебрах; он реализован на компьютерах *МИР-2*. Затем была разработана версия языка *Аналитик-74*, реализованная на компьютерах *МИР-3*. В настоящий момент язык *Аналитик* входит в состав системы компьютерной алгебры *АНАЛИТИК-2010*, совместно разрабатываемой Институтом проблем математических машин и систем НАН Украины и Полтавским техническим университетом (*Украина*).

В последующем, после появления ЭВМ БЭСМ-6 и серии ЕС ЭВМ, языков достаточно высокого уровня, было создано большое число *CAS*, различных по объему, областям применения и возможностям. Объединяющим центром по работам в области *CAS* в этот период явился ОИЯИ в Дубне; среди созданных в тот период отечественных *CAS* можно отметить такие, как *АНАЛИЗ, АЛЬКОР, АНАЛИТИК, АРАП, БОРА, ДИРАК* и др. Между тем, в силу целого ряда причин конкурентоспособных отечественных *CAS* в обозримом будущем не предвидится, делая актуальным освоение отечественными учеными, преподавателями и студентами новейших западных систем компьютерной алгебры (*CAS*), в целом математики (*CMS*). К таковым относятся системы *Mathematica* и *Maple* – общепризнанные мировые лидеры среди систем компьютерной математики, ориентированные, прежде всего, на массовый класс *PC*, вполне доступных достаточно широкому кругу отечественных пользователей из различных сфер деятельности.

## Глава 2. Общие сведения по встроенному языку программирования системы *Mathematica* (*Math-языку*)

Как известно, традиционные языки, с которых студенты обычно начинают освоение программирования (*C, C++, Fortran, Pascal, Basic*), являются процедурными языками. Процедурная программа пишется как список инструкций, говорящих компьютеру, что сделать шаг за шагом. Программные единицы включают главный/программный блок, подпрограммы, функции, процедуры. Процедурное программирование весьма подходит для разработки относительно небольших проектов; оно является наиболее естественным способом указать компьютеру, что требуется сделать для решения той или иной задачи, а так как *внутренний* язык самого компьютера (*машинный код*) также является процедурным, то *трансляция* кода на процедурном языке высокого уровня в машинный код является прямой и достаточно эффективной. Более того, *процедурное* программирование имеет встроенный способ разбивать большие списки инструкций в меньшие списки – *функции*. Языки данного типа являются процедурными. В целом, *Math-язык* является *процедурно-функциональным*, при этом, располагая достаточно развитым количеством конструкций из базового *функционального* программирования. Между тем, ввиду однотипности и весьма высокой конкуренции пакетов *Mathematica* и *Maple* вполне целесообразно на общем уровне сравнить их программные среды.

Пакет *Maple* способен решать довольно большое число, прежде всего, математически ориентированных задач вообще без программирования в общепринятом смысле, ибо средства языка пакета довольно *емки* в функциональном отношении, позволяя одним вызовом вычислять *интегралы*, брать *производные*, решать *диффуравнения* и их системы, и т.д. Поэтому вполне возможно ограничиться только описанием алгоритма решения своей задачи, разбитого на отдельные *последовательные* этапы, для которых *Maple* уже имеет готовые решения. При этом, *Maple* располагает большим набором процедур и функций, непосредственно решающих достаточно сложные задачи: интегрирование, дифференциальные уравнения и др. О многочисленных приложениях *Maple* в виде т.н. пакетов и говорить не приходится. Тем не менее, это вовсе не означает, что *Maple* не предполагает программирования. Имея собственный *встроенный* весьма развитый язык программирования (*в дальнейшем Maple-язык*), пакет позволяет программировать в своей среде самые разнообразные задачи из различных приложений, в том числе и средства, расширяющие стандартные средства собственно самого пакета. Несколько поясним данный аспект, которому в отечественной литературе уделяется, по нашему мнению, недостаточно внимания, причины которого рассматривались нами ранее.

Между тем, относительно *проблематики*, рассматриваемой в настоящей книге, вполне уместно сделать несколько весьма существенных замечаний. К большому сожалению, у многих пользователей современных математических пакетов, не исключая системы компьютерной алгебры – основной темы книги – бытует довольно распространенное мнение, что использование подобных средств не требует знания программирования, ибо все, что требуется для решения их задач, якобы уже имеется в данных средствах, и задача сводится только к выбору требуемого средства (*процедуры, модуля, функции и т.д.*). Такой подход к данным средствам носит в значительной степени дилетантский



характер, причины которого достаточно детально рассмотрены в нашей книге [44]. Действительно, пакет *Maple* способен решать весьма большое число задач вообще без программирования в общепринятом смысле данного понятия. Для этого достаточно только описать алгоритм решения и разбить его на отдельные этапы, которые *Maple* решает своими *стандартными* средствами. Тем не менее имеется немалое число задач, для которых средств *Maple* недостаточно; более того, для целого ряда задач требуется модификация стандартных средств либо доработка собственных, в ряде же случаев и вовсе требуется заменять стандартные, имеющие недостатки, на собственные. Все это и позволяет делать пользователю программная среда того либо иного пакета.

Прежде всего, с полным основанием можно говорить, что пакет *Maple* – это и система компьютерной алгебры, и язык программирования сверхвысокого уровня. В качестве языка программирования *Maple* имеет ряд атрибутов, характерных языкам *объектно-ориентированного* типа. *Maple*-язык является ярко выраженным процедурным языком с некоторыми элементами функционального программирования. Синтаксис *Maple*-языка достаточно близок к таким процедурным языкам, как *C* и *Pascal*; при этом, его интерактивность обеспечивает простоту освоения, удобство редактирования и отладки прикладных *Maple*-документов и программ. *Двухуровневая* лингвистическая поддержка *Maple* обеспечивается такими языками как *C* и *Maple*. В ряде публикаций встречается иная (не вполне обоснованная на наш взгляд) классификация, когда выделяются 3 языка – *реализации, входной и программирования*. Суть же этого состоит в следующем.

Действительно, ядро пакета *Maple* содержит набор высокоэффективных программ, в большинстве своем написанных на языке *C*. Более того, библиотека функций доступа к компонентам файловой системы компьютера было непосредственно заимствована из соответствующей библиотеки *C*. По нашим прикидкам доля программных средств пакета, написанных на *C*, не превышает 15%. Остальная масса программных средств *Maple* (функции, процедуры, модули), находящихся в различных библиотеках, написана на собственном *Maple*-языке. Уже ввиду сказанного весьма сомнительным выглядит утверждение, что *C* – язык реализации, а *Maple* – входной или язык программирования. Так как *Maple*-язык использован для реализации важнейших базовых средств пакета, то языками реализации являются и *C*, и *Maple*. При этом, с довольно определенными допущениями возможно говорить о *входном Maple-языке* и языке программирования пакета *Maple*. В основе своей *входной Maple-язык* пакета основывается на встроенном языке программирования, являясь его подмножеством, которое обеспечивает режим интерактивной работы с пакетом. Именно на *входном Maple-языке* в данном режиме пишутся и выполняются *Maple*-документы {*mws, mw*}-файлы. Тогда как некоторые конструкции *Maple*-языка допустимы лишь в процедурных выражениях, вызывая во *входном языке* ошибочные ситуации, как это иллюстрируют простые примеры [101]. В частности, что если *goto*-функция и *return*-предложение не допустимы во *входном языке* пакета, то в процедурных конструкциях *Maple*-языка их использование вполне корректно. Итак, по большому счету языками реализации пакета *Maple* являются как язык программирования *C*, так и собственный встроенный *Maple*-язык, *подмножество* которого и составляет собственно *входной язык* программирования пакета *Maple*.

*Входной язык* системы ориентирован, в первую очередь, на решение математически-

ориентированных задач практически любой сложности в интерактивном режиме. Он обеспечивает диалог пользователя со своей вычислительной компонентой, принимая запросы пользователя на обработку данных с их последующей обработкой и возвратом результатов в символьном, числовом и/или графическом видах. Входной *Maple*-язык является языком интерпретирующего типа и идеологически подобен языкам данного типа. Язык располагает большим числом математических и графических процедур и функций и другими средствами из обширных библиотек пакета. Его интерактивный характер позволяет довольно просто реализовывать интуитивный принцип решения своих задач, при котором ход решения можно пошагово верифицировать, получая в итоге требуемое решение. Уже введя *первые* предложения в текущий сеанс пакета, вы начинаете работать со входным *Maple*-языком программирования.

Среда программирования пакета обеспечивается *встроенным Maple*-языком, который является функционально полным *процедурным языком* программирования четвертого поколения (4GL). Он ориентирован, прежде всего, на эффективную реализацию как системных, так и задач пользователя из различных математически-ориентированных областей, расширение сферы приложений пакета, создание библиотек программных средств и т.д. Синтаксис *Maple*-языка наследует многие черты таких известных языков программирования как: *C, Fortran, Basic* и *Pascal*. Поэтому пользователю, имеющему в той либо иной мере опыт работы как с этими языками, так и с программированием вообще, не должно составить особого труда прилично освоить и *Maple*-язык.

*Maple*-язык пакета имеет вполне *традиционные* средства структурирования программ, включает в себя все команды и функции входного языка, ему доступны специальные операторы и функции пакета. *Maple*-язык, являясь как *универсальным*, так и *проблемно-ориентированным* языком программирования, характеризуется достаточно развитыми средствами для описания задач математического характера, возникающих во многих прикладных областях. В соответствии с языками этого класса структуры управляющей логики и данных *Maple*-языка в *значительной* мере отражают характеристику средств, в первую очередь, именно для математических приложений. Наследуя многие черты *C*-языка, *Maple*-язык обеспечивает как *числовые* вычисления с произвольной степенью точности, так и *символьные* вычисления, при этом поддерживая все *основные* операции традиционной математики. Между тем, следует привести одно весьма существенное пояснение, сводящееся к следующему, а именно.

Хорошо известно, что далеко не все задачи поддаются решению в символьном виде и приходится применять численные методы. Несмотря на то, что *Maple*-язык позволяет решать и такие задачи, его программы будут выполняться медленнее, чем созданные в среде языков компилирующего типа. Так что решение задач, требующих большого объема численных вычислений, в среде *Maple* может быть довольно неэффективным. Именно поэтому пакет *Maple* предоставляет интерфейс с языками такими, как *C, C#, MATLAB, Fortran, Java* и *VisualBasic*, а также интерфейс с пакетом *Ms Excel*.

Средства *Maple*-языка позволяют пользователю работать в среде пакета в 2 основных режимах, а именно: (1) на основе функциональных средств языка с использованием правил оформления и работы с *Maple*-документом предоставляется возможность на интерактивном уровне формировать и выполнять нужный алгоритм *пользовательской*

задачи без сколь-нибудь серьезного знания даже *основ* программирования, а подобно конструктору собирать из готовых функциональных компонентов входного языка на базе его синтаксиса требуемый вам алгоритм, включая его выполнение, отображение результатов на экране (*в обычном и/или графическом виде*), в файлах и в твердой копии (*правда, в общем понимании это как и любое решение можно называть программированием*), и (2) использовать всю мощь *Maple*-языка для создания развитых систем конкретного назначения и средств, расширяющих собственно саму среду *Maple*, чьи возможности определяются только собственными умениями и навыками пользователя. Между тем, первоначальное освоение *Maple*-языка не предполагает предварительного серьезного знакомства с основами программирования, хотя их знание весьма предпочтительно.

Реальная мощь *Maple*-языка обусловлена не только его управляющими структурами и структурами данных, но и всем богатством *функциональных* его средств (*встроенных, библиотечных, модульных*) и прикладных средств (*Maple-документов*), которые созданы к настоящему времени пользователями из различных прикладных областей, прежде всего, математических. Важнейшим преимуществом *Maple* является *открытость* его архитектуры, что способствовало созданию в достаточно короткие сроки обширного комплекса процедур и модулей, широким кругом пользователей из многих областей науки, образования, техники и т.д., которые значительно расширили как его область приложений, так и возможности. К их числу можно с полным основанием отнести и отмеченную в [45] *библиотеку*, содержащую более **850** средств, дополняющих средства пакета, устраняющих некоторые его *недоработки*, расширяющих ряд его стандартных средств и повышающих уровень совместимости версий пакета. Представленные в [45] средства используются достаточно широко как при работе с пакетом *Maple* в режиме интерактивном, так и при программировании различных задач в его среде. Средства представляют несомненный интерес при программировании различных задач в *среде Maple*, как упрощая собственно сам *процесс* программирования, так и делая его более эффективным и прозрачным с формальной точки зрения.

Таким образом, *Maple* – не просто высоко интеллектуальный калькулятор, способный аналитически решать множество задач, а легко обучаемая система, вклад в обучение которой вносят как сами разработчики пакета, так и его *многочисленные* пользователи. Очевидно, как бы ни была совершенна система, всегда найдется немало специальных задач, которые оказались вне сферы интересов разработчиков. Освоив относительно простой, но весьма эффективный *Maple*-язык, пользователь может сам изменять уже существующие процедуры под свои задачи или расширять пакет новыми средствами для решения нужных ему задач. Данные пользовательские средства можно включать в одну или несколько пользовательских библиотек, снабженной справочной базой, а затем логически сцепить с главной библиотекой пакета *Maple*, так что их средства на логическом уровне будут неотличимы от стандартных средств пакета. Именно таким образом и организована наша Библиотека [45]. И последнее, *Maple*-язык – наименее подверженная изменениям компонента пакета, поэтому ее освоение позволит весьма существенно пролонгировать *эффективное* использование пакета для решения задач, которые прямо не поддерживаются стандартными средствами пакета.

Так как *Maple*-язык является одновременно и языком *реализации* пакета, его освоение

и практическое программирование в его *среде* позволят не только весьма существенно повысить ваш уровень *использования* предоставляемых пакетом возможностей (*уровень владения пакетом*), но и глубже понять как *идеологию*, так и внутреннюю кухню самого пакета. Учитывая ведущие позиции *Maple* среди средств современной компьютерной алгебры и во многом распространенную его идеологию в этой области, вы получаете прекрасную и пролонгированную возможность весьма эффективного использования подобных ему средств в своей профессиональной деятельности, прежде всего, в таких областях, как математика, физика, информатика, техника и целый ряд других.

Прежде всего необходимо отметить, что *Maple-языку*, не взирая на *описательный* характер изложения, было уделено столько внимания лишь по той причине, что язык пакета *Mathematica* обладает многими из черт, что и пакет *Maple*. Поэтому далее мы рассмотрим на *содержательном* уровне некоторые принципиальные отличия, которые присущи именно языку пакета *Mathematica* (*в дальнейшем просто Math-языку пакета*).

Прежде всего, с полным основанием можно сказать, что *Mathematica* – это и система компьютерной алгебры, и язык программирования сверхвысокого уровня. В качестве языка программирования *Math-язык* имеет ряд атрибутов, характерных для языков объектно-ориентированного типа, являясь *процедурно-функциональным* языком. А в качестве назначения *Math-язык* с полным основанием возможно отнести к *проблемно-ориентированным* языкам сверхвысокого уровня, предназначенным, прежде всего, для обеспечения аналитических и численных вычислений. При этом, функциональность *Math-языка* весьма существенно отличает его от *Maple-языка*. Функциональные языки программирования делают акцент на правилах и сравнении образцов. В то время как они представляются не обладающими интуицией для пользователей, которые имеют опыт работы лишь с процедурными языками, функциональные языки обеспечивают краткие и естественные структуры программирования для тех, кто получил немного опыта. Функциональное программирование особенно полезно для математических приложений, где «*функция*» – достаточно хорошо определенное базовое понятие. В среде пакета *Maple* также можно использовать некоторые элементы *функционального* программирования, в частности, создавать чистые функции, однако это всего только его симуляция в достаточно ограниченных пределах. В книгах [100, 101] представлен целый ряд довольно интересных примеров, которые весьма наглядно иллюстрируют реализацию в среде *Maple* функций из системы *Mathematica* с использованием своего рода аналогов чистых функций (*pure function*).

Действительно, *функциональные* программы, вообще говоря, выполняются медленнее, чем *процедурные* программы, однако время выполнения – не самое главное в терминах эффективности. Время, требующееся для разработки програм и что еще более важно для их модификации, существенно меньше для функциональных программ, чем для процедурных программ. Данное обстоятельство весьма важно при проведении работ исследовательского характера. В частности, проведенный эксперимент по сравнению времени, затраченного на вычисления с *машинной* точностью с *матрицами*, *списками* и *тензорами* в среде пакетов *Matlab*, *Maple* и *Mathematica* показал явное преимущество последнего пакета. Более того, *Mathematica* использует механизм упаковки массивов, что дает не только временной выигрыш работы с ними, но и достаточно существенно

уменьшает требуемую для них память как оперативную, так и внешнюю.

Двухуровневая лингвистическая поддержка *Mathematica* обеспечивается языком *C* и *Math*-языком. Подобно ситуации с *Maple*, в *Mathematica* можно выделить два уровня языковой среды – язык реализации (*C + Math*-язык) и встроенный язык программирования (*Math*-язык). *Mathematica* аналогично пакету *Maple* реализована в основном на языке *Math*, т.е. собственном встроенном языке программирования, однако часть ее ядра и ряд важных функций в целях оптимизации написаны на языке программирования *C*. Таким образом, языком реализации *Mathematica* является и язык программирования *C*, показавший свою высокую эффективность как язык системного программирования. По нашей оценке доля программных средств пакета, написанных на *C*, не превышает  $8 \div 10\%$ . Тогда как остальная масса программных средств *Mathematica*, находящихся в различных библиотеках и пакетах, написана на собственном *Math*-языке. Это и дает основание рассматривать *C* и *Math* в качестве языков реализации пакета. При этом, с довольно определенными допущениями возможно говорить о входном *Math*-языке и языке программирования пакета *Mathematica*. Между тем, в отличие от пакета *Maple*, входной *Math*-язык, практически, совпадает со встроенным *Math*-языком пакета, что, в отличие от *Maple*, позволяет говорить о встроенном языке программирования, как о едином целом. Именно на встроенном *Math*-языке пишутся и выполняются в среде пакета *Mathematica*-документы и пакеты *{nb, m, mx}*-файлы. В отличие от *Maple*, все (за очень редким исключением) конструкции *Math*-языка допустимы как в процедурных конструкциях, так и на входе пакета *Mathematica*, как это иллюстрируют наглядные примеры, представляющие эквивалентные реализации в среде системы *Mathematica* в *Input*-режиме недопустимых примеров для системы *Maple* [100,101].

Подобно пакету *Maple*, пакет *Mathematica* является довольно сложным программным обеспечением, реализованным миллионами строк исходного кода, написанного в *C/C++*, *Java* и *Mathematica*. Программный код пакета на языке *C* фактически написан на специальном расширении языка *C*, которое поддерживает определенное управление памятью и объектно-ориентированные возможности. При этом, код оптимизирован, используя *Share* и *DumpSave*. При этом, программный код ядра пакета *Mathematica* по основным его функциональным составляющим процентуально можно охарактеризовать с определенной степенью погрешности как: поддержка встроенного *Math*-языка –  $30\%$ , алгебраические вычисления –  $20\%$ , числовые вычисления –  $20\%$ , графика и функции выхода –  $30\%$ . Исходный код ядра сохраняет процентуальное распределение данных составляющих независимо от используемой пакетом операционной платформы, тогда как объем исходного кода интерфейса пакета, написанного на языке *C++*, достаточно существенно зависит от используемой системой операционной платформы.

Пакет *Mathematica* использует вычислительную модель типа «client-server», в которой графический пользовательский интерфейс (GUI) и ядро пакета связываются между собой через протокол *MathLink*; с другими внешними средствами пакет поддерживает связь также по этому протоколу. Протокол *MathLink* поддерживает мульти-транспортные уровни (OSI), включая один, базирующийся на протоколе *TCP/IP*, и один, совместно использующий память. Наряду с указанным пакет использует ряд других протоколов. Вполне уместно упомянуть о такой весьма существенной характеристике для любого

программного средства, как совместимость «снизу-вверх». Несмотря на то, что с самого начала исходный код *Mathematica* изменялся весьма существенно от версии к версии, вплоть до того, что относительно самых первых версий остались неизменными лишь несколько процентов, а функциональные возможности пакета просто несопоставимы с первыми версиями, однако почти все программы, созданные в среде первых версий пакета *Mathematica* корректно выполняются и в старших версиях пакета. Тогда как в отношении пакета *Maple* это всегда было довольно острой проблемой. И не только на уровне версий пакета, но даже на уровне клонов одной и той же версии (например, для *Maple 9* режимов стандартный и классический). Именно для решения данной проблемы и был нами разработан комплекс процедур, обеспечивающий в значительной степени решение проблемы несовместимости. Естественно, в данном отношении *Mathematica* в большей степени отвечает требованиям к качественному программному продукту.

Следует отметить, особое неприятие у многих пользователей математических пакетов вызывает не весьма привычный синтаксис *Math*-языке. Действительно, его называют по-разному – и «нечеткий», и «странный», и «архаичный», и т.д., акцентируя внимание на том, что в противовес ему структура *Maple*-языка легко воспринимается для тех, у кого имеется даже небольшой опыт работы с процедурными языками. Частично с этим вполне можно согласиться, однако здесь следует иметь в виду, что «нечеткость» *Math*-языка во многом обусловлена как целым рядом причин исторического характера, так и его процедурно-функциональным гибридом, определив в нем ряд специфических парадигм, например, сравнение по образцам и т.д., что, однако, в целом ряде случаев оказывается весьма полезным при программировании в его среде как прикладных, так и системных средств. Правда, здесь имеются и не вполне убедительные решения как, например, кодирование функций с заглавных букв, скобки квадратные там, где для подавляющего большинства привычны круглые скобки, и т.д. И в этом отношении с определенной долей уверенности можно констатировать, что весь предыдущий опыт программирования не только не полезен, но и вступает в противоречие с парадигмой программной среды пакета *Mathematica*. Непривычность синтаксиса *Maple*-языка в течение довольно продолжительного времени будет портить настроение при наборе исходных кодов программ, правда, здесь имеются и свои плюсы в виде привычной для многих приложений прокрутки страниц текущего документа колесом мыши (в *Maple* это не поддерживается) и весьма удобной системы синтаксического анализа вводимого в *Input*-параграф текста, выполняемой в динамическом режиме, т.е. непосредственно в процессе посимвольного ввода. Подобной системой синтаксического анализа пакет *Maple* не располагает. Для пользователей *Maple* немало непривычного имеется и при наборе даже весьма элементарных выражений, например, если кодирование пробела между именем функции и скобкой, открывающей список фактических аргументов, в обоих языках допустимо, не вызывая ошибок, то кодирование за десятичной точкой пробела в *Maple* инициирует ошибочную ситуацию, тогда как в *Mathematica* пробел воспринимается как знак умножения, приводя зачастую к неожиданным результатам, как это очень наглядно иллюстрирует простой фрагмент, в котором пробел затенен:

```
> sin(42.47); => -0.9982894111  
> sin| (42.47); => -0.9982894111
```

```
> sin(42.47);  
Error, unexpected number  
In[2647]:= Sin[42.47]  
Out[2647]= -0.998289  
In[2648]:= Sin[42.47]  
Out[2648]= -0.998289  
In[2649]:= Sin[42.47]  
Out[2649]= 0.88187
```

Между тем, при всей первоначальной непривычности синтаксиса *Maple*-языка с ней вполне можно справиться в довольно непродолжительный срок, учитывая краткость и логичность его конструкций, выполняемых достаточно эффективно. Синтаксис его конструкций (*прежде всего значительных по объему*) требует определенного навыка для их восприятия. Привычному к синтаксису традиционных языков программирования пользователю *читабельность Math*-программ может оказаться весьма затруднительна. Многие примеры представленных в нашей книге [101] *процедурных* реализаций одной и той же задачи позволяют достаточно наглядно сравнивать такие *два* показателя, как *удобочитаемость* и *краткость*. В данном отношении, если исходный код *Mathematica* довольно явно отличается в сторону краткости, то исходный код *Maple*-процедуры в сторону большей удобочитаемости. И так как основу модульного программирования в обоих языках составляют базовые компоненты – процедуры/функции, здесь следует отдельно отметить существенно больший уровень читабельности *Maple*-процедур по отношению к *Mathematica*-процедурам. Резюмируя краткую характеристику языка *Math*, его синтаксис в целом можно объяснить тем обстоятельством, что *Mathematica* поддерживает процедурно-функциональный язык, сочетающий парадигмы 2 таких языков программирования, как *C++* и *APL*. Язык *APL* базируется на математической нотации и оказал весьма существенное влияние на разработчиков функциональных языков, электронных таблиц и ряда математических пакетов, включая *Mathematica*. Более того, парадигма *символьного* программирования *Mathematica* в качестве весьма важной составляющей включает понятие правил преобразования для произвольных символьных шаблонов. Поддерживающий их язык шаблонов позволяет очень удобно описывать достаточно общий набор классов выражений, делая возможным создавать удобочитаемые, изящные и эффективные программы. Вполне естественно, сочетание разных парадигм программирования и сопутствующих им синтаксисов делает коды, создаваемые в среде *Mathematica*, на первых порах несколько менее читательными.

Не детализируя более синтаксис *Math*-языка, с которым пользователь знакомится на первых шагах освоения пакета, отметим лишь, что если в *Maple* одной из важнейших базовых структур является *последовательность* выражений (*просто последовательность*), создаваемая на основе оператора *запятой* {,}, и составляет основу для создания многих типов структур (*вызовы функций, списки, множества и др.*), в *Mathematica*, практически, аналогичную роль играет *список*, являющийся удобной структурой функциональной парадигмы. И хотя в среде пакета *Maple* в определенной мере возможно реализовать функциональную парадигму, однако в большинстве случаев это потребует довольно серьезных усилий от пользователя и даже от достаточно искушенного пользователя.

Реальная мощь *Math*-языка обусловлена не только его управляющими структурами и структурами данных, но и всем богатством его функциональных средств (*встроенных, библиотечных, пакетных*) и прикладных средств (*в виде Math-пакетов*), которые созданы к настоящему времени пользователями из различных прикладных областей, прежде всего, математических. К их числу можно с полным основанием отнести и пакет [120], содержащий более 580 средств, дополняющих средства *Mathematica* и устраняющих некоторые ее недостатки наряду с расширением некоторых ее стандартных средств. Представленные в [120] средства использовались достаточно широко как при работе с *Mathematica* в интерактивном режиме, так и в программировании различных задач в его среде, зарекомендовав себя в качестве довольно полезных и эффективных. Более того, данные средства в прямом либо в модифицированном виде используются также и средствами, представляемыми настоящей книгой, с более детальным их описанием, в значительной степени ориентированным на лучшее освоение *Math*-языка пакета. В дальнейшем под «системными задачами» и разрешающими их средствами мы будем понимать те задачи, которые являются массовыми и представляют интерес в качестве средств для расширения процедурно-функциональной среды системы *Mathematica*.

Протокол *MathLink* пакета обеспечивает интерфейс не только между ядром и *GUI*, но и с другими языками: *C, AppleScript, Haskell, Visual Basic* и др. Ряд других протоколов обеспечивают интерфейс с рядом важных приложений. Между тем, *интегрированная среда разработки (IDE)* на основе *Eclipse* служит для работы с большими проектами в среде пакета, тогда как пакет *Maple* подобной *IDE* не имеет. Учитывая программный аспект, отметим, что *Mathematica Player* позволяет просматривать *Math*-документы (*файлы в cdf-формате*) без загрузки лицензионных версий *Mathematica*, тогда как для *Maple* необходима загрузка самого лицензионного пакета, достаточно существенно затрудняя просмотр даже доступных кодов. Правда, на основе файлов с документами {*mws, mw*} *Maple* и *m*-файлов с пакетами, в принципе, не составляет особого труда создавать перлюстраторы программных кодов, некоторые из которых представлены в наших книгах [100, 101]. Данные средства дают возможность перлюстрировать коды, созданные в среде обеих систем, вне их программной среды, не требуя самих систем.

Так как *Math*-язык является одновременно и языком *реализации* пакета, его освоение и практическое программирование в его среде позволят не только весьма существенно повысить ваш уровень использования предоставляемых пакетом возможностей (*уровень владения пакетом*), но и глубже понять как идеологию, так и внутреннюю кухню самого пакета. Ввиду ведущей роли *Mathematica* среди средств современной компьютерной алгебры и во многом распространенную его идеологию в этой области, вы получаете прекрасную и пролонгированную возможность весьма эффективного использования подобных ему средств в своей профессиональной деятельности, прежде всего, в таких областях, как математика, физика, информатика, техника и целый ряд других.

Сравнение систем программирования обоих пакетов производилось на основе *Maple 8÷11* и *Mathematica 7÷9*. Учитывая, что внутренний язык программирования, который составляет основу программирования пакетов, подвергается временным изменениям, как правило, в незначительной степени, то данный материал можно рассматривать в качестве достаточно пролонгированного. Между тем, преследовались цели не только



сравнительного характера, но также попытка в данном аспекте представить основные элементы систем *модульного программирования* обеих пакетов наряду с представлением ряда достаточно полезных средств как расширяющих программные средства пакетов, так и устраняющих обнаруженные недостатки и ошибки стандартных средств. Более того, рассмотрение и сравнительный анализ проводились лишь в рамках встроенных языков систем без привлечения средств, обеспечивающих интерфейс с внешними по отношению к системам программными средами [97,98]. Итак, сравнение языков *обеих* систем проводилось лишь в рамках их *встроенных* средств, исключая из рассмотрения средства, позволяющие использовать внешние программные средства, что повышает уровень объективности; т.е. производилось сравнение замкнутых программных сред обеих систем [97–101]. Следует иметь в виду, что именно в результате проведенного в [97–101] анализа обеих систем компьютерной математики появился наш *первый* пакет *AVZ\_Package* [90], содержащий средства, не только имитирующие средства системы *Maple*, отсутствующие в системе *Mathematica*, но и средства, как улучшающие либо расширяющие ее стандартные средства, так и средства расширяющие программную среду системы *Mathematica* в целом. Тогда как в рамках подготовки настоящей книги данный пакет был существенно переработан по указанным выше направлениям и в результате этой работы появился нынешний пакет *AVZ\_Package\_1* [120], чьи средства и представляет настоящая книга. Здесь целесообразно отметить следующий аспект.

В процессе программирования приложений в некоей программной среде, серьезный пользователь зачастую сталкивается с ситуацией, когда для его задач недостаточно ее стандартных средств или эти средства *неудовлетворительны* по требуемым критериям эффективности, функциональности, робастности и т.д. В таком случае вполне может потребоваться разработка собственных средств, как замещающих стандартные (*целый ряд программных систем являются закрытыми для пользователя, например, Mathematica*) средства, так и новых, отсутствующих в используемой программной среде. В качестве весьма наглядного примера можно привести создание нашего пакета *AVZ\_Package\_1*, когда в *процессе* программирования целого ряда его процедур и функций появлялась целесообразность допрограммирования новых средств, как упрощающих алгоритмы исходных средств, так и повышающих их прозрачность, реактивность и робастность. В тексте настоящей книги в целом ряде случаев *будут* проиллюстрированы подобные ситуации даже в ущерб принятой нами (*пусть даже в определенном контексте условной*) классификации типов средств по главам и разделам. Описанная технология работы с программной средой с течением времени позволяет создавать тот инструментарий, с которым пользователь имеет возможность более эффективно использовать для своих приложений. Более того, устоявшийся инструментарий может оказаться достаточно полезным для довольно широкого круга пользователей данной программной среды. В частности, наш пакет *AVZ\_Package\_1*, имеющий *FreeWare* лицензию, используется широким кругом пользователей *Mathematica*, подтверждая свою эффективность.

Система *Mathematica* в плане *процедурного программирования* в определенной степени уступает системе *Maple*, хоть и декларирует его как свою *особую* отличительную черту. Между тем, именно система *Maple* поддерживает встроенный язык ярко выраженного *процедурного* типа, тогда как система *Mathematica* использует смешанный *процедурно-*

функциональный язык, в основу которого положено комбинирование элементов таких языков как C++ и APL. Для более детального ознакомления с возможностями системы *Mathematica* читатель отсылается к нашим книгам [97-101] или руководству [117], где рассматриваются основные элементы процедурной и функциональной парадигмы в среде *Mathematica*. При этом, нами не рассматриваются средства программирования собственно *Math*-документов, а лишь системные средства, обеспечивающие массовые задачи процедурно-функционального программирования и расширяющие средства собственно самой системы *Mathematica*. В упомянутых книгах [100,101] представлено несколько версий целого ряда средств, что обусловлено рядом причин. Прежде всего, данный подход преследовал цель проиллюстрировать различные приемы, полезные в практическом программировании; во-вторых, данный подход достаточно хорошо иллюстрирует методику оптимизации программных реализаций в случае появления новых средств как стандартных, так и нестандартных. Наконец, в целом ряде случаев последующие версии, зачастую, довольно существенно расширяют функциональные возможности предыдущих версий. Все такие моменты достаточно поучительны при освоении программирования в среде системы *Mathematica*, учитывая ее встроенный *Math*-язык, достаточно отличный от привычных языков программирования.

Настоящая книга снабжена большим числом исходных кодов (*листингов*) процедур и функций, снабженных краткими описаниями (*справками*), позволяющими понимать их назначение и форматы вызовов. Их тексты крайне целесообразно не копировать, а набирать вручную, т.к. в это время у читателя работает, так называемая, «*моторная*» (*автоматическая*) память и происходит запоминание многих важных моментов. Более того, при данном подходе не только работает моторика, но также на основе кратких описаний сути алгоритмов, реализуемых процедурами и функциями, проясняются и многие важные особенности их реализации, уточняется понимание использованных алгоритмов. При этом, набирать тексты процедур и функций не обязательно один в один – экспериментируйте с кодом как можно больше, работайте как можно больше в среде системы, пытаясь внести что-то свое и результат не заставит себя долго ждать. В настоящей книге представлены средства различного назначения, они могут быть использованы как на практике при написании *реальных* проектов в среде системы, так и в использовании в качестве прототипов приемов программирования, примененных в них. Нужно лишь разобраться с логикой и принципом работы алгоритма, и далее в нужных ситуациях уже использовать их в своей практической работе в *Mathematica*. Со временем работа с этими алгоритмами и приемами дойдет у вас до автоматизма. Для удобства практической работы с этими средствами они оформлены в отдельный пакет [120], распространяемый по лицензии *Freeware*, который может быть полезен и в учебных целях, и при программировании прикладных задач различного типа. Его функциональные средства расширяют программную среду системы *Mathematica*, в целом ряде случаев устраняя ошибки и ограничения стандартных средств системы.

Следует еще раз акцентировать внимание на том, что целью настоящего издания было не систематическое изложение языка программирования в *Mathematica*, но, в первую очередь, представление как средств, расширяющих программную среду системы, так и ряда довольно полезных как стандартных, так и нестандартных приемов и методов

для повышения эффективности программирования различного рода приложений в ней. Вместе с тем, не являясь систематическим изложением системы *программирования* в среде *Mathematica*, с той либо иной степенью детализации представлены только те разделы программной среды *Mathematica*, которые непосредственно либо косвенно связаны со средствами, представленными в книге ниже. Поскольку цель изложения в значительной степени состояла в представлении новых средств, которые расширяют программную среду *Mathematica*, изложение в значительной степени действительно носит фрагментарный характер, что, на наш взгляд, вполне допустимо с учетом того уровня читателя, на который оно и рассчитано. Начинающему пользователю работы с системой на первых порах настоящая книга особого интереса не представит. Тогда как для пользователя, уже имеющего определенный опыт работы с *Mathematica*, она может представить вполне определенный интерес в плане дальнейшего повышения его профессионализма. Более того, в процессе работы над книгами настоящей серии [97-101] с появлением новых средств создавались и новые версии ряда предыдущих; в целом ряде случаев старые версии средств сохранялись в прилагаемом к книге пакете *AVZ\_Package\_1* [120], как содержащие ряд достаточно полезных подходов и приемов, ориентированных на программирование различных *приложений* в среде *Mathematica*. Надеемся, что предлагаемая книга представит вполне определенный интерес для тех пользователей системы *Mathematica*, которые в своей *профессиональной* деятельности намереваются использовать его с максимальной степенью отдачи. И прежде всего для тех пользователей, которым в ее среде предстоит создавать приложения, требующие нетривиального программирования. К таким приложениям можно отнести довольно большие проекты, а также задачи системного характера различного назначения.

При этом, в связи с разработкой нестандартных программных средств, подобных тем, которые составляют пакет *AVZ\_Package\_1*, могут возникать и вопросы, относящиеся к проблеме совместимости программного обеспечения. Однако, по нашему мнению такого типа вопросы являются надуманными. Действительно, разработка подобного типа приложений, включая расширяющие стандартную программную среду, совсем не нарушает принцип совместимости как и разработка любого приложения. Наличие такого приложения все равно требует версии, предполагается, лицензионной данной (*Mathematica*) либо иной среды, под которую оно было разработано. Поэтому, пакет *AVZ\_Package\_1*, имеющий *FreeWare*-лицензию, также совершенно не нарушает этот принцип совместимости, предоставляя пользователю более развитую в программном отношении среду *Mathematica + AVZ\_Package\_1*. Вопрос сводится к целесообразности использования чистой среды *Mathematica* или с применением расширений, впрочем, данный вопрос является вопросом именно целесообразности, а не совместимости. Он обусловлен лишь как теми задачами, которые стоят перед пользователем, его уровнем владения стандартной средой, так и потребностью в дополнительных средствах. Наш опыт использования пакета *AVZ\_Package\_1* совместно с *Mathematica 9.0* в разработке целого ряда приложений различного назначения достаточно однозначно подтвердил эффективность такого *симбиоза* – лишним хорошее программное средство не бывает. К данному выводу приводит нас и многолетний опыт программирования в целом на различных вычислительных платформах и в различных программных средах.

### Глава 3. Средства для режима текущего документа *Mathematica*

Документы текущего сеанса системы *Mathematica* (содержимое экрана, которое видит в процессе работы с системой пользователь) имеют *секционную* структуру, каждая секция которой в общем случае идентифицируется *входом* **In[n]** и *выходом* **Out[n]**, когда на *вход* подается *выражение*, которое требуется вычислить, а на *выходе* отображается сам *результат* данного вычисления. Более того, между этими точками могут появляться различного рода *сообщения*, связанные с данным вычислением. Например, возможно появление сообщений об обнаруженных системой ошибках при вычислении, как это весьма наглядно иллюстрирует следующий весьма простой пример:

```
In[2042]:= x = 0; Sin[x]/x
```

```
Power::infy: Infinite expression 1/0 encountered. >>
```

```
Infinity::indet: Indeterminate expression 0 ComplexInfinity encountered. >>
```

```
Out[2042]= Indeterminate
```

При этом, через гиперссылку «>>» обеспечивается выход на справочную страничку с более детальной информацией по возникшей ошибочной или особой ситуации. При этом, конструкция формата «**Out[n]=** », говорит о том, что в результате вычисления выражения *Mathematica* ничего не возвращает, в то время как конструкция формата «**Out[n]= Null**» сообщает о том, что в процессе вычисления выражения возвращается **Null**, т.е. ничего. Более того, при вычислениях на основе оператора «:=» отложенного присвоения также ничего не возвращается, включая конструкцию «**Out[n]=** », как это весьма наглядно иллюстрирует следующий простой пример:

```
In[2044]:= a := {78, 460}
```

```
In[2045]:= G[x_;/ IntegerQ[x]] := x^6
```

```
In[2046]:= a + G[78]
```

```
Out[2046]= {225199600782, 225199601164}
```

```
In[2047]:= G[(42/70)^3]
```

```
Out[2047]= G[27/125]
```

Комментируя данный пример, отметим, что вычисления в среде пакета полагают два уровня присвоения значений, а именно: *непосредственный* ( $x = a$ ) и *отложенный* ( $x := a$ ). В первом случае  $x$  получает значение  $a$  сразу после вычисления данного выражения, во втором – только в момент использования (обращения к)  $x$ . И здесь сразу же отметим, что в процессе работы с блоками/функциями/модулями результаты их вызовов могут быть *невычисленными*, т.е. возвращаются их вызовы только с упрощением значений, которые используются в качестве их *фактических* аргументов (*упрощения, допускаемые соглашениями программной среды системы Mathematica*).

Итак, в примере проиллюстрировано отсутствие «**Out[n]=** » в результате вычисления определения простой функции **G[x]** и отложенного присвоения для переменной  $a$ , а также возврат вызова функции **G** невычисленным в случае получения ею фактического аргумента типа, отличного от *Integer*. В процессе работы с текущими документами во многих случаях возникает потребность использования ранее вычисленных *выражений* в предыдущих *Input*-параграфах. Для данной цели служит %-оператор, для которого

формат кодирования имеет следующий весьма простой вид, а именно:

$\%, \% \%, \% \% \dots \% \% (k \text{ раз}), \% k$

определяющий возврат последнего, предпоследнего и  $k$ -го предыдущего результата вычислений в текущем сеансе системы. При этом, система *Mathematica* поддерживает две весьма полезные предопределенные глобальные переменные, а именно:

**\$Line** – определяет номер последнего **In**-параграфа текущего сеанса;

**\$HistoryLength** – определяет число предыдущих **In**-параграфов и **Out**-параграфов, сохраняемых в текущем сеансе пакета.

Более того, обе эти переменные допускают переопределения простым присвоением новых значений, как весьма наглядно иллюстрирует следующий простой пример:

```
In[2052]:= {$Line, $Line = 0, $Line}
```

```
Out[0]= {2052, 0, 0}
```

```
In[1]:= {$HistoryLength, $HistoryLength = 0, $HistoryLength}
```

```
Out[1]= {∞, 0, 0}
```

Для переменной **\$HistoryLength** значением по умолчанию является *бесконечность* ( $\infty$ ); используя для этой переменной меньшие установки, можно существенно экономить размер требуемой оперативной памяти для системы *Mathematica*.

В свою очередь глобальная переменная **\$Line1** в отличие от стандартной глобальной переменной **\$Line** определяет общее число **Out**-параграфов текущего сеанса пакета, включая результаты вычисления загруженных в него пакетов пользователя из файлов формата {"cdf", "nb"}. Фрагмент представляет исходный код и примеры применения.

```
In[460]:= $Line1 := Block[{a = "", c = "Out[", k = 1}, For[k, k < Infinity, k++,  
a = ToString[Out[k]]; If[a == c <> ToString[k] <> "]", Return[k]]]; k]  
In[461]:= $Line1  
Out[461]= 6077  
In[462]:= $Line  
Out[462]= 462
```

Приведенные примеры достаточно наглядно иллюстрируют ситуацию, когда после загрузки пакета пользователя значения **\$Line1** и **\$Line** могут достаточно существенно различаться: первое, определяя общее число сохраненных **Out**-параграфов, тогда как второе – число реально полученных **Out**-параграфов в текущем сеансе *Mathematica*.

Следует отметить, что %-операторы в системе *Mathematica* и системе *Maple* являются концептуально различными. Так, если в первом случае %-оператор непосредственно ассоциирован только с **Out**-параграфами текущего сеанса на глубину, допускаемую системой, что в целом ряде случаев работы в режиме *документа* представляется весьма полезным, то во втором случае %-оператор имеет глубину не более 3 и ассоциирован с результатами выполнения как предыдущих **Out**-параграфов, так и предложений. В данном контексте в *первом* случае использование %-оператора в блоке либо модуле, в целом, не имеет особого смысла, в то время как во *втором* – применение %-оператора в процедурах имеет вполне определенный смысл, что использовалось нами довольно

широко при создании библиотеки *UserLib* [45]. Примеры, представленные в [100,101], достаточно наглядно иллюстрирует вышесказанное.

Итак, реальной областью применимости *%*-оператора в *Mathematica* является текущий сеанс, исключая область и блоков, и модулей, в то время как областью применимости *%*-оператора в среде *Maple* является текущий сеанс в целом, вместе с тем, на глубину, не большую чем. Таким образом, *%*-операторы (*Ditto operators*), имея в *Mathematica* и *Maple* различные реальные области применимости, вместе с тем, обладают как своими недостатками, так и существенными преимуществами. Ранее уже отмечалось, путем изменения значения для переменной **\$HistoryLength** (сохраняется до переопределения, если установка размещена в инициализационном файле *init.m*) можно весьма существенно экономить размер оперативной памяти, требуемой для рабочей области *Mathematica*. По этой же причине накладываются и ограничения на историю вычислений путем ограничения количества реальных обновлений определений у процедур и функций. Вероятно, пользователь заметил, что после некоторой попытки обновить определение процедуры оно остается прежним. Это обстоятельство следует учитывать при отладке блоков/функций/модулей, периодически применяя к ним функции **Clear**, **ClearAll**.

В целом ряде случаев работы с большими документами возникает целесообразность в удалении из текущего сеанса ранее использованных **Out**-параграфов с ненужными в дальнейшем результатами. Эту операцию обеспечивает простая процедура **ClearOut**, чей вызов **ClearOut[x]** ничего не возвращает, одновременно удаляя **Out**-параграфы с номерами, определяемыми целым положительным числом либо их списком *x*. Ниже представлен исходный код процедуры с типичными примерами ее применения. Эта процедура в ряде случаев также обеспечивает выделение дополнительной памяти в рабочей области системы, что в случае больших документов довольно существенно.

```
In[2520]:= ClearOut[x_ /; PosIntQ[x] || PosIntListQ[x]] := Module[{a = Flatten[{x}], k = 1},
    Unprotect[Out]; For[k, k <= Length[a], k++, Out[a[[k]]] = .]; Protect[Out]; ]

In[2521]:= {Out[1508], Out[1510], Out[1511], Out[1515]}
Out[2521]= {42, 78, 2013, 460}

In[1522]:= ClearOut[{1508, 1510, 1511, 1515}]
In[1523]:= {Out[1508], Out[1510], Out[1511], Out[1515]}
Out[1523]= {%1508, %1510, %1511, %1515}
```

Исходный код достаточно прозрачен и каких-либо особых пояснений не требует.

С другой стороны, в ряде случаев работы в режиме документа появляется надобность замены **Out**-параграфов иным содержимым, что обеспечивает достаточно несложная процедура **ReplaceOut**, чей успешный вызов **ReplaceOut[x, y]** ничего не возвращает, в то же время выполняя замену содержимого существующих **Out**-параграфов, которые заданы целым положительным или их списком *x*, на новое выражение, определенное *y*-аргументом. Вызов предполагает паритет числа заменяемых **Out**-параграфов числу выражений, заменяющих их *текущие* значения; в противном случае вызов процедуры **ReplaceOut[x, y]** возвращает невычисленным. Следующий фрагмент представляет исходный код процедуры **ReplaceOut** с типичными примерами ее использования.

```

In[2025]:= Agn = 78
Out[2025]= 78

In[2026]:= ReplaceOut[x_;/; PosIntQ[x] || PosIntListQ[x], y___] :=
                Module[{a = Flatten[{x}], b = Flatten[{y}], k = 1},
                If[b != {}, If[Length[a] != Length[b], Defer[ReplaceOut[x, y]], Unprotect[Out];
                For[k, k <= Length[a], k++, Out[a[[k]]] = b[[k]]; Protect[Out];, ClearOut[x]]]

In[2027]:= ReplaceOut[2025, 460]
In[2028]:= Out[2025]
Out[2028]= 460
In[2029]:= ReplaceOut[2025]
In[2030]:= Out[2025]
Out[2030]= %2025
    
```

Вообще говоря, процедура **ReplaceOut** обобщает предыдущую процедуру **ClearOut**.

**Определение переменных в среде Mathematica.** Подобно большинству программных языков в системе *Mathematica* для значений используются *имена* (идентификаторы), давая возможность в дальнейшем обращаться к таким *именованным* выражениям по их именам; примеры ввиду их чрезвычайной простоты не приводятся. По оператору "=" производится *немедленное* присвоение одной или нескольким *переменным* требуемого значения, тогда как по конструкции "x = ." или функциям **Clear**, **ClearAll** переменная *x* очищается от приписанного ей значения, как иллюстрирует простой пример:

```

In[2030]:= t = 2013; x = y = z = 2013
Out[2030]= 2013
In[2031]:= {t, x, y, z}
Out[2031]= {2013, 2013, 2013, 2013}
In[2039]:= t =.; x =.; y =.; z =.
In[2040]:= {t, x, y, z}
Out[2040]= {t, x, y, z}
    
```

Для определения типа *присвоения*, примененного к *символу* (имени), можно применять достаточно простую процедуру **DefOp**, чей вызов **DefOp[w]** возвращает в строчном формате тип *присвоения*, примененного к имени *x*, закодированному тоже в строчном формате: (1) "Undefined" – имя *w* не определено, (2) "=" – к имени *w* было применено *немедленное* присвоение и (3) ":=" – к имени *w* применено *отложенное* присвоение. В приведенном фрагменте представлен исходный код процедуры с примерами.

```

In[2040]:= DefOp[x_;/; StringQ[x] && SymbolQ[x] || SymbolQ[ToExpression[x]], y___] :=
                Module[{a = DefOpt[x], b = {y}, c, d}, If[a == "Null", Return["Undefined"],
                c[h_] := StringTake[a, {Flatten[StringPosition[a, h]][[2]] + 1, -1}]];
                If[SuffPref[a, x <> "=", 1], d = "=", d = ":="];
                If[b != {} && ! HowAct[y], y = c[d]]; d]

In[2041]:= v = 78; g = 66; s := 46; Kr = 17; Art := 24; Res := a + b + c;
    
```

```
In[2042]:= Map[DefOp, {"v", "g", "s", "Kr", "Art", "Res", "Avz"}]
Out[2042]= {"=", "=", ":=", "=", ":=", ":=", "Undefined"}
In[2043]:= Clear[y]; {DefOp["Art", y], y}
Out[2043]= {":", "24"}
In[2044]:= Clear[y]; {DefOp["Res", y], y}
Out[2044]= {":", "a + b + c"}
In[2045]:= {x, y, z, h, g, v, s} := {a1, b1, c1, d1, f1, g1, h1};
In[2045]:= Map[DefOp, {"x", "y", "z", "h", "g", "v", "s"}]
Out[2046]= {":", ":=", ":=", ":=", ":=", ":=", ":="}
In[2046]:= {x, y, z, h, g, v, s} = {a2, b2, c2, d2, f2, g2, h2};
In[2047]:= Map[DefOp, {"x", "y", "z", "h", "g", "v", "s"}]
Out[2047]= {"=", "=", "=", "=", "=", "=", "="}
```

Тогда как вызов **DefOp**[*w*, *y*] через необязательный второй аргумент *y* – *неопределенная переменная* – возвращает выражение, присвоенное символу (*переменной*) *w*. Значение, присвоенное переменной *x*, остается ассоциированным с ней до его удаления по *x* = ., по функциям **Clear**, **ClearAll**, **Remove** или ее переопределения. Во избежание ошибок при работе с большим документом рекомендуется очищать переменные сразу же по мере отсутствия в них необходимости. *Имя* может иметь любую длину, но не может начинаться с цифры. В имени допустимы заглавные и строчные буквы; при этом, по соглашению встроенные объекты системы *Mathematica* начинаются с *заглавных* букв, поэтому рекомендуется использовать и пользовательские идентификаторы, которые начинаются со строчных букв. Однако, создавая свои библиотеки или пакеты, вполне допустимо для блоков, функций и модулей пользователя применять имена, которые начинаются с заглавных букв. Именно подобный подход использован нами в пакете *AVZ\_Package\_1* [120], содержащем средства, представленные в настоящей книге. При этом, для замены в выражениях подвыражений используется *правило замены* "*->*". Если к выражению применяется несколько правил, то они кодируются в виде списка. Для вычисления присвоений *Math*-язык располагает функцией **Definition**, вызов которой **Definition**[*x*] возвращает все определения, приписанные символу (*имени*) *x*. В случае нескольких определений, приписанных символу *x*, такой символ будет называться в дальнейшем нами *одноименным* и применение к такому символу функции **Map** или использование его в *списочной* структуре (*например, In-параграфы с номерами 2061, 2062*) визуализирует результаты некорректно, как это иллюстрируют примеры следующего фрагмента. В качестве выхода здесь можно использовать либо применение функции **Definition** к символам по отдельности, применять к результату **InputForm**-функцию либо нашу процедуру **DefOpt**, рассматриваемую в настоящей книге ниже. Наряду с данной процедурой рассматриваются и другие средства возврата определений.

```
In[2060]:= x = 78; y := 2013;
In[2061]:= {Definition[x], Definition[y]}
Out[2061]= {x = 78
In[2062]:= Map[Definition, {"x", "y"}]
Out[2062]= {x = 78
```



```
In[2063]:= InputForm[{Definition[x], Definition[y]}]
Out[2063]//InputForm=
      {x = 78, y := 2013}
In[2064]:= Map[DefOpt, {"x", "y"}]
Out[2064]= {"x = 78", "y := 2013"}
In[2064]:= Definition[x]
Out[2064]= x = 78
In[2064]:= Definition[y]
Out[2064]= y := 2013
```

В целом ряде случаев возникает необходимость очистки переменных текущего сеанса от полученных в результате динамической генерации значений. С этой целью можно использовать механизм, состоящий в *аккумуляции* в списке значений, переменные с которыми впоследствии должны быть *удалены* из текущего сеанса или *очищены* как от своих значений, так и атрибутов. Для этого можно воспользоваться функцией, вызов которой **ClearValues[x]** возвращает пустой список, одновременно удаляя из текущего сеанса все переменные, имеющие значения из списка *x*; тогда как вызов **ClearValues[x, y]** со вторым необязательным аргументом *y* – произвольным выражением – возвращает пустой список, однако такие переменные лишь очищаются от значений и атрибутов без удаления из текущего сеанса. Следующий фрагмент представляет исходный код функции **ClearValues** с наиболее типичными примерами ее использования.

```
In[2070]:= ClearValues[x_;/; ListQ[x], y___] := Select[Map[If[{y} == {}, Remove, ClearAll],
      Select[Names["*"], MemberQ[x, ToExpression[#]] &]], # != "Null" &]

In[2071]:= {a = 42, b = 78, c := 75, d = 460, h5 := 66, Kr = 17, Art = x + Sin[y]}
Out[2071]= {42, 78, Null, 460, Null, 17, 78 + Sin[2013]}
In[2072]:= ClearValues[{42, 78, 75, 460, 66, 17, x + Sin[y]}]
Out[2072]= {}
In[2073]:= Names["*"]
Out[2073]= {"ClearValues"}
In[2074]:= {a, b, c, d, h5, Kr, Art}
Out[2074]= {a, b, c, d, h5, Kr, Art}
In[2075]:= {a = 42, b = 78, c := 75, d = 460, h5 := 66, Kr = 17, Art = x + Sin[y]}
Out[2075]= {42, 75, Null, 450, Null, 16, x + Sin[y]}
In[2076]:= ClearValues[{42, 78, 75, 460, 66, 17, x + Sin[y]}, 78]
Out[2076]= {}
In[2077]:= Names["*"]
Out[2077]= {"a", "Art", "b", "c", "ClearValues", "d", "h5", "Kr"}

In[9]:= VarsValues[x_;/; ListQ[x]] := Select[Names["*"], MemberQ[x, ToExpression[#]] &]

In[10]:= {a = 42, b = 78, c := 75, d = 460, h5 := 66, Kr = 17, Art = x + Sin[y]}
Out[10]= {42, 75, Null, 450, Null, 16, x + Sin[y]}
In[11]:= VarsValues[{42, 78, 75, 460, 66, 17, x + Sin[y]}]
Out[11]= {"a", "Art", "b", "c", "d", "h5", "Kr"}
```

Во второй части фрагмента представлена довольно простая функция **VarsValues**, чей вызов **VarsValues[x]** возвращает список переменных в строчном формате, имеющих значения из списка *x*. Обе функции представляют определенный интерес при работе с переменными текущего сеанса пакета. Прежде всего, в целом ряде случаев возникает задача очистки памяти от большого числа динамически сгенерированных значений, в то время как в целом ряде случаев требуется определить переменные, получившие в текущем сеансе значения по любому оператору присваивания. Если в первом случае вопрос имеет самое прямое отношение к задаче оптимизации использования памяти, в то время как во втором важен с точки зрения адресного использования значений.

В ряде случаев на основе некоторого значения требуется определить имена, которым в текущем сеансе было присвоено данное значение. Данную задачу решает процедура **Nvalue[h]**, возвращающая список имен в строчном формате с заданным значением *h*. При этом, процедура отражает только те глобальные переменные, которые получили значения в текущем сеансе в *Input*-параграфах. В отсутствие таких имен процедура возвращает пустой список, т.е. {}. Следующий фрагмент представляет исходный код и наиболее типичные примеры использования процедуры **Nvalue**.

```
In[2025]:= Nvalue[x_] := Module[{a = {}, b = Names["`*"], k = 1}, For[k, k <= Length[b], k++,
    If[ToExpression[b[[k]]] == x, AppendTo[a, b[[k]]], Next[]]; a]
In[2026]:= {Agn, Avz, Art, Kr, Varn, $Art$, Vsv, $Kr$, Grsu} = {71, 66, 17, 24, 78, 460, Null,
    2013, a + b}; Map[Nvalue, {71, 66, 17, 24, 78, 460, Null, 2013, a + b}]
Out[2026]= {"Agn"}, {"Avz"}, {"Art"}, {"Kr"}, {"Varn"}, {"$Art$"}, {"Vsv"}, {"$Kr$"}, {"Grsu"}]
```

Следующая процедура является расширением функциональных возможностей ранее представленной процедуры **Nvalue**. Вызов процедуры **Nvalue1[x]** возвращает список имен переменных в строчном формате, которым в текущем сеансе системы присвоено значение *x*. Следующий фрагмент представляет исходный код **Nvalue1** с примерами.

```
In[2034]:= Nvalue1[x_] := Module[{a = {}, b = Select[Names["`*"], StringFreeQ[#, "$"] &], c,
    k = 1}, While[k <= Length[b],
    c = ToExpression["Attributes[" <> ToString1[b[[k]]] <> "]"];
    If[! MemberQ[c, Protected], AppendTo[a, b[[k]]], Null]; k++];
    Select[a, ToExpression[#] === x &]]
In[2035]:= {x, y, z, t, h, g, w, s, u} = {45, 78, 24, 17, 17, 17, 17, 17, 17}; Nvalue1[17]
Out[2035]= {"Art", "g", "h", "s", "t", "u", "w"}
In[2036]:= T = TimeUsed[]; m = Nvalue1[17]; TimeUsed[] - T
Out[2036]= 6.25
In[2037]:= m
Out[2037]= {"Art", "g", "h", "s", "t", "u", "w"}]
```

В частности, из последнего примера фрагмента следует, что процедура **Nvalue1** имеет не вполне удовлетворительные временные характеристики по причине используемого ею алгоритма, базирующегося на анализе 5181 (данное число определяется конкретикой текущего документа) активных объектов как системных, так и пользовательских.

Для определения значений, присвоенных переменным, довольно полезной является процедура, чей вызов **WhatValue[x]** возвращает значение, присвоенное переменной  $x$ ; на неопределенной переменной  $x$  возвращается **"Undefined"**, тогда как на системном  $x$  возвращается список формата {"System", " $x$ "}, а на локальной переменной  $x$  – список формата {"Local", " $x$ "}, результат вызова процедуры возвращается в *строчном* формате. Следующий фрагмент представляет исходный код процедуры **WhatValue** совместно с достаточно типичными примерами ее использования.

```
In[2044]:= WhatValue[x_] := Module[{a = ToString1[DefFunc[x]], c = "HeadPF["},
                                b = Quiet[ToString[HeadPF[x]]]],
  If[a == "Attributes[" <> ToString[x] <> "]" = {"Temporary"}, {"Local", x},
  If[HeadingQ[b], StringTake[a, {Flatten[StringPosition[a, b <> " := "][[2]] + 1, -1}],
  If[SuffPref[b, c, 1], StringTake[StringReplace[b, c -> "", 1], {1, -2}],
  If[SuffPref[a, "Attributes[" <> ToString[x] <> "]" = ", 1] &&
    b == ToString[x], {"System", b}, "Undefined"]]]]]

In[2045]:= Ag[x_] := Module[{}, x^2]; Sv[x_] := Block[{a}, a + x]; F[x_, y_] := x^2 + y^2
In[2046]:= Map[WhatValue, {460 + 78*# &, hg, Sin, Ag, Sv, 78, a + b, F, Gs}]
Out[2046]= {"460 + 78 #1 & ", "Undefined", {"System", "Sin"}, "Module[{}, x^2]",
  "Block[{a}, a + x]", "78", "a + b", "x^2 + y^2", "Undefined"}
In[2047]:= M = Module[{avz}, avz]; WhatValue[M]
Out[2047]= {"Local", avz$15239}
```

Вызов стандартной функции **Clear[x1, x2, x3, ..., xn]** очищает символы { $x_1, x_2, x_3, \dots, x_n$ } от присвоенных им выражений; при этом, исключение составляют только символы с *Protected*-атрибутом, как иллюстрируют достаточно простые примеры следующего фрагмента. В качестве полезного обобщения функций **Clear** и **ClearAll** вполне можно рассмотреть процедуру, чей вызов **Clear1[h, "x1", "x2", "x3", ..., "xn"]** возвращает **Null**, т.е. ничего, при  $h=1$  очищая символы { $x_1, x_2, x_3, \dots, x_n$ } от присвоенных им выражений с сохранением всех их атрибутов и опций, тогда как при  $h=2$ , очищая символы { $x_1, x_2, x_3, \dots, x_n$ } как от присвоенных им выражений, так и *всех* атрибутов и опций. Фрагмент представляет исходный код **Clear1** с типичными примерами ее использования.

```
In[2058]:= Clear1[x_;/ MemberQ[{1, 2}, x], y___] := Module[{a = {y}, b, c, d, k = 1},
  If[y === {}, Null, For[k, k <= Length[a], k++, b = a[[k]];
  d = Quiet[ToExpression["Attributes[" <> ToString1[b] <> "]"]];
  ToExpression["Quiet[ClearAttributes[" <> ToString1[b] <> ", " <>
  ToString[d] <> "]" <> "; Clear" <> If[x == 1, "", "All"] <> "[" <>
  ToString1[b] <> "]""]; If[x == 2, Null, Quiet[Check[ToExpression[
  "SetAttributes[" <> ToString1[b] <> ", " <> ToString[d] <> "]"], $Failed]]]]]]

In[2059]:= S[x_, y_] := x*y; SetAttributes[S, Protected]; Clear["S"]; {Attributes[S], S[5, 6]}
SetDelayed::write: Tag S in S[x_, y_] is Protected. >>
Clear::wrsym: Symbol S is Protected. >>
Out[2059]= {{Protected}, 30}
```

```
In[2060]:= ClearAll[S]; {Attributes[S], S[5, 6]}
ClearAll::wrsym: Symbol S is Protected. >>
Out[2060]= {{Protected}, 30}
In[2061]:= {x, y} = {78, 460}; Clear1[2]; Clear1[2, "78"]; {x, y}
Out[2061]= {78, 460}
In[2062]:= SetAttributes[G, Listable]; G[x_, y_] := x*y; SetAttributes[G, Protected];
Clear1[1, "G"]; {G, Attributes[G]}
Out[2062]= {G, {Listable, Protected}}
In[2063]:= SetAttributes[V, Listable]; V[x_, y_] := x*y; SetAttributes[V, Protected];
Clear1[2, "V"]; {V, Attributes[V]}
Out[2063]= {V, {}}
In[2064]:= Clear1[2, "V", "G"]; {V, G, Map[Attributes, {V, G}]}
Out[2064]= {V, G, {}, {}}
In[2065]:= {Clear1[2], Clear1[14, "x"], Clear1[], Clear1[1, 460]}
Out[2065]= {Null, Clear1[14, "x"], Clear1[], $Failed}
```

В качестве достаточно простого и полезного средства можно представить и функцию **UnDef**, вызов которой **UnDef[x]** возвращает **True**, если символ  $x$  не определен, и **False** в противном случае. Тогда как вызов **UnDef[x, y]** со 2-м необязательным аргументом – неопределенной переменной  $y$  – возвращает через  $y$  значение **Head1[x]**. Отметим, что в целом ряде случаев процедурного программирования **UnDef** оказывается довольно полезным средством. Следующий фрагмент представляет исходный код процедуры **UnDef** наряду с достаточно типичными примерами ее использования.

```
In[2090]:= UnDef[x_, y___] := Module[{a = {y}, g},
g[h_] := Quiet[If[Off[Definition::notfound]; (ToString[h] // Definition1) ==
"Null", On[Definition::notfound]; True, On[Definition::notfound]; False]];
Quiet[If[{y} != {} && g[{y}][[1]],
ToExpression[ToString[a][[1]]] <> " = Head1[" <> ToString1[x] <> "]"]; g[x]]
In[2091]:= x = 78; y = {a, b}; z = a + b; Map[UnDef, {t, h, x, y, z, 760}]
Out[2091]= {True, True, False, False, False, False}
In[2092]:= A[x_]; UnDef[x] := Block[{a, a = 460; a}; y := 2013; {A[y], A[78]}
Out[2092]= {A[2013], A[78]}
In[2093]:= L = {a, b, c, d, h, g, p, v, w}; Select[L, UnDef[#] &]
Out[2093]= {a, b, c, d, p, v, w}
```

Тут же вполне уместно еще раз отметить, что на примерах **UnDef1**, **UnDef2**, **UnDef3** – модификаций процедуры **UnDef** – проиллюстрировано принципиальное различие между процедурами типов *Module* и *Block* [97–100]. Поэтому тип процедуры следует выбирать достаточно осмотрительно, отдавая приоритет процедурам *Module*-типа. При этом, в качестве *вложенных* используются, как правило, процедуры *Module*-типа.

В целом ряде случаев возникает необходимость определения контекста произвольного символа. Данную задачу решает достаточно простая процедура, чей вызов **Affiliate[x]**

возвращает *контекст* для произвольного символа  $x$ , заданного в строчном формате, в то время как значение **"Undefined"** возвращается для полностью неопределенного для текущего сеанса символа. Более того, под «*полностью неопределенным*» понимается как конкретное значение, так и впервые используемый в текущем сеансе символ. Простой фрагмент представляет исходный код процедуры и ряд примеров ее использования, включая примеры, поясняющие саму суть понятия «*полностью неопределенный*».

```
In[2080]:= Affiliate[x_ /; StringQ[x]] := Module[{a = Quiet[Context[x]]},
      If[ToString[a] === "Context[" <> x <> "]", "Undefined",
      If[MemberQ[Contexts[], a] && ToString[DefFunc[x]] == "Null" ||
      Attributes[x] === {Temporary}, "Undefined", a]]]
In[2081]:= G = 66; Map[Affiliate, {"ProcQ1", "Sin", "G", "Z", "Affiliate"}]
Out[2081]= {"AladjevProcedures`", "System`", "Global`", "Undefined", "AladjevProcedures`"}
In[2082]:= {V, G = 71, 66}; Map[Affiliate, {"V", "G", "78", "Sin[17]", "Q", "Map"}]
Out[2082]= {"Undefined", "Global`", "Undefined", "Undefined", "Undefined", "System`"}
```

Вызов следующей довольно несложной процедуры **WhatObj[x]** возвращает значение в зависимости от местоположения  $x$ -символа, активизированного в текущем сеансе, а именно: **"Undefined"** – неопределенный символ; **"System"** – системная функция; **"CS"** – символ, чье определение было определено в текущем сеансе; **"Контекст"** – контекст, определяющий пакет, загруженный в текущий сеанс и содержащий определение  $x$ -символа; если же  $x$  имеет тип, отличный от *Symbol*, то вызов процедуры возвращается невычисленным. Следующий фрагмент представляет исходный код **WhatObj** наряду с наиболее типичными примерами ее использования.

```
In[2139]:= WhatObj[x_ /; SymbolQ[x]] := Module[{a = Quiet[Context[x]], t},
      If[a === "System", "System", If[a === "Global",
      If[MemberQ[{$Failed, "Undefined"}, PureDefinition[x]], "Undefined", "CS"], a]]]
In[2140]:= w[x_] := Block[{}, x]; Map[WhatObj, {Sin, a / b, ProcQ, t78, h460, w}]
Out[2140]= {"System", WhatObj[a/b], "AladjevProcedures`", "Undefined", "Undefined", "CS"}
```

Для тестирования символов на предмет отсутствия у них значений, нами определены две простые функции **HowAct** и **SymbolQ**. Первая из них корректно тестирует факт *определенности* переменной в текущем сеансе, как наглядно иллюстрируют примеры следующего фрагмента, однако на *локальных* переменных процедур вызов функции **HowAct** возвращает значение **True** независимо от наличия у них значений. С другой стороны, на *неопределенных* локальных переменных блоках **HowAct** возвращает **False**, как это также весьма наглядно иллюстрирует следующий простой фрагмент:

```
In[2020]:= HowAct[x_] := If[Quiet[Check[ToString[Definition[x]], True]] === "Null",
      False, If[Quiet[ToString[Definition[x]]] ===
      "Attributes[" <> ToString[x] <> "]" = {Temporary}, False, True]]
In[2021]:= SymbolQ[x_] := ! SameQ[Quiet[Check[ToExpression["Attributes[" <>
      ToString[x] <> "]", $Failed]], $Failed]]
```

```

In[2022]:= SymbolQ1[x_] := If[Length[ToCharacterCode[ToString[x]]] == 1, True, False]
In[2023]:= Map[SymbolQ1, {78, "G", "Kr", "6", "#", "!", ":", sv, avz}]
Out[2023]= {False, True, False, True, True, True, True, False, False}
In[2024]:= A[x_] := Module[{a, b=78}, {HowAct[a], HowAct[b], SymbolQ[a], SymbolQ[b]}]
In[2025]:= A[78460]
Out[2025]= {True, True, True, False}
In[2026]:= Map[HowAct, {78, IAN, "RANS", Cos, Args, TestArgsTypes, Label, HowAct}]
Out[2026]= {True, False, True, True, True, True, True, True}
In[2027]:= Map[SymbolQ, {78, IAN, "RANS", Cos, Args, TestArgsTypes, Label, HowAct}]
Out[2027]= {False, True, True, True, True, True, True, True}
In[2028]:= Avz[x_] := Module[{a, b = 78}, {SymbolQ[a], SymbolQ[b]}; Avz[78460]
Out[2028]= {True, False}
In[2029]:= {SymbolQ[Sin], HowAct[Sin], ValueQ[Sin]}
Out[2029]= {True, True, False}
In[2030]:= Agn[x_;/; StringQ[x]] := Module[{a}, FileExistsQ1[x, a]; a]
In[2031]:= Agn["BirthDay.doc"]
Out[2031]= a$2162234
In[2032]:= Av := Block[{a}, HowAct[a]]; Av
Out[2032]= False
In[2033]:= Ag := Block[{a = 460}, HowAct[a]]; Ag
Out[2033]= True
In[2034]:= Agn[x_;/; StringQ[x]] := Module[{a}, ClearAll[a]; FileExistsQ1[x, a]; a]
In[2035]:= Agn["BirthDay.doc"]
Out[2035]= {"C:\\Archive\\Miscellany\\Birthday.doc", "C:\\Temp\\Burthday.doc"}
In[2036]:= {x, y} = {Art, 78}; Map[SymbolQ, {Vsv, "Art", 2013, Sin[x], a + b, Ian, x, y}]
Out[2036]= {True, True, False, False, False, True, True, False}

```

Вызов простой, но недостаточно полезной функции **SymbolQ[x]** возвращает **True**, если *x* является *символом*, и **False** в противном случае. Функция используется в целом ряде средств, представленных в настоящей книге. Данный фрагмент представляет также в ряде случаев полезную функцию **SymbolQ1**, чей вызов **SymbolQ1[x]** возвращает **True**, если *x* является *одиночным символом*, и **False** в противном случае.

Именно по данной причине невозможно использовать внутренними процедурами в качестве фактических аргументов, для которых производится проверка по **! HowAct** на допустимость, локальных переменных главной процедуры. Для устранения этого недостатка предлагается модификация функции **HowAct**, представленная исходным кодом в предыдущем фрагменте. Модификация **HowAct** корректно тестирует факт *определенности* переменных, включая *локальные* переменные процедур. Приведенный пример иллюстрирует корректность выполнения **HowAct** в теле процедур. Поэтому в качестве стандарта следует использовать именно приведенную здесь модификацию. В некоторых представленных здесь средствах использование данной версии **HowAct** обусловило применение в их телах вызовов процедур и функций, чьи определения в

качестве формальных используют типированные **HowAct** аргументы, что позволяет использовать в качестве таких аргументов локальные переменные главных процедур. Читатель в качестве довольно полезного программного решения может использовать процедуру **HowAct** при программировании вложенных процедур; подобный подход использован рядом средств нашего пакета *AVZ\_Package\_1* для *Mathematica 7 ÷ 9* [120].

В целом ряде случаев возникает необходимость удаления из текущего сеанса некоего активного объекта, имеющего присвоенное значение, с возможностью последующего его восстановления в текущем либо другом сеансе. Решает эту задачу **ActRemObj**[*x*, *y*] процедура, чей вызов в зависимости от значения {"Act", "Rem"} второго фактического аргумента удаляет из текущего сеанса пакета объект, заданный его *именем* в строчном формате, или активирует его в текущем или другом сеансе соответственно. Фрагмент представляет исходный код процедуры с различными примерами ее использования. В ряде приложений данная процедура оказывается достаточно полезным средством.

```
In[2047]:= ActRemObj[x_/, StringQ[x], y_/, MemberQ[{"Act", "Rem"}, y]] :=
Module[{a = $HomeDirectory <> "\\\" <> x <> ".$ArtKr$", b,
c = ToString[Definition4[x]]}, If[c === "$Failed", $Failed,
If[HowAct[x] && y == "Rem", b = OpenWrite[a]; WriteString[a, c];
Close[b]; ClearAllAttributes[x]; Remove[x]; "Remove",
If[! HowAct[x] && y == "Rem", "Remove", If[FileExistsQ[a], b = OpenRead[a];
Read[b]; Close[b]; DeleteFile[a]; "Activate", Return[Defer[ActRemObj[x, y]]]]]]]

In[2048]:= Fam := {71, 66, 46, 17, 24}; SetAttributes[Fam, Protected]; Definition[Fam]
Out[2048]= Attributes[Fam] = {Protected}
Fam := {71, 66, 46, 17, 24}

In[2049]:= ActRemObj["Fam", "Rem"]; Definition[Fam]
Out[2049]= Null

In[2050]:= ActRemObj["Fam", "Act"]; Definition[Fam]
Out[2050]= Attributes[Fam] = {Protected}
Fam := {71, 66, 46, 17, 24}

In[2051]:= A[x_] := Module[{a = 460}, x + a]; A[x_, y_] := Module[{a = 78}, x + y + a]
In[2052]:= {A[100], A[100, 200]}
Out[2052]= {560, 378}

In[2053]:= ActRemObj["A", "Rem"]; Definition[A]
Out[2053]= Null

In[2054]:= ActRemObj["A", "Act"]
Out[2054]= "Activate"

In[2055]:= {A[100], A[100, 200]}
Out[2055]= {560, 378}
```

Успешное удаление объекта из текущего сеанса возвращает значение **"Remove"**, тогда как восстановление его в текущем сеансе пакета возвращает значение **"Activate"**. Если же файл, содержащий определение удаленного объекта *x*, не был найден в каталоге **\$HomeDirectory**, то вызов процедуры **ActRemObj** возвращается невычисленным. При этом, на недопустимом аргументе *x* вызов **ActRemObj**[*x*, *y*] возвращает **\$Failed**.

**Средства для работы с последовательными структурами.** Последовательности выражений (просто последовательности) в среде многих языков образуются на основе оператора запятой (,) и служат базой для определения многих типов данных (запросы процедур, списки, множества, индексы и др.). Между тем, в системе *Mathematica* данная структура в качестве самостоятельной отсутствует и в качестве ее выступает списочная структура; некоторые языки придерживаются той же концепции. В данном контексте следующий фрагмент представляет ряд средств, обеспечивающих работу с объектом **Seq[x]**, определяющим последовательность элементов *x*. Так, процедура **SeqToList[x]** обеспечивает конвертирование *Seq*-объекта *x* в список, вызов процедуры **ListToSeq[x]** обеспечивает конвертирование списка *x* в *Seq*-объект, вызов процедуры **SeqIns[x, y, z]** возвращает результат вставки в *Seq*-объект *x* элемента *y* (список, *Seq*-объект, выражение и др.) согласно заданной позиции *z* (*z* ≤ 0 – перед *x*, *z* ≥ Length[x] – после *x*, иначе после *z*-позиции в *x*); наконец, вызов процедуры **SeqDel[x, y]** возвращает результат удаления из *Seq*-объекта *x* элемента *y* (список, *Seq*-объект, выражение и др.). Средства для работы с *Seq*-объектами могут быть достаточно широко расширены, обеспечивая пользователя весьма полезным программным инструментарием. В определенном отношении они позволяют решать задачу совместимости с другими средствами, например, с *Maple*.

```
In[2040]:= A := Seq[a, b, c, d, h, r, x, y, z]
In[2041]:= SeqQ[x_] := Block[{a = ToString[x]}, If[StringLength[a] >= 4 &&
StringTake[a, {1, 4}] == "Seq[" && StringTake[a, {-1, -1}] == "]", True, False]]
In[2042]:= Map[SeqQ, {A, {a, b, c}, Agn}]
Out[2042]= {True, False, False}
In[2043]:= SeqToList[x_ /; SeqQ[x]] := ToExpression["{" <> StringTake[ToString[x],
{5, -2}] <> "}"]
In[2044]:= SeqToList[A]
Out[2044]= {a, b, c, d, h, r, x, y, z}
In[2045]:= ListToSeq[x_ /; ListQ[x]] := ToExpression["Seq[" <> StringTake[ToString[x],
{2, -2}] <> ""]
In[2046]:= ListToSeq[{42, 47, 71, 17, 24}]
Out[2046]= Seq[42, 47, 71, 17, 24]
In[2047]:= SeqIns[x_ /; SeqQ[x], y_, z_ /; IntegerQ[z]] := Module[{a = SeqToList[x], b = {},
c = If[SeqQ[y], SeqToList[y], y]}, If[z <= 0, b = Append[c, a],
If[z >= Length[a], b = Append[a, c], b = Join[a[[1 ;; z]], c,
a[[z + 1 ;; -1]]]]; ListToSeq[Flatten[b]]]
In[2048]:= SeqIns[A, {78, 460}, 24]
Out[2048]= Seq[a, b, c, d, h, r, x, y, z, 78, 460]
In[2049]:= SeqIns[A, Seq[42, 47, 71, 17, 24], 6]
Out[2049]= Seq[a, b, c, d, h, r, 42, 47, 71, 17, 24, x, y, z]
In[2050]:= SeqToList1[x_] := {x}
```



```
In[2051]:= x = 78; {SeqToList[a, b, c, d, r, h, p], SeqToList[]}
Out[2051]= {{a, b, c, d, r, h, p}, {}}

In[2052]:= SeqToString[h_] := StringTake[ToString[h], {2, -2}]

In[2053]:= {SeqToString[a, c + d, Sin[x], a^2 + (x + y)/78], SeqToString[], "VGS"}
Out[2053]= {"a, c + d, Sin[x], a^2 + (x + y)/78", "", "VGS"}

In[2054]:= SeqDel[x_;/; SeqQ[x], y_] := Module[{a = SeqToList[x], b = If[SeqQ[y],
    SeqToList[y], y]}, ListToSeq[Select[a, !MemberQ[Flatten[{b}], #] &]]]

In[2055]:= SeqDel[A, Seq[a, b, c, d]]
Out[2055]= Seq[h, r, x, y, z]
```

Дополнительно к функции **SeqToList** можно отметить простую функцию, чей вызов **SeqToList[a, b, c, ...]** возвращает список ее фактических аргументов, а также простую функцию, чей вызов **SeqToString[a, b, c, ...]** возвращает список аргументов в строчном формате, как иллюстрирует последний пример предыдущего фрагмента. Как ранее отмечалось, механизм, поддерживающий работу с объектами типа *последовательность* (*sequence*), в *Mathematica* отсутствует, что создает определенные затруднения. Между тем, система *Mathematica* предоставляет функцию **Sequence[a, b, ...]**, определяющую последовательность аргументов, автоматически передаваемых произвольным блоку, функции или модулю. И в данном контексте процедура **SequenceQ[s]** обеспечивает тестирование объектов, которые созданы на базе функции **Sequence**, возвращая **True**, если *s*-объект определен этой функцией, и **False** в противном случае; более того, имя *s*-объекта кодируется в строчном формате. Фрагмент представляет как исходный код процедуры **SequenceQ**, так и наиболее типичные примеры ее использования.

```
In[2399]:= S = Sequence[a, b]; G = Sequence[x, y]; Art := Sequence[c, d]; Kr = Sequence[z];
In[2300]:= SequenceQ[s_String] := Module[{a, b = " := ", d=Quiet[ToString[Definition[s]]],
    c = " = ", }, Quiet[Check[If[StringTake[StringReplace[d,
    {s <> b -> "", s <> c -> ""}], {1, 9}] == "Sequence["], True, False], False]]]

In[2301]:= Map[SequenceQ, {"S", "G", "Art", "Kr", "GrGu"}]
Out[2301]= {True, True, True, True, False}

In[2302]:= SeqUnion[x_] := Sequence[x]

In[2303]:= SeqUnion[a, b, c, Art, Kr, S, G]
Out[2303]= Sequence[a, b, c, c, d, z, a, b, x, y]
```

На базе стандартной функции **Sequence** можно создавать довольно простые средства, обеспечивающие работу с *последовательностными* структурами подобно пакету *Maple*; последний пример фрагмента представляет простую функцию **SeqUnion[x, y, ...]**, чей вызов возвращает результат слияния произвольного числа последовательностей. Эти функции наряду с рядом рассматриваемых достаточно полезны в работе с объектами типа «*последовательность*», чья структура системой не поддерживается и для работы с которой система *Mathematica* не располагает стандартными средствами.

Вызов стандартной функции **Sequence** $[x_1, x_2, x_3, \dots, x_n]$  определяет последовательность фактических аргументов  $x_j$  ( $j=1..n$ ), передаваемых некоторой функции. Между тем, с объектами типа «последовательность» *Mathematica* может работать опосредствованно, в частности, на основе *списочных* структур. В этой связи для расширения стандартной функции **Sequence** на списочные структуры определена процедура **Sequences** $[x]$ , чей вызов обеспечивает вставку в функцию аргументов  $x$ , заданных последовательностью либо списком; упрощенным вариантом процедуры служит функция **Sq**. Следующий фрагмент представляет исходные коды процедуры и функции с их применением.

```
In[2494]:= Sequences[x_] := Module[{a = Flatten[{x}], b, c},
    b = "Sequence[" <> ToString[a] <> "]; a = Flatten[StringPosition[b, {"(", "}"}]];
    ToExpression[StringReplace[b, {StringTake[b, {a[[1]], a[[1]]}] -> "",
    StringTake[b, {a[[-1]], a[[-1]]}] -> ""}]]]
In[2495]:= {F[Sequence[{x, y, z}], F[Sequences[{x, y, z}], F[Sequences[x, y, z]]]
Out[2495]= {F[{x, y, z}], F[x, y, z], F[x, y, z]}
In[9]:= Sq[x_List] := ToExpression["Sequence[" <> StringTake[ToString1[x], {2, -2}] <> "]"
In[10]:= Plus[Sq[{71, 66, 55, 46, 24, 17}]]
Out[10]= 279
```

При работе с последовательностными структурами достаточно полезной оказывается следующая процедура, обеспечивающая специального формата конвертацию строк в списки, и наоборот. Вызов процедуры **ListStrList** $[x]$  на списке  $x = \{a, b, \dots\}$  возвращает строку  $s$  формата "*ahbh ...*", тогда как  $x = \text{ListStrList}[s]$ , где  $h = \text{FromCharacterCode}[2]$ . В случае отсутствия в  $s$ -строке  $h$ -символа вызов **ListStrList** $[s]$  возвращает строку  $s$ .

```
In[2604]:= ListStrList[x_ /; StringQ[x] || ListQ[x]] := Module[{a = FromCharacterCode[2]},
    If[StringQ[x] && ! StringFreeQ[x, a], Map[ToExpression, StringSplit[x, a]],
    If[ListQ[x], StringTake[StringJoin[Map[StringJoin, Map[ToString1, x], a]], {1, -2}], x]]
In[2605]:= L = ListStrList[{Avz, 71, Agn, 66, Art, 24, Kr, 17, Vsv, 46}]
Out[2605]= "Avz□71□Agn□66□Art□24□Kr□17□Vsv□46"
In[2606]:= ListStrList[ListStrList[{Avz, 71, Agn, 66, Art, 24, Kr, 17, Vsv, 46}]]
Out[2606]= {Avz, 71, Agn, 66, Art, 24, Kr, 17, Vsv, 46}
In[2607]:= ListStrList["Avz71Agn66Art24Kr17Vsv46"]
Out[2607]= "Avz71Agn66Art24Kr17Vsv46"
In[2608]:= ListStrList[ListStrList[{"Avz", 71, "Agn", 66, "Art", 24, "Kr", 17, "Vsv", 46}]]
Out[2608]= {"Avz", 71, "Agn", 66, "Art", 24, "Kr", 17, "Vsv", 46}
```

Предыдущий фрагмент представляет исходный код процедуры **ListStrList** наряду с типичными примерами ее применения. В целом ряде случаев процедура **ListStrList** обеспечивает, например, решение задач весьма эффективного погружения процедур, разработанных в программной среде *Maple* с использованием последовательностных структур, достаточно широко используемых в этой среде в качестве одной из базовых структур данных, в программную среду пакета *Mathematica*. Ряд наших приложений довольно существенно использует данные процедуры, упрощая программирование.

## Глава 4. Дополнительные средства обработки выражений в программной среде системы *Mathematica*

Аналогично целому ряду программных систем и систем компьютерной математики *Mathematica* под «выражением» понимает все, с чем манипулирует (*графики, списки, формулы, строки, модули, функции, числа различного типа и др.*). Не взирая на то, что все эти выражения на первый взгляд достаточно существенно различаются, *Mathematica* представляет их в стандартном формате. Приведем простые примеры представления выражений в *полном формате (форме)*, а именно:

Выражение	Полный формат
$x + y + z$	<code>Plus[x, y, z]</code>
$x^n$	<code>Power[x, n]</code>
$\neg x$	<code>Not[x]</code>
$\{a, b, c\}$	<code>List[List[a, b, c]]</code>
$a \rightarrow b$	<code>Rule[a, b]</code>
$a = b$	<code>Set[a, b]</code>
$a \&\& b$	<code>And[a, b]</code>
$a := b$	<code>Null</code>

И только отложенное присвоение ( $:=$ ) не имеет полной формы. С целью определения *заголовка (его определяющего типа)* выражения используется функция **Head**, чей вызов **Head[expr]** возвращает заголовок выражения *expr*, например:

```
In[2129]:= Map[Head, {ProcQ, Sin, 78, a + b, Function[{x, y}, x + y], G[x], S[6], x*y, x^y}]
Out[2129]= {Symbol, Symbol, Integer, Plus, Function, G, S, Times, Power}
```

Для более точного определения *заголовков* нами была создана полезная модификация стандартной функции **Head** в виде процедуры **Head1**, расширяющая ее возможности, например, это относится к тестированию операций *отложенных* вычислений, когда на них возвращаются значения **SetDelayed** | | **TagSetDelayed**, блоков, функций, модулей.

```
In[2160]:= Head1[x_] := Module[{a, b, c = Quiet[Check[Attributes[x], {}]]},
  If[Quiet[SystemQ[x]], OptRes[Head1, System], If[c != {}, ClearAllAttributes[x]];
  b = Quiet[StringSplit[ToString[Definition[x]], "\n\n"]; a = If[! SameQ[b, {}] &&
  b[[-1]] == "Null", SetDelayed | | TagSetDelayed, If[PureFuncQ[x], PureFunction,
  If[Quiet[Check[FunctionQ[x], False]], Function, If[BlockQ[x], Block, If[BlockModQ[x],
  Module, Head[x]]]]]; {OptRes[Head1, a], Quiet[SetAttributes[x, c]]}][[1]]]
In[2161]:= G := S; Z[x_] := Block[{}, x]; F[x_] := x; Map[Head, {ProcQ, Sin, 6, a + b, # &, G,
  Z, Function[{x}, x], x*y, x^y, F}]
Out[2161]= {Symbol, Symbol, Integer, Plus, Function, Symbol, Symbol, Function, Times,
  Power, Symbol}
In[2162]:= G := S; Z[x_] := Block[{}, x]; F[x_] := x; Map[Head1, {ProcQ, Sin, 6, a + b, # &, G,
  Z, Function[{x}, x], x*y, x^y, F}]
Out[2162]= {Module, System, Integer, Plus, PureFunction, SetDelayed | | TagSetDelayed,
  Block, PureFunction, Times, Power, Function}
```

Предыдущий фрагмент представляет *исходный* код процедуры **Head1** с примерами ее типичного применения в сравнительном контексте относительно системной функции **Head**. Вызов **Head1[x]** возвращает заголовок выражения *x* в контексте {**Block, Function, Module, PureFunction, SetDelayed | TagSetDelayed, Symbol, System, Head[x]**}, как это иллюстрируют примеры предыдущего фрагмента, на которых очень наглядно видны функциональные различия обоих средств. В виду сказанного, процедура **Head1** имеет вполне определенное значение для более точной (*относительно системного стандарта*) классификации выражений по типу их заголовков. Итак, *Mathematica* рассматривает в качестве *выражения* все, что в нее вводится. Заголовок выражения можно получать по стандартной функции **Head**, но более дифференцированно по нашей процедуре **Head1**. На многих выражениях вызовы процедуры **Head1** и функции **Head** являются идентичными, тогда как на целом ряде их вызовы существенно различаются. Таким образом, понятие *выражения* является важным объединяющим принципом в системе, а тот факт, что каждый объект в системе имеет одинаковую внутреннюю структуру, позволяет охватить много областей сравнительно небольшим количеством базовых операций. Между тем, несмотря на то, что все выражения имеют одинаковую *базовую* структуру, *Mathematica* предоставляет множество различных функций для работы как с выражением в целом, так и с отдельными его компонентами.

*Средства тестирования корректности выражений.* *Mathematica* располагает рядом средств, обеспечивающих тестирование корректности синтаксиса выражений, среди которых пользователю доступны лишь две функции, а именно:

**SyntaxQ["x"]** – возвращает *True*, если *x* – синтаксически корректное выражение, и *False* в противном случае;

**SyntaxLength["x"]** – возвращает число *p* символов, начиная с самого начала строки "x", которое определяет синтаксически корректное выражение **StringTake["x", {1, p}]**; при этом, в случае *p > SyntaxLength["x"]* система декларирует, что вся строка "x" корректна, но формирование корректного выражения требует ее продолжения. На наш взгляд, это не совсем удобно при программной обработке выражений. Поэтому предложены расширения в виде функции **SyntaxQ1** и процедуры **SyntaxLength1**, исходные коды которых наряду с наиболее типичными примерами их применения представляются в нижеследующем фрагменте, а именно.

```
In[2029]:= SyntaxQ1[x_;/; StringQ[x]] := If[Quiet[ToExpression[x]] ===$Failed, False, True]

In[2030]:= Map[SyntaxQ1, {"(a+b/", "d[a[1]] + b[2]"}]
Out[2030]= {False, True}

In[2031]:= SyntaxLength1[x_;/; StringQ[x], y___] := Module[{a = "", b = 1, d, h = {},
c = StringLength[x]},
While[b <= c, d = Quiet[ToExpression[a = a <> StringTake[x, {b, b}]]];
If[! SameQ[d, $Failed], h = Append[h, StringTrim[a]]; b++];
h = DeleteDuplicates[h];
If[{y} != {} && ! HowAct[{y}][[1]], {y} = {h}]; If[h == {}, 0, StringLength[h][[-1]]]]
```

```

In[2032]:= SyntaxLength["(a+b/"]
Out[2032]= 7
In[2033]:= SyntaxLength1["(a+b/"]
Out[2033]= 0
In[2034]:= {SyntaxLength1["(a+b", p], p}
Out[2034]= {0, {}}
In[2035]:= SyntaxLength["d[a[1]] + b[2]"]
Out[2035]= 14
In[2036]:= SyntaxLength1["d[a[1]] + b[2]"]
Out[2036]= 14
In[2037]:= {SyntaxLength1["d[a[1]] + b[2]", g], g}
Out[2037]= {14, {"d", "d[a[1]]", "d[a[1]] + b", "d[a[1]] + b[2]"}}
```

Вызов **SyntaxLength1**[*x*] возвращает максимальный номер *p* позиции в строке *x* такой, что **ToExpression[StringTake**[*x*, {1, *p*}]] – синтаксически корректное выражение, иначе возвращается 0; вызов **SyntaxLength1**[*x*, *y*] через второй необязательный аргумент *y* – неопределенную переменную – дополнительно возвращает список подстрок строки *x*, представляющих корректные выражения. Выше приведены сравнительные примеры применения функции **SyntaxLength** и процедуры **SyntaxLength1**, которые достаточно наглядно иллюстрируют сказанное. Там же приведен простой пример программной реализации стандартной функции **SyntaxQ** в виде функции **SyntaxQ1**.

**Средства обработки выражений на уровне их компонентов.** Средства такой группы обеспечивают довольно эффективную дифференцированную обработку выражений. Унифицированная символьная архитектура пакета *Mathematica* дает возможность по непосредственному обобщению элементарно-ориентированных списочных операций на произвольные выражения, поддерживая операции как на отдельных термах, так и на наборах термов на заданных уровнях в деревьях выражений. Не вдаваясь детально во все средства, поддерживающие работу с составными частями выражений, приведем лишь список основных из них с кратким назначением. Тогда как с более детальным их описанием, включая допустимые форматы кодирования, можно познакомиться либо в справке по пакету, либо в соответствующей литературе по пакету, например, в [100, 121-140]. А поскольку списки являются подклассом класса всех выражений, не должно быть совсем уж неестественным то обстоятельство, что к выражениям применимы, на первый взгляд, сугубо списочные операции, часть из которых продублирована здесь. Представим некоторые из наиболее часто используемых средств данной группы:

**Part**[*x*, *k*] – получение *k*-й части произвольного выражения *x*;

**Variables**[*p*] – возвращает список *всех* независимых переменных полинома *p*; тогда как ее применение к произвольному выражению *p* имеет определенные ограничения;

Тогда как для получения всех независимых переменных произвольного выражения *x* вполне можно использовать довольно несложную функцию, чей вызов **UnDefVars**[*x*] возвращает список всех независимых переменных произвольного выражения *x*. Ниже представлены исходный код функции с примерами ее применения в сравнительном

контексте относительно стандартной функции **Variables**. В ряде случаев эта функция имеет определенные предпочтения относительно стандартной функции **Variables**.

```
In[2023]:= UnDefVars[x_] := Select[OP[x], Quiet[ToString[Definition[#]]] == "Null" &]
In[2023]:= UnDefVars[(x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z - G[h, t]]]
Out[2023]= {a, G, h, t, x, y, z}
In[2023]:= Variables[(x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z - G[h, t]]]
Out[2023]= {a, x, y, Cos[y], Log[x + y + z - G[h, t]], Sin[x]}
In[2023]:= H := 78; Map[UnDefVars, {a + 78*b, a*Sin[x]*Cos[y], {a, b}, a*F[h, g, s] + H}]
Out[2023]= {{a, b}, {a, x, y}, {a, b}, {a, F, g, h, s}}
In[2023]:= {Variables[(x + y)^3 + 6 z^2 - y z + 78], UnDefVars[(x + y)^3 + 6 z^2 - y z + 78]}
Out[2023]= {{x, y, z}, {x, y, z}}

In[2041]:= UnDefVars1[x_] := Select[ExtrVarsOfStr[ToString[x], 2], !SystemQ[#] &]
In[2042]:= H := 78; Map[UnDefVars1, {a + 78*b, a*Sin[x]*Cos[y], {a, b}, a*F[h, g, s] + H}]
Out[2042]= {"a", "b"}, {"a", "x", "y"}, {"a", "b"}, {"a", "F", "g", "h", "s"}}
```

В отличие от функции **UnDefVars**, вызов функции **UnDefVars1[x]**, представленной в конце предыдущего фрагмента, возвращает список всех независимых переменных в *строочном* формате произвольного выражения *x*. Предыдущий фрагмент представляет исходный код функции **UnDefVars1** с примерами ее применения.

**Length[x]** – возвращает количество частей выражения *x*; эквивалентна весьма простой функции **MaxParts[x\_] := Length[OP[x]]**;

**First[x]** – возвращает первую часть выражения *x*; эквивалентна **Part[x, 1]**;

**Last[x]** – возвращает последнюю часть выражения *x*; эквивалентна **Part[x, -1]**;

**Most[x]** – возвращает выражение *x* без последней части; эквивалентна **x[[1 ;; -2]]**;

**Rest[x]** – возвращает выражение *x* без первой части; эквивалентна **x[[2 ;; -1]]**;

**Take[x, n]** – возвращает части выражения *x*, определенные аргументами, начиная со второго *n*; имеет четыре формата кодирования и эквиваленты **x[[m ;; n]]**;

**Drop[x, n]** – возвращает выражение *x* без частей, определенных аргументами, начиная со второго *n*; имеет шесть форматов кодирования и эквиваленты **x[[m ;; n]]**;

**Extract[x, n]** – извлекает из выражения *x* часть, определенную аргументами, начиная со второго *n*; имеет три формата кодирования и эквиваленты с **Part**;

**Replace[x, r {, l}]** – возвращает результат применения правила *r* типа *a* → *b* или списка таких правил с целью преобразования выражения *x* как единого целого; применение 3-го необязательного аргумента *l* определяет применение правил *r* к частям *l*-уровня выражения *x*. Между тем, функция **Replace** имеет ряд ограничений, ряд из которых в значительной мере снимает процедура, чей вызов **Replace1[x, r]** возвращает результат применения правил *r* к независимым (*всем либо выборочно*) переменным выражения *x*. В случае обнаружения процедурой пустых правил печатается сообщение с указанием списка тех правил *r*, которые оказались пустыми, т.е. левые части которых не входят в список независимых переменных выражения *x*. Фрагмент представляет исходный код процедуры **Replace1** наряду с примерами ее типичного применения. При этом, дано сравнение с результатом применения функции **Replace** на том же самом выражении.

```

In[2052]:= Replace1[x_, y_ /; ListQ[y] && DeleteDuplicates[Map[Head, y]] == {Rule} | |
          Head[y] == Rule] := Module[{a = x/FullForm//ToString,
          b = UnDefVars[x], c, p, l, h = {}, r, k = 1,
          d = ToStringRule[DeleteDuplicates[Flatten[{y}]]]},
p = Mapp[RhsLhs, d, "Lhs"]; c = Select[p, ! MemberQ[Map[ToString, b], #] &];
If[c != {}, Print["Rules " <> ToString[Flatten[Select[d, MemberQ[c, RhsLhs[#,
          "Lhs"]] &]]] <> " are vacuous"]];
While[k <= Length[d], l = RhsLhs[d[[k]], "Lhs"]; r = RhsLhs[d[[k]], "Rhs"];
h = Append[h, {"[" <> l -> "[" <> r, " " <> l -> " " <> r, l <> "]" -> r <> "]" }]; k++];
Simplify[ToExpression[StringReplace[a, Flatten[h]]]]]

In[2053]:= X = (x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y]; Replace[X, {x -> a + b, a -> 78,
          y -> Cos[a], z -> Log[t]}]

Out[2053]= a Log[x + y] + (x^2 - y^2)/(Cos[y] + Sin[x])
In[2054]:= Replace1[X, {x -> a + b, a -> 78, y -> Cos[a], z -> Log[t], t -> c + d}]
          Rules {z -> (Log[t]), t -> (c + d)} are vacuous
Out[2054]= 78 Log[a + b + Cos[a]] + ((a + b)^2 - Cos[a]^2)/(Cos[Cos[a]] + Sin[a + b])
In[2055]:= Replace1[X, {x -> a + b, a -> 78, y -> Cos[a]}]
Out[2055]= 78 Log[a + b + Cos[a]] + ((a + b)^2 - Cos[a]^2)/(Cos[Cos[a]] + Sin[a + b])

```

В ряде случаев при преобразованиях выражений посредством подстановок возникает необходимость конвертации в строчный формат левых и правых частей правил типа  $"a \rightarrow b"$ . Данную задачу решает довольно простая процедура **ToStringRule**, чей вызов **ToStringRule[x]** возвращает правило либо список правил  $x$ , чьи левые и правые части имеют строчный формат. Более того, правая часть берется в круглые скобки. Именно эта процедура используется вышепредставленной процедурой **Replace1**. Следующий фрагмент представляет исходный код процедуры с примерами ее использования.

```

In[2023]:= ToStringRule[x_ /; ListQ[x] && DeleteDuplicates[Map[Head, x]] == {Rule} | |
          Head[x] == Rule] := Module[{a = Flatten[{x}], b = {}, c, k = 1},
While[k <= Length[a], c = a[[k]]; b = Append[b, ToString[RhsLhs[c, "Lhs"]] ->
          "(" <> ToString[RhsLhs[c, "Rhs"]] <> ")"]; k++]; If[ListQ[x], b, b[[1]]]]

In[2024]:= {ToStringRule[a -> b], ToStringRule[{a -> b, c -> d, m -> n}]}
Out[2024]= {"a" -> "(b)", {"a" -> "(b)", "c" -> "(d)", "m" -> "(n)"}}

In[2035]:= ToStringRule1[x_ /; ListQ[x] && DeleteDuplicates[Map[Head, x]] == {Rule} | |
          Head[x] == Rule] := Module[{a = Flatten[{x}], b},
b = Map[ToString[OP[#]][[1]] -> ToString[OP[#]][[2]] &, a]; If[Length[b] == 1, b[[1]], b]]

In[2036]:= {ToStringRule1[a -> b], ToStringRule1[{a -> b, c -> d, m -> n}]}
Out[2036]= {"a" -> "b", {"a" -> "b", "c" -> "d", "m" -> "n"}}

```

Завершает фрагмент процедура, чей вызов **ToStringRule1[x]** возвращает правило или список правил  $x$ , левые и правые части которых имеют строчный формат.

**ReplacePart**[ $x, n \rightarrow a$ ] – простейший формат вызова возвращает результат замены  $n$ -й части выражения  $x$  на  $a$ ; функция располагает еще четырьмя форматами.

В целом ряде случаев обработки выражений, в частности, вычисления производных  $x$  по подвыражениям выражения  $x$  либо интегрирования выражения  $x$ , когда в качестве переменной интегрирования  $x$  вместо простой переменной выступает подвыражение и т.д., весьма полезной оказывается процедура **Replace2**, вызов которой **Replace2**[ $x, y, z$ ] возвращает результат замены в выражении  $x$  все входящие в него подвыражения  $y$  на выражение  $z$ . При невозможности выполнить замену вызов **Replace2**[ $x, y, z$ ] возвращает значение **\$Failed**, что довольно удобно при необходимости обеспечения непрерывных вычислений. Следующий фрагмент представляет исходный код процедуры наряду с примерами ее применения. Там же представлены примеры процедур **D1** и **Integrate1**, которые базируются на процедуре **Replace2** и обеспечивают соответственно операции дифференцирования и интегрирования выражений по сложным переменным.

```
In[2028]:= Replace2[x_, y_, z_] := Block[{a = Map7[ToString, FullForm, {x, y, z}], b},
                                         SetAttributes[ToString1, Listable];
                                         b = ToExpression[StringReplace[a[[1]], a[[2]] -> a[[3]]];
                                         If[DeleteDuplicates[StringFreeQ[ToString1[Flatten[{b}], "\""]]] == {True},
                                         ClearAttributes[ToString1, Listable]; b, ClearAttributes[ToString1, Listable]; $Failed]]

In[2029]:= Replace[1 + 1/a, 1/a -> Sin[x]]
Out[2029]= 1 + 1/a

In[2030]:= Replace2[Sin[x + Cos[y]] + x^2*Log[x + y], x, h^2]
Out[2030]= h^4 Log[h^2 + y] + Sin[h^2 + Cos[y]]

In[2031]:= Replace2[Sin[x + Cos[y]] + 1/a^2 + x^2*Log[x + y], 1/a, h]
Out[2031]= $Failed

In[2032]:= D1[x_, y_] := Module[{a, b, c}, c = Replace2[D[b = Replace2[x, y, a];
                                         If[b === $Failed, Return[$Failed], b], a], a, y]; If[b === $Failed, $Failed, c]]

In[2033]:= {D1[Sin[x + Cos[y]] + x^2*Log[x + y], Cos[y]], D1[Sin[x], x]}
Out[2033]= {Cos[x + Cos[y]], Cos[x]}

In[2034]:= {D1[Sin[x + Cos[y]] + 1/a^2 + x^2*Log[x + y], 1/a], D1[Sin[x], x]}
Out[2034]= {$Failed, Cos[x]}

In[2035]:= Integrate1[x_, y_] := Module[{a, b, c}, c = Replace2[Integrate[b = Replace2[x, y, a];
                                         If[b === $Failed, Return[$Failed], b], a], a, y]; If[b === $Failed, $Failed, c]]

In[2036]:= {Integrate1[Sin[x + Cos[y]] + x^2*Log[x + y], Cos[y]], Integrate1[Tan[x], x]}
Out[2036]= {-Cos[x] Cos[Cos[y]] + x^2 Cos[y] Log[x + y] + Sin[x] Sin[Cos[y]], -Log[Cos[x]]}

In[2037]:= Replace2[1/a^2 + 1/(1/a + Cos[x]), 1/a, Sin[x]]
Out[2037]= $Failed

In[2038]:= Subs[1/a^2 + 1/(1/a + Cos[x]), 1/a, Sin[x]]
Out[2038]= Sin[x]^2 + 1/(Cos[x] + Sin[x])
```

Между тем, не взирая на простоту и изящность реализаций процедур **Replace2**, **D1** и



**Integrate1**, говорить об их универсальности не приходится именно из-за ограничений процедуры **Replace2**, по целому ряду соображений здесь не рассматриваемых. Однако заинтересованному читателю в качестве весьма полезного упражнения настоятельно рекомендуется рассмотреть этот вопрос детальнее. Между тем, 2 последних примера фрагмента иллюстрируют как ограниченность **Replace2**, так и большие возможности процедуры **Subs**, рассматриваемой нами несколько ниже. В любом случае процедуры **Replace2**, **D1** и **Integrate1** в ряде случаев достаточно полезны, включая примененные при их программировании некоторые весьма полезные приемы. Например, довольно полезным оказывается прием временного приписывания *Listable*-атрибута процедуре **ToString1**, расширяя ее применимость на списки в качестве аргументов.

**Append[x, a]** – возвращает результат дополнения выражения  $x$  подвыражением  $a$ ; при этом, дополнение производится на основе конструкции **Head[x][x, a]**;

**Prepend[x, a]** – вызов возвращает результат расширения выражения  $x$  подвыражением  $a$  слева; при этом, дополнение производится на основе конструкции **Head[x][a, x]**; так как пакет использует упомянутый выше принцип *полного* представления выражений, то в целом ряде случаев имеет место соотношение **Append[x, a] ≡ Prepend[x, a]**, как это весьма наглядно иллюстрируют следующие весьма простые примеры, а именно:

In[2164]:= **X = (a + b)/(c + d); {Append[X, Sin[x]], Prepend[X, Sin[x]]}**

Out[2164]= **{{(a + b) Sin[x]/(c + d), ((a + b) Sin[x]/(c + d))}**

In[2165]:= **Y = {a + b + c + d}; {Append[Y, 78], Prepend[Y, 78]}**

Out[2165]= **{{a + b + c + d, 78}, {78, a + b + c + d}}**

**Insert[x, a, n]** – простейший формат вызова возвращает результат вставки выражения  $a$  по месту  $n$ -й части выражения  $x$ , раздвигая его; функция имеет еще два формата;

**Delete[x, n]** – простейший формат вызова возвращает результат удаления выражения  $n$ -й части выражения  $x$ , упрощая его; функция имеет еще два формата кодирования; более того, в обоих функциях **Insert** и **Delete** вычисление  $n$ -й позиции производится, исходя из полной формы выражений, ранее упомянутой и принятой в пакете;

**DeleteCases[x, p]** – простейший формат вызова возвращает результат удаления всех частей выражения  $x$ , соответствующих шаблону  $p$ ; функция имеет еще два формата;

**Position[x, p]** – возвращает список позиций, на которых в выражении  $x$  появляются все подвыражения, соответствующие шаблону  $p$ ; функция имеет 3 формата кодирования; при этом, возвращаемый функцией список позиций имеет форму, вполне пригодную для использования функциями **Extract**, **ReplacePart** и **MapAt**, но не **Part**, например:

In[2188]:= **X = (a^b + Sin[x])/(c^d - Cos[x]); Position[X, \_^\_]**

Out[2188]= **{{1, 1, 1}, {1}, {2, 1}}**

In[2189]:= **ReplacePart[X, {%%[[1]] -> agn, %[[3]] -> agn}]**

Out[2189]= **(agn + Sin[x])/(agn - Cos[x])**

**MapAt[F, x, n]** – простейший из форматов вызова возвращает результат применения  $F$  к части выражения  $x$ , находящейся на позиции  $n$ ; функция имеет еще два формата; и в данном случае  $n$ -я позиция определяется, исходя из полной формы выражения  $x$ ;

**Map[F, x]** – простейший формат вызова возвращает результат применения функции  $F$  ко *всем* частям выражения  $x$ , находящимся на *первом* уровне; функция **Map** имеет еще

один формат кодирования, достаточно полезный в целом ряде приложений;

**MapAll**[ $F, x$ ] – вызов функции возвращает результат применения функции  $F$  ко всем подвыражениям выражения  $x$ ; достаточно полезна в целом ряде приложений;

**Level**[ $x, n$ ] – простейший формат вызова возвращает список всех подвыражений для  $x$  выражения на уровнях от 1 до  $n$ ; функция имеет еще один формат кодирования;

В качестве полезного обобщения функции **Level** можно рассматривать и процедуру, чей вызов **Levels**[ $x, h$ ] возвращает список всех подвыражений для  $x$  выражения на всех его *возможных* уровнях, тогда как через *второй* аргумент  $h$  – *независимую* переменную – возвращается максимальное число уровней выражения  $x$ . Вообще говоря, имеет место следующее определяющее соотношение **Levels**[ $x, h$ ]  $\equiv$  **Level**[ $x, \text{Infinity}$ ], однако в случае процедуры **Levels** дополнительно возвращается максимальный уровень выражения  $x$ . Фрагмент представляет исходный код процедуры **Levels** с примером ее применения в сравнительном контексте с функцией **Level** со вторым **Infinity**-аргументом.

```
In[2068]:= Levels[x_, h_]; ToString[Definition[h]] == "Null" := Module[{a = {}, b, k = 1},
    While[k < Infinity, b = Level[x, k]; If[a == b, Break[], a = b]; k++]; h = k - 1; a]
In[2069]:= {Levels[(x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z - G[h, t]], g], g}
Out[2069]= {{a, x, y, z, -1, h, 7, G[h, 7], -G[h, 7], x + y + z - G[h, 7], Log[x + y + z - G[h, 7]],
    a Log[x + y + z - G[h, 7]], x, 2, x^2, -1, y, 2, y^2, -y^2, x^2 - y^2, y, Cos[y], x,
    Sin[x], Cos[y] + Sin[x], -1, 1/(Cos[y] + Sin[x]), (x^2 - y^2)/(Cos[y] + Sin[x])}, 6}
In[2070]:= Level[(x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z - G[h, t]], Infinity]
Out[2070]= {{a, x, y, z, -1, h, 7, G[h, 7], -G[h, 7], x + y + z - G[h, 7], Log[x + y + z - G[h, 7]],
    a Log[x + y + z - G[h, 7]], x, 2, x^2, -1, y, 2, y^2, -y^2, x^2 - y^2, y, Cos[y], x,
    Sin[x], Cos[y] + Sin[x], -1, 1/(Cos[y] + Sin[x]), (x^2 - y^2)/(Cos[y] + Sin[x])}, 6}
In[2079]:= ExprOnLevels[x_] := Module[{a = {}, k = 1}, While[k <= Depth[x],
    a = Append[a, MinusList[Level[x, k], Level[x, k - 1]]]; k++]; a[[1 ;; -2]]]
In[2080]:= X = (x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z - G[h, t]]; ExprOnLevels[X]
Out[2080]= {{a Log[x + y + z - G[h, t]], (x^2 - y^2)/(Cos[y] + Sin[x])}, {a, Log[x + y + z -
    G[h, t]], x^2 - y^2, 1/(Cos[y] + Sin[x])}, {x + y + z - G[h, t], x^2, -y^2, Cos[y] +
    Sin[x], -1}, {x, y, z, x, 2, -1, y^2, Cos[y], Sin[x], -1}, {G[h, t], y, 2, y, x, -1}, {h, t}}
```

Тогда как вызов довольно простой процедуры **ExprOnLevels**[ $x$ ] возвращает *вложенный* список, элементами которого являются списки подвыражений выражения  $x$ , которые расположены на каждом из его уровней с 1-го до последнего. Предыдущий фрагмент завершается исходным кодом процедуры с типичным примером ее применения.

**Depth**[ $x$ ] – возвращает общее число уровней в выражении  $x$ . Относительно массивов, векторов и матриц функция анализирует их подобно соответствующим им спискам; по сравнению с вышепредставленной процедурой **Levels** функция **Depth** определяет на том же самом выражении  $x$  максимальное число его уровней на единицу больше:

```
In[2280]:= Clear[t]; {Levels[a + b + c^2, t], t, Depth[a + b + c^2]}
Out[2280]= {{a, b, c, 2, c^2}, 2, 3}
```

**Sow** и **Reap** – обеспечивают удобный способ для накопления списка промежуточных результатов при вычислении выражений. Функции **Sow** и **Reap** имеют 3 и 4 формата кодирования соответственно, охватывая весьма существенный диапазон приложений при программировании как прикладных, так и системных задач.

Для удобства обработки структуры выражения достаточно удобной представляется и функция, вызов **FullForm[x]** которой возвращает полную форму выражения  $x$  без его специального синтаксиса, как весьма наглядно иллюстрирует простой пример:

```
In[2032]:= ToString[FullForm[x^2 + Sqrt[1/a^2 + 1/a]]]
Out[2032]= "Plus[Power[Plus[Power[a, -2], Power[a, -1]], Rational[1, 2]], Power[x, 2]]"
In[2033]:= ToExpression[%]
Out[2033]= Sqrt[1/a^2 + 1/a] + x^2
```

Обратный переход к стандартной входной форме весьма прост и проиллюстрирован в последнем примере предыдущего фрагмента. Приведенное средство полезно как для структурного анализа, так и для представления выражений. Итак, любое выражение может рассматриваться как некоторая линейная хорошо обрабатываемая структура, облегчающая его анализ и манипуляции с ним программными средствами. Указанная возможность используется и рядом наших средств, манипулирующих выражениями.

**Reverse[x, {n}]** – вызов функции в зависимости от фактических аргументов возвращает результат обращения подвыражений выражения  $x$  в целом либо подвыражений на  $n$ -уровне; в целом же, функция располагает тремя форматами кодирования. Между тем, данная функция имеет вполне определенный смысл, в основном, только для списков, выполняя *обращение* их элементов. Тогда как, например, для формульных выражений данная функция зачастую не имеет смысла ввиду использования пакетом, т.н. *полной формы (FullForm)* представления выражений, которая основывается на используемой системе приоритетов. Следующий простой фрагмент иллюстрирует сказанное:

```
In[2040]:= X = (x^2 - y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z - G[h, t]]; Reverse[X]
Out[2040]= a Log[x + y + z - G[h, t]] + (x^2 - y^2)/(Cos[y] + Sin[x])
In[2041]:= Reverse[X, 3]
Out[2041]= (-1)^(Cos[y] + Sin[x]) (x^2 - y^2) + a Log[x + y + z - G[h, t]]
In[2042]:= Y = {{a, b, c}, {c, d, g}, {x, y, z}}; Reverse[Y]
Out[2042]= {{x, y, z}, {c, d, g}, {a, b, c}}
```

В целом ряде случаев достаточно полезной оказывается *глобальная* переменная, вызов которой **FullFormF[]** возвращает список наиболее часто используемых эквивалентов функций и операторов, используемых в полной форме представления выражений. В первую очередь данная переменная полезна при *анализе* выражений, представленных в *полной* форме. Фрагмент представляет код переменной с примером ее применения.

```
In[2047]:= FullFormF[] := {And, AngleBracket, CapitalDifferentialD, CenterDot, Complex,
Condition, Congruent, Decrement, DifferentialD, DoubleVerticalBar, DirectedEdge,
Element, Equal, Equivalent, Exists, ForAll, Function, GreaterFullEqual, Greater,
GreaterEqual, Implies, Increment, Infinity, Intersection, LeftArrow, LessEqual, Less,
```

```

LessFullEqual, LessLess, MessageName, MinusPlus, Not, NotDoubleVerticalBar,
NotCongruent, NotElement, NotExists, NotReverseElement, NotSubset, NotSuperset,
Or, Part, Pattern, Plus, PlusMinus, Power, Product, Proportion, Proportional, Rational,
Repeated, ReverseElement, RoundImplies, Rule, RuleDelayed, SameQ, Span,
StringJoin, Subset, SubsetEqual, Sum, Superset, SupersetEqual, TildeEqual, TildeTilde,
Times, UndirectedEdge, Unequal, Union, UpArrow, VerticalBar, VerticalSeparator}
In[2048]:= FullFormF[]
Out[2048]= {And, AngleBracket, CapitalDifferentialD, CenterDot, Complex, Condition,
Congruent, Decrement, DifferentialD, DoubleVerticalBar, DirectedEdge,
Element, Equal, Equivalent, Exists, ForAll, Function, GreaterFullEqual,
Greater, GreaterEqual, Implies, Increment,  $\infty$ , ..., VerticalSeparator}
In[2049]:= Map4[FreeQ2, Map[FullForm, {a + b*Cos[x], a^b, a -> b}], {Plus, Power, Rule}]
Out[2049]= {False, False, False}
In[2202]:= FreeQ1[x_, y_] := Module[{h}, Quiet[FreeQ[Subs[x, y, h = Unique["ArtKr"]], h]]]
In[2203]:= {FreeQ1[1/(x^2 + b), x^2], FreeQ[1/x^2, x^2]}
Out[2203]= {False, True}
In[2204]:= {FreeQ1[1/x^3 + b, x^3], FreeQ[1/x^3 + b, x^3]}
Out[2204]= {False, True}
In[2205]:= {FreeQ1[a/Sqrt[x], Sqrt[x]], FreeQ[a/Sqrt[x], Sqrt[x]]}
Out[2205]= {False, True}
In[2206]:= {FreeQ1[{42, 47, Sqrt[x], 17, 24}, Sqrt[x]], FreeQ[{42, 47, Sqrt[x], 17, 24}, Sqrt[x]]}
Out[2206]= {False, False}
In[2250]:= FreeQ2[x_, p_] := If[ListQ[p], If[DeleteDuplicates[Map10[FreeQ, x, p]] ===
{True}, True, False], FreeQ[x, p]]
In[2251]:= L = {a, b, c, d, f, g, h}; {FreeQ[L, {a, d, h}], FreeQ2[L, {a, d, h}]}
Out[2251]= {True, False}
In[2252]:= {FreeQ[Cos[x] + Sin[x], {Sin, Cos}], FreeQ2[Cos[x] + Sin[x], {Sin, Cos}]}
Out[2252]= {True, False}
In[2253]:= {FreeQ[Cos[x] + Sin[x], Sin], FreeQ2[Cos[x] + Sin[x], Sin]}
Out[2253]= {False, False}

```

Предыдущий фрагмент представляет довольно простую процедуру **FreeQ1**, которая существенно расширяет стандартную функцию **FreeQ**, обеспечивая более широкое тестирование вхождений в выражение подвыражений. Вызов **FreeQ1**[*x*, *y*] возвращает значение **True**, если выражение *x* не содержит подвыражений *y*, в противном случае возвращается **False**. Фрагмент представляет как исходный код процедуры **FreeQ1**, так и типичные примеры ее применения в сравнительном контексте с функцией **FreeQ**.

Завершает данный фрагмент функция **FreeQ2**, расширяющая стандартную функцию **FreeQ** дополнительно и на список в качестве второго фактического аргумента. Вызов функции **FreeQ2**[*x*, *p*] возвращает **True**, если выражение *x* не содержит подвыражения

*p* либо подвыражений из списка *p*, в противном случае возвращается **False**. Фрагмент представляет исходный код функции с примерами применения и в *сравнении* с **FreeQ**. Данная функция используется и примерами фрагмента, и во многих приложениях.

Используя функцию **FullForm**, обеспечивающую представление выражений в полной форме, можно получить достаточно простую процедуру, решающую задачу замены в выражениях заданных подвыражений. Вызов **Replace3[x, y, z]** процедуры возвращает результат *замены* в выражении *x* *всех* вхождений в него подвыражений *y* на выражения *z*; в качестве аргументов {*y, z*} могут выступать отдельные выражения либо их списки. Фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2062]:= Replace3[x_, y_, z_] := Module[{a, b = Flatten[{y}], c = Flatten[{z]}, d,
      {a, b} = {FullForm[x] // ToString, Map12[ToString, Map12[FullForm, {b, c}]]};
      d = Min[Map[Length, {b, c}]]; If[d < Length[b[[1]]], Print["Subexpressions " <>
      ToString[InputForm[ToExpression[b[[1]]][[d + 1 ;; -1]]]] <> " were not replaced"];
      ToExpression[StringReplace[a, Map9[Rule, b[[1]][[1 ;; d]], b[[2]][[1 ;; d]]]]]

In[2063]:= Replace3[x^2 + Sqrt[1/a^2 + 1/a - Sin[1/a]], 1/a, Cos[h]]
Out[2063]= x^2 + Sqrt[1/a^2 + Cos[h] - Sin[Cos[h]]]
In[2064]:= Replace3[1/(1 + 1/a) + Cos[1/a + Sin[1/a]]*(c + 1/a)^2, 1/a, F[h] + d]
Out[2064]= 1/(1 + d + F[h]) + Cos[d + F[h] + Sin[d + F[h]]] (c + d + F[h])^2
In[2065]:= Replace3[x^2 + Sqrt[1/a^2 + 1/a - Sin[1/a]], {1/a, 1/b}, Cos[h]]
      Subexpressions {b^(-1)} were not replaced
Out[2065]= x^2 + Sqrt[1/a^2 + Cos[h] - Sin[Cos[h]]]
```

При этом, в случае аргументов {*y, z*} в форме списка для них выбирается общая длина, определяемая соотношением **Min[Map[Length, {y, z}]]**, позволяя избегать возможных особых и ошибочных ситуаций, требующих программной обработки, но с выводом на экран соответствующей диагностической информации, как иллюстрирует пример.

Таким образом, для обработки выражений различных типов на основе составляющих их частей можно в довольно широких пределах использовать большинство *списочных* операций, часть из которых рассмотрена выше и будет рассматриваться далее в книге. Посредством данных функций возможно управлять структурой выражений разными способами, обеспечивая решение конкретных прикладных и системных задач. Далее будет представлен ряд и других средств манипулирования с выражениями.

В ряде случаев требуется выполнить *обмен* значений переменных с соответствующим обменом всех их атрибутов. Например, переменные *x* и *y*, имеющие значения **70** и **65**, должны получить значения **42** и **47** соответственно с соответствующим обменом *всеми* их атрибутами. Следующая процедура **VarExch[L]** решает данную задачу, возвращая значение **Null**, т.е. ничего. В качестве фактического аргумента выступает либо список *двух имен* переменных, которые обмениваются значениями и атрибутами, в строчном формате, либо вложенный список из подсписков указанного типа; в любом случае все элементы пар должны быть определенными, иначе вызов возвращает значение **Null** с выводом соответствующего диагностического сообщения. Нижеследующий фрагмент представляет исходный код процедуры с типичными примерами ее использования.

```

In[2545]:= VarExch[L_List /; Length[L] == 2 || ListListQ[L] && Length[L[[1]]] == 2] :=
Module[{Kr, k=1}, Kr[p_List] := Module[{a = Map[Attributes, p], b, c, m, n},
ToExpression[{"ClearAttributes[" <> StrStr[p[[1]]] <> "," <>
ToString[a[[1]]] <> "], "ClearAttributes[" <> StrStr[p[[2]]] <> "," <>
ToString[a[[2]]] <> ""]}]; {b, c} = ToExpression[{"ToString[Definition[" <>
StrStr[p[[1]]] <> "]", "ToString[Definition[" <> StrStr[p[[2]]] <> ""]}"];
If[MemberQ[{b, c}, "Null"], Print[VarExch::"Both actual arguments should
be defined but uncertainty had been detected: ", p]; Return[], Null];
{m, n} = Map4[StringPosition, Map[StrStr, {b, c}], Map[StrStr, {" := ", " = "}]];
{n, m} = {StringTake[b, {1, m[[1]][[1]] - 1}] <> StringTake[c, {n[[1]][[1]], -1}],
StringTake[c, {1, n[[1]][[1]] - 1}] <> StringTake[b, {m[[1]][[1]], -1}];
ToExpression[{n, m}];
Map[ToExpression, {"SetAttributes[" <> StrStr[p[[1]]] <> "," <>
ToString[a[[2]]] <> "], "SetAttributes[" <> StrStr[p[[2]]] <> "," <>
ToString[a[[1]]] <> ""]}]; If[! ListListQ[L], Kr[L],
For[k, k <= Length[L], k++, Kr[L[[k]]]]; ]

In[2546]:= Agn = 65; Avz := 70; Art := 24; Kr = 17; SetAttributes["Agn", Protected];
SetAttributes["Art", {Protected, Listable}]

In[2547]:= Map[Attributes, {"Agn", "Avz", "x", "y", "Art", "Kr"}]
Out[2547]= {{Protected}, {}, {}, {}, {Listable, Protected}, {}}

In[2548]:= VarExch[{"Avz", "Agn"}, {"x", "y"}, {"Art", "Kr"}]
VarExch::Both actual arguments should be defined but
uncertainty had been detected: {x, y}

In[2549]:= {Avz, Agn, Art, Kr}
Out[2549]= {65, 70, 17, 24}

In[2550]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[2550]= {{}, {Protected}, {}, {Listable, Protected}}

In[2551]:= Rename[x_String /; HowAct[x], y_ /; ! HowAct[y]] := Module[{a, c, d,
b = Flatten[{PureDefinition[x]}], If[! SameQ[b, {$Failed}], a = Attributes[x];
c = ClearAllAttributes[x]; d = StringLength[x];
c = Map[ToString[y] <> StringTake[#, {d + 1, -1}] &, b];
Map[ToExpression, c]; Clear[x]; SetAttributes[y, a]]]

In[2552]:= fm = "Art_Kr"; SetAttributes[fm, {Protected, Listable}]; {fm, Attributes[fm]}
Out[2552]= {"Art_Kr", {Listable, Protected}}

In[2553]:= Rename["fm", Tampere]
In[2554]:= {Tampere, Attributes[Tampere], fm}
Out[2554]= {"Art_Kr", {Listable, Protected}, fm}

In[2555]:= s = HowAct[Q]; SetAttributes[Q, {Listable, Protected}]; {s, Attributes[Q]}
Out[2555]= {False, {Listable, Protected}}

```

```

In[2556]:= Rename[Tampere, Q]
Out[2556]= Rename["Art_Kr", Q]

In[2557]:= VarExch1[L_List /; Length[L] == 2 | | ListListQ[L] && Length[L[[1]]] == 2] :=
Module[{Art, k = 1, d}, Art[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
$Aborted]], b, c, m, n}, If[a == $Aborted, Return[Defer[VarExch1[L]], Null];
If[HowAct[$Art$], b = $Art$; Clear[$Art$]; m = 1, Null];
If[HowAct[$Kr$], c = $Kr$; Clear[$Kr$]; n = 1, Null];
ToExpression[{"ClearAttributes[" <> StrStr[p[[1]]] <> ", " <>
ToString[a[[1]]] <> "]", "ClearAttributes[" <> StrStr[p[[2]]] <> ",
" <> ToString[a[[2]]] <> "]}];
ToExpression[{"Rename[" <> StrStr[p[[1]]] <> ", " <> "$Art$" <> "]",
"Rename[" <> StrStr[p[[2]]] <> ", " <> "$Kr$" <> "]}];
ToExpression["Clear[" <> StrStr[p[[1]]] <> ", " <> StrStr[p[[2]]] <> "];
ToExpression[{"Rename[" <> StrStr["$Kr$"] <> ", " <> p[[1]] <> "]",
"Rename[" <> StrStr["$Art$"] <> ", " <> p[[2]] <> "]}];
Map[ToExpression, {"SetAttributes[" <> StrStr[p[[1]]] <> ", " <>
ToString[a[[2]]] <> "]", "SetAttributes[" <> StrStr[p[[2]]] <> ", " <>
ToString[a[[1]]] <> "]}];
If[m == 1, $Art$ = b, Null]; If[n == 1, $Kr$ = c, Null]; ];
If[! ListListQ[L], Art[L], For[k, k <= Length[L], k++, Art[L[[k]]]]]]

In[2558]:= Agn = 65; Avz := 70; Art := 24; Kr = 17; SetAttributes["Agn", Protected];
SetAttributes["Art", {Protected, Listable}];

In[2559]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[2559]= {{Protected}, {}, {Listable, Protected}, {}}
In[2560]:= {$Art$, $Kr$} = {78, 460}; VarExch1[{"Agn", "Avz"}, {"x", "y"}, {"Art", "Kr"}]
In[2561]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]}
Out[2561]= {{460, 78, 17, 24}, {{}, {Protected}, {}, {Listable, Protected}}}
In[2562]:= {x, y, $Art$, $Kr$}
Out[2562]= {x, y, $Art$, $Kr$}

```

Процедура **Rename**[*x*, *y*] в штатном режиме возвращает *Null*, т.е. ничего, обеспечивая замену имени *x* определенного объекта на имя *y* с сохранением всех атрибутов этого объекта. Если *y*-аргумент определяет имя *определенного* объекта или *неопределенное* имя с приписанными ему атрибутами, вызов процедуры возвращается невычисленным. В случае же *первого* аргумента *x*, недопустимого для переименования, вызов процедуры возвращает *Null*, т.е. ничего; процедура **Rename** успешно обрабатывает и *одноименные* объекты *x* типа **Block**, **Function** либо **Module**.

Иную реализацию алгоритма процедуры **VarExch**, базирующуюся на использовании процедуры **Rename** и глобальных переменных, иллюстрирует процедура **VarExch1**, в принципе, допускающая тот же тип фактического аргумента, но в отличие от первой

процедуры при обнаружении *неопределенных* элементов списка *L* либо его подписков вызов **VarExch1** возвращается *невычисленным* без вывода какого-либо *диагностического* сообщения. Между тем, здесь вполне уместно еще раз акцентировать наше внимание на таком важном аспекте, как использование в процедурах *глобальных* переменных, в целом ряде случаев позволяющих, порой, существенно упрощать программирование процедур. Такой механизм можно реализовать следующим образом, а именно: перед использованием в процедуре глобальной переменной она тестируется на предмет ее определенности (*наличия присвоенного ей значения*); в случае неопределенности данной переменной она *свободно* используется в процессе выполнения алгоритма процедуры, однако перед выходом из процедуры (*если через нее не предусмотрено выхода*) ее следует вновь сделать неопределенной. Если же глобальная переменная *до* входа в процедуру уже имела значение, оно должно быть сохранено в *локальной* переменной до ее *первого* использования в процедуре, тогда как перед выходом из процедуры значение данной переменной должно быть восстановлено. Однако, механизм глобальных переменных в среде системы *Mathematica* не является универсальным, вполне корректно работая в случае вычислений определений процедур, содержащих глобальные переменные, в текущем сеансе в *Input*-параграфе; между тем, как в общем случае не поддерживается при загрузке в текущий сеанс системы процедур, содержащих *глобальные* переменные, в частности, из *nb*-файлов данных с последующей активизацией их содержимого.

В качестве примера приведем фрагмент, в котором в текущий сеанс загружается наш пакет *AVZ\_Package\_1* [120] с процедурой **VarExch1** предыдущего фрагмента, которая использует две переменные **\$Art\$** и **\$Kr\$** в качестве *глобальных*, после чего процедура используется для обмена значений для переменных *Agn, Avz, Art* и *Kr*, из которых две переменные *Agn, Art* имеют *Protected*-атрибут. Из результата вызова этой процедуры следует, что исходные значения (*до вызова процедуры*) глобальных переменных **\$Art\$** и **\$Kr\$** не сохраняются в результате выполнения процедуры **VarExch1**. Более того, даже значения для переменных *Agn* и *Avz* получаются некорректными (*причина этого здесь не рассматривается*). Поэтому для устранения подобной ситуации нами предлагается процедура, вызов **NbCallProc[x]** которой переактивизирует в текущем сеансе системы блок, функцию или модуль *x*, чье определение находилось в загруженном в текущий сеанс системы *nb*-файле (*в данном случае в файле AVZ\_Package\_1.nb*), с возвратом **Null**, т.е. ничего. Следующий фрагмент представляет исходный код процедуры **NbCallProc** наряду с наиболее типичными примерами ее использования.

```
In[2010]:= Agn = 66; Avz := 71; Art := 24; Kr = 17; SetAttributes["Agn", Protected];
SetAttributes["Art", {Protected, Listable}];
In[2011]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[2011]= {{Protected}, {}, {Listable, Protected}, {}}
In[2012]:= {$Art$, $Kr$} = {78, 460}; VarExch1[{"Agn", "Avz"}, {"x", "y"}, {"Art", "Kr"}]
In[2013]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]}
Out[2013]= {{460, 78, 17, 24}, {{}, {Protected}, {}, {Listable, Protected}}}
In[2014]:= {x, y, $Art$, $Kr$}
Out[2014]= {x, y, $Art$, $Kr$}
```



```

In[2015]:= NbCallProc[x_;/; BlockFuncModQ[x]] := Module[{a = SubsDel[StringReplace[
    ToString1[DefFunc[x]], "\n\n" -> ";", "\"" <> ToString[x] <> "\"", {"[", " "], -1}],
    Clear[x]; ToExpression[a]]

In[2016]:= NbCallProc[VarExch1]
In[2017]:= Unprotect[Agn, Avz, Art, Kr, $Art$, $Kr$]
Out[2017]= {"Avz", "Kr"}
In[2018]:= Clear[Agn, Avz, Art, Kr, $Art$, $Kr$]
In[2019]:= Agn = 66; Avz := 71; Art := 24; Kr = 17; SetAttributes["Agn", Protected];
    SetAttributes["Art", {Protected, Listable}];

In[2020]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[2020]= {{Protected}, {}, {Listable, Protected}, {Listable}}
In[2021]:= {$Art$, $Kr$} = {78, 460}; VarExch1[{"Agn", "Avz"}, {"x", "y"}, {"Art", "Kr"}]
In[2022]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]}
Out[2022]= {{71, 66, 17, 24}, {{}, {Protected}, {Listable}, {Listable, Protected}}}
In[2023]:= {x, y, $Art$, $Kr$}
Out[2023]= {x, y, 78, 460}
    
```

Следует отметить, вызов процедуры **NbCallProc[x]** переактивирует в текущем сеансе все определения блоков, функций и модулей с *идентичным* именем *x* и с различными заголовками. Эти определения предварительно должны быть загружены из *nb*-файла в текущий сеанс и активизированы по функции *Evaluate Notebook*. Успешный вызов процедуры **NbCallProc[x]** возвращает *Null*, т.е. ничего, переактивизируя средство *x*.

Предыдущий фрагмент представляет как исходный код процедуры **NbCallProc**, так и результат использования процедуры **VarExch1** для обмена значений у переменных *Agn*, *Avz*, *Art* и *Kr* после вызова **NbCallProc[VarExch1]**. Полученные здесь результаты достаточно наглядно показывают, что процедура **VarExch1**, содержащая глобальные переменные и загруженная из *nb*-файла с последующей его активацией (*no Evaluate Notebook*), выполняется совершенно корректно и с корректным функционированием механизма глобальных переменных, восстанавливающих свои значения после выхода из процедуры **VarExch1**. Процедура **NbCallProc** имеет ряд интересных приложений и, прежде всего, при необходимости использования процедур, активированных именно в *Input*-параграфе текущего сеанса системы *Mathematica*.

На примере реализации процедуры **VarExch1** иллюстрируется и такой очень важный момент, как обработка ошибочных ситуаций при назначении *локальным* переменным начальных значений. Естественно, если *присваиваемые* начальные значения зависят от получаемых процедурой при вызове фактических аргументов, то подобные ситуации вполне возможны. Для обработки подобных ситуаций возможно использовать целый ряд механизмов, некоторые из которых иллюстрируются следующим фрагментом.

```

In[2494]:= Agn = 66; SetAttributes[Agn, {Protected, Listable}]; Attributes[Agn];
In[2495]:= Art24[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
    Return[Defer[Art24[p]]]]], b = 460, c}, c = N[Sin[78]]; {a, b, c}]
    
```

```

In[2496]:= Art24[{"Agn", 71}]
Out[2496]= {Art24[{"Agn", 71}], 460, 0.513978}
In[2497]:= Art24[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
Print[Defer[Art24[p]]]; Abort[], b = 460, c], c = N[Sin[78]]; {a, b, c}]
In[2498]:= Art24[{"Agn", 71}]
          Art24[{"Agn", 71}]
Out[2498]= $Aborted
In[2499]:= Art24[p_List] := Module[{a = Quiet[Check[Map[Attributes, p], $Aborted]],
          b = 460, c], If[a == $Aborted, Return[Defer[Art24[p]], Null]; c = N[Sin[78]]; {a, b, c}]
In[2500]:= Art24[{"Agn", 71}]
Out[2500]= Art24[{"Agn", 71}]

```

Из фрагмента следует, что использование для выхода из процедуры в случае ошибки при вычислении начального значения для *локальной* переменной функции **Return** не дает ожидаемого результата, т.е. не обеспечивается *выход* из процедуры, не выполняя тело процедуры, что, в общем, недопустимо. Более приемлем в данном случае второй вариант, однако наиболее удобен для последующей обработки вызова процедуры 3-й вариант, который и был использован при реализации процедуры **VarExch1**.

В целом ряде случаев перед обновлением определений объектов (*процедуры, функции, переменные и др.*) требуется проверка на наличие у них атрибута **Protected**, что вполне обеспечивает очень простая функция, чей вызов **ProtectedQ[x]** возвращает **True**, если объект *x* имеет атрибут **Protected**, и **False** в противном случае. В качестве аргумента *x* может выступать произвольное корректное выражение. Нижеследующий фрагмент представляет исходный код функции **ProtectedQ** наряду с рядом некоторых довольно типичных примеров ее использования.

```

In[2030]:= ProtectedQ[x_] := If[MemberQ[Attributes1[x], Protected], True, False]
In[2031]:= g = 78; Protect[g]; Map[ProtectedQ, {Sin, Protect, AVZ, HowAct, 460, Map, "g"}]
Out[2031]= {True, True, False, False, False, True, True}
In[2032]:= Avz = 78; S = a + b; Protect[ArtKr]; Map[ProtectedQ, {"Avz", "S", Sin, ArtKr}]
Out[2032]= {False, False, True, True}

```

Как уже отмечалось, списочная структура является одной из базовых в *Mathematica* в еще большей степени, чем у ее конкурента *Maple*. И *Maple*, и в еще большей степени *Mathematica* располагают довольно развитым набором средств обработки списочных структур. Одним из таких довольно важных средств является *конвертация* выражений в списки; для *Maple* такое средство имеет вид **convert(Exp, list)**, тогда как *Mathematica* аналогичного средства не имеет, и в качестве его может выступить процедура, вызов которой **ToList[Exp]** возвращает результат конвертации выражения *Exp* в список. При этом, в случае строки *Exp* производится *конвертация* в посимвольный список, в случае списка *Exp* возвращается *Exp*, тогда как в остальных случаях конвертация базируется на стандартной функции **Map** пакета. Следующий фрагмент представляет исходный код процедуры **ToList** наряду с довольно типичными примерами ее использования.

```

In[2070]:= ToList[expr_] := Module[{a, b, c = {}, d, k = 1, n},
    If[StringQ[expr], Characters[expr], If[ListQ[expr], expr,
        a = ToString[InputForm[Map[b, expr]]];
        d = StringSplit[a, ToString[b] <> "["];
        For[k, k <= Length[d], k++, n = d[[k]];
        c = Append[c, StringTake[n, {1, Flatten[StringPosition[n, "]"][[-1] - 1]}]];
        ToExpression[c]]]

In[2071]:= ToList[(a*Sin[x] + g[b])/(c + d) + (d + c)/(Cos[y] + h)]
Out[2071]= {(c + d)/(h + Cos[y]), (g[b] + a Sin[x])/(c + d)}
In[2072]:= ToList[Sqrt[a*Sin[x] + g[b])/(c + d) + (d + c)/(Cos[y] + h)]
Out[2072]= {(c + d)/(h + Cos[y]), Sqrt[g[b] + a Sin[x])/(c + d)}
In[2073]:= x = "qwertyuiopasdfghjklzxcvbnmxyzavzagnvsv"; ToList[x]
Out[2073]= {"q", "w", "e", "r", "t", "y", "u", "i", "o", "p", "a", "s", "d", "f", "g", "h", "j", "k", "l",
    "z", "x", "c", "v", "b", "n", "m", "x", "y", "z", "a", "v", "z", "a", "g", "n", "v", "s", "v"}
In[2074]:= x = {a, b, c, d, e, f, g, h, a, v, z, a, g, n, A, r, t, K, r}; ToList[x]
Out[2074]= {a, b, c, d, e, f, g, h, a, v, z, a, g, n, A, r, t, K, r}

```

Среда *Maple* располагает двумя полезными средствами манипуляции с выражениями типа уравнение, неравенство, отношение, диапазон, вызовы *lhs(Ex)* и *rhs(Ex)* которых возвращают левую и правую части выражения *Ex* соответственно. Точнее, вызов *lhs(Ex)*, *rhs(Ex)* возвращает соответственно значения *op(1, Ex)*, *op(2, Ex)*. Тогда как *Mathematica* подобными полезными процедурами не располагает. Данный недостаток восполняет процедура **RhsLhs**, чей исходный код с примерами использования приведены ниже.

```

In[2080]:= RhsLhs[x_] := Module[{a = Head[{x}][[1]], b = ToString[InputForm[{x}][[1]]], d,
    h = {x}, c = {{Greater, ">"}, {GreaterEqual, ">="}, {And, "&&"}, {LessEqual, "<="},
    {Unequal, "!="}, {Or, " | "}, {Rule, "->"}, {Less, "<"}, {Plus, {"+", "-"}}, {Power, "^"},
    {Equal, "=="}, {Span, ";;"}, {NonCommutativeMultiply, "***"}, {Times, {"*", "/"}}},
    If[Length[h] < 2 || ! MemberQ[{"Lhs", "Rhs"}, h[[2]]],
    Return[Defer[RhsLhs[x]]], Null];
    If[! MemberQ[Select[Flatten[c], ! StringQ[#] &], a] || a == Symbol,
        Return[Defer[RhsLhs[x]]], Null];
        d = StringPosition[b, Flatten[Select[c, #[[1]] == a &], 1][[2]]];
        If[Length[h] >= 3 && ! HowAct[h[[3]]],
            ToExpression[ToString[h[[3]]] <> "=" <> ToString[a], Null];
            ToExpression[If[h[[2]] == "Lhs", StringTrim[StringTake[b, {1, d[[1]][[1]] - 1}]],
                StringTrim[StringTake[b, {d[[1]][[2]] + 1, -1}]]]]

In[2081]:= {RhsLhs[(a + b)*d -> c, "Lhs", x], x}
Out[2081]= {(a + b) d, Rule}
In[2082]:= RhsLhs[a + b -> c^5, "Rhs"]
Out[2082]= c^5

```

```
In[2083]:= {RhsLhs[a + b^c + d, "Rhs", y], y}
Out[2083]= {b^c + d, Plus}
In[2084]:= {RhsLhs[a/d + b^c, "Rhs", z], z}
Out[2084]= {a/d, Plus}
In[2085]:= {RhsLhs[a == c + d, "Rhs", h], h}
Out[2085]= {c + d, Equal}
In[2086]:= {RhsLhs[a/c^d, "Rhs", t], t}
Out[2086]= {c^d, Times}
In[2087]:= {RhsLhs[Art_Kr, "Rhs", g], g}
Out[2087]= {RhsLhs[Art_Kr, "Rhs", g], g}
In[2088]:= {RhsLhs[78 ;; 460, "Rhs", s], s}
Out[2088]= {460, Span}
In[2089]:= {RhsLhs[a && b, "Rhs", g], g}
Out[2089]= {b, And}
In[2090]:= {RhsLhs[a ** b, "Rhs", v], v}
Out[2090]= {b, NonCommutativeMultiply}
```

Вызов процедуры **RhsLhs**[*x*, *y*] в зависимости от значения {"Lhs", "Rhs"} 2-го аргумента *y* возвращает соответственно левую либо правую части выражения *x* по отношению к оператору **Head**[*x*], тогда как вызов **RhsLhs**[*x*, *y*, *t*] дополнительно через *неопределенную* переменную *t* возвращает оператор **Head**[*x*], относительно которого и производилось разбиение выражения *x* на левую и правую части. Процедура **RhsLhs** довольно легко модифицируется в свете расширения анализируемых операторов **Head**[*x*]. Оставляем это читателю в качестве довольно полезного практического упражнения. Более того, следующий фрагмент представляет процедуру **RhsLhs1**, являющуюся *функционально* эквивалентной модификацией процедуры **RhsLhs**; вызов процедуры **RhsLhs1**[*x*, *y*] в зависимости от значения {"Lhs", "Rhs"} 2-го аргумента *y* возвращает соответственно левую либо правую части выражения *x* по отношению к оператору **Head**[*x*], тогда как вызов **RhsLhs1**[*x*, *y*, *h*] дополнительно через *неопределенную* переменную *h* возвращает оператор **Head**[*x*], относительно которого производилось разбиение выражения *x* на *левую* и *правую* части. Следующий фрагмент представляет исходный код **RhsLhs1** с некоторыми примерами ее наиболее типичного использования.

```
In[2100]:= RhsLhs1[x_, y_] := Module[{a = Head[x], b = {x, y}, d, c = {{Greater, ">"},
                                     {GreaterEqual, ">="}, {And, "&&"}, {LessEqual, "<="},
                                     {Unequal, "!="}, {Or, "||"}, {Rule, "->"}, {Less, "<"},
                                     {Plus, {"+", "-"}}, {Power, "^"}, {Equal, "=="}, {Span, ";;"},
                                     {NonCommutativeMultiply, "***"}, {Times, {"*", "/"}}}},
  If[! MemberQ[Flatten[c], a], Return[Defer[RhsLhs1[x, y]], d = Level[x, 1]];
  If[Length[b] > 2 && ! HowAct[b[[3]]], ToExpression[ToString[b[[3]]] <>
    " = " <> ToString1[Flatten[Select[c, #[[1]] === a &]]], Null];
  If[b[[2]] == "Lhs", d[[1]], If[b[[2]] == "Rhs", d[[2]], Defer[RhsLhs1[x, y]]]]
```

```
In[2101]:= RhsLhs1[Sin[x]]
Out[2101]= RhsLhs1[Sin[x]]
In[2102]:= Mapp[RhsLhs1, {a > b, a + b, a^b, a*b, a -> b, a <= b, a || b, a && b}, "Rhs"]
Out[2102]= {b, b, b, b, b, b, b, b}
In[2103]:= Mapp[RhsLhs1, {a > b, a + b, a^b, a*b, a -> b, a <= b, a || b, a && b}, "Lhs"]
Out[2103]= {a, a, a, a, a, a, a, a}
In[2104]:= {RhsLhs1[a || b, "Rhs", w], w}
Out[2104]= {b, {Or, " || "}}
```

При этом, в случае невозможности выделения частей вызовов обеих процедур **RhsLhs** и **RhsLhs1** возвращается невычисленным, как весьма наглядно иллюстрируют простые примеры последних двух фрагментов.

*Maple* располагает средствами тестирования выражений на нижеследующие типы:

```
{!, `*, `+, `., `::, `<, `<=, `<>, `=, `@, `@@, `^, `| |, `and, `or, `xor, `implies, `not`}
```

В *Mathematica* средства такого довольно широкого диапазона отсутствуют и в данной связи ниже представлена процедура, чей вызов **TwoHandQ[x]** возвращает **True**, если *x* выражение имеет один из нижеследующих типов, а именно:

```
{"+", ">=", "<=", "&&", " || ", "-", "^", "***", "<", "==", "!=", ">", "->"}
```

и **False** в противном случае; более того, если вызов **TwoHandQ[x,y]** возвращает **True**, то через *второй* необязательный аргумент *y* – *неопределенную переменную* – возвращается тип выражения *x*. Нижеследующий фрагмент представляет исходный код процедуры наряду с наиболее типичными примерами ее использования.

```
In[2034]:= TwoHandQ[x_] := Module[{a = ToString[InputForm[{x}][[1]]],
    b = {"+", ">=", "<=", "&&", " || ", "-", "^", "***", "<", "==", "!=", ">", "->"}, c, d = {x}},
    c = StringPosition[a, b]; If[StringFreeQ[a, "->"] && StringFreeQ[a, ">="] &&
    Length[c] > 2 || Length[c] == 0, False, If[Length[d] > 1 && !HowAct[d[[2]]] &&
    !ProtectedQ[d[[2]]], ToExpression[ToString[d[[2]]] <> "=" <>
    ToString[Head[{x}][[1]]], Return[Defer[TwoHandQ[x]]]; True]]

In[2035]:= {TwoHandQ[a3 <= w, h], h}
Out[2035]= {True, LessEqual}
In[2036]:= {TwoHandQ[a -> b, t], t}
Out[2036]= {True, Rule}
In[2037]:= {TwoHandQ[a == b && c, z], z}
Out[2037]= {True, And}
In[2038]:= {TwoHandQ[a != b, p], p}
Out[2038]= {True, Unequal}
In[2039]:= Clear[z]; {TwoHandQ[a < b && c, z], z}
Out[2039]= {True, And}
In[2040]:= {TwoHandQ[a >= b, r], r}
Out[2040]= {True, GreaterEqual}
```

```
In[2041]:= Clear[p]; {TwoHandQ[a | | b + c, p], p}
Out[2041]= {True, Or}
In[2042]:= Clear[p]; {TwoHandQ[a -> b + c && d, p], p}
Out[2042]= {True, Rule}
```

В среде *Maple* определен тип индексированных выражений, что в целом ряде случаев дает возможность более эффективно обрабатывать выражения данного типа. В [97-99] представлено средство вычисления индексов для выражений такого типа, тогда как в пакете *Mathematica* подобные средства отсутствуют. Для устранения этого недостатка предлагаются две достаточно простые процедуры, а именно **IndexedQ** и **Index**. Вызов процедуры **IndexedQ[Quiet[x]]** возвращает *True*, если *x* – индексированное выражение, и *False* в противном случае. Тогда как вызов **Index[Quiet[x]]** возвращает индекс в случае индексированности выражения *x*, иначе вызов возвращается невычисленным. Простой фрагмент представляет исходные коды процедур и примеры их применения. Данные средства оказываются достаточно полезными при обработке ряда индексированных выражений, в частности, при обработке списочных структур, в частности, массивов.

```
In[2006]:= IndexedQ[x_] := Module[{a = Quiet[ToString[x]], b}, b = StringPosition[a, "[[[";
If[StringTake[a, {-2, -1}] == "[[" &&
Plus[SymbolQ[StringTake[a, {1, b[[1]][[1]] - 1]]], True, False]]
In[2007]:= {IndexedQ[Quiet[17[[2 ;; 4]]], IndexedQ[Quiet[ArtKr[[17 ;; 24]]]]}
Out[2007]= {False, True}
In[2008]:= Index[x_ /; IndexedQ[x]] := Module[{a = Quiet[ToString[x]], b, c},
b = Flatten[StringPosition[a, "[[["];
ToExpression[StringTake[a, {b[[2]] + 1, -3}]]]
In[2009]:= {Index[Quiet[VSV[[17 ;; 24]]], Index[Quiet[SVEGAL[[78]]]]}
Out[2009]= {17 ;; 24, 78}
In[2010]:= Index[Quiet[456[[1 ;; 6]]]]
Out[2010]= Index[460[[1 ;; 6]]]
In[2011]:= H[1] = 78; H[2] = 47; H[6] = 17; H[9] = 24; H[17] = 42; H[24] = 71;
In[2012]:= (H[6] + H[2] + H[1])/(H[9] + H[24] + H[17])
Out[2012]= 142/137
In[2013]:= ArrayInd[x_ /; StringQ[x]] := Module[{a = StringSplit[ToString[InputForm[
DefFunc[x]]], "\n\n"]}, If[a == {"Null"}, {}, a]]
In[2014]:= Map[ArrayInd, {"H", "G"}]
Definition::notfound: Symbol G not found. >>
Out[2014]= {"H[1] = 78", "H[2] = 47", "H[6] = 17", "H[9] = 24", "H[17] = 42", "H[24] = 71"}, {}}
```

Во многих случаях вычисления используют множества значений, из которых базовым примером является список *L*, к чьим элементам можно обращаться по конструкциям вида *L[[n]]* и *L[[n ;; m]]*, где *n*, *m* – позиции элементов в списке. Между тем, списочная структура имеет недостаток, требуя в момент ее определения задания *всех* элементов.

Однако пакет дает возможность указания в качестве элементов множества только тех, которые известны в данный момент. Для этой цели можно использовать конструкции вида  $H[n]$ , присвоение значений которым формирует требуемый массив с именем  $H$ , к элементам которого впоследствии можно обращаться по тем же конструкциям  $H[n]$ . Эти конструкции в определенном смысле аналогичны *индексированным* выражениям в *Maple*. Для получения всех элементов массива  $H$  достаточно воспользоваться простой процедурой **ArrayInd**, исходный код которой с примерами приведены в предыдущем фрагменте. Вызов процедуры **ArrayInd[H]** возвращает список формата {"H[n1]=c1", ..., "H[nk]=ck"}, где  $H$  – имя массива, заданного строчным форматом, который был создан описанным выше способом, и  $c_j$  ( $j=1..k$ ) – значения его элементов, как весьма наглядно иллюстрируют примеры предыдущего фрагмента. Если  $H$  не является *массивом* выше указанного типа, вызов **ArrayInd** возвращает либо список с определением объекта  $H$ , заданного в строчном формате, либо пустой список, т.е. {}, с выводом сообщения.

Тут же вполне уместно отметить одну довольно существенную особенность, которая имеет место в *Mathematica*, а именно. Как видно из нижеследующего фрагмента, при вызове **ArrayInd** на неопределенном символе «W» результат возвращается корректно, однако с выводом сообщения, что данный символ не определен в текущем сеансе, что подтверждает и результат вызова функции **Names**. И только после его вычисления в текущем сеансе, обеспечиваем его ввод в «поле зрения» системы. Этим *Mathematica*, на наш взгляд, достаточно существенно отличается от пакета *Maple* и, прежде всего, при разработке средств *системного* характера, ориентированных на массовое применение.

```
In[2031]:= ArrayInd["W"]
          Definition::notfound: Symbol W not found. >>
Out[2031]= {}
In[2032]:= Names["*"]
Out[2032]= {"d", "d$", "h", "H", "h$", "p", "PacletFind", "PacletInformation", "S", "y"}
In[2033]:= W; ArrayInd["W"]
Out[2033]= {}
In[2034]:= Names["*"]
Out[2034]= {"d", "d$", "h", "H", "h$", "p", "PacletFind", "PacletInformation", "S", "W", "y"}
In[2035]:= Characters1[x_] := Characters[x // ToString]; Characters1[Art24Kr17]
Out[2035]= {"A", "r", "t", "2", "4", "K", "r", "1", "7"}
```

Завершает фрагмент простая, но в целом ряде случаев достаточно полезная функция **Characters1**, обобщающая стандартную функцию **Characters** на аргументы, отличные от строк. Вызов **Characters1[x]** эквивалентен вызову **Characters[x // ToString]**.

В отличие от *Maple*, в системе *Mathematica* средства для работы с индексированными выражениями отсутствуют. Для устранения этого недостатка могут быть предложены достаточно простые процедуры, а именно **IndexedQ**, **Index** и **ArrayInd**, рассмотренные выше. Тогда как здесь представим весьма простую процедуру **IndexQ**, вызов которой **IndexQ[x]** возвращает *True*, если  $x$  – *индексированное* выражение, и *False* в противном случае; при этом, аргумент  $x$  задается в строчном формате, где под *индексированным*

понимается любое выражение в приведенной форме, которое завершается *индексной* скобкой `"]]`. Следующий фрагмент представляет исходные коды *двух* реализаций для процедуры **IndexQ** наряду с наиболее типичными примерами их использования.

```
In[2623]:= IndexQ[x_String] := If[! SuffPref[x, "[", 1] && StringDependQ[x, {"[", ""]}] &&
    EvenQ[StringCount[x, {"[", ""]}] && StringTake[x, {-2, -1}] == "]", True, False]
In[2624]:= Map[IndexQ, {"{a, {b, x, y}, c}[[2, 4]]", "(a + b)", "V[[78]]", "G[[47, 66]] + Sv",
    "V[[a[[b]] + c[[d]]]"][[a + b]]", "Sin[x] + V[[78]]", "G*L[[47]][[66]]", "L[[47]][[66]] + S"}]
Out[2624]= {True, False, True, False, True, True, True, False}
In[2625]:= IndexQ[x_String] := If[StringTake[x, {-2, -1}] != "]", False, True]
In[2626]:= Map[IndexQ, {"{a, {b, x, y}, c}[[2, 4]]", "(a + b)", "V[[70]]", "G[[47, 66]] + Sv",
    "V[[a[[b]] + c[[d]]]"][[a + b]]", "Sin[x] + V[[78]]", "G*L[[47]][[66]]", "L[[47]][[66]] + S"}]
Out[2626]= {True, False, True, False, True, True, True, False}
```

Как следует из приведенного фрагмента, под *индексированными* в контексте функции **IndexQ** понимается достаточно широкий круг выражений, предполагая дальнейшую детализацию данного понятия, что представляется важным для ряда приложений. В частности, на основе этой функции довольно несложно программируется достаточно полезная в целом ряде задач работы с индексированными выражениями процедура с исходным кодом и примерами использования, представленными фрагментом.

```
In[2675]:= Indices[x_ /; IndexQ[x]] := Module[{a = "]", b = "[", h = x, c = {}, g, t, k, y,
    p = StringLength[x]},
    While[p >= 4, For[k = p - 2, k >= 1, k--, c = StringTake[h, {k, p}];
        If[StringCount[c, a] == StringCount[c, b], d = Append[d, {k, c}];
            Break[]]; k = d[[-1]][[1]]; h = StringTake[h, {1, k - 1}];
            If[IndexQ[h], p = StringLength[h]; Continue[], Break[]];
        g = Reverse[Map[{#[[1]], #[[2]], #[[1]] + StringLength[#[[2]]] &, d}]; y = g[[-1]][[2]];
        For[k = Length[g] - 1, k >= 1, k--, If[g[[k]][[3]] == g[[k + 1]][[1]], y = g[[k]][[2]] <> y]; y]
In[2676]:= Indices["(Art[[24]] + Kr[[17]])*a[[a + b]][[b[[k]] + j]][[c, d]]"]
Out[2676]= "[[a + b]][[b[[k]] + j]][[c, d]]"
In[2677]:= Indices["a[[a + b]][[b[[k]] + j]][[c, d]]"]
Out[2677]= "[[a + b]][[b[[k]] + j]][[c, d]]"
```

Вызов процедуры **Indices[x]** возвращает *индексную* составляющую индексированного выражения *x*, заданного в строчном формате, в противном случае вызов возвращается невычисленным, как наглядно иллюстрируют примеры предыдущего фрагмента. В то же время рассмотрение всех функций, поддерживаемых *Mathematica* для работы с массивами, таблицами, векторами и матрицами, нами здесь не обсуждается равно как и множество функций по работе со списочными структурами, как самостоятельными объектами. Заинтересованный читатель отсылается к справочной информации или к соответствующей литературе по *Mathematica*, в частности, [59,60,99,102,106,121-140].



Следующая процедура представляет определенный интерес в работе с выражениями индексированного формата, а именно *Symbol[expression]*. Вызов процедуры **Ind[x]** на данного типа выражениях возвращает список вида *{Symbol, {expression}}*; при этом, в качестве выражения (*expression*) могут выступать и последовательности. В противном случае возвращается упрощенное выражение *x*. Следующий фрагмент представляет исходный код процедуры **Ind** и некоторые типичные примеры ее использования.

```
In[2024]:= Ind[x_] := Module[{a = ToString[InputForm[x]], b},
    b = Flatten[StringPosition[a, {"[", "["}]]; If[b == {} || StringTake[a, -1] != "]", x,
    Quiet[Check[Map[ToExpression, {StringTake[a, {1, b[[1]] - 1}], "{" <>
    StringTake[a, {b[[1]] + 1, b[[-1]] - 1}] <> "}"], x]]]
In[2025]:= Map[Ind, {Sin[x], (a + b)/(c + d), Sin[x] + Cos[x], AG[a + b], SV[x, y, z]}]
Out[2025]= {{Sin, {x}}, (a + b)/(c + d), Cos[x] + Sin[x], {AG, {a + b}}, {SV, {x, y, z}}}
```

На основе предыдущих процедур **ToList** и **Ind** программируется нижеследующая **OP** процедура, чей вызов **OP[x]** возвращает список атомарных элементов, составляющих выражение *x*. Следующий фрагмент представляет исходный код процедуры наряду с некоторыми достаточно типичными примерами ее использования.

```
In[2020]:= OP[expr_] := Module[{a = ToString[InputForm[expr]], b = {}, c, d, k, h},
    If[StringTake[a, {-1, -1}] == "]", a = Flatten[Ind[expr]],
    a = DeleteDuplicates[Quiet[To�ist[expr]]];
    Label[ArtKr]; d = Length[a];
    For[k = 1, k <= Length[a], k++, h = a[[k]]; c = Quiet[To�ist[h]];
    If[MemberQ[DeleteDuplicates[c], $Failed], AppendTo[b, Ind[h]],
    AppendTo[b, c]]; a = DeleteDuplicates[Flatten[b]];
    If[d == Length[a], Sort[a], b = {}; Goto[ArtKr]]]
In[2021]:= OP[(a + b)/(c + g[d])]
Out[2021]= {-1, a, b, c, d, g}
In[2022]:= OP[Cos[x] + Sin[y]*Tan[z]]
Out[2022]= {Cos, Sin, Tan, x, y, z}
In[2023]:= OP[Sqrt[(a + b)/(c + d)] + Sin[x]*Cos[y]]
Out[2023]= {-1, 1/2, a, b, c, Cos, d, Sin, x, y}
In[2024]:= OP[(Log[(a + b)/(c + d)] + Sin[x]*Cos[y])/(G[h, g, t] - w^2)]
Out[2024]= {-1, 2, a, b, c, Cos, d, g, G, h, Log, Sin, t, w, x, y}
In[2025]:= Map[OP, {{Sin[x]}, G[h, g, t], A[m, p]/G[t, q]}]
Out[2025]= {{Sin, x}, {g, G, h, t}, {-1, A, G, m, p, q, t}}
```

В *Mathematica* отсутствует прямой эквивалент встроенной *op*-функции *Maple*, но его можно с точностью до аксиоматики систем определить следующей процедурой, что в целом ряде случаев достаточно удобно при программировании приложений:

```
In[2072]:= Op[x_] := Module[{a, b}, a = {}; If[ListQ[x], a = x,
    Do[a = Insert[a, Part[x][[b]], -1], {b, Length[x]}]; a]
```

```
In[2073]:= Op[Sin[x] + Cos[x]]
Out[2073]= {Cos[x], Sin[x]}
In[2074]:= Op[Sin[x]*Cos[x]]
Out[2074]= {Cos[x], Sin[x]}
In[2075]:= Op[{1, 2, 3, 4, 5, 6, 7, 8, 9}]
Out[2075]= {1, 2, 3, 4, 5, 6, 7, 8, 9}
In[2076]:= Op[Sqrt[a + b]]
Out[2076]= {1 + a, 1/2}
In[2077]:= Op[Sqrt[a + b] + Sin[x] - c/d]
Out[2077]= {Sqrt[1 + a], -(c/d), Sin[x]}
In[2078]:= Op[(x + y*Cos[x])/(y + x*Sin[y])]
Out[2078]= {x + y Cos[x], 1/(y + x Sin[y])}
In[2079]:= Map[Op, {Sin[x], Cos[a + b], 1/(a + b)}]
Out[2079]= {{x}, {a + b}, {a + b, -1}}
In[2080]:= Op[f[g[a, b], h[c, d, e, j, k, l]]]
Out[2080]= {g[a, b], h[c, d, e, j, k, l]}
```

Несложно убедиться, что полученные результаты вызовов **Op**-процедуры идентичны с аналогичными вызовами *op*-функции в среде *Maple* с учетом того, что *Mathematica* не поддерживает структуру типа последовательность, которая заменяется списком.

В целом ряде приложений несомненный интерес представляет определенный аналог *Maple*-процедуры *whattype(e)*, возвращающей тип выражения *e*, который может быть одним из базовых *Maple*-типов данных, как хорошо иллюстрируют примеры из [99]. В качестве подобного аналога в среде *Mathematica* выступает *одноименная* процедура, чей вызов **WhatType[x]** возвращает тип объекта *x* одного из базовых типов {"Module", "DynamicModule", "Block", "Complex", "Integer", "Rational", "Real", "Times", "Plus", "List", "Power", "And", "Rule", "Condition", "StringJoin", "UndirectedEdge", "Alternatives", ...}.

```
In[2069]:= WhatType[x_;/; StringQ[x]] := Module[{a = Quiet[Head[ToExpression[x]]],
      b = t, d, c = $Packages}, If[a === Symbol, Clear[t]; d = Context[x];
      If[d == "Global`,", d = Quiet[ProcFuncBIQ[x, t]];
      If[d === True, Return[{t, t = b}][[1]], Return[{"Undefined", t = b}][[1]],
      If[d == "System`", Return[{d, t = b}][[1]], Null]],
      Return[{ToString[a], t = b}][[1]]]; If[Quiet[ProcFuncBIQ[x, t]],
      If[MemberQ[{"Module", "DynamicModule", "Block"}, t], Return[{t, t = b}][[1]],
      t = b; ToString[Quiet[Head[ToExpression[x]]]], t = b; "Undefined"]]
```

```
In[2070]:= t = 460; x = 78; y := 42.47; z := a + b; Map[WhatType, {"Kr", "x", "y", "z", "ProcQ",
      "Sin", "F[r]", "WhatType"}]
Out[2070]= {"Undefined", "Integer", "Real", "Plus", "Module", "System`", "F", "Module"}
In[2071]:= Map[WhatType, {"a^b", "a**b", "3 + 5*I", "{42, 47}", "a&&b", "a -> b"}]
Out[2071]= {"Power", "NonCommutativeMultiply", "Complex", "List", "And", "Rule"}
In[2072]:= Map[WhatType, {"a_/; b", "a <> b", "a <-> b", "a | b", "a`b"}]
Out[2072]= {"Condition", "StringJoin", "UndirectedEdge", "Alternatives", "Undefined"}
```

Предыдущий фрагмент представляет исходный код процедуры наряду с примерами по ее применению для выявления типов различных объектов. Однако, тут же следует отметить, что процедура **WhatType** не поддерживает исчерпывающего тестирования типов, однако на ее основе несложно расширять класс проверяемых типов данных. В данной связи такая задача предлагается продвинутому читателю в качестве довольно полезного упражнения по развитию навыков программирования в *Mathematica*.

Функция **ReplaceAll (/.)** системы *Mathematica* имеет весьма существенные ограничения по отношению к замене подвыражений относительно уже весьма простых выражений, как это иллюстрируют примеры нижеследующего фрагмента. В качестве замены для данной функции можно предложить процедуру **Subs[x,y,z]**, которая функционально эквивалентна стандартной функции **ReplaceAll (/.)**, но избавлена от ряда недостатков второй. Вызов процедуры **Subs[x,y,z]** возвращает результат подстановок в выражение *x* выражений *z* вместо вхождений в него подвыражений *y*. При этом, если в качестве *x* выступает произвольное корректное выражение, допускаемое *Math*-языком пакета, то в качестве второго и третьего аргументов, определяющих подстановки формата *y* → *z*, допустима одинарная подстановка или их множество, кодируемое в виде *y* ≡ {*y*<sub>1</sub>, *y*<sub>2</sub>, ..., *y*<sub>*n*</sub>} и *z* ≡ {*z*<sub>1</sub>, *z*<sub>2</sub>, ..., *z*<sub>*n*</sub>}, определяя множество подстановок {*y*<sub>1</sub> → *z*<sub>1</sub>, *y*<sub>2</sub> → *z*<sub>2</sub>, ..., *y*<sub>*n*</sub> → *z*<sub>*n*</sub>}, выполняемых последовательно в порядке кодирования при вызове процедуры **Subs**. Следующий фрагмент представляет как исходный код процедуры **Subs**, так и целый ряд примеров ее использования на тех выражениях и с теми типами подстановок, где она превосходит стандартную функцию **ReplaceAll (/.)** системы *Mathematica*. Данные примеры достаточно наглядно иллюстрируют преимущества процедуры **Subs**.

```
In[2068]:= Subs[x_, y_, z_] := Module[{d, k = 2, subs}, subs[m_, n_, p_] :=
Module[{a, b, c, h, t}, If[! HowAct[n], m /. n -> p, {a, b, c, h} =
First[{Map[ToString, Map[InputForm, {m, n, p, 1/n}]]];
t = Simplify[ToExpression[StringReplace[StringReplace[
a, b -> "(" <> c <> ")", h -> "1/" <> "(" <> c <> ")]]];
If[t === m, m /. n -> p, t]];
If[! ListQ[y] && ! ListQ[z], subs[x, y, z], If[ListQ[y] && ListQ[z] &&
Length[y] == Length[z], d = subs[x, y[[1]], z[[1]];
For[k, k <= Length[y], k++, d = subs[d, y[[k]], z[[k]]]; d, Defer[Subs[x, y, z]]]]]

In[2069]:= (c + x^2)/x^2 /. x^2 -> a
Out[2069]= (a + c)/x^2
In[2070]:= Subs[(c + x^2)/x^2, x^2, a]
Out[2070]= (a + c)/a
In[2071]:= (c + b^2)/x^2 /. x^2 -> Sqrt[z]
Out[2071]= (b^2 + c)/x^2
In[2072]:= Subs[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[2072]= (b^2 + c)/Sqrt[z]
In[2074]:= (a + x^2)/(b + a/x^2) /. x^2 -> Sqrt[a + b]
Out[2074]= (a + Sqrt[a + b])/(b + a/x^2)
```

```

In[2075]:= Subs[(a + x^2)/(b + a/x^2), x^2, Sqrt[a + b]]
Out[2075]= (a + Sqrt[a + b])/(b + a/Sqrt[a + b])
In[2076]:= (a + x^2)/(b + 1/x^2) /. x^2 -> Sqrt[a + b]
Out[2076]= (a + Sqrt[a + b])/(b + 1/x^2)
In[2077]:= Subs[(a + x^2)/(b + 1/x^2), x^2, Sqrt[a + b]]
Out[2077]= (a + Sqrt[a + b])/(b + 1/Sqrt[a + b])
In[2078]:= Replace[1/x^2 + 1/y^3, {{x^2 -> a + b}, {y^3 -> c + d}}]
Out[2078]= {1/x^2 + 1/y^3, 1/x^2 + 1/y^3}
In[2079]:= Subs[1/x^2 + 1/y^3, {x^2, y^3}, {a + b, c + d}]
Out[2079]= 1/(a + b) + 1/(c + d)
In[2080]:= Replace[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {{x^2 -> a + b}, {y^3 -> c + d}}]
Out[2080]= {Sqrt[Cos[1/y^3] + Sin[1/x^2]], Sqrt[Cos[1/y^3] + Sin[1/x^2]]}
In[2081]:= Subs[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {x^2, y^3}, {a + b, c + d}]
Out[2081]= Sqrt[Cos[1/(c + d)] + Sin[1/(a + b)]]
In[2082]:= With[{x = a + c, y = b}, Module[{}, x^2 + y]
Out[2082]= b + (a + c)^2
In[2083]:= With[{x^2 = a + c, y = b}, Module[{}, x^2 + y]
      With::lvset: Local variable specification {x^2=a+c, y=b} contains x^2=a+c, ... >>
Out[2083]= With[{x^2 = a + c, y = b}, Module[{}, x^2 + y]
In[2084]:= Subs[Module[{}, x^2 + y], {x, y}, {a + c, b}]
Out[2084]= b + (a + c)^2
In[2085]:= Subs[Module[{}, x^2 + y], {x^2, y}, {a + c, b}]
Out[2085]= a + b + c
In[2086]:= Replace[(a + x^2/y^3)/(b + y^3/x^2), {{y^3 -> m}, {x^2 -> n}}]
Out[2086]= {(a + x^2/y^3)/(b + y^3/x^2), (a + x^2/y^3)/(b + y^3/x^2)}
In[2087]:= Subs[(a + x^2/y^3)/(b + y^3/x^2), {y^3, x^2}, {m, n}]
Out[2087]= n (a m + n)/(m (m + b n))
In[2088]:= Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
      Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]
In[2089]:= Df[(a + x^2)/(b + a/x^2), x^2]
Out[2089]= (a^2 + 2 a x^2 + b x^4)/(a + b x^2)^2
In[2090]:= Df[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]
Out[2090]= (-4 x - 2 Sqrt[y] + y)/((2 + Sqrt[y])^3 y^(3/2))
In[2081]:= D[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]
      General::ivar: Sqrt[y] is not a valid variable. >>
Out[2081]=  $\partial_y ((x + \sqrt{y})/(y + 2 \sqrt{y})^2)$ 
In[2082]:= Df[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[2082]= ((d + 2 Sqrt[x])/Sqrt[a + Sqrt[x]] - 8 (Sqrt[a + Sqrt[x]] + x))/(2 (d + 2 Sqrt[x])^3)
In[2083]:= Df[(x + Sqrt[x + b])/(d + 2*Sqrt[x + b])^2, Sqrt[x + b]]
Out[2083]= (d - 2 (2 x + Sqrt[b + x]))/(d + 2 Sqrt[b + x])^3

```

```

In[2084]:= ReplaceAll1[x_, y_, z_] := Module[{a, b, c}, If[! HowAct[y], x/. y -> z,
  c = If[MemberQ[{Plus, Times, Power}, Head[z]], "(" <> ToString[InputForm[z]] <> ")",
    ToString[z]]; {a, b} = Map[ToString, Map[InputForm, {x, y}]];
  If[StringLength[b] == 1, ReplaceAll[x, y -> z], ToExpression[StringReplace[a, b -> c]]]]]

In[2085]:= {ReplaceAll[c/x^2 + x^2, x^2 -> t], ReplaceAll[(1 + c/x^2)/(b + x^2), x^2 -> t]}
Out[2085]= {t + c/x^2, (1 + c/x^2)/(b + t)}

In[2086]:= {ReplaceAll1[c/x^2 + x^2, x^2, a+b], ReplaceAll1[(1 + c/x^2)/(b + x^2), x^2, c+d]}
Out[2086]= {a + b + c/(a + b), (1 + c/(c + d))/(b + c + d)}

In[2087]:= Df1[x_, y_] := Module[{a, b, c = "$Sart23$Kr16$$"}, If[! HowAct[y], D[x, y],
  {a, b} = Map[ToString, Map[InputForm, {x, y}]];
  Simplify[ToExpression[StringReplace[ToString[InputForm[
    D[ToExpression[StringReplace[a, b -> c]], ToExpression[c]]]], c -> b]]]]]

In[2088]:= Df2[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
  Simplify[ReplaceAll1[D[ReplaceAll1[x, y, a], a], a, y]]]]

In[2089]:= Df1[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[2089]= ((d + 2 Sqrt[x])/Sqrt[a + Sqrt[x]] - 8 (Sqrt[a + Sqrt[x]] + x))/(2 (d + 2 Sqrt[x])^3)
In[2090]:= Df2[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[2090]= ((d + 2 Sqrt[x])/Sqrt[a + Sqrt[x]] - 8 (Sqrt[a + Sqrt[x]] + x))/(2 (d + 2 Sqrt[x])^3)
In[2091]:= Df2[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[2091]= -(((a - c) x^2)/(1 + c)^2)
In[2092]:= Df1[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[2092]= -(((a - c) x^2)/(1 + c)^2)
In[2093]:= Df[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[2093]= -((2 (a - c) x^6)/(1 + c x^4)^2)
In[2094]:= Df2[(a + b)/(Sin[x] + Cos[x]), Sin[x] + Cos[x]]
Out[2094]= -((a + b)/(Cos[x] + Sin[x])^2)
In[2095]:= Df2[Cos[x]/(Sin[x] + Cos[x]), Cos[x]]
Out[2095]= Sin[x]/(Cos[x] + Sin[x])^2

```

В качестве простого примера применения нашей процедуры **Subs** представлена одна версия процедуры **Df[W, X]**, обеспечивающая дифференцирование выражения **W** по любому его подвыражению **X** и достаточно существенно расширяющая стандартную функцию **D** пакета; примеры иллюстрируют некоторые возможности процедуры **Df**. В завершение фрагмента представлена процедура **ReplaceAll1[x, y, z]**, функционально эквивалентная стандартной функции **ReplaceAll**, но избавленная от ряда недостатков *второй*. Затем на основе процедур **ReplaceAll1** и **StringReplace** представлены варианты процедуры **Df**, а именно процедуры **Df1** и **Df2**, использующие ряд полезных приемов программирования. Вместе с тем, они в ряде случаев оказываются более полезны, чем процедура **Df**, что достаточно наглядно иллюстрируют приведенные выше примеры. При этом, процедуры **Df**, **Df1** и **Df2** достаточно существенно расширяют стандартную

функцию **D**. Данный фрагмент представляет исходные коды отмеченных процедур и некоторые примеры их применения там, где они превосходят стандартные функции **D**, **ReplaceAll**, **Rule** и **With** программной среды системы *Mathematica*.

Вполне естественным представляется нам получение подобного расширения также и для стандартной функции **Integrate** системы *Mathematica*, которая имеет достаточно существенные ограничения по использованию произвольных выражений в качестве переменной интегрирования. С данной целью были предложены два варианта такого расширения в виде несложных процедур **Int** и **Int1**, базирующихся на рассмотренной процедуре **Subs**, чьи исходные коды наряду с примерами их типичного применения представляет нижеследующий фрагмент, а именно:

```
In[2041]:= Int[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y],
                                     Simplify[Subs[Integrate[Subs[x, y, a], a], a, y]]]]
In[2042]:= Int1[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y],
                                     Simplify[ReplaceAll1[Integrate[ReplaceAll1[x, y, a], a], a, y]]]]
In[2043]:= {Int[Sin[a + 1/x^2] + c/x^2, 1/x^2], Int1[Sin[a + 1/x^2] + c/x^2, 1/x^2]}
Out[2043]= {-Cos[a + 1/x^2] + c Log[1/x^2], c/x^4 - Cos[a + 1/x^2]}
In[2044]:= {Int[Sin[n/x^2] + m/x^2, x^2], Int1[Sin[n/x^2] + m/x^2, x^2]}
Out[2044]= {-n CosIntegral[n/x^2] + m Log[x^2] + x^2 Sin[n/x^2],
             -n CosIntegral[n/x^2] + m Log[x^2] + x^2 Sin[n/x^2]}
In[2045]:= {Int[(a*x^2+b/x^2)/(c*x^2+d/x^2), x^2], Int1[(a*x^2+b/x^2)/(c*x^2+d/x^2), x^2]}
Out[2045]= {(a x^2)/c + ((b c - a d) ArcTan[(Sqrt[c] x^2)/Sqrt[d]])/(c^(3/2) Sqrt[d]),
             (a x^2)/c + ((b c - a d) ArcTan[(Sqrt[c] x^2)/Sqrt[d]])/(c^(3/2) Sqrt[d])}
In[2046]:= Integrate[(a*x^2 + b/x^2)/(c*x^2 + d/x^2), x^2]
Integrate::ilim: Invalid integration variable or limit(s) in x^2. >>
Out[2046]= Integrate[(b/x^2 + a*x^2)/(d/x^2 + c*x^2), x^2]
In[2047]:= Integrate[Sin[a + 1/x^2] + c/x^2, 1/x^2]
Integrate::ilim: Invalid integration variable or limit(s) in 1/x^2. >>
Out[2047]= Integrate[c/x^2 + Sin[a + 1/x^2], 1/x^2]
In[2048]:= Quiet[{Integrate[(a + b)/x^2, x^2], Int[(a + b)/x^2, x^2]}]
Out[2048]= {\[Integral](a + b)/x^2 \[DifferentialD]x^2, (a + b) Log[x^2]}
```

Таким образом, на примерах определенного типа имеют преимущества те либо иные наши процедуры, вместе с тем полностью *идентичные* на тех выражениях, на которых соответствующие стандартные функции **ReplaceAll**, **D** и **Integrate** возвращают вполне корректные результаты. При этом, представленные варианты реализации процедур **Df**, **Df1**, **Df2**, **Int**, **Int1**, **ReplaceAll1** и **Subs** иллюстрируют различные приемы, полезные в ряде задач практического программирования в среде языка системы *Mathematica* и, в первую очередь, задач системного характера.

Представленные варианты реализации процедур **Df**, **Df1**, **Df2**, **Int**, **Int1**, **ReplaceAll1** и **Subs** иллюстрируют различные приемы, полезные в ряде задач программирования в среде языка системы *Mathematica* и, в первую очередь, задач системного характера. В следующем фрагменте приведены средства, имеющие как самостоятельное значение,

так и ряд довольно полезных приемов в программировании приложений различного типа. Предваряют данный фрагмент две простые и полезные функции, используемые в следующих процедурах фрагмента. Вызов функции **ListRulesQ[x]** возвращает **True**, если является списком правил формата  $a \rightarrow b$ , и **False** в противном случае. Тогда как функция **Map17** обобщает стандартную функцию **Map** на случай списка правил как ее второго фактического аргумента. Вызов функции **Map17[F, {{a  $\rightarrow$  b, c  $\rightarrow$  d, ...}}**, где **F** – символ, возвращает результат нижеследующего формата, а именно:

$$\{F[a] \rightarrow F[b], F[c] \rightarrow F[d], \dots\}$$

не требуя каких-либо дополнительных пояснений ввиду его прозрачности. Тогда как вызов процедуры **Diff[x, y, z, ...]** возвращает результат дифференцирования выражения  $x$  по обобщенным переменным  $\{y, z, \dots\}$ , в качестве которых допустимы произвольные выражения. Результат возвращается в упрощенном виде на основе функции **Simplify**. Вызов процедуры **Integral[x, y, z, ...]** возвращает результат интегрирования выражения  $x$  по обобщенным переменным  $\{y, z, \dots\}$ , в качестве которых допустимы произвольные выражения. Результат возвращается в упрощенном виде на основе функции **Simplify**. Фрагмент представляет исходные коды указанных средств с примерами применения.

```
In[21]:= ListRulesQ[x_List] := DeleteDuplicates[Map[RuleQ[#] &, Flatten[x]]] === {True}
In[22]:= ListRulesQ[{a -> b, c -> d, t -> g, w -> v, h}]
Out[22]= False
In[23]:= ListRulesQ[{a -> b, c -> d, t -> g, w -> v, h -> 78}]
Out[23]= True
In[24]:= Map17[x_, y_ /; RuleQ[y] | | ListRulesQ[y]] := If[RuleQ[y], Map[x, y],
                                                    Map[Map[x, #] &, y]]
In[25]:= Map17[F, a -> b]
Out[25]= F[a] -> F[b]
In[26]:= Map17[F, {a -> b, c -> d, t -> g, w -> v, h -> 78}]
Out[26]= {F[a] -> F[b], F[c] -> F[d], F[t] -> F[g], F[w] -> F[v], F[h] -> F[78]}
In[32]:= Diff[x_, y_] := Module[{a = Map[ToString, Map[InputForm, {y}]], c = {}, d = {},
                                b = Length[{y}], t = {}, k = 1, h = x, n = g}, Clear[g];
    While[k <= b, AppendTo[c, Unique[g]]; AppendTo[d, ToString[c[[k]]]];
                AppendTo[t, a[[k]] -> d[[k]]];
        h = ToExpression[StringReplace[ToString[h // InputForm], t[[k]]]];
                h = D[h, c[[k]]];
        h = ReplaceAll[h, Map[ToExpression, Part[t[[k]], 2] -> Part[t[[k]], 1]]]; k++;
        g = n; Map[Clear, c]; h]
In[33]:= Diff[Sin[x^2]*Cos[1/b^3], 1/b^3, x^2]
Out[33]= -Cos[x^2]*Sin[1/b^3]
In[34]:= Diff[(a + b)/(c + d), a + b, c + d]
Out[34]= -(1/(c + d)^2)
```

```

In[35]:= Diff[(a + b) + m/(c + d), a + b, 1/(c + d)]
Out[35]= 0
In[63]:= Diff[(1/b + Cos[a + Sqrt[c + d]])/(Tan[1/b] - 1/c^2), 1/b, 1/c^2, Sqrt[c + d]]
Out[63]= (2*Sec[1/b]^2*Sin[a + Sqrt[c + d]])/(- (1/c^2) + Tan[1/b])^3
In[81]:= Diff[1/Sqrt[a + b]*(a + b) + Tan[Sqrt[a + b] + c], Sqrt[a + b], a + b]
Out[81]= (Sec[Sqrt[a + b] + c]^2*Tan[Sqrt[a + b] + c])/Sqrt[a + b]

In[1257]:= Integral[x_, y_] := Module[{a = Map[ToString, Map[InputForm, {y}]], c = {},
    b = Length[{y}], d = {}, t = {}, k = 1, h = x, n = g}, Clear[g];
    While[k <= b, AppendTo[c, Unique[g]]; AppendTo[d, ToString[c[[k]]]];
        AppendTo[t, a[[k]] -> d[[k]]];
        h = ToExpression[StringReplace[ToString[h // InputForm], t[[k]]]];
        h = Integrate[h, c[[k]]];
        h = ReplaceAll[h, Map[ToExpression, Part[t[[k]], 2] -> Part[t[[k]], 1]]]; k++];
    g = n; Map[Clear, c]; h]

In[1258]:= g = 78; Integral[Sin[x^2] + Cos[1/b^3], 1/b^3, x^2]
Out[1258]= -(Cos[x^2]/b^3) + x^2*Sin[1/b^3]
In[1259]:= Integral[Sin[x] + Cos[x], x]
Out[1259]= -Cos[x] + Sin[x]
In[1260]:= Integral[(Sin[x] + Cos[y])*z, x, y, z]
Out[1260]= -(1/2)*z^2*(y*Cos[x] - x*Sin[y])
In[1261]:= Integral[(a + b)/(c + d), a + b, c + d]
Out[1261]= (1/2)*(a + b)^2*Log[c + d]
In[1262]:= Integral[(a + b) + m/(c + d), a + b, 1/(c + d)]
Out[1262]= ((a + b)*(a*(c + d) + b*(c + d) + 2*m))/(2*(c + d)^2)
In[1263]:= Integral[(a + b)/(c + d), a + b, c + d, c + d]
Out[1263]= (1/2)*(a + b)^2*(c + d)*(-1 + Log[c + d])
In[1264]:= Integral[(a + b)/(c + d) + Log[c + d], a + b, c + d, c + d]
Out[1264]= (1/4)*(a + b)*(c + d)*(-2*a - 2*b - 3*c - 3*d + 2*(a + b + c + d)*Log[c + d])

```

При этом, процедуры **Diff** и **Integral** имеют определенные ограничения, требующие при использовании соответствующей осмотрительности; некоторое представление о таких ограничениях иллюстрирует следующий весьма простой пример, а именно:

```

In[3322]:= Diff[(a + b*m)/(c + d*n), a + b, c + d]
Out[3322]= -(m/((c + d)^2*n))
In[3323]:= Integral[(a + b*m)/(c + d*n), a + b, c + d]
Out[3323]= ((a + b)^2*m*Log[c + d])/(2*n)

```

С целью устранения данных недостатков были созданы две модификации функций **Replace** и **ReplaceAll** в лице процедур **Replace4** и **ReplaceAll2** соответственно. Данные процедуры не только расширяют стандартные средства, но и позволяют кодировать предыдущие две процедуры **Diff** и **Integral** с более широким диапазоном корректных



приложений в контексте использования обобщенных переменных интегрирования и дифференцирования. Вызов **Replace4**[ $x, a \rightarrow b$ ] возвращает результат применения к  $x$  выражению подстановки  $a \rightarrow b$ , когда в качестве ее левой части  $a$  допускается любое выражение. При отсутствии в выражении  $x$  вхождений подвыражения  $a$  возвращается исходное выражение  $x$ . В отличие от предыдущей, вызов **ReplaceAll2**[ $x, r$ ] возвращает результат применения к  $x$  выражению правила  $r$  или последовательного применения правил из списка  $r$ ; в качестве левых частей правил допускаются любые выражения. В следующем фрагменте представлены исходные коды процедур **Replace4**, **ReplaceAll2** с наиболее типичными примерами их использования.

```
In[4]:= Replace4[x_, r_ /; RuleQ[r]] := Module[{a, b, c, h}, {a, b} = {ToString[x // InputForm],
Map[ToString, Map[InputForm, r]]};
c = StringPosition[a, Part[b, 1]];
If[c == {}, x, If[Head[Part[r, 1]] === Plus,
h = Map[If[#[[1]] === 1 || MemberQ[{" ", "(", "[", "{"},
StringTake[a, {#[[1]] - 1, #[[1]] - 1}]]] && #[[2]] === StringLength[a] ||
MemberQ[{" ", ")", "]", "}", ",", "."], StringTake[a, {#[[2]] + 1, #[[2]] + 1}]], #] &, c],
h = Map[If[#[[1]] === 1 ||
! Quiet[SymbolQ[StringTake[a, {#[[1]] - 1, #[[1]] - 1}]]] &&
(#[[2]] === StringLength[a] ||
! Quiet[SymbolQ[StringTake[a, {#[[2]] + 1, #[[2]] + 1}]]]), #] &, c]];
h = Select[h, ! SameQ[#, Null] &];
ToExpression[StringReplacePart[a, "(" <> Part[b, 2] <> ")", h]]]

In[5]:= Replace4[(c + d*x)/(c + d + x), c + d -> a + b]
Out[5]= (c + d*x)/(a + b + x)
In[6]:= Replace[Sqrt[a + b*x^2*d + c], x^2 -> a + b]
Out[6]= Sqrt[a + c + b*d*x^2]
In[7]:= Replace4[Sqrt[a + b*x^2*d + c], x^2 -> a + b]
Out[7]= Sqrt[a + c + b*(a + b)*d]

In[8]:= ReplaceAll2[x_, r_ /; RuleQ[r] || ListRulesQ[r]] := Module[{a = x, k = 1},
If[RuleQ[r], Replace4[x, r], While[k <= Length[r], a = Replace4[a, r[[k]]]; k++]; a]

In[9]:= ReplaceAll[Sqrt[a + b*x^2*d + c], {x^2 -> a + b, a + c -> avz}]
Out[9]= Sqrt[avz + b*d*x^2]
In[10]:= ReplaceAll2[Sqrt[a + b*x^2*d + c], {x^2 -> a + b, a + c -> avz}]
Out[10]= Sqrt[avz + b*(a + b)*d]
In[11]:= ReplaceAll2[x*y*z, {x -> 42, y -> 78, z -> 460}]
Out[11]= 1506960
In[12]:= ReplaceAll2[Sin[a + b*x^2*d + c*x^2], x^2 -> a + b]
Out[12]= Sin[a + (a + b)*c + b*(a + b)*d]
```

```

In[34]:= Diff[x_, y_] := Module[{a = x, a1, a2, a3, b = Length[{y}], c = {}, d, k = 1, n = g},
    Clear[g]; While[k <= b, d = {y}[[k]]; AppendTo[c, Unique[g]];
    a1 = Replace4[a, d -> c[[k]]]; a2 = D[a1, c[[k]]];
    a3 = Replace4[a2, c[[k]] -> d]; a = a3; k++; g = n; Simplify[a3]]

In[35]:= Diff[(a + b)/(c + d + x), a + b, c + d]
Out[35]= -(1/(c + d + x)^2)
In[36]:= Diff[(a + b*m)/(c + d*n), a + b, c + d]
Out[36]= 0

In[37]:= Integral[x_, y_] := Module[{a = x, a1, a2, a3, b = Length[{y}], c = {}, d, k = 1, n = g},
    Clear[g]; While[k <= b, d = {y}[[k]]; AppendTo[c, Unique[g]];
    a1 = Replace4[a, d -> c[[k]]]; a2 = Integrate[a1, c[[k]]];
    a3 = Replace4[a2, c[[k]] -> d]; a = a3; k++; g = n; Simplify[a3]]

In[38]:= Integral[(a + b*m)/(c + d*n), a + b, c + d]
Out[38]= ((a + b)*(c + d)*(a + b*m))/(c + d*n)
In[39]:= Integral[Sqrt[a + c + b*(a + b)*d], a + c, a + b]
Out[39]= (2/3)*(a + b)*(a + c + a*b*d + b^2*d)^(3/2)
In[40]:= Integral[Sqrt[a + c + h*g + b*d], c + h, b*d]
Out[40]= (2/3)*(c + h)*(a + c + b*d + g*h)^(3/2)
In[41]:= Integral[(a + c + h*g + b*d)/(c + h), c + h, b*d]
Out[41]= (1/2)*b*d*(2*a + 2*c + b*d + 2*g*h)*Log[c + h]
In[42]:= Integral[(c + h*m)/(c + h), c + h, m]
Out[42]= (c + h*m)^2/(2*(c + h))

```

На основе процедуры **Replace4** возможно расширить процедуры **Diff**, **Integral**. Вызов процедуры **Diff**[*x*, *y*, *z*, ...] возвращает результат дифференцирования выражения *x* по обобщенным переменным {*y*, *z*, *h*, ...}, в качестве которых допускаются произвольные выражения. Результат возвращается в упрощенном виде на основе функции **Simplify**. Вызов процедуры **Integral**[*x*, *y*, *z*, ...] возвращает результат интегрирования выражения *x* по обобщенным переменным {*y*, *z*, ...}, в качестве которых допустимы произвольные выражения. Результат возвращается в упрощенном виде на основе функции **Simplify**. Фрагмент представляет исходные коды указанных средств с примерами применения. Представленные выше средства достаточно полезны во многих случаях манипуляций с алгебраическими выражениями, которые базируются на системе правил, включая их символьные дифференцирование и интегрирование по обобщенным переменным, в качестве которых могут выступать произвольные алгебраические выражения.

Вызов простой функции **GV**[] возвращает список специальных символов и глобальных переменных текущего релиза, инициализированных как самой системой, так и пакетами пользователя, загруженными в текущий сеанс; в частности, в используемом релизе **9.0** их общее количество составляет **175**. Следующий фрагмент представляет исходный код функции **GV** наряду с некоторыми примерами ее использования, а именно:

```

In[2783]:= GV[] := Select[Names["*"], StringTake[#, 1] == "$" &]
In[2784]:= GV[]
Out[2784]= {"$Aborted", "$ActivationGroupID", "$ActivationKey", "$AddOnsDirectory",
"$ActivationUserRegistered", ..., "$UserName", "$Version", "$VersionNumber"}
In[2785]:= Length[%]
Out[2785]= 175
In[2786]:= Length[Select[GV[], Attributes[#] != {} &]]
Out[2786]= 84
In[2787]:= Length[Select[GV[], Attributes[#] == {Protected} &]]
Out[2787]= 30
In[2788]:= DeleteDuplicates[Select[Map[Attributes, Select[GV[], Attributes[#] != {} &]],
# != {Protected} &]]
Out[2788]= {{Locked, Protected}, {Protected, ReadProtected}, {HoldAll, Protected}, {Locked,
Protected, ReadProtected}}
    
```

Так, последние 3 примера фрагмента констатируют, из 175 глобальных переменных и специальных символов системы и пользовательских лишь 84 наделены специальными атрибутами; в частности, 30 наделены лишь *Protected*-атрибутом. Тогда как остальные приходятся на три дополнительных типа атрибутов {**Locked**, **ReadProtected**, **HoldAll**}, представленных в последнем примере предыдущего фрагмента.

Следующая процедура **SEQ** служит аналогом одноименной встроенной функции *seq* системы *Maple*, генерирующая последовательности значений. Следующий фрагмент представляет исходный код процедуры **SEQ** наряду с примерами ее использования.

```

In[2334]:= SEQ[x_, y_ /; SymbolQ[y], z_ /; Head[z] == Span] := Module[{a = ToString[z],
b = {}, c, d = ToString[y], p}, c = ToExpression[StringSplit[a, " ;; "]];
If[DeleteDuplicates[Map[NumberQ, c]] != {True} | |
DeleteDuplicates[Map[Positive, c]] != {True}, Return[Defer[Seq[x, y, z]]],
If[Length[c] > 2 && c[[3]] == 0, Return[Defer[Seq[x, y, z]]],
If[c[[1]] <= c[[2]], p = 1, p = 2]]; For[y = c[[1]], If[p == 1, y <= c[[2]],
y >= c[[2]] - If[p == 1 && Length[c] == 2 | | p == 2 && Length[c] == 2, 0,
c[[3]] - 1]], If[Length[c] == 2, If[p == 1, y++, y--],
If[p == 1, y += c[[3]], y -= c[[3]]], b = Append[b, x]];
{ToExpression["Clear[" <> d <> "]", b][[2]]]

In[2335]:= SEQ[F[k], k, 17 ;; 24]
Out[2335]= {F[17], F[18], F[19], F[20], F[21], F[22], F[23], F[24]}
In[2336]:= SEQ[F[t], t, 1 ;; 75 ;; 6]
Out[2336]= {F[1], F[7], F[13], F[19], F[25], F[31], F[37], F[43], F[49], F[55], F[61], F[67], F[73]}
In[2337]:= SEQ[F[t], t, 100 ;; 90]
Out[2337]= {F[100], F[99], F[98], F[97], F[96], F[95], F[94], F[93], F[92], F[91], F[90]}
In[2338]:= SEQ[F[t], t, 42.71 ;; 100 ;; 6.47]
Out[2338]= {F[42.71], F[49.18], F[55.65], F[65.12], F[68.59], F[75.06], F[81.531], F[88.], F[94.47]}
    
```

```
In[2339]:= {SEQ[F[k], k, 42 ;; 71 ;; 0], SEQ[F[k], k, 42 ;; 78 ;; 6]}
Out[2339]= {SEQ[F[k], k, 42 ;; 71 ;; 0], {F[42], F[48], F[54], F[60], F[66], F[72], F[78]}}
In[2340]:= SEQ[F[k], k, 42 ;; 71 ;; -6]
Out[2340]= SEQ[F[k], k, 42 ;; 71 ;; -6]
```

Вызов процедуры **SEQ**[*x*, *y*, *z*] возвращает список значений *x*[*y*], где *y* изменяется или в пределах *z*=*m*;;*n*, или в пределах *z*=*m*;;*n*;;*p* с шагом *p*; при этом, значения {*m*,*n*,*p*} могут принимать лишь положительные числовые значения; при *m* <= *n* значение *p* считается положительным, иначе отрицательным. Из примеров предыдущего фрагмента очень наглядно просматривается принцип формирования списка значений в зависимости от формата 3-го аргумента. При нулевом либо отрицательном значении 3-го аргумента вызов процедуры **SEQ**[*x*, *y*, *z*] возвращается невычисленным, как достаточно наглядно иллюстрируют последние примеры предыдущего фрагмента.

Несложная функция **AtomicQ**[*x*] возвращает **True**, если *x* – выражение, не делимое на подвыражения, и **False** иначе; тогда как функция **StrExprQ**[*x*] возвращает **True**, если *x* – строка, содержащая корректное выражение, и **False** в противном случае. Следующий фрагмент представляет исходные коды указанных средств с примерами применения.

```
In[2625]:= AtomicQ[x_] := Module[{F}, If[Map[F, x] === x, True, False]]
In[2626]:= Map[AtomicQ, {Sin, a + b, 78, Sqrt[c + d], a/b, a^x}]
Out[2626]= {True, False, True, False, False, False}
In[2629]:= StrExprQ[x_String] := Module[{a = True}, a && Quiet[Check[ToExpression[x],
                                                                    a = False]]; a]
In[2630]:= ExpressionQ[x_String] := If[Quiet[Check[ToExpression[x], $Failed]] ===
                                                                    $Failed, False, True]
In[2631]:= Map[ExpressionQ, {"/a/b", "a^^b", "a (a + b)", "Avz=78=460", "(a + b/", "a++"}]
Out[2631]= {False, False, True, False, False, False}
In[2632]:= Map[StrExprQ, {"/a/b", "a^^b", "a (a + b)", "Avz=78=460", "(a + b/", "a++"}]
Out[2632]= {False, False, True, False, False, False}
```

Настоящий фрагмент завершает функция **ExpressionQ**, представляющая собой иную реализацию **StrExprQ**; вызов **ExpressionQ**[*x*] возвращает **True**, если *x* является строкой, содержащей корректное выражение, и **False** в противном случае. В качестве развития предыдущих средств служит процедура, чей вызов **ExprsInStrQ**[*x*, *y*] возвращает **True**, если строка *x* содержит корректные выражения, и **False** в противном случае. Тогда как через второй необязательный аргумент *y* – неопределенную переменную – возвращается список выражений, находящихся в *x*. Следующий фрагмент представляет исходный код процедуры **ExprsInStrQ** наряду с типичными примерами ее использования.

```
In[2160]:= ExprsInStrQ[x_ /; StringQ[x], y___] := Module[{a = {}, b = StringLength[x], c = 1,
                                                                    d, k = 1, j},
For[k = c, k <= b, k++, For[j = k, j <= b, j++, d = StringTake[x, {k, j}];
If[! SymbolQ[d] && ! SameQ[Quiet[Check[ToExpression[d], $Failed]], $Failed],
```

```

a = Append[a, d]]; c++;
a = Mapp[StringTrim1, Mapp[StringTrim1, Map[StringTrim, a], "+", "", "-", ""];
a = DeleteDuplicates[Map[StringTrim, Select[a, ! SymbolQ[ToExpression[#]] &&
! NumericQ[ToExpression[#]] &]]];
If[a == {}, False, If[{y} != {} && ! HowAct[{y}][[1]], {y} = {a}]; True]]

In[2161]:= ExprsInStrQ["avz71ransian46078"]
Out[2161]= True
In[2162]:= ExprsInStrQ["a (c + d) - b^2 = Sin[x] h"]
Out[2162]= True
In[2163]:= {ExprsInStrQ["a (c + d) - b^2 = Sin[x] h", t], t}
Out[2163]= {True, {"a*(c + d)", "a*(c + d) - b", "a*(c + d) - b^2", "(c + d)", "(c + d) - b",
"(c + d) - b^2", "c + d", "b^2", "Sin[x]", "Sin[x] h", "in[x]", "in[x] h", "n[x]", "n[x] h"}}
In[2164]:= {ExprsInStrQ["n*(a+c)/c ", h1], h1}
Out[2164]= {True, {"n*(a+c)", "n*(a+c)/c", "(a+c)", "(a+c)/c", "a+c"}}

```

В целом ряде задач манипулирования с выражениями, включая дифференцирование и интегрирование по *обобщенным* переменным в символьном виде, довольно актуален вопрос определения состава выражения по *подвыражениям*, входящим в него, включая произвольные переменные. Данную задачу решает процедура **ExprComp**, чей вызов **ExprComp[x]** возвращает множество всех подвыражений, составляющих выражение *x*, тогда как вызов **ExprComp[x, z]**, где *второй* необязательный аргумент *z* – *неопределенная* переменная, через *z* дополнительно возвращает вложенный список подвыражений *x* выражения по уровням, начиная с первого уровня. Следующий фрагмент приводит исходный код процедуры **ExprComp** с типичными примерами ее использования.

```

In[3329]:= ExprComp[x_, z_] := Module[{a = {x}, b, h = {}, F, q, t = 1},
F[y_List] := Module[{c = {}, d, p, k, j = 1},
For[j = 1, j <= Length[y], j++, k = 1;
While[k < Infinity, p = y[[j]]; a = Quiet[Check[Part[p, k], $Failed]];
If[a === $Failed, Break[], If[! SameQ[a, {}], AppendTo[c, a]]; k++]; c];
q = F[a]; While[q != {}, AppendTo[h, q]; q = Flatten[Map[F[{}]] &, q]];
If[{z} != {} && ! HowAct[z], z = Map[Select[#, ! NumberQ[#] &] &, h]];
Sort[Select[DeleteDuplicates[Flatten[h], Abs[#1] === Abs[#2] &], ! NumberQ[#] &]]]

In[3330]:= ExprComp[(1/b + Cos[a + Sqrt[c + d]])/(Tan[1/b] - 1/c^2)]
Out[3330]= {a, 1/b, b, -(1/c^2), c, d, Sqrt[c + d], c + d, a + Sqrt[c + d], Cos[a + Sqrt[c + d]],
1/b + Cos[a + Sqrt[c + d]], Tan[1/b], 1/(-(1/c^2) + Tan[1/b]), -(1/c^2) + Tan[1/b]}
In[3331]:= ExprComp[(1/b + Cos[a + Sqrt[c + d]])/(Tan[1/b] - 1/c^2), g]
Out[3331]= {a, 1/b, b, -(1/c^2), c, d, Sqrt[c + d], c + d, a + Sqrt[c + d], Cos[a + Sqrt[c + d]],
1/b + Cos[a + Sqrt[c + d]], Tan[1/b], 1/(-(1/c^2) + Tan[1/b]), -(1/c^2) + Tan[1/b]}
In[3332]:= g
Out[3332]= {{1/b + Cos[a + Sqrt[c + d]], 1/(-(1/c^2) + Tan[1/b])}, {1/b, Cos[a + Sqrt[c + d]],

```

```

-(1/c^2) + Tan[1/b]], {b, a + Sqrt[c + d], -(1/c^2), Tan[1/b]}, {a, Sqrt[c + d],
1/c^2, 1/b}, {c + d, c, b}, {c, d}}
In[3334]:= ExprComp[Log[1/b + Cos[a + Sqrt[c + d]]]/(Tan[1/b]*(m + n) - 1/c^2), v]
Out[3334]= {a, 1/b, b, -(1/c^2), c, d, Sqrt[c + d], c + d, a + Sqrt[c + d], m, n, m + n,
Cos[a + Sqrt[c + d]], 1/b + Cos[a + Sqrt[c + d]], Log[1/b + Cos[a + Sqrt[c + d]]],
Tan[1/b], (m + n)*Tan[1/b], 1/(-(1/c^2) + (m + n)*Tan[1/b]),
-(1/c^2) + (m + n)*Tan[1/b]}
In[3335]:= v
Out[3335]= {{Log[1/b + Cos[a + Sqrt[c + d]]], 1/(-(1/c^2) + (m + n)*Tan[1/b])}, {1/b +
Cos[a + Sqrt[c + d]], -(1/c^2) + (m + n)*Tan[1/b]}, {1/b, Cos[a + Sqrt[c + d]],
-(1/c^2), (m + n)*Tan[1/b]}, {b, a + Sqrt[c + d], 1/c^2, m + n, Tan[1/b]},
{a, Sqrt[c + d], c, m, n, 1/b}, {c + d, b}, {c, d}}

```

Выражение может быть образовано при помощи арифметических операторов типов: **Plus** (+, -), **Times** (\*, /), **Power** (^), **Indexed** (индексов) или **Function** (функции). При этом, выражение  $a - b$  имеет тип "+" с операндами  $a$ ,  $-b$ ; тогда как выражение  $a/b$  - тип "/" с операндами  $a$ ,  $b$ ; выражение  $a^b$  имеет тип "^" с операндами  $a$ ,  $b$ ; выражение  $a[b]$  имеет тип "Function" и выражение  $a[[b]]$  - тип "Indexed". В этом смысле возможно использовать некий показатель **Cost** для оценки сложности вычисления выражений. **Cost** определяется как полином от переменных, в качестве которых выступают имена указанных трех операторов, **Indexed** и **Function**, с неотрицательными целочисленными коэффициентами. Процедура **Cost** обеспечивает вычисление данного показателя; ее исходный код с примерами использования представляет следующий фрагмент.

```

In[2255]:= Cost[x_] := Module[{a = ToString[InputForm[x]], b = {"+", "-", {"*", "/"}, {"^"}, {"["}, c,
d = {}, h, f = {Plus, Times, Power, Indexed, Function}, k = 1, j, t},
If[StringFreeQ[a, Flatten[{b, "["}]], 0, c = Map[StringCount[a, #] &, b];
While[k <= 3, h = c[[k]]; If[h != 0, AppendTo[d, {f[[k]], h}]; k++];
If[Set[b, StringCount[a, "["] > 0, AppendTo[d, {f[[4]], b}]; t = StringPosition[a, "["];
If[t != {}, t = Map[#[[1]] &, t]; t = Select[Map[If[StringTake[a, {# - 1, # - 1}] != "[" &&
StringTake[a, {# + 1, # + 1}] != "[", #] &, t], ! SameQ[#, Null] &];
If[t != {}, AppendTo[d, {f[[5]], Length[t]}];
b = StringPosition[a, "-"]; {t, b, h} = {0, Map[#[[1]] &, b], StringLength[a];
For[k = 1, k <= Length[b], k++, c = "";
For[j = b[[k]], j <= h, j++, c = c <> StringTake[a, {j, j}];
If[StringCount[c, "["] === StringCount[c, ""]],
f = Quiet[Check[ToExpression[c], $Failed]];
If[f === $Failed, Continue[], If [NumberQ[f], t = t + 1]; Break[]];
d = If[t != 0 && d[[1]][[1]] === Plus, d[[1]][[2]] = d[[1]][[2]] - t; d, d];
Plus[Sequences[Map[#[[2]]*#[[1]] &, d]]]]
In[2256]:= Cost[(1/v + (1 + (1 + x)*x)*x)*x]
Out[2256]= 3*Plus + Power + 3*Times

```

```

In[2257]:= Cost[z^(h*n - 2) + t^3]
Out[2257]= 3*Plus + 2*Power + Times
In[2258]:= Cost[(z^(h*n - 2) + t^3)/(x*y + c)]
Out[2258]= 4*Plus + 2*Power + 3*Times
In[2259]:= Map[Cost, {42.47, 78*d + p^g, AvzAgnVsv}]
Out[2259]= {0, Plus + Power + Times, 0}
In[2260]:= Cost[(z^(h*n[6] - 2) + t^3)/(x*y + c[78])]
Out[2260]= 2*Function + 4*Plus + 2*Power + 3*Times
In[2261]:= Cost[(z^(-2) + t^(-3))/(x*y + c[x])]
Out[2261]= Function + 2*Plus + 2*Power + 2*Times
In[2262]:= Cost[(a + Sin[-a + v] + x[b[[-78 ;; 460]]]) // Quiet]
Out[2262]= 2*Function + Indexed + 4*Plus
    
```

Вызов **Cost**[*x*] возвращает показатель **Cost** вышеуказанного формата для произвольного алгебраического выражения *x*; при отсутствии для *x* операторов возвращается 0. При этом, процедура достаточно несложно детализируется относительно подсчета числа операторов **Plus**, что оставляем заинтересованному читателю в качестве упражнения.

В целом ряде случаев программирования приложений различного назначения средства настоящей главы оказываются достаточно полезными. Данные средства расположены в нашем пакете *AVZ\_Package\_1* [120] в файле *mx*-формата "*AVZ\_Package\_1.mx*". При необходимости использования его функций вполне достаточно такой файл загрузить в текущий сеанс по функции **Get**, обеспечив все средства из этого файла доступными любому документу на весь период текущего сеанса работы с пакетом. Несколько более детально вопрос создания библиотек, содержащих пользовательские и процедуры, и функции, наряду с доступом к ним рассматривается в конце настоящей монографии. В качестве стандартного механизма *Mathematica* для организации пользовательских средств является сохранение их в пакетах, рассматриваемых ниже. Данный механизм *Mathematica* является довольно эффективным, весьма широко используемым самой системой и пользователями. Довольно широко использовался данный подход и нами. Однако, в целом ряде случаев более предпочтительным оказывается все же подход на основе библиотечной организации, рассматриваемой несколько ниже. Библиотечная организация довольно широко используется и системой, однако нами предлагаются подходы на основе нестандартных организаций библиотек как довольно простых, так и весьма удобных для использования и обеспечения программной поддержки. Между тем, система *Maple* в этом отношении представляется нам более предпочтительной, предоставляя пользователю механизм ведения библиотек, практически, аналогичный механизму ведения стандартной библиотеки системы. Более того, наряду с собственно созданием библиотеки система предоставляет также очень простой механизм ведения соответствующей ей справочной базы, очень легко интегрируемой в стандартную базу системы, что в целом делает библиотеку пользователя на уровне стандартной. Между тем, средства *Mathematica* также позволяют программировать механизмы организации пользовательских библиотек, подобные *Maple*, с обеспечением уровня средств из них на логическом уровне стандартных средств системы. Некоторые подходы к решению данной проблемы рассматриваются в настоящей книге несколько ниже.

## Глава 5. Дополнительные средства обработки символов и строчных структур в программной среде системы *Mathematica*

Не взирая на то, что *Mathematica* располагает достаточно большим набором средств для работы со строчными структурами, зачастую имеется потребность в средствах, в наборе системы отсутствующих. Некоторые из таких средств представлены в данной главе; среди них имеются как простые, так и более сложные, появившиеся в процессе программирования задач разного назначения в качестве сопутствующих функций и процедур, облегчающих либо упрощающих процесс программирования.

В настоящей главе на ряде примеров иллюстрируется *оформление* процедур системы *Mathematica*, которое отражает основные ее элементы и принципы, что позволяет с учетом вышерассмотренного материала непосредственно приступать к созданию, на первых порах, относительно несложных процедур пользователя разного назначения, базирующихся на обработке строчных структур данных. Наряду с этим, целый ряд примеров *Mathematica*-процедур различного назначения представлен также в других главах книги. Здесь рассматриваются только процедуры т.н. «системного» характера, предназначенные для обработки строчных структур, которые, однако, представляют также самый непосредственный *прикладной* интерес в программировании различных приложений. Более того, здесь представлены такие процедуры и функции, которые располагают вполне обозримым объемом *исходного* кода, что позволяет проводить их относительно несложный анализ. Их анализ может послужить достаточно полезным упражнением читателю, как начинающему программирование в среде *Mathematica*, так и уже имеющему достаточно серьезный опыт в данном направлении. Далее под *системными* средствами мы будем понимать как средства собственно самой системы, так и наши средства, ориентированные на массовый характер применения.

Следует отметить, что строчные структуры представляют особый интерес не только в качестве базовых структур данных, с которыми работает система и пользователь, но и в качестве основы, в частности, *динамической* генерации объектов в среде *Mathematica*, включая процедуры и функции. Механизм такой динамической генерации довольно прост и состоит в следующем. При выполнении алгоритма, реализуемого, например, процедурой, в зависимости от фактических аргументов или от вычисляемых значений некоторых переменных возникает необходимость выполнения вычисления с ранее не предусмотренными параметрами, например, вычисления цикла нужной *вложенности*. В этом случае можно сгенерировать строчную структуру, описывающую требуемый циклический объект, с последующим применением к ней **ToExpression**-функции. В целом ряде рассматриваемых в книге средств был использован данный механизм, во многих случаях позволяя получать достаточно эффективные решения.

В частности, вызов процедуры **SuffPref**[*S*, *s*, *n*] обеспечивает тестирование строки *S* на предмет начинаться (*n* = 1), завершаться (*n* = 2) подстрокой или подстроками из списка *s*, или (*n* = 3) быть с обеих концов ограниченной подстроками из *s*. При установлении данного факта **SuffPref** возвращает значение **True**, в противном случае возвращается **False**. Тогда как довольно простая функция **StrStr**[*x*] обеспечивает возврат выражения



$x$ , отличного от строки, в строчном формате и двойной строки в противном случае. В целом ряде случаев функция **StrStr** полезна в работе со строчными конструкциями, в частности, со стандартной функцией **StringReplace**.

```
In[2010]:= StrStr[x_] := If[StringQ[x], "\" <> x <> \"", ToString[x]]
In[2011]:= Map[StrStr, {"RANS", a + b, IAN, {71, 66, 46}, F[x,y]}]
Out[2011]= {"\"RANS\", \"a + b\", \"IAN\", \"{71, 66, 46}\", \"F[x, y]"}
In[2012]:= SuffPref[S_String, s_;/; StringQ[s] || ListQ[s] &&
DeleteDuplicates[Map[StringQ, s]] == {True}, n_;/; MemberQ[{1, 2, 3}, n]] :=
Module[{a, b, c, k = 1}, If[StringFreeQ[S, s], False, b = StringLength[S];
c = Flatten[StringPosition[S, s]]; If[n == 3 && c[[1]] == 1 && c[[-1]] == b, True,
If[n == 1 && c[[1]] == 1, True, If[n == 2 && c[[-1]] == b, True, False]]]]
In[2013]:= SuffPref["IAN_RANS_RAC_REA_78_460", "78_460", 2]
Out[2013]= True
In[2014]:= SuffPref["IAN_RANS_RAC_REA_78_460", {"IAN_RANS", "IAN"}, 1]
Out[2014]= True
In[2015]:= SuffPref["IAN_RANS_RAC_REA_78_460_IAN", {"IAN_RANS", "IAN"}, 3]
Out[2015]= True
```

Следующая процедура **Spos**[ $x$ ,  $y$ ,  $p$ ,  $dir$ ] вычисляет номер позиции первого вхождения *однобуквенной* строки  $y$  в строку  $x$  *влево* ( $dir=0$ ) либо *вправо* ( $dir=1$ ) от заданной позиции  $p$ . Если строка  $y$  не входит в строку  $x$  в указанном от позиции  $p$  направлении, то вызов процедуры **Spos** возвращает нулевое значение. В противном случае вызов процедуры **Spos** возвращает номер позиции первого вхождения *однобуквенной* строки  $y$  в строку  $x$  *влево* ( $dir=0$ ) или *вправо* ( $dir=1$ ) от заданной позиции  $p$ ; при этом, номер позиции будет отсчитываться от начала строки  $x$ . При этом, процедура **Spos** обрабатывает основные как *ошибочные*, так и *особые* ситуации, возвращая на них **False**-значение. Следующий фрагмент представляет исходный код процедуры **Spos** с примерами ее применения.

```
In[2020]:= Spos[x_String, y_String, p_Integer, dir_Integer] := Module[{a, b, c},
If[StringLength[y] > 1 || dir != 0 && dir != 1, Return[False], b = StringLength[x]];
If[p < 1 || p > b, False, If[p == 1 && dir == 0, c = 0, If[p == b && dir == 1, c = 0,
If[dir == 0, For[a = p, a >= 1, a -= 1, If[StringTake[x, {a}] == y, Return[a], c]],
For[a = p, a <= b, a += 1, If[StringTake[x, {a}] == y, Return[a], c]]]]];
If[a == 0 || a == b + 1, 0, a]]
In[2021]:= Q:= "AVZ78RANS460IAN2013"; {Spos[Q, "A", 10, 0], Spos[Q, "4", 3, 1],
Spos[Q, "0", 1, 1], Spos[Q, "Z", 19, 0], Spos[Q, "W", 19, 0], Spos[Q, "P", 1, 1]}
Out[2021]= {7, 10, 12, 3, 0, 0}
```

В целом ряде случаев возможностей стандартных функций **StringReplace** и **Replace**, возвращающих соответственно результаты замены подстрок в строчных выражениях и подвыражений в алгебраических выражениях, оказывается недостаточно. Поэтому была запрограммирована процедура, чей вызов **StringReplace2**[ $S$ ,  $s$ ,  $Exp$ ] возвращает

результат замены *всех* вхождений в строку  $S$  ее подстроки  $s$  на выражение  $Exp$ ; при этом, заменяемые подстроки  $s$  не должны быть ограничены буквами. В случае, если строка  $S$  не содержит вхождений  $s$ , то вызов процедуры возвращает исходную строку  $S$ , тогда как на пустой строке  $S$  возвращается пустая строка. В определенном смысле процедура **StringReplace2** совмещает возможности указанных системных функций. Следующий фрагмент представляет как исходный код процедуры **StringReplace2**, так и наиболее типичные примеры ее использования.

```
In[2267]:= StringReplace2[S_;/; StringQ[S], s_;/; StringQ[s], Exp_] := Module[{b, c, d, k = 1,
    a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]],
    b = Quiet[Select[StringPosition[S, s],
    ! MemberQ[a, StringTake[S, {#[[1]] - 1, #[[1]] - 1}]] &&
    ! MemberQ[a, StringTake[S, {#[[2]] + 1, #[[2]] + 1}]] &]],
    StringReplacePart[S, ToString[Exp], b]]

In[2268]:= StringReplace2["Length[\"abSin[x]\"] + Sin[x] + ab - Sin[x]*6", "Sin[x]", "a^b"]
Out[2268]= "Length[\"abSin[x]\"] + a^b + ab - a^b*6"
In[2269]:= StringReplace2["Length[\"abSin[x]\"] + Cos[x] + ab - Cos[x]*6", "Sin[x]", "a^b"]
Out[2269]= "Length[\"abSin[x]\"] + Cos[x] + ab - Cos[x]*6"
In[2270]:= StringReplace2["", "Sin[x]", "(a+b)"]
Out[2270]= ""
```

В дополнение к стандартной функции **StringReplace** и предыдущей **StringReplace2** в целом ряде случаев довольно полезной оказывается процедура **StringReplace1[S, L, P]**, которая возвращает результат подстановки в строку  $S$  подстроки из списка  $P$  вместо ее подстроки, определяемых позициями *вложенного* списка  $L$  *ListList*-типа. В ряде случаев процедура **StringReplace1** дополняет системную функцию **StringReplace**. Следующий фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2331]:= StringReplace1[S_;/; StringQ[S], L_;/; ListListQ[L] && Length[L[[1]]] == 2 &&
    MatrixQ[L, IntegerQ] && Sort[Map[Min, L]] [[1]] >= 1, P_;/; ListQ[P]] :=
    Module[{a = {}, b, k = 1, If[Sort[Map[Max, L]] [[-1]] <= StringLength[S] &&
    Length[P] == Length[L], Null, Return[Defer[StringReplace1[S, L, P]]];
    For[k, k <= Length[L], k++, b = L[[k]];
    a = Append[a, StringTake[S, {b[[1]], b[[2]]}] -> ToString[P[[k]]]]];
    StringReplace[S, a]]

In[2332]:= StringReplace1["avz123456789agn", {{4, 7}, {8, 10}, {11, 12}}, {" RANS ", Tampere,
    Sqrt[(a + b)*(c + d)]}]
Out[2332]= "avz RANS TampereSqrt[(a + b) (c + d)]agn"
```

Для работы со строчными структурами вполне определенный интерес представляет процедура **SubsDel**, чей вызов **SubsDel[S, x, y, p]** возвращает результат удаления из  $S$ -строки *всех* подстрок, которые ограничены справа (*слева*) подстрокой  $x$  и слева (*справа*) первым встреченным символом в строчном формате из списка  $y$ ; более того, поиск  $y$ -

символа производится влево ( $p = -1$ ) или вправо ( $p = 1$ ). При этом, удаляемые подстроки будут содержать подстроку  $x$  с одного конца и первый символ до встреченного из  $y$  с другого конца. Более того, если в процессе поиска до конца строки не были найдены символы из списка  $y$ , то в результате удаляется оставшаяся часть исходной строки  $S$ .

```
In[2321]:= SubsDel[S_;/ StringQ[S], x_;/ StringQ[x], y_;/ ListQ[y] &&
DeleteDuplicates[Map[StringQ, y]] == {True} &&
Plus[Sequences[Map[StringLength, y]]] == Length[y],
p_;/ MemberQ[{-1, 1}, p]] :=
Module[{b, c = x, d, h = StringLength[S], k},
If[StringFreeQ[S, x], Return[S], b = StringPosition[S, x][[1]]];
For[k = If[p == 1, b[[2]] + 1, b[[1]] - 1],
If[p == 1, k <= h, k >= 1], If[p == 1, k++, k--],
d = StringTake[S, {k, k}];
If[MemberQ[y, d] || If[p == 1, k == 1, k == h], Break[],
If[p == 1, c = c <> d, c = d <> c]; Continue[]];
StringReplace[S, c -> ""]]

In[2322]:= S = "Subs[AladjevProceduresAndFunctions`Subs`x_,
AladjevProceduresAndFunctions`Subs`y_,AladjevProceduresAndFunctions`Subs`z_]";
Out[2322]= "Subs[AladjevProceduresAndFunctions`Subs`x_,
AladjevProceduresAndFunctions`Subs`y_,AladjevProceduresAndFunctions`Subs`z_]";
In[2323]:= SubsDel[S, "Subs`", {"[", " ", "}"}, -1]
Out[2323]= "Subs[x_,y_,z_]"
In[2324]:= SubsDel["12345avz6789", "avz", {"8"}, 1]
Out[2324]= "1234589"
In[2325]:= SubsDel["12345avz6789", "avz", {"8", 9}, 1]
Out[2325]= SubsDel["12345avz6789", "avz", {"8", 9}, 1]
In[2326]:= SubsDel["123456789avz6789", "avz", {"5"}, 1]
Out[2326]= "123456789"
```

Предыдущий фрагмент представляет исходный код процедуры **SubsDel**[ $S, x, y, p$ ] с наиболее типичными примерами ее использования. Данная процедура используется целым рядом процедур и функций из нашего пакета *AVZ\_Package\_1* [120].

Простая функция **StrOfSymb1Q**[ $S, A$ ] возвращает **True**, если строка  $S$  содержит только символы из множества  $A$ , и **False** в противном случае. Данная функция была создана для системы *Mathematica* в качестве полезного аналога трех *одноименных* процедур из нашей Библиотеки процедур [45] для системы *Maple*. В то время как представляемая процедура **SubDelStr**[ $x, L$ ] обеспечивает удаление из строки  $x$  всех подстрок, которые ограничены номерами позиций, заданных *списком списков*  $L$  (*список ListList-мина*) из двухэлементных подсписков. На некорректных кортежах из фактических аргументов вызов процедуры возвращается невычисленным. Следующий фрагмент представляет исходные коды для обоих средств с типичными примерами их использования.

```
In[2568]:= StrOfSymbQ[x_ /; StringQ[x], A_ /; ListQ[A]] := If[DeleteDuplicates[Map3[
    MemberQ, Map[ToString, A], Characters[x]]] == {True}, True, False]

In[2569]:= StrOfSymbQ["1343256789", {1, 2, 4, 5, 7, 9}]
Out[2569]= False
In[2570]:= StrOfSymbQ["41343825567289", {1, 3, 2, 4, 5, 6, 7, 8, 9}]
Out[2570]= True

In[2826]:= SubDelStr[x_ /; StringQ[x], L_ /; ListListQ[L]] := Module[{k = 1, a = {}},
    If[! L == Select[L, ListQ[#] && Length[#] == 2 &] ||
        L[[-1]][[2]] > StringLength[x] || L[[1]][[1]] < 1,
        Return[Defer[SubDelStr[x, L]],
        For[k, k <= Length[L], k++, a = Append[a, StringTake[x, L[[k]]] -> ""]];
    StringReplace[x, a]]

In[2827]:= SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}}]
Out[2827]= "1269dfdh"
In[2828]:= SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}, {40, 42}}]
Out[2828]= SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}, {40, 42}}]
```

Для получения подстрок строки, заданных их позициями конца и начала, *Mathematica* располагает функцией **StringTake**, допускающей 6 форматов кодирования. Однако, в целом ряде случаев более удобным оказывается получение подстрок, ограниченных не позициями, а заданными подстроками. Для данной цели служат две функционально идентичные процедуры *StringTake1* и *StringTake2*, чьи исходные коды представлены в нижеследующем фрагменте. Процедуры в целях иллюстрации реализуют различные алгоритмы, методы реализации которых могут быть полезны при программировании задач обработки объектов в строчном формате. Вызов процедуры **StringTake1** *StringTake1*[1 | 2][x, y] возвращает список подстрок строки x, ограниченных ее подстроками y; в качестве 2-го аргумента может выступать как произвольное выражение, так и их список. Фрагмент представляет исходные коды процедур **StringTake1** и **StringTake2** наряду с наиболее типичными примерами их использования.

```
In[2747]:= StringTake1[x_ /; StringQ[x], y_] := Module[{b = {}, c, k = 1, a = Map[ToString,
    Map[InputForm, y]]}, c = Sort[StringPosition[x, a]];
    b = {StringTake[x, {1, c[[1]][[1]] - 1}]}; For[k, k <= Length[c] - 1, k++,
    b = Append[b, StringTake[x, {c[[k]][[2]] + 1, c[[k + 1]][[1]] - 1}]]];
    Select[Append[b, StringTake[x, {c[[k]][[2]] + 1, -1}]], # != "" &]]

In[2748]:= StringTake1["ransianavzagnvsvartkr", {ian, agn, art}]
Out[2748]= {"rans", "avz", "vsv", "kr"}
In[2749]:= StringTake1["ransianavzagnvsvartkr", {ian, 460, art, 78}]
Out[2749]= {"rans", "avzagnvsv", "kr"}
In[2750]:= StringTake1["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]
Out[2750]= {"s", "avz", "vsv"}
```

```

In[2751]:= StringTake2[x_;/; StringQ[x], y_] := Module[{b = {}, k = 1, a = Map[ToString,
      Map[InputForm, y]]}, For[k, k <= Length[a], k++,
      b = Append[b, ToString1[a[[k]]] <> "->" <> "\", \""];
      StringSplit[ StringReplace[x, ToExpression[b], ", "]]

In[2752]:= StringTake2["ransianavzagnvsvartkr", {ian, agn, art}]
Out[2752]= {"rans", "avz", "vsv", "kr"}
In[2753]:= StringTake2["ransianavzagnvsvartkr", {ian, 460, art, 78}]
Out[2753]= {"rans", "avzagnvsv", "kr"}
In[2754]:= StringTake2["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]
Out[2754]= {"s", "avz", "vsv"}

```

С другой стороны, вызов процедуры **StrDelEnds**[*W*, *h*, *p*] возвращает усеченную слева строку *W* подстрокой *h* при *p* = 1, усеченную справа строку *W* подстрокой *h* при *p* = 2, усеченную с *обеих* концов строку *W* при *p* = 3; тогда как при других значениях *p* вызов процедуры **StrDelEnds**[*W*, *h*, *p*] возвращается невычисленным. Следующий фрагмент представляет исходный код процедуры с некоторыми примерами ее использования.

```

In[2042]:= StrDelEnds[S_String, h_String, p_;/; MemberQ[{1, 2, 3}, p]] := Module[{a, k, s},
      If[! SuffPref[S, h, p], S,
      If[p == 3, StringTrim[S, h ...], {s, a, k} = {S, StringLength[h], 1};
      While[SuffPref[s, h, p], s = StringTake[s,
      If[p == 1, {a + 1, -1}, {1, StringLength[s] - a}]]; k++ 1]; s]]

In[2043]:= StrDelEnds["avzavzxxxxx123456789yyyyyavzavzavz", "avz", 1]
Out[2043]= "xxxxx123456789yyyyyavzavzavz"
In[2044]:= StrDelEnds["avzavzxxxxx123456789yyyyyavzavzavz", "avz", 2]
Out[2044]= "avzavzxxxxx123456789yyyyy"
In[2045]:= StrDelEnds["avzavzxxxxx123456789yyyyyavzavzavz", "avz", 3]
Out[2045]= "xxxxx123456789yyyyy"
In[2046]:= StrDelEnds["avzavzxxxxx123456789yyyyyavzavzavz", "agn", 78]
Out[2046]= StrDelEnds["avzavzxxxxx123456789yyyyyavzavzavz", "agn", 78]

```

Для работы со строчными структурами достаточно полезной оказывается процедура **InsertN**[*S*, *L*, *n*] (аналог одноименной процедуры для *Maple*, рассмотренной в [45]), которая обеспечивает вставку в строку *S* подстрок из списка *L* после ее позиций из списка *n*. В случае *n* = {< 1 | ≥ StringLength[*S*]} подстрока помещается перед началом строки *S* или в ее конец соответственно. Фрагмент представляет исходный код процедуры **InsertN** наряду с примерами ее применения. Предполагается, что фактические аргументы *L* и *n* могут содержать различное число элементов, в данном случае лишние элементы *n* игнорируются. Вызов с недопустимыми аргументами возвращается невычисленным.

```

In[2583]:= InsertN[S_String, L_;/; ListQ[L], n_;/; ListQ[n] && Length[n] ==
      Length[Select[n, IntegerQ[#] &]] :=
      Module[{a = Map[ToString, L], b, c = FromCharacterCode[2],

```

```

d = Characters[S], k = 1, p, m = DeleteDuplicates[Sort[n]],
b = Map[c <> ToString[#] &, Range[1, Length[d]]];
b = Riffle[d, b]; p = Min[Length[a], Length[m]];
While[k <= p, If[m[[k]] < 1, PrependTo[b, a[[k]]],
If[m[[k]] > Length[d], AppendTo[b, a[[k]]],
b = ReplaceAll[b, c <> ToString[m[[k]]] -> a[[k]]]; k++];
StringJoin[Select[b, ! SuffPref[#, c, 1] &]]

In[2584]:= InsertN["123456789Rans_Ian", {Agn, Avz, Vsv, Art, Kr}, {6, 9, 3, 0, 3, 17}]
Out[2584]= "Agn123Avz456Vsv789ArtRans_IanKr"
In[2585]:= InsertN["123456789", {a, b, c, d, e, f, g, h, n, m}, {4, 2, 3, 0, 1, 5, 6, 7, 8, 9, 17}]
Out[2585]= "a1b2c3d4e5f6g7h8n9m"

```

При этом, обработка строки  $S$  выполняется относительно списка позиций для вставок  $m$ , определяемого согласно следующего соотношения  $m = \text{DeleteDuplicates}[\text{Sort}[n]]$ .

В противоположность предыдущей процедуре следующая процедура  $\text{DelSubStr}[S, L]$  обеспечивает удаление из строки  $S$  подстрок, чьи позиции заданы списком  $L$ . Список  $L$  может быть как вложенности  $0$ , так и  $1$ , например,  $\{\{3, 4\}, \{7\}, \{9\}\}$  либо  $\{1, 3, 5, 7, 15\}$ .

```

In[2738]:= DelSubStr[S_String, L_ /; ListQ[L] && MemberQ[{0, 1}, MaxNestLevel[L]]] :=
Module[{a = If[MaxNestLevel[L] == 1, Select[Sort[Map[Sort, L]],
MemberQ[Range[1, StringLength[S]], First[#]] &&
MemberQ[Range[1, StringLength[S]], First[#]] &,
Map[List, Sort[L]]], c = S, d = {}, h, k = 1},
If[Sort[Flatten[L]]][[-1]] > StringLength[S], S, For[k, k <= Length[a], k++,
d = Append[d, If[Length[a[[k]]] == 2, a[[k]], {a[[k]][[1]], a[[k]][[1]]}]; d = Sort[d];
For[k = 1, k <= Length[a], k++, h = d[[k]];
c = StringReplacePart[c, "", h]; d = d - Last[h] + First[h] - 1]; c]]

In[2739]:= DelSubStr["123456789abcdfh", {{3, 4}, {7}, {9}, {13}, {15}}]
Out[2739]= "12568abcf"
In[2740]:= DelSubStr["123Rans4IAN786460", {{4, 8}, {9, 11}, {12}}]
Out[2740]= "12386460"
In[2741]:= DelSubStr["123456789abcdgh", Range[1, 15]]
Out[2741]= ""
In[2742]:= DelSubStr["123Rans4IAN785460", {4, 8}]
Out[2742]= "123ansIAN785460"
In[2743]:= DelSubStr["123Rans4IAN785460", {{4, 8}}]
Out[2743]= "123IAN785460"
In[2744]:= DelSubStr["123456789abcdgh", {6}]
Out[2744]= "12345789abcdgh"
In[2745]:= DelSubStr["123456789abcdgh", Range[1, 17]]
Out[2745]= "123456789abcdgh"

```

Процедура позволяет удалять из строк как их подстроки, определенные списками их граничных позиций, так и отдельные символы, определенные списками их позиций. Предыдущий фрагмент представляет исходный код процедуры и ее применения.

Весьма простая, но довольно полезная функция **SortString**[*x*, *y*] возвращает результат посимвольной сортировки *x*-строки в соответствии со вторым *y*-аргументом, который принимает значение {*Greater* | *Less*}. Следующий фрагмент представляет исходный код функции **SortString** наряду с довольно типичными примерами ее использования.

```
In[2013]:= SortString[x_/: StringQ[x], y_/: MemberQ[{Greater, Less}, y]] := StringJoin[
      Sort[Characters[x], y[ToCharacterCode[#1][[1]], ToCharacterCode[#2][[1]]] &]]
In[2014]:= SortString["AVZ321AGN456Art789Kr", Less]
Out[2014]= "123456789AAAGKNVZrrt"
In[2015]:= SortString["AVZ321AGN456Art789Kr", Greater]
Out[2015]= "trrZVNKGAAA987654321"
```

Ранее уже отмечалось, что некоторые функциональные средства пакета нуждаются в доработке как с целью *расширения* сферы применения, так и устранения недостатков. К таким средствам в полной мере можно отнести такие весьма широко используемые функции, как *D*, *Integrate* и *Replace*, не обеспечивающие корректного решения в ряде довольно простых случаев. Представленные здесь процедуры расширяют указанные средства на целый ряд случаев, не обеспечиваемых данными функциями пакета. Это в полной мере относится и к такой важной функции, как **ToString**[*x*], возвращающей результат конвертирования *x*-выражения в строчный формат.

Данная стандартная процедура пакета *некорректно* конвертирует в строчный формат выражения, содержащие строчные подвыражения, если их кодировать стандартным способом. По этой причине нами была определена относительно простая процедура **ToString1**[*x*], возвращающая результат корректного конвертирования произвольного *x*-выражения в *строчный* формат, как наглядно иллюстрирует следующий фрагмент, представляющий исходный код процедуры **ToString1** с примерами ее применения. В целом ряде приложений данная процедура оказывается весьма востребованной.

```
In[2720]:= ToString1[x_] := Module[{a = "$Art24Kr17$.txt", b = "", c, k = 1}, Write[a, x];
      Close[a]; For[k, k < Infinity, k++, c = Read[a, String];
      If[SameQ[c, EndOfFile], Return[DeleteFile[Close[a]]; b],
      b = b <> StrDelEnds[c, " ", 1]]]
In[2721]:= Kr[x_] := Module[{a = "Art_Kr", b = " = "}, a <> b <> ToString[x]]
In[2722]:= ToString[Definition[Kr]]
Out[2722]= "Kr[x_] := Module[{a = Art_Kr, b = = }, a<>b<>ToString[x]]"
In[2723]:= ToExpression[%]
ToExpression::sntx: Invalid syntax in or before "Kr[x_] := Module[{a = Art_Kr, b = = } ... ".
Out[2723]= $Failed
In[2724]:= ToString1[Definition[Kr]]
Out[2724]= "Kr[x_] := Module[{a = \"Art_Kr\", b = \" = \"}, StringJoin[a, b, ToString[x]]]"
```

```
In[2725]:= ToExpression[%]; Kr[2]
Out[2725]= "Art_Kr = 2"

In[2748]:= ToString2[x_] := Module[{a}, If[ListQ[x], SetAttributes[ToString1, Listable];
      a = ToString1[x]; ClearAttributes[ToString1, Listable]; a, ToString1[x]]]

In[2749]:= ToString2[a + b/71 - Sin[460.78]]
Out[2749]= "-0.8594128004980472 + a + b/71"

In[2750]:= ToString2[{{71, 66}, {46, {a, b, {x, y}, c}, 51}, {24, 17}}]
Out[2750]= {{{"71", "66"}, {"46", {"a", "b", {"x", "y"}, "c"}, "51"}, {"24", "17"}}
```

Непосредственное применение процедуры **ToString1** позволяет в целом ряде случаев, порой, достаточно существенно упрощать алгоритмы программируемых задач. При этом, примеры предыдущего фрагмента наглядно иллюстрируют применение обоих средств на конкретном примере, подчеркивающим преимущества нашей процедуры. Тогда как процедура **ToString2** расширяет предыдущую процедуру на списки любого уровня вложенности. Так, вызов процедуры **ToString2[x]** на аргументе  $x$ , отличном от списка, эквивалентен вызову **ToString1[x]**, а на списке  $x$  – вызову **ToString1[x]**, которая наделена *Listable*-атрибутом. Завершает данный фрагмент исходный код процедуры **ToString2** наряду с наиболее типичными примерами ее использования.

Следующая простая процедура является весьма полезным расширением стандартной процедуры **StringPosition**; вызов **StringPosition1[x, y]** возвращает список, элементами которого выступают списки элементов  $y$ , как первых элементов, с их подписками как начальных, так и конечных позиций вхождения в строку  $x$  в качестве ее подстроки.

```
In[2263]:= StringPosition1[x_ /; StringQ[x], y_] := Module[{b = {}, c = 1, d,
      a = Flatten[{If[ListQ[y], Map[ToString, y], ToString[y]}]},
      For[c, c <= Length[a], c++, d = a[[c]];
      AppendTo[b, {d, Flatten[StringPosition[x, d]}]];
      Sort[Select[b, #[[2]] != {} &]]]

In[2264]:= StringPosition1["abcdefghpstw", {abc, cde, hp, stw}]
Out[2264]= {{{"abc", {1, 3}}, {"cde", {3, 5}}, {"hp", {8, 9}}, {"stw", {10, 12}}}

In[2265]:= StringPosition1["abcdefghpstw", {x, y, z, v}]
Out[2265]= {}
```

Вторым аргументом в вызове процедуры может выступать как отдельное выражение, так и их список. Выше приведен исходный код процедуры и пример ее применения. При отсутствии вхождений  $y$  в строку  $x$  возвращается пустой список, т.е. {}.

Следующий фрагмент представляет достаточно полезную процедуру, вызов которой **SubStr[S, p, a, b, r]** возвращает подстроку строки  $S$ , которая слева ограничена первым символом, отличным от символа  $a$  либо от символов из списка  $a$ , и справа ограничена символом, отличным от символа  $b$  или от символов из списка  $b$ . Между тем, как через аргумент  $r$  в случае ошибочной ситуации возвращается соответствующее сообщение, диагностирующее возникшую ошибочную ситуацию. Значение аргумента  $p$  должно



находиться в интервале 1 .. [StringLength[S]. Следующий фрагмент представляет код и достаточно типичные примеры использования процедуры SubStr.

```
In[2379]:= SubStr[S_;/ StringQ[S], p_;/ IntegerQ[p], a_;/ CharacterQ[a] || ListQ[a] &&
DeleteDuplicates[Map[CharacterQ, a]] == {True},
b_;/ CharacterQ[b] || ListQ[b] &&
DeleteDuplicates[Map[CharacterQ, b]] == {True}, r_;/ ! HowAct[r]] :=
Module[{c = Quiet[StringTake[S, {p, p}]], k, t},
If[p >= 1 && p <= StringLength[S],
For[k = p + 1, k <= StringLength[S], k++, t = StringTake[S, {k, k}];
If[If[CharacterQ[b], t != b, ! MemberQ[b, t]], c = c <> t; Continue[], Break[]];
For[k = p - 1, k >= 1, k--, t = StringTake[S, {k, k}];
If[If[CharacterQ[a], t != a, ! MemberQ[a, t]], c = c <> t; Continue[], Break[]]; c,
r = "Argument p should be in range 1.." <> ToString[StringLength[S]] <>
" but received " <> ToString[p]; $Failed]]
```

```
In[2380]:= SubStr["12345abcdefg6789sewrt", 14, "3", "r", Error]
```

```
Out[2380]= "45abcdefg6789sew"
```

```
In[2381]:= SubStr["12345abcdefg6789sewrt", 14, {"1", "2", "0"}, {"s", "w", "x"}, Error]
```

```
Out[2381]= "345abcdefg6789"
```

```
In[2382]:= SubStr["12345abcdefg6789sewrt", 22, "0", "x", Error]
```

```
Out[2382]= $Failed
```

```
In[2383]:= Error
```

```
Out[2383]= "Argument p should be in range 1..21 but received 22"
```

```
In[2384]:= SubStr["12345ab3cdefgr6789sewrt", 7, "3", "r", Err]
```

```
Out[2384]= "45ab3cdefg"
```

В целом ряде задач обработки *выражений* довольно актуальной является задача *разного* рода выделений того или иного типа выражений из *строчных* конструкций. В данном отношении определенный интерес представляет процедура, исходный код которой с примерами ее типичного использования представляет нижеследующий фрагмент.

```
In[2075]:= ExprOfStr[x_String, n_;/ IntegerQ[n] && n > 0, m_;/ MemberQ[{-1, 1}, m],
L_;/ ListQ[L]] := Module[{a = "", b, k}, If[n >= StringLength[x],
Return[Defer[ExprOfStr[x, n, m, L]], Null];
For[k = n, If[m == -1, k >= 1, k <= StringLength[x]],
If[m == -1, k--, k++], If[m == -1, a = StringTake[x, {k, k}] <> a,
a = a <> StringTake[x, {k, k}]]; b = Quiet[ToExpression[a]];
If[b === $Failed, Null, If[If[m == -1, k == 1, k == StringLength[x]] ||
MemberQ[L, Quiet[StringTake[x, If[m == -1, {k - 1, k - 1}, {k + 1, k + 1}]]],
Return[a], Null]]; $Failed]
```

```
In[2076]:= P[x_, y_] := Module[{a, P1}, P1[z_, h_] := Module[{n}, z^2 + h^2]; x*y + P1[x, y]]
```

```
In[2077]:= x = DefFunc1[ToString[P]]; {ExprOfStr[x, 44, 1, {" ", ";", ", ", ""}],
                                          ExprOfStr[x, 39, -1, {" ", ";", ", ", ""}]}
Out[2077]= {"Module[{n}, z^2 + h^2]", "P1[z_, h_]"}
In[2078]:= y = "123456789abcdefghAVZ_2013"; ExprOfStr[y, 78, -1, {" ", ";", ", ", ""}]
Out[2078]= ExprOfStr["123456789abcdefghAVZ_2013", 78, -1, {" ", ";", ", ", ""}]
In[2079]:= ExprOfStr[x, 10, 1, {" ", ";", ", ", ""}]
Out[2079]= $Failed
In[2080]:= ExprOfStr["12345678;F[(a + b)/(c + d)]; AVZ_2013", 28, -1, {"^", ";", ""}]
Out[2080]= "F[(a + b)/(c + d)];"
```

Вызов процедуры **ExprOfStr** $[w, n, m, L]$  возвращает результат извлечения из строки  $w$ , ограниченной ее  $n$ -й позицией и концом, *первого* корректного выражения с условием, что поиск производится *влево* ( $m=-1$ )/*вправо* ( $m=1$ ) от заданной позиции и следующий или предыдущий за найденным выражением символ должен принадлежать списку  $L$ . Вызов данной процедуры возвращается в *строчном* формате; в отсутствие *корректного* выражения возвращается **\$Failed**, в то время как вызов процедуры на недопустимых *фактических* аргументах возвращается невычисленным. Приведенные в предыдущем фрагменте примеры достаточно наглядно иллюстрируют вышесказанное.

Процедура **ExprOfStr1** представляет собой одну полезную *модификацию* предыдущей процедуры; ее вызов **ExprOfStr1** $[x, n, p]$  возвращает минимальную по длине *подстроку* строки  $x$ , в которой граничным элементом является символ в  $n$ -й позиции строки  $x$ , содержащую корректное выражение. При этом, поиск такой подстроки производится вправо от  $n$ -й позиции и до конца строки  $x$  при  $p = 1$ , и влево от  $n$ -й позиции строки и до начала строки при  $p = -1$ . В случае отсутствия такой подстроки вызов процедуры **ExprOfStr1** $[x, n, p]$  возвращает **\$Failed**, в то время как на недопустимых *фактических* аргументах вызов возвращается *невычисленным*. Следующий фрагмент представляет исходный код процедуры наряду с типичными примерами ее использования.

```
In[2145]:= ExprOfStr1[x_String, n_ /; IntegerQ[n] && n >= 1 && n <= StringLength[x],
                                          p_ /; MemberQ[{-1, 1}, p]] :=
Module[{a = StringTake[x, {n, n}], b = StringLength[x], k},
  For[k = If[p == -1, n - 1, n + 1], If[p == -1, k >= 1, k <= b],
    If[p == -1, k--, k++], If[Quiet[ToExpression[a]] === $Failed,
      If[p == -1, a = StringTake[x, {k, k}] <> a,
        a = a <> StringTake[x, {k, k}]]; Continue[], Return[a]]; $Failed]

In[2146]:= x = "123{a+b}, F[c+d]"; ExprOfStr1[x, 4, 1]
Out[2146]= "{a+b}"
In[2147]:= x = "123{a+b}, F[c+d+Sin[a+b]]"; ExprOfStr1[x, 25, -1]
Out[2147]= "F[c+d+Sin[a+b]]"
In[2148]:= x = "123{a+b}, [c+d]"; ExprOfStr1[x, 15, -1]
Out[2148]= $Failed
In[2149]:= x = "123{a+b}, [c+d]"; ExprOfStr1[x, 17, -1]
Out[2149]= ExprOfStr1["123{a+b}, [c+d]", 17, -1]
```

В определенном отношении к процедуре **ExprOfStr** примыкает и процедура **ExtrExpr**, чей вызов **ExtrExpr[S, N, M]** возвращает в строчном формате корректное выражение, которое содержится в подстроке строки *S*, ограниченной позициями с номерами *N* и *M*. При отсутствии корректного выражения возвращается пустой список, т.е. {}.

```
In[2621]:= ExtrExpr[S_;/; StringQ[S], N_;/; IntegerQ[N], M_;/; IntegerQ[M]] :=
Module[{a = StringLength[S], b, c, d, k = -1, Res = {}},
  If[(1 <= M <= a && N <= M), Return[$Failed], Null];
  Label[b]; If[N + k > M, Res = Select[DeleteDuplicates[Res], # != "Null" &];
  Return[If[Res == {}, {}, Res][[-1]]], k = k + 1];
  c = Quiet[ToString[ToExpression[StringTake[S, {N, N + k}]]]];
  If[c == "$Failed", Goto[b], Res = Append[Res, c]; Goto[b]]

In[2622]:= ExtrExpr["z=(Sin[x+y] + Log[x])+G[x,y];", 4, 13]
Out[2622]= "Sin[x + y]"
In[2623]:= ExtrExpr["z=(Sin[x+y] + Log[x])+F[x,y];", 1, 21]
Out[2623]= "Log[x] + Sin[x + y]"
In[2624]:= ExtrExpr["z = (Sin[x + y] + Log[x]) + F[x, y];", 1, 36]
Out[2624]= "F[x, y] + Log[x] + Sin[x + y]"
```

Предыдущий фрагмент представляет исходный код процедуры наряду с примерами ее использования. Процедура **ExtrExpr** оказывается достаточно полезным средством в целом ряде приложений, связанных, прежде всего, с выделением выражений из строк.

*Строчная* является одной из базовых структур как в пакете *Maple*, так и в *Mathematica*, для обеспечения работы с которой оба пакета располагают целым рядом достаточно эффективных средств. Однако, если *Maple* наряду с относительно небольшим набором встроенных средств располагает расширенным набором средств из пакетного модуля **StringTools** и целым рядом средств из нашей библиотеки [45], то *Mathematica* в этом отношении располагает менее представительным набором средств. Между тем, набор ее стандартных средств позволяет достаточно просто программировать недостающие как *Maple*-аналоги средств, так и другие требующиеся средства обработки строчных структур. В книгах [97–101] и в пакете [120] представлен ряд средств, являющихся как аналогами средств системы *Maple*, так и других средств представляющих достаточно определенный интерес для задач обработки строчных структур. В частности, среди которых можно отметить такие, как *Border*, *CatN*, *IsMonotonic*, *IsPermutation*, *LeftFold*, *LongestCommonSubSequence*, *LongestCommonSubString*, *MaximalPalindromicSubstring*, *Overlap*, *PalindromeQ*, *PrefixQ*, *RightFold*, *SuffixQ*. Исходные коды данных средств не представляются, с ними возможно познакомиться в упомянутых литературе и пакете [120], тогда как ниже приводится назначение каждого из перечисленных средств.

Вызов одноименной с *Maple*-процедурой процедуры **Border[x]** возвращает *обрамление* строки *x*, т.е. *максимальной* подстроки, являющейся для *x* и префиксом, и суффиксом. Тогда как вызов **CatN[s, n]** возвращает результат *n*-кратной конкатенации строки *s*. В свою очередь, 2 процедуры **LeftFold** и **RightFold** представляют аналоги одноименных *Maple*-процедур. Вызов процедуры **RightFold[F, id, s]** выполняет итерацию процедуры

$F$  над строкой  $s$ , составляя композицию последовательных результатов справа вместе с начальным значением  $id$ ; в свою очередь вызов процедуры **LeftFold**[ $F, id, s$ ] выполняет итерацию процедуры  $F$  над строкой  $s$ , составляя композицию результатов слева вместе с начальным значением  $id$ . Тогда как вызов процедуры **OverLap**[ $x, y$ ], одноименной с *Maple*-процедурой, возвращает длину наложения между строками  $x$  и  $y$ ; при этом, под наложением понимается наибольший суффикс  $x$ , являющийся одновременно также префиксом для  $y$ . Более того, вызов с тремя аргументами процедуры **OverLap**[ $x, y, z$ ], где  $z$  неопределенный символ, возвращает через  $z$  само наложение. При отсутствии наложения вызов процедуры возвращает **\$Failed**, тогда как при других ошибочных и особых ситуациях вызов процедуры возвращается *невычисленным*. При этом, в целом, *Math*-процедура **OverLap** оказывается реактивнее своего *Maple*-аналога. Между тем, вызов процедуры **LongestCommonSubString**[ $x, y$ ], одноименной с *Maple*-процедурой, возвращает список подстрок максимальной длины, общих для строк  $x, y$ . Процедура **LongestCommonSubSequence**[ $x, y$ ] подобна предыдущей, однако ее вызов возвращает список максимальных подпоследовательностей, общих для строк  $x$  и  $y$ . В то время как вызов функции **{PrefixQ | SuffixQ}**[ $x, y$ ] возвращает значение *True*, если строка  $x$  для  $y$ -строки является {префиксом | суффиксом}, в противном случае возвращается значение *False*. В свою очередь простая функция **PalindromeQ**[ $x$ ] возвращает *True*, если строка  $x$  является *палиндромом*, и значение *False* в противном случае. С другой стороны, по вызову процедуры **MaximalPalindromicSubstring**[ $x$ ] возвращается список из подстрок максимальной длины, являющихся *палиндромами* (перевертеньями).

Вызов процедуры **IsPermutation**[ $x$ ] возвращает *True*, если строка  $x$  есть *перестановка* из символов, ее составляющих, и *False* в противном случае; при этом, строка полагается *перестановкой*, если строка содержит строго по одному вхождению составляющих ее символов. Наконец, вызов одноименной с *Maple*-процедурой полезной процедуры **IsMonotonic**[ $x$ ] возвращает *True*, если все символы строки  $x$  согласно их *ASCII*-кодам находятся в порядке возрастания/убывания, и *False* в противном случае. При вызове процедуры **IsMonotonic**[ $x, t$ ] с двумя аргументами в случае основного результата *True* через второй неопределенный символ  $t$ , возвращается *порядок* следования символов в строке  $x$  {"Increase", "Decrease"}. При этом, в случае пустой строки  $x$  вызов процедуры **IsMonotonic**[ $x, t$ ] возвращает *True*; тогда как через символ  $t$  возвращается "Decrease". В остальных же случаях вызов процедуры **IsMonotonic** возвращается невычисленным.

В отличие от функции **StringFreeQ** вызов процедуры **StringDependQ**[ $x, y$ ] возвращает *True*, если строка  $x$  содержит вхождения подстроки либо подстрок, заданных списком  $y$ , и *False* в противном случае. Тогда как вызов процедуры **StringDependQ**[ $x, y, z$ ] при наличии третьего необязательного аргумента – неопределенной переменной – через него дополнительно возвращает список подстрок, не имеющих вхождений в строку  $x$ .

```
In[2611]:= StringDependQ[x_/, StringQ[x], y_/, StringQ[y] | | ListStrQ[y], z_] :=
Module[{a = Map3[StringFreeQ, x, Flatten[{y}]], b = {}, c = Length[y], k = 1},
If[DeleteDuplicates[a] == {False}, True, If[{z} != {} && Definition1[{z}][[1]] == "Null",
ListAssignP[{z}, 1, While[k <= c, b = Append[b, If[a[[k]], y[[k]]]; k++];
Select[b, # != "Null" &]]]; False]]
```

```
In[2612]:= Clear[t]; {StringDependQ["abcd", {"a", "d", "g", "s", "h", "t", "w"}, t], t}
Out[2612]= {False, {"g", "s", "h", "t", "w"}}
In[2613]:= Clear[t]; {StringDependQ["abgschtdw", {"a", "d", "g", "s", "h", "t", "w"}, t], t}
Out[2613]= {True, t}
```

Процедура имеет ряд полезных приложений в задачах обработки *строчных* структур. Фрагмент представляет исходный код процедуры с примерами ее использования.

В целом ряде задач обработки строчных структур возникает необходимость замены не просто подстроки, а подстроки, ограниченных заданными подстроками. Одну из таких задач решает процедура, вызов которой **StringReplaceS[S, s1, s2]** возвращает результат подстановки в строку *S* вместо вхождений в нее подстроки *s1*, ограниченных строками "*x*" (**StringLength["x"] = 1**) слева и справа из указанных множеств *L* и *R* соответственно, подстроками *s2*; в отсутствие таких вхождений вызов возвращает *S*. Нижеследующий фрагмент представляет исходный код процедуры **StringReplaceS** наряду с примером ее достаточно типичного использования.

```
In[2191]:= StringReplaceS[S_String, s1_String, s2_String] := Module[{a = StringLength[S],
    b = StringPosition[S, s1], c = {}, k = 1, p,
    L = Characters["`!@#%^&*(){}:\\"\\| <>?~-=+[];'. , 1234567890"],
    R = Characters["`!@#%^&*(){}:\\"\\| <>?~-=+[];'. , "],
    If[b == {}, S, While[k <= Length[b], p = b[[k]];
        If[Quiet[(p[[1]] == 1 && p[[2]] == a) ||
            (p[[1]] == 1 && MemberQ[R, StringTake[S, {p[[2]] + 1, p[[2]] + 1}])] ||
            (MemberQ[L, StringTake[S, {p[[1]] - 1, p[[1]] - 1}]) &&
            MemberQ[R, StringTake[S, {p[[2]] + 1, p[[2]] + 1}])] ||
            (p[[2]] == a && MemberQ[L, StringTake[S, {p[[1]] - 1, p[[1]] - 1}])],
            c = Append[c, p]]; k++]; StringReplacePart[S, s2, c]]

In[2192]:= StringReplaceS["abc& c + bd6abc - abc78*abc", "abc", "xyz"]
Out[2192]= "xyz& c + bd6xyz - abc78*xyz"
```

Данная процедура, в частности, достаточно полезна при обработке определений как блоков, так и модулей в плане работы с их аргументами и *локальными* переменными.

Следующая процедура представляет определенный интерес при программировании целого ряда задач, имеющих дело с определениями объектов в строчных форматах. В данном контексте вызов процедуры **ExtrName[x, n, p]** возвращает подстроку строки *x*, которая ограничена позицией *n* с одной стороны и включает лишь символы, которые допустимо использовать в составе имен объектов. Тогда как третий аргумент *p* задает направление выделения подстроки (*p* = 1 - *вправо* и *p* = -1 - *влево* от *p* позиции; при этом, символ в самой позиции *p* игнорируется) в исходной строке *x*.

```
In[2018]:= ExtrName[x_String, n_ /; IntegerQ[n] && n > 0, p_ /; MemberQ[{-1, 1}, p]] :=
    Module[{a, b, c = "", k, d = Flatten[{CharacterRange["a", "z"],
        CharacterRange["A", "Z"], "$",
```

```

CharacterRange["\[ScriptA]", "\[ScriptZ]"],
CharacterRange["\[ScriptCapitalA]", "\[ScriptCapitalZ]"],
CharacterRange["\[Alpha]", "\[Sampi]"],
CharacterRange["\[CapitalAlpha]", "\[CurlyPhi]"],
CharacterRange["\[GothicA]", "\[GothicZ]"],
Map[ToString, Range[0, 9]]],
For[k = If[p == 1, n + 1, n - 1], If[p == 1, k <= StringLength[x], k >= 1],
If[p == 1, k++, k--], a = StringTake[x, {k, k}];
If[MemberQ[d, a], If[p == 1, c = c <> a, c = a <> c]; Continue[], Break[]]; c]
In[2019]:= ExtrName["12345 xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h[a+b]; gh", 14, -1]
Out[2019]= "xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h"
In[2020]:= ExtrName["12345 xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h[a+b]; gh", 6, 1]
Out[2020]= "xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h"

```

Предыдущий фрагмент представляет как исходный код процедуры **ExtrName**, так и типичные примеры ее применения. Кстати, процедура **ExtrName** довольно несложно расширяется также на символы, допускаемые в качестве составляющих *имен* объектов в программной среде системы *Mathematica*.

В целом ряде случаев при обработке *строчных* конструкций необходимо извлекать из них подстроки, ограниченные символом {"", т.е. «строки в строках». Эту задачу решает процедура, чей вызов **StrFromStr[x]** возвращает список таких подстрок, находящихся в строке *x*; в противном случае вызов **StrFromStr[x]** возвращает пустой список, т.е. {}. В следующем фрагменте представлены как исходный код, так и примеры применения.

```

In[3050]:= StrFromStr[x_;/; StringQ[x]] := Module[{a = "", b, c = {}, k = 1},
b = DeleteDuplicates[Flatten[StringPosition[x, a]]];
For[k, k <= Length[b] - 1, k++,
c = Append[c, ToExpression[StringTake[x, {b[[k]], b[[k + 1]]}]]]; k = k + 1]; c]
In[3051]:= StrFromStr["12345\"678abc\"xyz\"910\"mnph"]
Out[3051]= {"678abc", "910"}
In[3052]:= StrFromStr["123456789"]
Out[3052]= {}

```

Вызов весьма простой процедуры **DelSuffPref[x, y, n]** обеспечивает возврат результата усечения строки *x* на подстроку *y* слева (*n* = 1), справа (*n* = 2) или с обоих концов (*n* = 3). Следующий фрагмент представляет исходный код процедуры с примерами.

```

In[3137]:= DelSuffPref[x_;/; StringQ[x], y_;/; StringQ[y], n_;/; MemberQ[{1, 2, 3}, n]] :=
Module[{a = StringLength[y]}, If[! SuffPref[x, y, n], x,
StringTake[x, {{a + 1, -1}, {1, -(a + 1)}, {a + 1, -(a + 1)}}][[n]]]]
In[3138]:= DelSuffPref["avzransianArtKravz", "avz", 1]
Out[3138]= "ransianArtKravz"

```

```
In[3139]:= DelSuffPref["avzransianArtKravz", "avz", 2]
Out[3139]= "avzransianArtKr"
In[3140]:= DelSuffPref["avzransianArtKravz", "avz", 3]
Out[3140]= "ransianArtKr"
```

Вызов процедуры **StrSub**[*x*, *y*, *z*, *n*] возвращает результат замены в строке *x* вхождений подстрок *y* на подстроки *z* согласно номерам вхождений *n*, определяемых или целым положительным числом или их списком. В случае отсутствия вхождений *y* в строку *x* или недопустимого значения *n* вызов процедуры **StrSub** возвращает значение **\$Failed**. Фрагмент представляет исходный код процедуры **StrSub** с примерами применения.

```
In[2226]:= StrSub[x_String, y_String, z_String, n_ /; PosIntQ[n] || PosIntListQ[n]] :=
Module[{a = StringPosition[x, y], b = Sort[Flatten[{n}]]},
If[a == {} || Length[a] < b[[-1]], $Failed, StringReplacePart[x, z, Part[a, b]]]
In[2227]:= StrSub["avzArtKravzRansIanavz", "avz", "agn66", {1, 3}]
Out[2227]= "agn66ArtKravzRansIanagn66"
In[2228]:= StrSub["avzArtKravzRansIanavz", "avz71", "agn66", {1, 3}]
Out[2228]= $Failed
In[2229]:= PrefCond[x_String, y_String] := Module[{a = Flatten[StringPosition[x, y]]},
If[a == {}, "", StringTake[x, {1, a[[1]] - 1}]]
In[2230]:= s = "rans ian 460 78"; s1 = " "; {PrefCond[s, s1], PrefCond["agn", "vsv"]}
Out[2230]= {"rans ian", ""}
```

Завершает фрагмент довольно простая, но полезная процедура **PrefCond**, чей вызов **PrefCond**[*x*, *y*] возвращает результат выделения из строки *x* подстроки, ограниченной началом строки *x* и первым вхождением в нее подстроки *y*; в противном случае вызов процедуры **PrefCond** возвращает пустую строку, т.е. "".

В отличие от функции **StringSplit**[*x*, *y*], вызов процедуры **StringSplit1**[*x*, *y*] выполняет семантическое разбиение строки *x* символом *y* на элементы возвращаемого списка. В данном случае семантизм сводится к тому, что в возвращаемый список помещаются лишь те подстроки строки *x*, которые содержат корректные выражения; в отсутствие таких подстрок вызов процедуры возвращает пустой список. Следующий фрагмент представляет исходный код процедуры **StringSplit1** наряду с некоторыми примерами ее применения. Процедура **StringSplit1** оказывается довольно полезным средством, в частности, при программировании средств обработки заголовков блоков, функций и модулей. Имеется целый ряд и других интересных приложений данной процедуры.

```
In[2250]:= StringSplit1[x_ /; StringQ[x], y_ /; StringQ[y] || StringLength[y] == 1] :=
Module[{a = StringSplit[x, y], b, c = {}, d, p, k = 1, j = 1},
d = Length[a]; Label[G]; For[k = j, k <= d, k++, p = a[[k]];
If[! SameQ[Quiet[ToExpression[p]], $Failed], c = Append[c, p], b = a[[k]];
For[j = k, j <= d - 1, j++, b = b <> y <> a[[j + 1]];
If[! SameQ[Quiet[ToExpression[b]], $Failed], c = Append[c, b]; Goto[G, Null]]]; c]
```

```
In[2251]:= StringSplit["x_String, y_Integer, z_/; MemberQ[{1, 2, 3, 4, 5}, z] | |  
IntegerQ[z], h_, s_String, s_/; StringQ[y] | | StringLength[y] == 1", ","]  
Out[2251]= {"x_String", " y_Integer", " z_/; MemberQ[{1, \" 2\", \" 3\", \" 4\", \" 5}\", \" z] | |  
IntegerQ[z]\", \" h_\", \" s_String\", \" s_/; StringQ[y] | | StringLength[y] == 1\"}  
In[2252]:= StringSplit1["x_String, y_Integer, z_/; MemberQ[{1, 2, 3, 4, 5}, z] | | IntegerQ[z],  
h_, s_String, h_/; StringQ[y] | | StringLength[y] == 1", ","]  
Out[2252]= {"x_String", " y_Integer", " z_/; MemberQ[{1, 2, 3, 4, 5}, z] | | IntegerQ[z]\", \" h_\",  
\"s_String\", \"h_/; StringQ[y] | | StringLength[y] == 1\"}  
In[2029]:= StringSplit2[x_String, y_String] := Module[{a = FromCharacterCode[17]},  
StringSplit[StringReplace[x, y -> a <> y], a]]  
In[2030]:= StringSplit["12345135187651897451", "1"]  
Out[2030]= {"2345", "35", "8765", "89745"}  
In[2031]:= StringSplit2["12345135187651897451", "1"]  
Out[2031]= {"12345", "135", "18765", "189745", "1"}
```

Сравнительный анализ **StringSplit** и **StringSplit1** говорит в пользу последней. Между тем, вызов процедуры **StringSplit2** $[x, y]$  в отличие от стандартной функции включает в возвращаемый список подстроки строки  $x$ , начинающиеся с разделителя  $y$ . Целый ряд задач, имеющих дело с обработкой строчных структур, делает довольно полезной процедуру **SubsStr** $[x, y, h, t]$ , чей исходный код с примером применения представляет нижеследующий фрагмент, а именно:

```
In[99]:= SubsStr[x_/; StringQ[x], y_/; StringQ[y], h_/; ListQ[h], t_/; MemberQ[{0, 1}, t]] :=  
Module[{a = Map[ToString, h], b}, If[StringFreeQ[x, y], Return[x],  
b = If[t == 1, Map3[StringJoin, y, a], Mapp[StringJoin, a, y]]];  
If[StringFreeQ[x, b], Return[x], StringReplace[x, Map9[Rule, b, h]]]
```

```
In[100]:= SubsStr["Module[{a$ = $CallProc, b$, c$}, x + StringLength[y] + b$*c$; b$ - c$;  
b$^c$; a$]", "$", {"", "]", "[", ""}, "", ";", "*", "^", "-"], 1]
```

```
Out[100]= "Module[{a = $CallProc, b, c}, x + StringLength[y] + b*c; b - c; b^c; a]"
```

Вызов процедуры **SubsStr**[*x*, *y*, *h*, *t*] возвращает результат замены в строке *x* вхождений подстроки, образованных конкатенацией (справа при *t*=1 или слева при *t*=0) подстроки *y* со строками из списка *h*, на строки из списка *h* соответственно. При невозможности проведения замены возвращается исходная строка *x*. Процедура **SubsStr** оказывается довольно полезным средством, например, при программировании средств обработки тела процедуры в строчном формате, содержащего локальные переменные. Тогда как вызов процедуры **SubsBstr**[*S*, *x*, *y*] возвращает список всех непересекающихся строк в строке *S*, ограниченных символами *x* и *y*, иначе возвращается пустой список, т.е. {}. В фрагменте представлен исходный код **SubsBstr** с примерами ее использования.

```
In[2438]:= SubsBstr[S_/, StringQ[S], x_/, CharacterQ[x], y_/, CharacterQ[y]] :=  
Module[{a = {}, c, h, n, m, s = S, p, t},  
c[s_ p_ t_] := DeleteDuplicates[Map10[StringFreeQ, s, {p, t}]] == {False};
```



```

While[c[s, x, y], n = StringPosition[s, x, 1][[1]][[1]]; s = StringTake[s, {n, -1}];
m = StringPosition[s, y, 1]; If[m == {}, Return[], m = m[[1]][[1]];
a = Append[a, h = StringTake[s, {1, m}]]; s = StringReplace[s, h -> ""]; Continue[]; a]
In[2439]:= SubstrBstr["123452333562675243655", "2", "5"]
Out[2439]= {"2345", "23335", "2675", "24365"}
In[2440]:= SubstrBstr["123452333562675243655", "9", "5"]
Out[2440]= {}

```

Нижеследующая процедура **SubStrSymbolParity** представляет несомненный интерес при обработке определений *блоков/функций/модулей*, заданных в строчном формате.

```

In[2533]:= SubStrSymbolParity[x_;/; StringQ[x], y_;/; CharacterQ[y], z_;/; CharacterQ[z],
d_;/; MemberQ[{0, 1}, d], t___;/; t == {} || PosIntQ[{t}][[1]]] :=
Module[{a, b = {}, c = {y, z}, k = 1, j, f, m = 1, n = 0, p, h},
If[{t} == {}, f = x, f = StringTake[x, If[d == 0, {t, StringLength[x]}, {1, t}]];
If[Map10[StringFreeQ, f, c] != {False, False} || y == z, Return[],
a = StringPosition[f, If[d == 0, c[[1]], c[[2]]]];
For[k, k <= Length[a], k++, j = If[d == 0, a[[k]][[1]] + 1, a[[k]][[2]] - 1]; h = If[d == 0, y, z];
While[m != n, p = Quiet[Check[StringTake[f, {j, j}], Return[$Failed]]];
If[p == y, If[d == 0, m++, n++];
If[d == 0, h = h <> p, h = p <> h],
If[p == z, If[d == 0, n++, m++];
If[d == 0, h = h <> p, h = p <> h],
If[d == 0, h = h <> p, h = p <> h]];
If[d == 0, j++, j--]];
b = Append[b, h]; m = 1; n = 0; h = ""; b]
In[2534]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 0]
Out[2534]= {"{abcdfgh}", "{rans}", "{ian}"}
In[2535]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 0, 7]
Out[2535]= {"{rans}", "{ian}"}
In[2536]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1]
Out[2536]= {"{abcdfgh}", "{rans}", "{ian}"}
In[2537]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1, 24]
Out[2537]= {"{abcdfgh}", "{rans}"}
In[2538]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1, -78]
Out[2538]= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1, -78]

```

Вызов процедуры **SubStrSymbolParity**[*x, y, z, d*] с *четырьмя* фактическими аргументами возвращает список подстрок строки *x*, ограниченных односимвольными строками {*y, z*} (*y* ≠ *z*); при этом, поиск таких подстрок производится слева направо при *d* = 0, тогда как при *d* = 1 поиск в строке *x* производится справа налево. При этом, вызов процедуры **SubStrSymbolParity**[*x, y, z, d, t*] с *пятым* *необязательным* аргументом, в качестве которого

выступает *целое* положительное число  $t > 0$ , обеспечивает поиск в подстроке  $x$ , которая ограничена позицией  $t$  и концом строки при  $d = 0$  и началом строки, и  $t$  при  $d = 1$ . При получении недопустимых фактических аргументов вызов процедуры возвращается невычисленным, тогда как при невозможности выделения требуемых подстрок вызов процедуры возвращает **\$Failed**. Предыдущий фрагмент представляет исходный код с наиболее типичными примерами применения процедуры **SubStrSymbolParity**. Эта процедура оказывается довольно полезной, в частности, в решении задач выделения в определениях процедур списка локальных переменных, заголовков процедур и т.д. Между тем, во многих случаях вполне можно обойтись более простой и реактивной версией этой процедуры, а именно процедурой, чей вызов **SubStrSymbolParity1** $[x, y, z]$  с 3 фактическими аргументами возвращает список подстрок строки  $x$ , ограниченных односимвольными строками  $\{y, z\}$  ( $y \neq z$ ); при этом, поиск таких подстрок выполняется слева направо. В отсутствие данных подстрок вызов возвращает пустой список, т.е.  $\{\}$ . Следующий фрагмент представляет исходный код процедуры **SubStrSymbolParity1** наряду с наиболее типичными примерами ее использования.

```
In[2023]:= SubStrSymbolParity1[x_String, y_ /; CharacterQ[y], z_ /; CharacterQ[z]] :=
Module[{a = DeleteDuplicates[Flatten[StringPosition[x, y]]], c = {}, d, k = 1, j, p,
      b = DeleteDuplicates[Flatten[StringPosition[x, z]]],
  If[a == {} || b == {}, {}, For[k, k <= Length[a], k++, p = StringTake[x, {a[[k]], a[[k]]}];
    For[j = a[[k]] + 1, j <= StringLength[x], j++, p = p <> StringTake[x, {j, j}];
    If[StringCount[p, y] == StringCount[p, z], c = Append[c, p]; Break[{}]; c]]

In[2024]:= SubStrSymbolParity1["Definition2[Function[{x, y}, x Sin[y]]", "{", "}"]
Out[2024]= {"{x, y}"}

In[2025]:= SubStrSymbolParity1["G[x_String, y_ z_ /; ListQ[z]]:= Block[{}, {x,y,z}]", "[", "]"
Out[2025]= {"[x_String, y_ z_ /; ListQ[z]]", "[z]", "[{, {x, y, z}]"}

```

Следующая достаточно простая процедура является весьма полезной модификацией процедуры **SubStrSymbolParity1**; ее вызов **StrSymbParity** $[S, s, x, y]$  возвращает список, чьими элементами являются подстроки строки  $S$ , имеющие формат  $sw$ , при условии паритета *минимального* числа вхождений в подстроку  $w$  символов  $x, y$  ( $x \neq y$ ). В случае отсутствия таких подстрок или идентичности символов  $x, y$  вызов возвращает пустой список, т.е.  $\{\}$ . Фрагмент представляет ее исходный код с примерами использования.

```
In[2175]:= StrSymbParity[S_String, S1_String, x_ /; StringQ[x] && StringLength[x] == 1,
      y_ /; StringQ[y] && StringLength[y] == 1] := Module[{b = {}, c = S1, d, k = 1, j,
      a = StringPosition[S, S1]},
  If[x == y || a == {}, {}, For[k, k <= Length[a], k++,
    For[j = a[[k]][[2]] + 1, j <= StringLength[S], j++, c = c <> StringTake[S, {j, j}];
    If[StringCount[c, x] != 0 && StringCount[c, y] != 0 &&
      StringCount[c, x] == StringCount[c, y], AppendTo[b, c]; c = S1; Break[{}]; b]]

In[2176]:= StrSymbParity["12345[678]9[abcd]", "34", "[", "]"]
Out[2176]= {"345[678]"}

```

```
In[2177]:= StrSymbParity["12345[6[78]9", "34", "[", "]"]
Out[2177]= {}
In[2178]:= StrSymbParity["12345[6[78]9", "34", "[", "]"]
Out[2178]= {}
In[2179]:= StrSymbParity["12345[678]9[ab34cd[x]34[a, b]", "34", "[", "]"]
Out[2179]= {"345[678]", "34cd[x]", "34[a, b]"}
```

Процедуры **SubStrSymbolParity**, **SubStrSymbolParity1** и **StrSymbParity** оказываются достаточно полезными, в частности, при обработке определений блоков и модулей, заданных в строчном формате. Используются рядом средств пакета *AVZ\_Package\_1*.

Нижеследующая процедура **SubsStrLim** представляет вполне определенный интерес в целом ряде приложений, которые достаточно существенно используют процедуру выделения из строк подстрок вполне определенного формата. Следующий фрагмент представляет исходный код процедуры **SubsStrLim** с примерами ее использования.

```
In[2542]:= SubsStrLim[x_;/; StringQ[x], y_;/; StringQ[y] && StringLength[y] == 1,
              z_;/; StringQ[z] && StringLength[z] == 1] :=
Module[{a, b = x <> FromCharacterCode[6], c = y, d = {}, p, j, k = 1, n, h},
  If[! StringFreeQ[b, y] && ! StringFreeQ[b, z], a = StringPosition[b, y];
    n = Length[a]; For[k, k <= n, k++, p = a[[k]][[1]]; j = p;
  While[h = Quiet[StringTake[b, {j + 1, j + 1}]]; h != z, c = c <> h; j++]; c = c <> z;
  If[StringFreeQ[StringTake[c, {2, -2}], {y, z}], d = Append[d, c]; c = y];
  Select[d, StringFreeQ[#, FromCharacterCode[6]] &]]

In[2543]:= SubsStrLim["1234363556aaa36", "3", "6"]
Out[2543]= {"36", "3556", "36"}
In[2544]:= SubsStrLim[DefOpt["SubsStrLim"], "{", "}"]
Out[2544]= {"{}", "{j + 1, j + 1}", "{2, -2}", "{y, z}" }
In[2545]:= SubsStrLim["1234363556aaa363", "3", "3"]
Out[2545]= {"343", "363", "3556aaa3", "363"}
```

Вызов **SubsStrLim**[*x*, *y*, *z*] возвращает список подстрок строки *x*, которые ограничены символами {*y*, *z*} при условии, что сами такие символы не входят в данные подстроки, исключая их концы. Фрагмент представляет исходный код процедуры с типичными примерами ее применения. Например, процедура **SubsStrLim** довольно полезна при необходимости выделения в строчном представлении определений блоков, функций и модулей целого ряда составляющих их компонент, ограниченных определенными символами, порой, существенно упрощая ряд процедур обработки их определений.

Тогда как вызов процедуры **SubsStrLim1**[*x*, *y*, *z*], являющейся полезной модификацией процедуры **SubsStrLim**, возвращает список подстрок строки *x*, которые ограничены символами {*y*, *z*} при условии, что сами данные символы либо не входят в подстроки, исключая их концы, или вместе с их концами имеют *одинаковое* число вхождений пар {*y*, *z*}. Процедура **SubsStrLim1** имеет целый ряд достаточно полезных приложений. В следующем фрагменте представлены исходный код процедуры **SubsStrLim1** наряду

с достаточно типичными примерами ее применения. Процедура **SubsStrLim1**, порой, весьма существенно упрощает ряд процедур обработки определений средств.

```
In[2214]:= SubsStrLim1[x_;/StringQ[x], y_;/StringQ[y] && StringLength[y] == 1,
              z_;/StringQ[z] && StringLength[z] == 1] :=
Module[{a, b = x <> FromCharacterCode[6], c = y, d = {}, p, j, k = 1, n, m, h},
  If[!StringFreeQ[b, y] && !StringFreeQ[b, z],
    a = StringPosition[b, y]; {n, m} = {Length[a], StringLength[x]};
    For[k, k <= n, k++, p = a[[k]][[1]];
    For[j = p + 1, j <= m, j++, h = StringTake[b, {j, j}];
    If[h != z, c = c <> h, c = c <> z];
  If[Length[DeleteDuplicates[Map10[StringCount, c, {y, z}]]] == 1,
    d = Append[d, c]; c = y; Break[], Continue[{}]]]; d]

In[2215]:= SubsStrLim1["art[kr[xyz]sv][rans]78[[460]]", "[", "]"]
Out[2215]= {"[kr[xyz]sv]", "[xyz]", "[rans]", "[[460]]", "[460]"}
In[2216]:= SubsStrLim1["G[x_] := Block[{a = 78, b = 460, c = 2013}, (a+b+c)*x]", "{", "}"]
Out[2216]= {"{a = 78, b = 460, c = 2013}"}
```

Строчные шаблоны довольно часто используются для *извлечения* некоторых структур из текстовых строк. В целом ряде случаев текстовые строки могут содержать символы пробела, перевода на новую строку, табуляции, которые не нужно учитывать, с такой целью в пакете определен шаблон *Whitespace*, определяющий последовательности из произвольной комбинации данных символов, как иллюстрирует следующий пример:

```
In[2225]:= StringReplace["ab \ncd fg hk \nm np\nagn", WhitespaceCharacter -> ""]
Out[2225]= "abcdfghkmnpagn"
```

В котором обеспечивается уплотнение строки за счет удаления из нее всех указанных символов. В данном свете представляется довольно полезной процедура **RedSymbStr**, чей вызов **RedSymbStr[x, y, z]** возвращает результат замены всех подстрок, состоящих из символа *y*, строки *x* на символ либо строку *z*. В случае отсутствия вхождений *y* в *x*, вызов процедуры возвращает строку *x* неизменной. Следующий фрагмент приводит исходный код процедуры **RedSymbStr** наряду с примерами ее использования.

```
In[2202]:= RedSymbStr[x_String, y_;/SymbolQ1[y], z_String] :=
Module[{a = StringPosition[x, y], b},
  If[StringFreeQ[x, y], x, b = Map[#[[1]] &, a]];
  b = Sort[DeleteDuplicates[Map[Length, Split[b, #2 - #1 == 1 &]]], Greater];
  b = Mapp[Rule, Map3[StringMultiple, y, b], z]; StringReplace[x, b]]

In[2203]:= RedSymbStr["a b c d ef gh x y z", " ", " "]
Out[2203]= "a b c d ef gh x y z"
In[2204]:= RedSymbStr["a b c d ef gh x y z", " ", ""]
Out[2204]= "abcdefghxyz"
```

```
In[2205]:= RedSymbStr["a b c d ef gh x y z", "", "GGG"]
Out[2205]= "aGGGbGGGcGGGdGGGefGGGghGGGxGGGyGGGz"
In[2206]:= RedSymbStr["a b c d ef gh x y z", "x", "GGG"]
Out[2206]= "a b c d ef gh GGG y z"
```

Строчные шаблоны, как правило, применяются к подстрокам, находящимся в любой позиции строки. Между тем, в некоторых случаях целесообразно указать, что данные шаблоны могут применяться только к подстрокам в определенных позициях. Можно сделать это включением символов **StartOfString** в строчных шаблонах, тогда как для обозначения конца строки в шаблонах используется символ **EndOfString**. Более того, система располагает еще рядом средств программирования строчных шаблонов, здесь не рассматриваемым. Заинтересованный читатель отсылается к работам [100-102,118] либо к справке по системе. Механизм строчных шаблонов обеспечивает достаточно серьезный метод производить *разнообразную* обработку строчных структур. При этом, знакомство со специальными языками обработки строк во многих случаях позволяет определять строчные шаблоны с помощью нотации *регулярных* выражений, которые определяются в системе на базе функции **RegularExpression**. Данная функция может применяться для представления типов строк в функциях **StringMatchQ**, **StringCases**, **StringReplace** и **StringSplit**. Функция **RegularExpression** поддерживает стандартный синтаксис *регулярных* выражений формата, который используется в типичных языках программирования обработки строк, в частности, в языке **Perl**. Использование в среде системы *регулярных* выражений для расширения механизма *строчных* шаблонов здесь не рассматривается и читатель отсылается к указанным выше изданиям.

Следующая процедура в определенной степени носит вспомогательный характер, но в то же время расширяет стандартную функцию **StringTrim**, имея и самостоятельное значение. Вызов **StringTrim1**[*x*, *y*, *z*] возвращает результат усечения строки *x* слева на подстроку *y* и справа на подстроку *z*. При условии *y* = *z* = "" вызов **StringTrim1**[*x*, "", ""] эквивалентен вызову **StringTrim**[*x*]. Процедуру **StringTrim1** можно достаточно легко расширить на случай списков в качестве аргументов *y* и *z*, что оставляется читателю в качестве полезного упражнения. Следующий фрагмент представляет исходный код и примеры достаточно типичного использования процедуры **StringTrim1**.

```
In[2105]:= StringTrim1[x_String, y_/, StringQ[y], z_/, StringQ[z]] := Module[{a = x},
  If[y == z == "", a = StringTrim[a], If[SuffPref[a, y, 1], a = StringReplace[a, y -> "", 1]];
  If[SuffPref[a, z, 2], a = StringReplace[a, z -> "", StringCount[a, z]]]; a]
In[2106]:= StringTrim1["1112222222333", "111", "333"]
Out[2106]= "2222222"
In[2107]:= StringTrim1["1112222222333", "", "333"]
Out[2107]= "1112222222"
In[2108]:= StringTrim1[" 1112222222333 ", "", ""]
Out[2108]= "1112222222333"
```

Строки, генерируемые ранее рассмотренной процедурой **ToString1**, подобно случаю списков можно назвать **StringString** (*строки строк*) и для их тестирования может быть

В задачах обработки строчных структур достаточно полезной оказывается процедура **StringMultiple**, которая обеспечивает кратную конкатенацию строк. Вызов процедуры **StringMultiple[s, p]** возвращает строку - результат *p*-кратной конкатенации *s*-строки. В то время как вызов процедуры **StringMultiple[s, p, h]** на трех аргументах (*h* в качестве 3-го необязательного аргумента выступает произвольное выражение, в частности, строка) возвращает результат *p*-кратной конкатенации *s*-строки, разделенной *h* в строчном формате. В случае получения недопустимого кортежа фактических аргументов вызов процедуры возвращается невычисленным, как иллюстрирует следующий фрагмент, представляющий исходный код процедуры наряду с примерами ее использования.

С целью упрощения реализуемого процедурой **DefInPackage** алгоритма уже по ходу программирования оказалось целесообразным определить процедуру, вызов которой **SubsPosSymb** $[x, n, y, z]$  возвращает подстроку строки  $x$ , ограниченную справа (*слева*) позицией  $n$ , и слева (*справа*) символом из списка  $y$ ; при этом, поиск в строке  $x$  делается *слева направо* ( $z = 0$ ) и *справа налево* ( $z = 1$ ). Тогда как вызов на недопустимых аргументах возвращается невычисленным. Исходный код процедуры **SubsPosSymb** с примерами

ее довольно типичного использования представляет нижеследующий фрагмент.

```
In[2042]:= SubsPosSymb[x_;/ StringQ[x], n_;/ PosIntQ[n], y_;/ ListQ[y] &&
DeleteDuplicates[Map[CharacterQ, y]] == {True}, z_;/ z == 0 || z == 1] :=
Module[{a = "", k = n, b}, If[n > StringLength[x], Return[Defer[SubsPosSymb[x, n, y, z]]],
While[If[z == 0, k >= 1, k <= StringLength[x]], b = StringTake[x, {k, k}];
If[! MemberQ[y, b], If[z == 0, a = b <> a, a = a <> b], Break[]]; If[z == 0, k--, k++]; a]]
In[2043]:= SubsPosSymb["123456789abcdfght", 5, {"g"}, 1]
Out[2043]= "56789abcdf"
In[2044]:= SubsPosSymb["123456789abcdfght", 16, {"z"}, 0]
Out[2044]= "123456789abcdfgh"
```

Данная процедура представляет определенный интерес в качестве самостоятельного средства обработки строчных структур, что и характеризуют ее приложения. Между тем, механизм, использованный при программировании процедуры **DefInPackage** и состоящий в создании под него процедуры **SubsPosSymb**, является вполне типичным при разработке программного обеспечения пользователя, когда возникает некоторая целесообразность создания собственного программного инструментария, который не только расширяет собственно сам набор стандартных средств *Mathematica*, но также и их функциональные характеристики. Именно таким способом в значительной мере и создавались оба наших пакета *AVZ\_Package* [90] и *AVZ\_Package\_1* [120].

Достаточно простая процедура **ListStrToStr** представляет несомненный интерес при работе со списками в строчном формате, точнее, вызов процедуры **ListStrToStr[x]**, где аргумент *x* имеет формат {"a", "b", "c", ...}, конвертирует *x* в строку формата "a, b, c, ...", если вызов процедуры использует единственный фактический аргумент *x*; если вызов процедуры использует в качестве второго аргумента произвольное выражение, вызов возвращает строку формата "abcd ...". Следующий фрагмент представляет исходный код процедуры **ListStrToStr** с наиболее типичными примерами ее использования.

```
In[2328]:= ListStrToStr[x_;/ ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True},
p___] := Module[{a = ""}, If[{p} == {}, Do[a = a <> x[[k]] <> ", ",
{k, Length[x]}]; StringTake[a, {1, -3}], StringJoin[x]]]
In[2329]:= ListStrToStr[{"a", "b", "c", "d", "h", "t", "k", "Art", "Kr"}]
Out[2329]= "a, b, c, d, h, t, k, Art, Kr"
In[2330]:= ListStrToStr[{"(a+b)", "*", "Sin[x]", " - ", "(c - d)", "*", "Cos[y]", " == ", "460"}, 6]
Out[2330]= "(a+b)*Sin[x] - (c - d)*Cos[y] == 460"
```

Следующая функция обеспечивает тестирование строки *x* на предмет наличия в ней корректного выражения; при этом, тестируется вся строка полностью. Вызов **ExprQ[x]** возвращает **True**, если строка *x* содержит выражение, и **False** в противном случае.

```
In[2051]:= ExprQ[x_String] := If[Quiet[ToExpression[x]] === $Failed, False, True]
In[2052]:= Map[ExprQ, {"(a+b)", "(a^c+b)", "a\\b+c", "a**b+cos[x]/c", a + b}]
Out[2052]= {True, False, False, True, ExprQ[a + b]}
```

```
In[2064]:= CharacterQ[c_] := !NumberQ[c] && StringLength[ToString[c]] == 1
In[2065]:= Map[CharacterQ, {Agn, "78", G, S, "V", 6}]
Out[2065]= {False, False, True, True, True, False}
```

Процедура **ExprQ** оказывается полезной при программировании процедур/функций (прежде всего, системного характера), ориентированных на обработку различного рода выражений, представленных в строчном формате. Тогда как вызов простой функции **CharacterQ[x]** возвращает **True**, если *x* определяет символ с *характеристическим* кодом из диапазона 0 .. 255, в противном случае возвращается **False**. Выше представлен код с примером применения. Функция **CharacterQ** имеет ряд важных приложений. Вызов процедуры **MixCaseQ[x]** возвращает **True**, если строка *x* содержит буквы (допустимы специальные символы) на разных регистрах, и **False** в противном случае; если же строка *x* содержит только специальные символы, то возвращается "Special Characters"; ниже представлен исходный код процедуры наряду с примерами ее использования.

```
In[2014]:= MixCaseQ[x_ /; StringQ[x]] := Module[{a, b, k}, {a, b} = {Characters[x], {}};
For[k = 1, k <= Length[a], k++, If[! LetterQ[a[[k]]], Null,
b = Append[b, If[UpperCaseQ[a[[k]]], 1, 2]]];
b = Length[DeleteDuplicates[b]];
If[b == 0, "Special Characters", If[b == 1, False, True]]]
In[2015]:= Map[MixCaseQ, {Agn, "Asv", "RANS", "j", "∞"}]
Out[2015]= {MixCaseQ[Agn], True, False, "Special Characters", "Special Characters"}
```

В целом ряде случаев достаточно полезной оказывается процедура, обеспечивающая конвертацию строк определенной структуры в списки строк. В частности, подобные задачи возникают при обработке формальных аргументов и локальных переменных. Данную задачу эффективно решает процедура **StrToList**, представленная ниже.

Процедура предназначена для конвертации строк структуры "{xxxxxxx ... x}" в список строк, полученных из строки "xxxxxxx ... x", разделенной символами запятой ",". При отсутствии в исходной строке обоих ограничивающих символов {"", ""} строка будет конвертироваться в список символов в соответствии с вызовом **Characters["xxxxx ... x"]**. Следующий фрагмент представляет исходный код процедуры **StrToList** с ее наиболее типичными примерами использования.

```
In[2190]:= StrToList[x_ /; StringQ[x]] := Module[{a, b = {}, c = {}, d, h, k = 1, j,
y = If[StringTake[x, {1, 1}] == "{" &&
StringTake[x, {-1, -1}] == "}", StringTake[x, {2, -2}], x]},
a = DeleteDuplicates[Flatten[StringPosition[y, "="] + 2];
d = StringLength[y];
If[a == {}, Map[StringTrim, StringSplit[y, ","],
While[k <= Length[a], c = ""; j = a[[k]];
For[j, j <= d, j++, c = c <> StringTake[y, {j, j}];
If[! SameQ[Quiet[ToExpression[c]], $Failed] &&
```



```

(j == d || StringTake[x, {j + 1, j + 1}] == ","),
b = Append[b, c -> ToString[Unique[$Art$Kr$]]]; Break[]]; k++];
h = Map[StringTrim, StringSplit[StringReplace[y, b, ","]];
Mapp[StringReplace, h, RevRules[b]]]]

In[2191]:= StrToList["Kr, a = 78, b = {x, y, z}, c = {n, m, {42, 47, 67}}"]
Out[2191]= {"Kr", "a = 78", "b = {x, y, z}", "c = {n, m, {42, 47, 67}}"}
In[2192]:= StrToList["{a, b = 78, c = {m, n}}"]
Out[2192]= {"a", "b = 78", "c = {m, n}"}
In[2193]:= Map[StrToList, {{a, b, c, d}}, {"a, b, c, d"}]
Out[2193]= {{{"a", "b", "c", "d"}, {"a", "b", "c", "d"}}}
In[2194]:= RevRules[x_ /; RuleQ[x] || ListQ[x] && DeleteDuplicates[Map[RuleQ, x]] ==
{True}] := Module[{a = Flatten[{x}], b},
b = Map[#[[2]] -> #[[1]] &, a]; If[Length[b] == 1, b[[1]], b]]
In[2195]:= RevRules["{x, y, z}" -> "ArtKr$18438"]
Out[2195]= "ArtKr$18438" -> "{x, y, z}"
In[2196]:= RevRules[{x -> a, y -> b, z -> c, h -> g}]
Out[2196]= {a -> x, b -> y, c -> z, g -> h}

```

Процедура предназначена для конвертации строк структуры "{xx ... x}" или "xx ... x" в список строк, полученных из строк указанного формата, разделенных символами "=" и/или запятой ",". Примеры фрагмента довольно наглядно иллюстрируют принцип выполнения процедуры и формат возвращаемых ею результатов конвертации строк. Процедура достаточно полезна в целом ряде задач обработки *локальных* переменных. Более того, завершает фрагмент довольно простая и, вместе с тем, полезная процедура, чей вызов **RevRules[x]** возвращает правило или список правил, обратных к правилам, определенным аргументом *x* – правило формата *a -> b* либо их список.

Для целого ряда задач обработки строк и символов весьма целесообразно определить две новые функции, а именно: **SymbolGreater** и **SymbolLess** в качестве аналогов для системных функций операций **Greater** и **Less** соответственно, аргументами которых выступают *числовые* значения. Тогда как в качестве аргументов функций **SymbolLess** и **SymbolGreater** выступают символы и *сравнение* аргументов производится на основе их кодов. Вызов **SymbolLess[x, y]** (**SymbolGreater[x, y]**) возвращает **True**, если  $x < y$  ( $x > y$ ) в упомянутом контексте, и **False** в противном случае. Фрагмент приводит исходные коды обеих функций наряду с типичными примерами их применения. В частности, данные функции полезны в задачах специальной сортировки символьных списков.

```

In[2010]:= SymbolGreater[x_ /; SymbolQ[x], y_ /; SymbolQ[y]] :=
Greater[ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[x]]]],
ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[y]]]]]]
In[2011]:= Map7[SymbolGreater, Sequences, {{Avz, Agn}, {Rans, Ian}, {"a", "c"}, {"x", "z"}}]
Out[2011]= {True, True, False, False}

```

```
In[2014]:= SymbolLess[x_;/; SymbolQ[x], y_;/; SymbolQ[y]] :=
    Less[ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[x]]]],
        ToExpression[StringJoin[Map[ToString, ToCharacterCode[ToString[y]]]]]]
In[2015]:= Map7[SymbolLess, Sequences, {{Avz, Agn}, {Rans, Ian}, {"a", "c"}, {"x", "z"}}]
Out[2015]= {False, False, True, True}
In[2016]:= Sort[{Rans, IAN, Avz, Agn, Vsv, Kr, Art}, SymbolGreater[#1, #2] &]
Out[2016]= {Rans, Vsv, Avz, Art, Agn, IAN, Kr}
In[2017]:= Sort[{Rans, IAN, Avz, Agn, Vsv, Kr, Art}, SymbolLess[#1, #2] &]
Out[2017]= {Kr, IAN, Agn, Art, Avz, Vsv, Rans}
```

Из последних 2 примеров применения функций видно, реализуемая ими сортировка в общем случае отличается от классической лексикографической сортировки, что для целого ряда довольно важных приложений оказывается достаточно полезным.

Вызов простой функции **SortQ[s]** возвращает **True**, если *s* является отсортированной посимвольно строкой, и **False** в противном случае. Функция представляет интерес в задачах, связанных с обработкой строк и выражений в строчном формате. Фрагмент представляет исходный код функции **SortQ** наряду с примерами ее использования.

```
In[2019]:= SortQ[s_;/; StringQ[s]] := SameQ[s, StringJoin[Sort[Characters[s]]]]
In[2020]:= Map[SortQ, {"dsertyuhgfd", "123a4b54c321d", "123456789abcd"}]
Out[2020]= {False, False, True}
```

Следующая процедура оказывается довольно полезной в задачах обработки строчных структур. Вызов процедуры **SortRevStr[x, y]** возвращает результат обращения строки *x*, если *y* = *Reverse*, или результат посимвольной сортировки строки *x* в возрастающем порядке, если *y* = *Sort*; тогда как вызов **SortRevStr[x, y, z]** с третьим необязательным *z* = *SymbolGreater* аргументом возвращает результат посимвольной сортировки строки *x* в убывающем порядке. Следующий фрагмент представляет исходный код процедуры **SortRevStr** наряду с наиболее типичными примерами ее использования.

```
In[2153]:= SortRevStr[x_;/; StringQ[x], y_;/; MemberQ[{Reverse, Sort}, y], z_] :=
    Module[{a = Characters[x]}, If[y === Reverse, StringJoin[y[a]],
        StringJoin[If[{z} != {} && z === SymbolGreater, Sort[a, z[#1, #2] &], Sort[a]]]]]
In[2154]:= SortRevStr["qwertyuiopasdfghjklzxcvbnm", Reverse]
Out[2154]= "mnbvcxzlkhgfsapoiuytrewq"
In[2155]:= SortRevStr["qwertyuiopasdfghjklzxcvbnm", Sort]
Out[2155]= "abcdefghijklmnopqrstuvwxyz"
```

Процедуры **LeftFold** и **RightFold** представляют аналоги одноименных *Maple*-процедур. Вызов **RightFold[F, id, s]** выполняет итерацию процедуры *F* над строкой *s*, возвращая композицию последовательных результатов справа вместе с начальным значением *id*; в свою очередь вызов **LeftFold[F, id, s]** выполняет итерацию процедуры *F* над строкой *s*, составляя композицию результатов слева вместе с начальным значением *id*. Фрагмент представляет исходные коды обеих процедур наряду с примерами их применения.

```

In[2017]:= LeftFold[F_, id_, s_ /; StringQ[s]] := Module[{a = StringLength[s], c = "", k = 1,
    b = ToString[F] <> "[", For[k, k <= a, k++,
    c = c <> b <> ToString1[StringTake[s, {k, k}]] <> ", "];
    ToExpression[c <> ToString[id] <> CatN[""], a]]

In[2018]:= LeftFold[F, Z, "abcdefgh"]
Out[2018]= F["a", F["b", F["c", F["d", F["e", F["f", F["g", F["h", Z]]]]]]]]]]]]

In[2019]:= RightFold[F_, id_, s_ /; StringQ[s]] := Module[{a = StringLength[s], c = "", k = 1,
    b = ToString[F] <> "[",
    For[k, k <= a, k++, c = c <> b <> ToString1[StringTake[s, {-k, -k}]] <> ", "];
    ToExpression[c <> ToString[id] <> CatN[""], a]]

In[2020]:= RightFold[F, Z, "abcdefgh"]
Out[2020]= F["h", F["g", F["f", F["e", F["d", F["c", F["b", F["a", Z]]]]]]]]]]]]

```

Как системные, так и наши средства обработки строчных структур, представленные в настоящей книге, образуют эффективный инструментарий для обработки объектов данного типа. Так, следующий фрагмент представляет процедуру **StructProcFunc**, чей вызов **StructProcFunc[x]** возвращает простой либо вложенный список, чьи элементы в зависимости от типа {"Block", "Module", "Function"} фактического аргумента  $x$  имеют формат {Type, Heading, Locals, Body} для {"Block", "Module"} и {Type, Heading, Body} для "Function"; при этом, под функцией понимается такой объект  $x$ , что **QFunction[x]=True**. Фрагмент представляет исходный код процедуры и пример применения, из которых формат возвращаемого процедурой **StructProcFunc** результата предельно прозрачен.

```

In[3223]:= StructProcFunc[x_ /; ProcQ[x] | QFunction[x]] := Module[{c, d, h = {}, p, k = 1, t,
    a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}],
    c = Map9[StringReplace, a, Map[StringJoin[#, " := " ] -> "" &, b]];
    While[k <= Length[b], d = c[[k]]; If[SuffPref[d, "Module[", 1], t = "Module",
        If[SuffPref[d, "Block[", 1], t = "Block", t = ""]];
        If[t != "", AppendTo[h, {t, b[[k]], p = SubStrSymbolParity1[d, {"(", "}"}][[1]];
        StrToList[p], StringReplace[StringTake[d, {1, -2}], t <> "[" <> p <> ", " -> ""]]],
        AppendTo[h, {"Function", b[[k]], StringReplace[d, b[[k]] <> " := " -> ""]]]; k++;
        If[Length[h] == 1, h[[1]], h]]

In[3224]:= Agn[x_] := Block[{a = 78, b = 460}, x^2*a*b]; Agn[x_, y_] := x + y
In[3225]:= Agn[x_, y_, z_] := Module[{a = 78}, a*(x + y + z)]
In[3226]:= StructProcFunc[Agn]
Out[3226]= {{{"Block", "Agn[x_]", {"a = 78", "b = 460"}, "x^2*a*b"}, {"Function", "Agn[x_, y_]",
    "x + y"}, {"Module", "Agn[x_, y_, z_]", {"a = 78"}, "a*(x + y + z)"}}}
In[3227]:= Avz[x_] := Module[{a = 78, b = Stack[_]}, a + x; b; $InBlockMod]
In[3228]:= StructProcFunc[Avz]
Out[3228]= {"Module", "Avz[x_]", {"a = 78", "b = Stack[_]", "a + x; b; $InBlockMod"}}

```

Процедура выявляет общую структурную организацию объектов  $x$  указанного типа. Вышеприведенные средства обработки *строчных* структур, аналогичные средствам *Maple*, базировались как на достаточно широко используемых стандартных средствах системы *Mathematica*, так и на наших средствах, представленных в настоящей книге, весьма наглядно демонстрируя относительную простоту программирования в среде *Math*-языка средств, аналогичных средствам *Maple* как своего основного конкурента. Между тем, наличие в *Mathematica* достаточно развитого набора средств для работы с символьными строчными *шаблонами* позволяют создавать достаточно эффективные и развитые системы обработки *строчных* структур, которые по целому ряду довольно важных показателей превосходят возможности *Maple*. При этом, *строчные* шаблоны в *Mathematica* используют те же принципы, что и для представления шаблонов строк текста. Пользователь может представлять строчную структуру, как *последовательность* символов, и применять к ней *общие* механизмы шаблонов. Более того, дополнительно к общим шаблонам пользователю предоставляется набор строчных шаблонов более специального назначения, существенно расширяя возможности обработки строчных структур. Мы не будем акцентировать внимание на этом вопросе, отсылая читателя к соответствующей литературе и справочной системе *Mathematica*. Итак, наш опыт по использованию данных систем для программирования средств (как *прикладных*, так и *системных*) обработки *строчных* структур показал, что стандартные средства *Maple* по ряду существенных показателей уступают однотипным средствам *Math*-языка; более того, именно *Math*-язык предоставляет *набор* более развитых и эффективных средств по программированию как *прикладных* средств, включающих обработку строк, так и *системных*, расширяющих стандартные средства данного назначения. Ряд примеров данного типа приведен выше. Для системы *Maple* нами также был создан целый ряд средств обработки *строчных* структур [45], ряд из которых (*либо их аналоги*) появились впоследствии в более старших релизах *Maple*; между тем, программирование средств данного типа в среде *Math*-языка оказывается более простым не только в связи с более развитыми средствами для решения задач этого типа, но и *процедурно-функциональной парадигмой* языка, допускающего использование аппарата чистых функций. При этом, запрограммированные в *Math*-языке средства обработки *строчных* структур не только более эффективны во временном отношении, но и для их программирования система *Mathematica* предлагает более развитые функциональные средства. Дополнительно к представленным здесь и ранее средствам обработки строчных структур достаточно мощный механизм строчных шаблонов, позволяющий говорить о *шаблонной* форме программирования, предоставляет достаточно развитые средства обработки строк по уровню, существенно не уступающие *специализированным* языкам обработки текстов. Детальнее с механизмом *шаблонов* возможно познакомиться либо в справочной базе системы, либо в книгах [57-60,100,101]; данный механизм достаточно эффективен при создании различного типа функциональных конструкций, ориентированных как на символьные, так и на численные вычисления. Ряд поучительных иллюстрационных примеров использования рассмотренного механизма приведен в книгах [98,100,101], а также в настоящей книге и пакете *AVZ\_Package\_1* [120]. Итак, в главе представлен ряд средств, расширяющих стандартные функции системы, ориентированные на работу со *строчными* структурами в самом широком их понимании.

## Глава 6. Дополнительные средства обработки списков и последовательностей в программной среде системы *Mathematica*

При программировании многих приложений является целесообразным использовать не отдельные выражения, а их совокупности, оформляемые в форме *списков*, формат которых очень прост, а именно: последовательность выражений, разделенных знаком запятой «*,*», берется в фигурные скобки, например,  $\{a, b, c, d\}$ . При такой организации вместо выполнения вычислений над отдельными выражениями имеется возможность выполнять требуемые операции как над *списками*, как едиными объектами, так и над их отдельными элементами. Списки различных типов представляют собой важную и одну из наиболее часто используемых структур в программной среде *Mathematica*. В программной среде системы многие функции имеют атрибут *Listable*, говорящий о том, что блок, функция, модуль либо оператор *F* с данным атрибутом автоматически применимы к каждому элементу списка, используемого соответственно в качестве их аргумента или операнда. Для тестирования наличия у объекта этого атрибута может использоваться довольно простая функция, вызов которой с примерами применения представляет нижеследующий фрагмент, а именно:

```
In[2250]:= ListableQ[x_] := MemberQ[Quiet[Check[Attributes[x], {}]], Listable]
In[2251]:= Map[ListableQ, {78, Sin, Power, Plus, Log, a + b}]
Out[2251]= {False, True, True, True, True, False}
In[2252]:= a = N[{78, 42, 2013}]; {a, Sin[a]}
Out[2252]= {{78., 42., 2013.}, {0.773891, -0.916522, 0.981986}}
In[2253]:= {a^2, a + 78, a + {x1, y1, z1}}
Out[2253]= {{6084., 1764., 4.05217*10^6}, {156., 120., 2091.}, {78. + x1, 42. + y1, 2013. + z1}}
In[2269]:= a = {78, 42, 2013}; a + {x1, x2}
Thread::tdlen: Objects of unequal length in {78, 42, 2013} + {x1, x2} cannot
be combined. >>
Out[2269]= {x1, x2} + {78, 42, 2013}
In[2270]:= a^{x1, x2}
Thread::tdlen: Objects of unequal length in {78, 42, 2013}^{x1, x2} cannot
be combined. >>
Thread::tdlen: Objects of unequal length in {Log[78], Log[42], Log[2013]} {x1, x2}
cannot be combined. >>
Out[2270]= {78, 42, 2013}^{x1, x2}
In[2271]:= ListOp[a, {x1, x2}, Power]
Out[2271]= {78^x1, 42^x2, 2013}
In[2272]:= ListOp[{x1, x2}, a, Plus]
Out[2272]= {78 + x1, 42 + x2, 2013}
```

Между тем, для ряда операций, имеющих *Listable*-атрибут, требуется соответствие по длине списков-операндов, иначе инициируются соответствующие ошибочные либо особые ситуации, как иллюстрируют примеры фрагмента. И лишь процедура *ListOp*,

рассматриваемая несколько ниже, позволяет устранять подобные ситуации. Поясним данную ситуацию несколько детальнее, учитывая важность *Listable*-атрибута.

Как уже ранее отмечалось, *списки* являются одними из *центральных* структур в пакете *Mathematica*, используемые для представления совокупностей, множеств, массивов и последовательностей всех видов. Списки могут иметь любую структуру и достаточно большой размер. Свыше тысячи встроенных функций пакета *Mathematica* оперирует непосредственно со *списочными* структурами, делая *списки* весьма мощным средством как *символьной*, так и *численной* обработки данных. В отличие от *Maple*, *Mathematica* располагает целым рядом достаточно эффективных функций работы со *списочными* структурами различного вида, в частности, большинство *встроенных* функций пакета имеют атрибут *Listable*, который определяет то, что они эффективно применимы по отдельности к каждому элементу списка, а в случае их множества к соответствующим элементам каждого списка. В то же время все аргументы *List*-типа в функции такого типа должны быть *идентичной* длины, в противном случае инициируется ошибочная или особая ситуация, как довольно наглядно было проиллюстрировано предыдущим фрагментом. С целью устранения такого достаточно существенного недостатка была предложена процедура **ListOp**[*x, y, z*], вызов которой возвращает список, элементами которого являются результаты *применения z-процедуры/функции* к соответствующим элементам списков *x* и *y*; при этом, в случае *различных* длин таких списков описанная процедура применяется к обоим спискам в рамках минимальной длины списков, при этом оставляя остальные элементы большего списка без изменения. Нижеследующий фрагмент представляет исходный код процедуры **ListOp** и примеры ее применения, достаточно наглядно иллюстрирующие сказанное. Процедура **ListOp** в значительной степени предполагает в качестве 3-го аргумента *чистые* функции, что в значительной мере позволяет расширить класс функций в качестве *третьего* аргумента, например, на логические функции, как иллюстрируют последние три примера фрагмента.

```
In[2413]:= ListOp[x_/, ListQ[x], y_/, ListQ[y], z_/, HowAct[z]] := Module[{a = Length[x],
    b = Length[y], c, d = {}, k = 1, c = Min[a, b];
    For[k, k <= c, k++, d = Append[d, z[x[[k]], y[[k]]]];
    Flatten[{d, x[[c + 1 ;; -1]], y[[c + 1 ;; -1]]}]

In[2414]:= ListOp[{x, y, z, h, w}, {a, b, c, d, e}, #1 + #2 &]
Out[2414]= {a + x, b + y, c + z, d + h, e + w}
In[2415]:= ListOp[{x, y, z, h, w}, {a, b, c}, #1^#2 &]
Out[2415]= {x^a, y^b, z^c, h, w}
In[2416]:= ListOp[{x, y, z, h, w}, {a, b, c, d, e}, #1 * #2 &]
Out[2416]= {a x, b y, c z, d h, e w}
In[2417]:= ListOp[{x, y, z, h, w}, {}, #1 * #2 &]
Out[2417]= {x, y, z, h, w}
In[2418]:= {True, False, True} && {True, True, False}
Out[2418]= {True, False, True} && {True, True, False}
In[2419]:= ListOp[{True, False, True}, {True, True, False}, #1 && #2 &]
Out[2419]= {True, False, False}
```

```
In[2420]:= ListOp[{True, False, True}, {True, True, False}, #1 || #2 &]
Out[2420]= {True, True, True}
```

Вызов функции **ListableQ**[*x*] возвращает **True**, если объект *x* имеет **Listable**-атрибут, и **False** в противном случае. В принципе, любой процедуре/функции арности 1 можно приписать **Listable**-атрибут, обеспечивая ее корректный вызов на списках в качестве ее фактического аргумента, как наглядно иллюстрирует следующий фрагмент:

```
In[2569]:= ProcQ[{ProcQ, ListOp, DefOp, Globals, DirName, Mapp, Attrib}]
          SetAttributes::sym: Argument {} at position 2 is expected to be a symbol. >>
Out[2569]= False
In[2570]:= SetAttributes[ProcQ, {Listable}]
In[2571]:= ProcQ[{ProcQ, ListOp, DefOp, Globals, DirName, Mapp, Attrib}]
Out[2571]= {True, True, True, True, False, True, True}
```

Если вызов процедуры **ProcQ** (рассматривается ниже) без **Listable**-атрибута на списке *x* инициирует ошибку, в результате присвоения ей данного атрибута вызов **ProcQ**[*x*] на списке *x* в качестве аргумента процедуры возвращает *вполне* ожидаемый результат, как весьма наглядно иллюстрирует предыдущий фрагмент. На формальном уровне для блока, функции или модуля *F* арности единица можно отметить нижеследующее *определяющее* соотношение, а именно: **Map**[*F*, {*a*, *b*, *c*, *d*, ...}]  $\equiv$  {*F*[*a*], *F*[*b*], *F*[*c*], *F*[*d*], ...}, где в левой части процедура *F* может быть как с **Listable**-атрибутом, так и без него, тогда как в правой части предполагается наличие **Listable**-атрибута для блока, модуля либо функции *F*. Таким образом, для блоков, функций либо модулей *F* без атрибута **Listable** для получения его эффекта используется системная функция **Map**.

Для обеспечения наличия **Listable**-атрибута у блока/функции/модуля пользователя, стандартной функции *h* может оказаться достаточно полезной достаточно несложная процедура **ListableC**, чей вызов **ListableC**[*h*] возвращает список атрибутов *h*-объекта, выполняя установку **Listable**-атрибута у объекта *h* (блок/функция/модуль пользователя или стандартная функция). Следующий фрагмент приводит исходный код **ListableC** процедуры наряду с наиболее типичными примерами ее использования, а именно:

```
In[2649]:= ListableC[x_;/; SystemQ[x] || ProcQ[x] || QFunction[ToString[x]] :=
          Module[{b = If[MemberQ[Attributes[x], Protected], "Protected", "Null"]},
          a = Attributes[x], If[MemberQ[a, Listable], a, Unprotect[x]; SetAttributes[x, Listable];
          If[b == "Protected", Protect[x]; Attributes[x]]]

In[2650]:= Attributes[FromCharacterCode]
Out[2650]= {Protected}
In[2651]:= ListableC[FromCharacterCode]
Out[2651]= {Listable, Protected}
In[2652]:= ListableC[ProcQ]
Out[2652]= {Listable}
In[2653]:= FromCharacterCode[{42, 47, 67, 71, 66, 46, 24, 17, 78, 460}]
Out[2653]= {"*", "/", "C", "G", "B", ".", "□", "□", "N", "□"}
```

Система *Mathematica* при манипулировании со списочными структурами имеет ряд недостатков, среди которых является невозможность прямого присвоения элементам списка выражений, как иллюстрирует следующий весьма простой пример:

```
In[2012]:= {a, b, c, d, h, g, s, x, y, z}[[6]] = 78
```

```
Set::setps: {a, b, c, d, h, g, s, x, y, z} in the part assignment is not a symbol. >>
```

```
Out[2012]= 78
```

```
In[2013]:= g
```

```
Out[2013]= g
```

Для упрощения реализации процедур, использующих подобные прямые присвоения элементам списка выражений, используется следующая процедура **ListAssignP**, вызов которой **ListAssignP[x, n, y]** возвращает обновленное значение списка *x*, базирующееся на результатах присвоения значения *y* либо одинарного, либо списка значений *n*-м элементам списка *x*, в качестве которых может выступать одна позиция списка либо их список. Более того, в случае, если списки *n* и *y* имеют разные длины, то выбирается их минимальное значение. Фрагмент представляет исходный код **ListAssignP** наряду с примерами ее применения. **ListAssignP** расширяет функциональные возможности программной среды системы *Mathematica*, выполняя вполне корректное присвоение элементам списка выражений, чего среда системы в полной мере не обеспечивает, как иллюстрирует приведенный выше простой пример.

```
In[2693]:= ListAssignP[x_ /; ListQ[x], n_ /; PosIntQ[n] | | PosIntListQ[n], y_] :=
Module[{a = DeleteDuplicates[Flatten[{n}]], b = Flatten[{y}], c, k = 1},
  If[a[[-1]] > Length[x], Return[Defer[ListAssignP[x, n, y]]],
  c = Min[Length[a], Length[b]];
  While[k <= c, Quiet[Check[ToExpression[ToString[x][[a[[k]]]]] <> " = "
    <> ToString1[If[ListQ[n], b[[k]], y]], Null]]; k++];
  If[NestListQ1[x], x[[-1]], x]]

In[2694]:= Clear[x, y, z]; ListAssignP[{x, y, z}, 3, 78]
Out[2694]= {x, y, 78}
In[2695]:= Clear[x, y, z]; ListAssignP[{x, y, z}, 6, 78]
Out[2695]= ListAssignP[{x, y, z}, 6, 78]
In[2696]:= Clear[x, y, z]; ListAssignP[{x, y, z}, {2, 3}, {71, 66}]
Out[2696]= {x, 71, 66}
In[2597]:= Clear[x, y, z]; ListAssignP[{x, y, z}, 3, {42, 71, 2013}]
Out[2597]= {42, 71, 2013}
```

Наряду с процедурой **ListAssignP** оказалось вполне целесообразным дополнительно определить простую функцию, вызов которой **ListStrQ[x]** возвращает **True**, если все элементы списка *x* - выражения в строчном формате, и **False** в противном случае.

```
In[2599]:= ListStrQ[x_ /; ListQ[x]] := SameQ[DeleteDuplicates[Map[StringQ, x]], {True}]

In[2600]:= Map[ListStrQ, {"a", "b", "a", "a", "b"}, {"a", "b", a, "a", b}, {"Art", "Kr"}]
Out[2600]= {True, False, True}
```



Достаточно полезной в процедурном программировании представляется следующая процедура, вызов которой **ListAssign**[*x*, *y*] обеспечивает присвоение значений списка *x* сгенерированным переменным формата *y\$nnn*, возвращая вложенный список, чей *первый* элемент определяет список сгенерированных переменных "*y\$nnn*" в строчном формате, тогда как *второй* – список присвоенных им значений из списка *x*. Фрагмент представляет исходный код процедуры наряду с примерами ее использования.

```
In[2220]:= ListAssign[x_/, ListQ[x], y_/, SymbolQ[y]] := Module[{a = {}, b},
    Do[a = Append[a, Unique[y]], {k, Length[x]}; b = Map[ToString, a];
    ToExpression[ToString[a] <> "=" <> ToString1[x]]; {b, a}

In[2221]:= ListAssign[{a + b, Sin[x], "agn", {m, n}}, y]
Out[2221]= {"y$5667", "y$5668", "y$5669", "y$5670"}, {a + b, Sin[x], "agn", {m, n}}
In[2222]:= ListAssign[{46, 24, 17, 66, 71}, h]
Out[2222]= {"h$5678", "h$5679", "h$5680", "h$5681", "h$5682"}, {46, 24, 17, 66, 71}
```

Прежде всего, процедура **ListAssign** наряду с рядом подобных представляет интерес в задачах динамической генерации переменных с присвоением им значений.

В *Mathematica* для *группировки* выражений наряду с *простыми списками* используются и более сложные *списочные* структуры – *вложенные* списки, чьими элементами также являются списки (*подсписки*). В этом отношении особый интерес представляют списки *ListList*-типа, чьи подсписки имеют *одинаковую* длину. При этом, для простых списков система располагает тестирующей функцией; вызов **ListQ**[*x*] возвращает **True**, если *x* – список, и **False** в противном случае. Тогда как для тестирования вложенных списков нами определены функции **NestListQ**, **NestListQ1**, **ListListQ**, рассматриваемые ниже.

Для тестирования вложенных списков достаточно полезной представляется простая функция, вызов которой **NestListQ**[*x*] возвращает **True**, если *x* – вложенный список, и **False** в противном случае. При этом, под *вложенным* понимается список *x*, у которого *все* элементы являются списками. При этом, в программировании зачастую возникает целесообразность дополнительно к функции определить и функцию **NestListQ1**, чей вызов **NestListQ1**[*x*] возвращает **True**, если по крайней мере один из элементов списка *x* является его подсписком, и **False** в противном случае. Тогда как достаточно простая функция **NestQL** эквивалентна функции **NestListQ**, при вызове возвращая **True**, если аргумент является вложенным списком, и **False** в противном случае, однако реализует несколько иной алгоритм; при этом, определения *обеих* функций используют чистые функции и весьма полезны в работе со списками. Наконец, вызов простой функции **ListListQ**[*x*] возвращает **True**, если *x* является списком списков (*ListList*), иначе **False** возвращается; при этом, под списком *ListList*-типа понимается вложенный список, в котором все элементы являются списками, имеющими *одинаковую* длину. Следующий фрагмент представляет как исходные коды функций **NestListQ**, **NestListQ1**, **NestQL** и **ListListQ**, так и типичные примеры их применения. Данные средства довольно часто используются в составе тестирующих составляющих заголовков процедур и функций как из нашего пакета *AVZ\_Package\_1* [120], так и в различного рода блоках, функциях и модулях, в первую очередь, применяемых в задачах системного характера [97–101].

```

In[2010]:= NestListQ[x_] := SameQ[ListQ[x] && DeleteDuplicates[Map[ListQ, x]], {True}]
In[2011]:= Map[NestListQ, {{a, c, c, {m, n}}, {a, b, c}, {{a, b}, {a, b}}, {}]}
Out[2011]= {True, False, True, False}
In[2012]:= NestListQ1[x_] := ! SameQ[ListQ[x] && Length[Select[x, ListQ[#] &]], 0]
In[2013]:= Map[NestListQ, {{}, {}, {a, b}, {a, b, {c}}, {{a, b}, {c, d}}]}
Out[2013]= {False, True, False, False, True}
In[2014]:= Map[NestListQ1, {{}, {}, {a, b}, {a, b, {c}}, {{a, b}, {c, d}}]}
Out[2014]= {False, True, False, True, True}
In[2015]:= NestQL[L_ /; ListQ[L]] := MemberQ[Map[ListQ[#] &, L], True]
In[2016]:= {NestQL[{a, b, c}], NestQL[{a, {b, x}, c}], NestQL[{{a, b, x, c}}]}
Out[2016]= {False, True, True}
In[2017]:= ListListQ[x_] := If[x != {} && ListQ[x] && Length[Select[x, ListQ[#] &&
                                Length[#] == Length[x[[1]]] &]] == Length[x], True, False]
In[2018]:= Map[ListListQ, {{a, {b}, c}, {{a}, {b}, {c}}, {{a, b}, {c, d}}, {}]}
Out[2018]= {False, True, True, False}

```

Дополнительно к вышеперассмотренным тестирующим функциям представим еще ряд полезных функций этого же класса, которые довольно полезны в программировании средств для обработки списочных структур произвольной организации. В частности, следующий фрагмент определяет функцию **BinaryListQ[L]**, возвращающую **True** на *бинарном* списке *L* (список может быть вложенным), в противном случае возвращается **False**. Вызов функции **IntegerListQ[x]** возвращает **True**, если список *x*, включая все его подписки любого уровня *вложенности*, является *числовым* списком, иначе возвращает **False**. Представим еще 2 простые функции, тестирующие *списочные* структуры. Вызов функции **ListNumericQ[x]** возвращает **True**, если *x*, включая *все* его подписки любого уровня *вложенности*, является *числовым* списком, и **False** в противном случае. Более того, рассматривается не сам список *x*, а список **N[x]**, что в целом ряде случаев весьма актуально. В то время как вызов **ListExprHeadQ[w, h]** возвращает **True**, если список *w* содержит лишь элементы, удовлетворяющие условию **Head[a] = h**, и **False** в противном случае. Вызов функции **ListSymbolQ[x]** возвращает **True**, если все элементы списка *x*, включая его подписки любого уровня *вложенности*, имеют тип *Symbol*, в противном случае возвращается **False**. Вызов функции **PosIntQ[n]** возвращает **True**, если *n* – целое положительное число, в противном случае возвращается **False**. Функция полезна как сама по себе, так и используется функцией **PosIntListQ**. Наконец, вызов **PosIntListQ[x]** возвращает **True**, если *x* является списком целых положительных чисел, в противном случае возвращается **False**. В дальнейшем данные тесты, определяя, по сути дела, *тип* могут применяться при тестировании фактических аргументов в целом ряде важных процедур и функций. Перечисленные и ряд других средств часто используются при обработке *списочных* структур. Более того, многие из них используются и средствами нашего пакета **AVZ\_Package\_1** [120]. Следующий фрагмент представляет исходные коды вышеперечисленных функций с примерами их типичного использования.

```

In[3017]:= BinaryListQ[L_] := ListQ[L] &&
           MemberQ[{0, 1, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]]
In[3018]:= BinaryListQ[{{1, 0, 1, 0, {1, 1, 0, 1}}, {0, 1}}]
Out[3018]= True
In[1060]:= IntegerListQ[x_] := ListQ[x] && Length[Select[x, IntegerQ[#] &]] == Length[x]
In[1063]:= Gs[x_ /; IntegerListQ[x]] := Plus[Sequences[x]]
In[1064]:= Gs[{46, 24, 17, 66, 71}]
Out[1064]= 224
In[2008]:= IntegerListQ[{42, {78, 460, 6}, 46, {71, 66, {24, 17}}}]
Out[2008]= True
In[1574]:= ListNumericQ[x_ /; ListQ[x]] := SameQ[DeleteDuplicates[Map[NumericQ,
           N[Flatten[x]]]], {True}]
In[1575]:= Map[ListNumericQ, {{42, 460.78, 17/24, Sqrt[2]}, {42, 460.78, a, Sqrt[2]}}]
Out[1575]= {True, False}
In[1576]:= ListExprHeadQ[x_ListQ, h_] := SameQ[DeleteDuplicates[Map[Head, x]], {h}]
In[1577]:= {ListExprHeadQ[{(a + b), (c - d)}, Plus], ListExprHeadQ[{a*b, c/d}, Times]}
Out[1577]= {True, True}
In[1645]:= ListSymbolQ[x_List] := SameQ[DeleteDuplicates[SymbolQ /@ Flatten[x]],
           {True}]
In[1646]:= ListSymbolQ[{Avz, Agn, "a", "b", Ian, "Rans", "x", "y"}]
Out[1646]= True
In[3210]:= PosIntQ[n_] := If[IntegerQ[n] && n > 0, True, False]
In[3211]:= Map[PosIntQ, {24, 17, a + b, 71, 47.66, -460, 0}]
Out[3211]= {True, True, False, True, False, False, False}
In[3212]:= PosIntListQ[L_List] := SameQ[DeleteDuplicates[Map[PosIntQ, L]], {True}]
In[3213]:= Map[PosIntListQ, {{24, 17, 71, 66, 46, 6}, {24, 17, a + b, 71, 47.66, -460, 0}}]
Out[3213]= {True, False}

```

Достаточно простая и вместе с тем полезная процедура **SelectPos** обеспечивает выбор из списка элементы по заданным позициям. Вызов **SelectPos**[*x*, *y*, *z*] возвращает список с элементами списка *x*, чьи номера позиций отличны от элементов списка *y* при *z* = 1, в то время как при *z* = 2 возвращается список с элементами списка *x*, номера позиций которых совпадают с элементами целочисленного списка *y*. В следующем фрагменте представлен исходный код процедуры **SelectPos** с примерами ее использования.

```

In[2096]:= SelectPos[x_ /; ListQ[x], y_ /; ListQ[y] &&
           DeleteDuplicates[Map[IntegerQ[#] && # > 0 &, y]] == {True},
           z_ /; MemberQ[{1, 2}, z]] := Module[{a = {}, k = 1}, While[k <= Length[x],

```

```

If[If[z == 1, ! MemberQ[y, k], MemberQ[y, k]], AppendTo[a, x[[k]]]; k++]; a]
In[2097]:= SelectPos[{a, b, c, d, e, f, g, h, m, n, p}, {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}, 2]
Out[2097]= {a, c, e, g, m, p}
In[2098]:= SelectPos[{a, b, c, d, e, f, g, h, m, n, p}, {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}, 1]
Out[2098]= {b, d, f, h, n}

```

Следует иметь ввиду, что номера позиций списка *y*, вне *диапазона* позиций элементов списка *x* игнорируются, не инициируя ошибочной ситуации, что достаточно удобно для обеспечения непрерывного выполнения приложений без обработки ситуаций.

Довольно простая процедура **ListPosition** расширяет стандартную функцию **Position** на список в качестве второго фактического аргумента. Вызов **ListPosition**[*x*, *y*], где *x* – простой список и *y* – список произвольных выражений, возвращает *вложенный* список, чьи элементы определяют списки *позиций* элементов списка *y* в списке *x*. Следующий фрагмент представляет исходный код процедуры с примерами ее использования.

```

In[2108]:= ListPosition[x_/_; ! NestListQ1[x], y_/_; ListQ[y]] := Module[{a = {}, c = Length[y],
k = 1}, While[k <= c, AppendTo[a, Flatten[Position[x, y[[k]]]]; k++]; a]
In[2109]:= ListPosition[{1, 2, 3, A, G, 2, V, 1, 3, S, G, K, G, 1, G}, {1, 2, 3, G}]
Out[2109]= {{1, 8, 14}, {2, 6}, {3, 9}, {5, 11, 13, 15}}
In[2110]:= ListPosition[{1, 2, 3, {A, G, 2, V, 1, 3, S, G}, K, G, 1, G}, {1, 2, 3, G}]
Out[2110]= ListPosition[{1, 2, 3, {A, G, 2, V, 1, 3, S, G}, K, G, 1, G}, {1, 2, 3, G}]

```

Из ряда приложений, в частности, из примера применения процедуры **Range2** видно [101], что принятого в системе присвоения значений списку переменных не делается и такую задачу решает довольно полезная процедура **AssignToList**[*y*, *z*, *n*], чей вызов присваивает значения элементов списка *y* соответствующим элементам программно сгенерированного списка {*z*<sub>1</sub>, *z*<sub>2</sub>, *z*<sub>3</sub>, ..., *z*<sub>*n*</sub>} (*n* ≥ 1). Следующий фрагмент представляет исходный код процедуры **AssignToList** с типичными примерами ее использования.

```

In[2368]:= AssignToList[y_/_; ListQ[y], z_/_; ! HowAct[z], n_/_; IntegerQ[n] && n >= 1] :=
Module[{a, k = 1},
If[Length[y] < n, Return[Defer[AssignToList[y, z, n]], a = Range2[z, n];
For[k, k <= n, k++, ToExpression[ToString[a[[k]]] <> "=" <> ToString[y[[k]]]]; a]
In[2369]:= {AssignToList[Range[14], z, 14], z1, z3, z5, z6, z9, z10, z11, z12, z13, z14}
Out[2369]= {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}, 1, 3, 5, 6, 9, 10, 11, 12, 13, 14}
In[2370]:= {AssignToList[CharacterRange["A", "H"], x, 7], x1, x2, x3, x4, x5, x6, x7, x8}
Out[2370]= {{A, B, C, D, E, F, G, H}, A, B, C, D, E, F, G, H}

```

Наряду с простыми система поддерживает работу с *вложенными* списками различного формата. Для решения целого ряда задач, имеющих дело с вложенными списками, в некоторых случаях могут возникать вопросы, не решаемые прямыми стандартными средствами, требуя в таких ситуациях программирования задач средствами, которые предоставляются системой *Mathematica*. Это довольно наглядно иллюстрируется на примере задачи, состоящей в определении количества элементов, отличных от *списка*,

на каждом уровне вложенности списка как простого списка (уровень вложенности 0), так и вложенного. Данную задачу может решить процедура, вызов которой **ElemLevelsN[x]** возвращает вложенный список, элементами которого являются 2-х элементные списки, первый элемент которых определяет уровень вложенности, тогда как второй – число элементов данного уровня с типом, отличным от *List*. Полезная модификация данной процедуры – процедура **ElemLevelsL**, вызов которой **ElemLevelsL[x]** также возвращает вложенный список, элементы которого – 2-х элементные списки, чей первый элемент определяет уровень вложенности, тогда как второй элемент – список элементов такого уровня с типом, отличным от *List*. При этом, читатель в качестве довольно полезного упражнения может объединить данные процедуры в одну, например, возвращая один из результатов через необязательный аргумент. Следующий фрагмент представляет исходные коды обоих средств с наиболее типичными примерами их использования.

```
In[2633]:= ElemLevelsN[x_ /; ListQ[x]] := Module[{a = x, c = {}, m = 0, n, k = 0},
      While[NestListQ1[a], n = Length[Select[a, ! ListQ[#] &]];
      c = Append[c, {k++, n - m}]; m = n; a = Flatten[a, 1]; Continue[];
      Append[c, {k++, Length[a] - m}]]

In[2634]:= L = {a, b, a, {d, c, s}, a, b, {b, c, {x, y, {v, g, z, {78, {460, {}, 71}}, a, k, a}, z}, b}, c, b};
In[2635]:= ElemLevelsN[L]
Out[2635]= {{0, 7}, {1, 6}, {2, 3}, {3, 6}, {4, 1}, {5, 2}, {6, 0}}
In[2644]:= Map[ElemLevelsN, {{}, {a, b, c, d, r, t, y, c, s, f, g, h, 71, 78, 460, s, a, q, w}}]
Out[2644]= {{{0, 0}}, {{0, 19}}}

In[2674]:= ElemLevelsL[x_ /; ListQ[x]] := Module[{a = x, c = {}, m = {}, n, k = 0},
      While[NestListQ1[a], n = Select[a, ! ListQ[#] &];
      c = Append[c, {k++, MinusList[n, m]}]; m = n; a = Flatten[a, 1];
      Continue[]; Append[c, {k++, MinusList[a, m]}]]

In[2675]:= ElemLevelsL[L]
Out[2675]= {{0, {a, b, a, a, b, c, b}}, {1, {d, s}}, {2, {x, y, z}}, {3, {v, g, k}},
      {4, {78}}, {5, {460, 71}}, {6, {}}}
```

Следующая процедура обеспечивает возврат всех возможных подсписков вложенного списка. Вызов **SubLists[x]** возвращает список всех возможных подсписков вложенного списка *x* с учетом их вложенности. Если список *x* является простым, вызов процедуры **SubLists[x]** возвращает *пустой* список, т.е. {}. Нижеследующий фрагмент представляет исходный код процедуры **SubLists** наряду с типичными примерами ее применения.

```
In[2039]:= SubLists[x_ /; ListQ[x]] := Module[{a, b, c = {}, k = 1}, If[! NestListQ1[x], {},
      a = ToString[x]; b = DeleteDuplicates[Flatten[StringPosition[a, "{"]]];
      While[k <= Length[b], AppendTo[c,
      SubStrSymbolParity1[StringTake[a, {b[[k]], -1}], "{", "}"][[1]]]; k++];
      DeleteDuplicates[ToExpression[c[[2 ;; -1]]]]]

In[2040]:= L = {a, b, a, {d, c, s}, a, b, {b, c, {x, y, {v, g, z, {78, {460, {}, 71}}, a, k, a}, z}, b}, c, b};
```

```
In[2041]:= SubLists[Flatten[L]]
Out[2041]= {}
In[2042]:= SubLists[L]
Out[2042]= {{d, c, s}, {b, c, {x, y, {v, g, z, {78, {460, {}, 71}}, a, k, a}, z}, b}, {x, y, {v, g, z, {78, {460, {}, 71}}, a, k, a}, z}, {v, g, z, {78, {460, {}, 71}}, a, k, a}, {78, {460, {}, 71}}, {460, {}, 71}, {}
In[2043]:= SubLists[{a, b, {c, d, {g, h, {g, s}}, {n, m}}, {78, 460}}]
Out[2043]= {{c, d, {g, h, {g, s}}, {n, m}}, {g, h, {g, s}}, {g, s}, {n, m}, {78, 460}}
```

Как отмечалось выше, многие функции работают со списками как *едиными объектами*, однако в целом ряде случаев имеется настоятельная необходимость оперирования с *отдельными* их элементами, что достигается или упомянутой функцией **Part** формата, соответствующего задаче, либо с использованием индексации, имеющей следующий формат, а именно: **L[[k]], L[[k]][[j]] ... [[p]]**; при этом, использование функции **Part** при значительной глубине индексации представляется более компактным и наглядным. Вложенные списки могут представлять достаточно широкий спектр структур данных (*векторы, таблицы, массивы, матрицы и др.*). Между тем, в целом ряде случаев требуется сведение их к более простым спискам, имеющим меньший уровень вложенности. Эту задачу решает функция **Flatten**, располагая четырьмя форматами кодирования. Так, на ее основе можно создавать достаточно эффективные средства обработки *списочных* структур. В качестве примера приведем процедуру **MaxLevel**, чей вызов **MaxLevel[x]** возвращает *максимальный уровень* вложенности списка *x* (при этом, *уровень вложенности простого списка полагается равным 0*). В данном контексте вызов простой, но довольно полезной процедуры **MaxNestLevel[L]**, обеспечивает возврат максимального уровня вложенности списка *L*; при этом, уровень **0** определяется для *невложенного* списка, т.е. списка, элементы которого не являются списками. Следующий фрагмент представляет исходные коды данных процедур, а также типичные примеры их использования.

```
In[2562]:= MaxLevel[x_;/; ListQ[x]] := Module[{a = x, k = 0},
      While[NestListQ1[a], k++; a = Flatten[a, 1]; Continue[]]; k]
In[2563]:= Map[MaxLevel, {{a, b}, {a, {b, c, d}}, {{{a, b, c}}}, {a, {{c, {d}, {{h, g}}}}}]
Out[2563]= {0, 1, 2, 4}
In[2564]:= MaxLevel[{a, b, c, d, f, g, h, s, r, t, w, x, y, z}]
Out[2564]= 0
In[2581]:= ListLevels[x_;/; ListQ[x]] := Module[{a = x, b, c = {}, k = 1}, If[! NestListQ1[x], 0,
      While[NestListQ1[a], b = Flatten[a, 1]; If[Length[b] >= Length[a],
        c = Append[c, k++], c = Append[c, k]]; a = b; Continue[]]; c]]
In[2582]:= ListLevels[{a, b, c, d, f, g, h, s, r, t, w, x, y, z}]
Out[2582]= 0
In[2583]:= ListLevels[{a, b, c, {d, f, g, {h, s, {z, y, g}, r}, t}, w, {x, {{{a, b, c}}}, y}, z]}
Out[2583]= {1, 2, 3, 4}
In[2586]:= MaxNestLevel[L_;/; ListQ[L]] := Module[{a = Flatten[L], b = L, c = 0},
      While[! a == b, b = Flatten[b, 1]; c = c + 1]; c]
```

```
In[2587]:= L = {{a, {b, {m, {x, y}, n}, x}, c, {{{{{{{66, 71}}}}}}}}; Map[MaxNestLevel, {L, {a, b, c}}]
Out[2587]= {8, 0}
In[2588]:= L = {m, {m, {b, {m, {x, y}, n}, x}, c, {{{{{{{66, m, 71}}}}}}}}; MemberQ[L, x]
Out[2588]= False
```

В представленном примере определение процедуры **MaxLevel** использует функцию, чей вызов **ListQ[x]** возвращает **True**, если  $x$  – список, и **False** в противном случае, в то время как вызов функции **NestListQ1[x]** возвращает **True**, если  $x$  – вложенный список, и **False** в противном случае. Для более детального тестирования объектов на *List*-тип нами были дополнительно определены тестирующие функции **ListListQ**, **NestListQ**, **NestQL** и **PosIntListQ**, рассмотренные выше. При необходимости пользователь имеет возможность существенно расширять данный перечень с учетом потребности своих приложений либо системных задач. Фрагмент содержит процедуру, вызов которой **ListLevels[x]** возвращает список уровней вложенности списка  $x$ ; при этом, на простом списке, включая пустой список, вызов процедуры **ListLevels** возвращает **0**. Более того, между упомянутыми средствами имеют место определяющие соотношения:

$$\text{Flatten}[x] \equiv \text{Flatten}[x, \text{MaxLevel}[x]] \qquad \text{MaxLevel}[x] \equiv \text{ListLevels}[x][[-1]]$$

Для присвоения одного и того же значения переменным можно использовать весьма простую конструкцию  $x = y = z = \dots = a$ , тогда как для присвоения переменным разных значений можно использовать конструкцию вида  $\{x, y, z, \dots\} = \{a, b, c, \dots\}$  при условии, что длины обоих списков *идентичны*, иначе инициируется ошибочная ситуация. Для устанения данного недостатка может быть использована процедура **ListsAssign[x, y]**, возвращающая результат присвоения значений списка  $y$  списку  $x$ .

```
In[2066]:= ListsAssign[x_List, y_List] := Module[{a = Min[Map[Length, {x, y}]], b, c, d = {},
    k = 1}, If[a == 0, Return[x], Off[Set::setraw]; Off[Set::write]; Off[Set::wrsym]];
    While[k <= a, {b, c} = {x[[k]], y[[k]]}; d = Append[d, b = c]; k++]; x = {Sequences[d[[1 ;; a]]],
    Sequences[x[[a + 1 ;; -1]]]}; On[Set::setraw]; On[Set::write]; On[Set::wrsym]; x]

In[2067]:= L1 = {x, 78, a + b, Sin, t, s}; L2 = {a, b, c, w, 71}; ListsAssign[L1, L2]
Out[2067]= {a, 78, a + b, Sin, 71, s}
In[2068]:= {x, y, z, h, g, w, t} = {a, b, c}
Set::shape: Lists {x, y, z, h, g, w, t} and {a, b, c} are not the same shape. >>
Out[2068]= {a, b, c}
In[2069]:= ListsAssign[{x, y, z, h, g, w, t}, {a, b, c}]; {x, y, z}
Out[2069]= {a, b, c}

In[2070]:= ListAppValue[x_List, y_] := Quiet[x = PadLeft[{}, Length[x], y]]

In[2071]:= x = 78; ListAppValue[{x1, y, z, h, g, w, t}, 71]; {x1, y, z, h, g, w, t, x}
Out[2071]= {71, 71, 71, 71, 71, 71, 71, 78}
In[2072]:= Clear[x, y, z, h, g, w, t]; {x, y, z, h, g, w, t} = 78; {x, y, z, h, g, w, t}
Set::shape: Lists {x, y, z, h, g, w, t} and 78 are not the same shape. >>
Out[2072]= {x, y, z, h, g, w, t}
```

Таким образом, вызов **ListsAssign**[ $x$ ,  $y$ ] возвращает *обновленный* присвоениями список  $x$ ; при этом, процедура обрабатывает ошибочные и особые ситуации, обусловленные присвоениями  $x = y$ . Тогда как для присвоения одного и того же значения элементам списка может применяться функция **ListAppValue**, вызов **ListAppValue**[ $x$ ,  $y$ ] которой обеспечивает присвоение выражения  $y$  элементам списка  $x$ . Предыдущий фрагмент представляет исходные коды указанных средств наряду с примерами их применения.

Вызов процедуры **DelEl**[ $L$ ,  $w$ ,  $N$ ] возвращает усеченный слева список  $L$  элементами  $w$  при  $N=1$ , усеченный справа список  $L$  элементами  $w$  при  $N=2$ , и при  $N=3$  усеченный с обоих концов; при других значениях  $N$  вызов **DelEl** возвращается невычисленным.

```
In[2355]:= DelEl[L_;/; ListQ[L], x_, N_;/; MemberQ[{1, 2, 3}, N]] :=
Module[{a, b = Length[L], k, Art},
  Art[Z_, y_, p_;/; MemberQ[{1, 2}, p]] := Module[{b = Length[Z], k},
    If[b == 0, Defer[DelEl[Z, y, p]], If[p == 1, For[k = 1, k <= b, k++,
      If[Z[[k]] == y, Null, Return[Take[Z, {k, b}]]],
    For[k = b, 1 <= k, k--, If[Z[[k]] == y, Null, Return[Take[Z, {1, k}]]], Null]];
  If[N == 1 | | N == 2, Art[L, x, N], Art[Art[L, x, 1], x, 2]]]

In[2356]:= DelEl[{a, a, a, a, a, 66, 71, 46, 17, 42, 47, 24, a, a, a, a, a}, a, 1]
Out[2356]= {66, 71, 46, 17, 42, 47, 24, a, a, a, a, a}
In[2357]:= DelEl[{a, a, a, a, a, 66, 71, 46, 17, 42, 47, 24, a, a, a, a, a}, a, 2]
Out[2357]= {a, a, a, a, a, 66, 71, 46, 17, 42, 47, 24}
In[2358]:= DelEl[{a, a, a, a, a, 66, 71, 46, 17, 42, 47, 24, a, a, a, a, a}, a, 3]
Out[2358]= {66, 71, 46, 17, 42, 47, 24}
```

Вопросы работы со списочными структурами весьма актуальны, т.к. такие структуры являются *базовыми* в среде системы. Система располагает довольно развитым набором функций для работы с этими структурами, однако в ряде случаев возникают задачи, требующие дополнительного программирования. Например, следующая процедура предназначена для *группировки* элементов списка по их кратности. Вызов процедуры **GroupIdentMult**[ $x$ ] возвращает вложенный список следующего формата, а именно:

$$\{\{n_1\}, \{x_1, x_2, \dots, x_a\}\}, \{\{n_2\}, \{y_1, y_2, \dots, y_b\}\}, \dots, \{\{n_k\}, \{z_1, z_2, \dots, z_c\}\}$$

где  $\{x_i, y_j, z_p\}$  – элементы списка  $x$  и  $n_t$  – соответствующие им кратности. Следующий фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2997]:= GroupIdentMult[x_List] := Module[{a = Gather[x], b},
  b = Map[{DeleteDuplicates[#][[1]], Length[#]} &, a];
  b = Map[DeleteDuplicates[#] &, Map[Flatten,
    Gather[b, SameQ[#1[[2]], #2[[2]]] &]]];
  b = Map[{#[[1]], Sort[#[[2] ;; -1]]] &, Map[Reverse,
    Map[If[Length[#] > 2, Delete[Append[#, #[[2]], 2], #] &, b]]];
  b = Sort[b, #1[[1]][[1]] > #2[[1]][[1]] &];
  If[Length[b] == 1, Flatten[b, 1], b]]
```



```

In[2998]:= L = {a, c, b, a, a, c, g, d, a, d, c, a, c, c, h, h, h, h, h};
In[2999]:= GroupIdentMult[L]
Out[2999]= {{{5}, {a, c, h}}, {{1}, {b, g}}, {{2}, {d}}}
In[3000]:= GroupIdentMult[{a, a, a, a, a, a}]
Out[3000]= {{6}, {a}}
In[3001]:= GroupIdentMult[{a, a, a, a, a, a, b, b, b, b, b, b}]
Out[3001]= {{6}, {a, b}}
In[3002]:= L = RandomInteger[42, 71]
Out[3002]= {5, 40, 27, 14, 30, 29, 8, 29, 3, 40, 13, 37, 41, 30, 14, 40, 16, 27, 14, 40, 17, 24, 36, 20,
22, 26, 32, 36, 33, 6, 4, 42, 23, 30, 21, 21, 10, 33, 15, 21, 26, 24, 2, 9, 40, 21, 29, 17, 38,
22, 23, 40, 41, 9, 20, 31, 14, 28, 21, 35, 25, 40, 19, 23, 32, 19, 26, 34, 40, 4, 8}
In[3003]:= GroupIdentMult[L]
Out[3003]= {{{8}, {40}}, {{5}, {21}}, {{4}, {14}}, {{3}, {23, 26, 29, 30}}, {{2}, {4, 8, 9, 17, 19, 20, 22, 24,
27, 32, 33, 36, 41}}, {{1}, {2, 3, 5, 6, 10, 13, 15, 16, 25, 28, 31, 34, 35, 37, 38, 42}}}
In[3004]:= GroupIdentMult[{}]
Out[3004]= {}

```

Элементы возвращаемого резульатного вложенного списка *отсортированы* в порядке убывания кратностей групп элементов исходного списка *x*. Вызов **GroupIdentMult[{}]** возвращает пустой список, т.е. {}. Будучи дополнительными средствами для работы со списочными структурами – базовыми в программной среде пакета – данные средства оказываются достаточно полезными в целом ряде приложений различного характера. Между тем, в книге представлены и другие средства, которые можно весьма успешно использовать при обработке списочных структур различного формата и назначения. Ряд средств уже рассмотрен выше, другие будут рассмотрены ниже, а также имеется немало средств, представленных в книге, прямо не ассоциированных со списочными структурами, но вполне приемлемых для работы с отдельными форматами списков.

В обработке *списочных* структур интерес представляет задача группировки элементов вложенных списков *ListList*-типа на основе *n*-х элементов их подсписков. Эта задача решается следующей процедурой, чей вызов **ListListGroup[x, n]** возвращает *вложенный* список – результат группировки *ListList*-списка *x* по *n*-му элементу его подсписков. Фрагмент представляет исходный код процедуры с примерами ее использования.

```

In[2369]:= ListListGroup[x_/, ListListQ[x], n_/, IntegerQ[n] && n > 0] := Module[{a = {},
b = {}, k = 1, If[Length[x][[1]] < n, Return[Defer[ListListGroup[x, n]]],
For[k, k <= Length[x], k++, b = Append[b, x[[k]][[n]]];
b = DeleteDuplicates[Flatten[b]]];
For[k = 1, k <= Length[b], k++, a = Append[a, Select[x, #[[n]] == b[[k]] &]]]; a]
In[2370]:= ListListGroup[{{78, 2}, {460, 6}, {17, 2}, {24, 2}, {71, 6}}, 2]
Out[2370]= {{{78, 2}, {17, 2}, {24, 2}}, {{460, 6}, {71, 6}}}
In[2371]:= ListListGroup[{{78, 2}, {460, 6}, {17, 2}, {24, 2}, {71, 6}}, 6]
Out[2371]= ListListGroup[{{78, 2}, {460, 6}, {17, 2}, {24, 2}, {71, 6}}, 6]

```

На недопустимых фактических аргументах вызов возвращается невычисленным.

Следующий фрагмент представляет процедуру **ListToString[x,y]**, которая возвращает результат конвертирования в *единую* строку всех элементов списка *x*, не учитывая его вложенности, разделенных строкой *y*; в то время как строка *x* конвертируется в *список* подстрок строки *x*, разделенных строкой *y*. Фрагмент представляет как исходный код процедуры **ListToString**, так и наиболее типичные примеры ее использования.

```
In[2013]:= ListToString[x_;/; ListQ[x] || StringQ[x], y_;/; StringQ[y]] :=
Module[{a, b = {}, c, d, k = 1},
  If[ListQ[x], a = Flatten[x]; For[k, k < Length[a], k++, c = a[[k]];
    b = Append[b, ToString1[c] <> y]];
  a = StringJoin[Append[b, ToString1[a[[-1]]]],
  a = FromCharacterCode[14]; d = a <> StringReplace[x, y -> a] <> a;
  c = Sort[DeleteDuplicates[Flatten[StringPosition[d, a]]]];
  For[k = 1, k < Length[c], k++, b = Append[b, StringTake[d, {c[[k]] + 1,
    c[[k + 1]] - 1}]]]; ToString[b]]

In[2014]:= ListToString[{a + b, {"Agn", 66}, Kr, 17, Art, 24, "RANS", {{{Avz | | 71}}}, "&"]
Out[2014]= "a + b&\"Agn\"&66&Kr&17&Art&24&\"RANS\"&Avz | | 71"
In[2015]:= ListToString["a + b&\"Agn\"&66&Kr&17&Art&24&Avz | | 71", "&"]
Out[2015]= {a + b, "Agn", 66, Kr, 17, Art, 24, Avz | | 71}
```

**Матричные** объекты непосредственно представляются *вложенными списками ListList*-типа, для обработки которых пакет располагает целым рядом средств, включая также работу с матрицами специального типа. Каждый список *L ListList*-типа может быть представлен соответствующей матрицей размерности **Length[L] \* Length[L[[1]]]**. Итак, вектора и матрицы могут быть представлены *простыми* списками и *списками ListList*-типа соответственно. Для обеспечения работы с *вложенными* списками был предложен ряд полезных средств, представленных выше. В качестве лишь одного из этих средств рассмотрим процедуру, чей вызов **TransListList[x]** возвращает результат *транспозиции* списка *x ListList*-типа. Процедура **TransListList** работает по принципу *переключателя*; исходный код **TransListList** с примерами ее применения представлены фрагментом:

```
In[2063]:= TransListList[L_;/; ListListQ[L]] := Module[{a = Length[L], b = Length[L[[1]]],
  c = {}, d = {}, k = 1, j}, For[k, k <= b, k++,
  For[j = 1, j <= a, j++, d = Append[d, L[[j]][[k]]]; c = Append[c, d]; d = {}]; c]

In[2064]:= L = {{a0, b0, c0}, {a1, b1, c1}, {a2, b2, c2}, {a3, b3, c3}, {a4, b4, c4}}; TransListList[L]
Out[2065]= {{a0, a1, a2, a3, a4}, {b0, b1, b2, b3, b4}, {c0, c1, c2, c3, c4}, {d0, d1, d2, d3, d4}}
In[2066]:= Map[MatrixForm, {L, %}]
Out[2066]= 
$$\begin{pmatrix} a0 & b0 & c0 \\ a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \\ a4 & b4 & c4 \end{pmatrix}, \begin{pmatrix} a0 & a1 & a2 & a3 & a4 \\ b0 & b1 & b2 & b3 & b4 \\ c0 & c1 & c2 & c3 & c4 \end{pmatrix}$$

```

Наряду к таким объектам, как списки и матрицы, группирующим данные, в полной мере можно отнести таблицы (*Table*) и массивы (*Array*), для которых предлагается ряд системных функциональных средств обработки как над объектами в целом, так и над их отдельными элементами. Ряд средств в данном направлении предложен и нами. В то же время вопросы этого типа здесь не рассматриваются, читатель отсылается или к соответствующей литературе [121-140], либо к справке по системе *Mathematica*. Здесь же сделан акцент на списочных структурах, как основе группирующих объектов.

В целом ряде случаев возникает необходимость выполнять *присвоения* выражений, чье число заранее не известно и которое определяется в процессе некоторых вычислений, например, циклического характера, переменным. Задачу решает достаточно простая процедура **ParVar**. Вызов процедуры **ParVar[x, y]** обеспечивает присвоение элементов списка *y* списку сгенерированных на основе символа *x* *переменных* с возвратом списка в *строчном* формате этих переменных. Следующий фрагмент представляет исходный код процедуры **ParVar** наряду с достаточно типичным примером ее использования.

```
In[2010]:= ParVar[x_ /; SymbolQ[x], y_ /; ListQ[y]] := Module[{a = {}, b, k = 1},
               For[k, k <= Length[y], k++, a = Append[a, Unique[x]]];
               b = ToString[a]; {b, ToExpression[b <> "=" <> ToString1[y]]}][[1]]

In[20110]:= W = ParVar[GS, {71, 66, 46, 24, 17}]
Out[2011]= "{GS$2660, GS$2661, GS$2662, GS$2663, GS$2664}"
In[2012]:= ToExpression[W]
Out[2012]= {71, 66, 46, 24, 17}
```

В целом ряде задач, имеющих дело со *списками*, возникает необходимость вычисления *разности* между двумя списками *x* и *y*, которая определяется как список, чьи элементы входят в список *x*, но не входят в *y*. Итак, вызов функции **MinusList[x, y]** возвращает результат *вычитания* списка *y* из списка *x*, которое состоит в удалении из списка *x* *всех вхождений* элементов из списка *y*. Тогда как вызов процедуры **MinusList1[x, y]**, которая является модификацией предыдущей, возвращает результат *вычитания* списка *y* из *x*, которое состоит в паритетном *удалении* из списка *x* вхождений элементов из списка *y*, т.е. число элементов, удаляемых из списка *x*, строго соответствуют их числу в списке *y*. Следующий фрагмент представляет исходные коды функции **MinusList** и процедуры **MinusList1** наряду с наиболее типичными примерами их использования.

```
In[2035]:= MinusList[x_ /; ListQ[x], y_ /; ListQ[y]] := Select[x, ! MemberQ[y, #] &]

In[2036]:= MinusList[{"P3", Art, "a", Kr, "P3", "b", "P1", Avz, "P2", "m", Kr, "P2", Agn, "n",
                    "P3", g, Art, h, g}, {"P3", Art, "P1", "P2", g, "P2", Kr, h}]
Out[2036]= {"a", "b", Avz, "m", Agn, "n"}
In[2037]:= {MinusList[{}, {}], MinusList[{a, b, c, d}, {a, b, c, d}]}
Out[2037]= {{}, {}}

In[2038]:= MinusList1[x_ /; ListQ[x], y_ /; ListQ[y]] := Module[{a = x, k = 1},
               While[k <= Length[y], a = ReplacePart[a, Flatten[Position[a, y[[k]]]][[1]] -> Null]; k++];
               Select[a, ! SameQ[#, Null] &]]
```

```
In[2039]:= MinusList1[{x, a, x, y, b, c, 78, x, d, a, b, x, y, 78}, {a, b, c, 78, x, y, a, b, x}]
Out[2039]= {x, d, x, y, 78}
In[2040]:= {MinusList1[{}, {}], MinusList1[{a, b, c, d}, {a, b, c, d}]}
Out[2040]= {{}, {}}
```

Оба средства представляют интерес и в качестве вспомогательных при решении ряда наших процедур [120], а также самостоятельный интерес при обработке списков.

В процессе программирования ряда процедур доступа к файлам данных выяснилась целесообразность запрограммировать полезную и в других приложениях процедуру, вызов которой **PosSubList**[*x*, *y*] возвращает вложенный список начальных и конечных элементов для вхождения в простой список *x* кортежа элементов, указанных списком *y*. Следующий фрагмент представляет исходный код процедуры с примерами.

```
In[2260]:= PosSubList[x_;/ ListQ[x], y_;/ ListQ[y]] := Module[{a = ToString[x], d,
    b = ToString[y], c = FromCharacterCode[16]}, d = StringTake[b, {2, -2}];
    If[! StringFreeQ[a, d], b = StringReplace[a, d -> c <> ", " <>
        StringTake[ToString[y][[2 ;; -1]]], {2, -2}];
        Map[{#, # + Length[y] - 1} &,
            Flatten[Position[ToExpression[b], ToExpression[c]]], {}]]
In[2261]:= PosSubList[{a, a, b, a, a, a, b, a, x, y, z, a, b, a}, {a, b, a}]
Out[2261]= {{2, 4}, {6, 8}, {12, 14}}
In[2262]:= PosSubList[{a, a, b, a, a, a, b, a, x, y, z, a, b, a}, {a, x, a}]
Out[2262]= {}
```

Подход еще раз наглядно иллюстрирует побудительные мотивы и предпосылки для создания *инструментария* пользователя, расширяющего программную среду системы *Mathematica*. Относительно процедуры **PosSubList** необходимо еще раз отметить, что она служит для решения задачи определения *позиций* вхождения кортежа элементов, указанного списком, в простой список, как весьма наглядно иллюстрируют примеры фрагмента. В качестве полезного упражнения рекомендуется расширить процедуру на допустимость использования в качестве ее аргументов вложенных списков.

Многие языки программирования обеспечивают стандартные средства конвертации *строк* в *списки*, в свою очередь, для обеспечения конвертации *символов* в *списки* можно предложить простую функции **SymbolToList**, возвращающую результат *конвертации* символа в список. В ряде приложений данное простое средство оказывается довольно полезным. Следующий фрагмент представляет исходный код функции **SymbolToList** наряду с некоторыми наиболее типичными примерами ее использования.

```
In[2008]:= SymbolToList[x_;/ SymbolQ[x]] := Map[ToExpression, Characters[ToString[x]]]
In[2009]:= SymbolToList[Art24Kr17]
Out[2009]= {A, r, t, 2, 4, K, r, 1, 7}
In[2010]:= SymbolToList[$RANS24IAN17AVZArt2013]
Out[2010]= {$, R, A, N, 2, 4, I, A, N, 1, 7, A, V, Z, A, r, t, 2, 0, 1, 3}
```

Вызов стандартной функции **Tuples**[*L*, *n*] возвращает список всевозможных кортежей длины *n*, составленных из элементов списка *L*. Для получения сравнительной оценки реактивности систем *Mathematica* и *Maple* именно алгоритм, реализующий данную процедуру в среде обеих систем, был нами положен в основу подобного сравнения. И действительно, в целом, *реактивность* того либо иного алгоритма определяется как реактивностью самой системы, так и самой реализацией алгоритма. В качестве весьма наглядного примера можно привести *Maple*-процедуру *tuples*, рассмотренную в [99], и реализованный ее алгоритм в *Mathematica*-процедуре **Tuples1**, которая приведена следующим фрагментом. Обе процедуры возвращают *список* всевозможных кортежей длины *n* в строчном формате, составленных из символов некоторого алфавита *A*.

```
In[2321]:= Tuples1[x_ /; ListQ[x], n_ /; IntegerQ[n]] := Module[{c = "", d = "", f = Res, h = {},
    t, t$$$$, j = 1, a = Map[ToString1, Map[ToString, x]], m = Length[x], b = ""},
    Res = {}; For[j = 1, j <= n, j++, h = Append[h, "t$$$$" <> ToString[j]];
    b = b <> "For[t$$$$" <> ToString[j] <> "=1, t$$$$" <> ToString[j] <>
    " <= " <> ToString[m] <> ", t$$$$" <> ToString[j] <> "++,"; d = d <> "];";
    For[j = 1, j <= n, j++, c = c <> ToString[a] <> "[[t$$$$" <> ToString[j] <> "]" <> ",";
    c = "Res = Append[Res, ToString1[StringJoin[" <> StringTake[c, {1, -2}] <> "]]]";
    ToExpression[b <> c <> d]; {Res, Res = f}][[1]]

In[2322]:= Tuples1[{1, 2, 3, 4}, 3]
Out[2322]= {"111", "112", "113", "114", "121", "122", "123", "124", "131", "132", "133", "134",
    "141", "142", "143", "144", "211", "212", "213", "214", "221", "222", "223", "224",
    "231", "232", "233", "234", "241", "242", "243", "244", "311", "312", "313", "314",
    "321", "322", "323", "324", "331", "332", "333", "334", "341", "342", "343", "344",
    "411", "412", "413", "414", "421", "422", "423", "424", "431", "432", "433", "434",
    "441", "442", "443", "444"}

In[2323]:= t = TimeUsed[]; Tuples1[{0, 1}, 14]; TimeUsed[] - t
Out[2323]= 27.64
In[2324]:= t = TimeUsed[]; Tuples1[{0, 1}, 17]; TimeUsed[] - t
Out[2324]= 992.460
In[2325]:= t = TimeUsed[]; Tuples[{0, 1}, 17]; TimeUsed[] - t
Out[2325]= 0.547
In[2326]:= t = TimeUsed[]; Tuples[{"0", "1", "2"}, 17]; TimeUsed[] - t
    No more memory available.
    Mathematica kernel has shut down.
    Try quitting other applications and then retry.
In[2327]:= t = TimeUsed[]; Tuples[{0, 1, 2, 3}, 17]; TimeUsed[] - t
    Tuples::toomany: The length of the output of Tuples[{0, 1, 2, 3}, 17] should be a
    machine integer. >>
> t := time(): tuples({1, 2}, 14): time() - t;  => 17.891                                #Maple 11
> t := time(): tuples({1, 2}, 17): time() - t;  => 1024.720
```

Полученные временные оценки показывают, что на данном алгоритме *Maple* имеет вполне определенные преимущества, довольно быстро теряемые как с ростом длины

$n$  кортежей, так и с ростом мощности алфавита  $A$ . Между тем, в целом было замечено, что с ростом вложенности циклических конструкций временные издержки в системе *Mathematica* растут заметно быстрее, чем на тех же конструкциях в среде *Maple*. Для этого был проведен эксперимент с процедурами **NestCycles**, запрограммированными в среде обеих систем и чей алгоритм состоит в реализации простейшей циклической конструкции заданной глубины вложенности  $n$ ; результаты данного эксперимента со всей очевидностью подтверждают вышесказанное [101].

Следует отметить, использованный в обоих процедурах прием довольно эффективен при необходимости динамической генерации программных кодов. И еще достаточно важный момент следует иметь ввиду, а именно. При сравнении использовались лишь идентично реализованные алгоритмы, реализующие процедуры **Tuples1**, **NestCycles**, тогда как, например, реализация стандартной функции **Tuples** является значительно более *реактивной*. Правда, результаты вызова функции **Tuples** довольно существенно зависят как от типа элементов списка  $L$ , так и от его длины, как достаточно наглядно иллюстрируют два примера предыдущего фрагмента. Итак, обе системы достаточно чувствительны к размеру доступной памяти в их рабочей области.

Следующая процедура несколько расширяет стандартную функцию **Gather** системы, оказываясь достаточно полезной в целом ряде довольно важных приложений. Вызов процедуры **Gather1[L, n]** возвращает вложенный список *ListList*-типа, образованный на основе *ListList*-списка  $L$  путем группировки его подсписков по их  $n$ -му элементу. В этом контексте оказывается довольно полезной еще одна модификация стандартной процедуры **Gather** – **Gather2**, чей вызов **Gather2[L]** возвращает либо простой список, либо список *ListList*-типа, определяющий только кратные элементы списка  $L$  вместе с их кратностями. При этом, при отсутствии в  $L$  кратных элементов вызов процедуры возвращает пустой список, т.е.  $\{\}$ . Следующий фрагмент представляет исходный код обеих процедур наряду с некоторыми типичными примерами их использования.

```
In[2470]:= Gather1[L_ /; ListListQ[L], n_ /; IntegerQ[n]] := Module[{a = {}, b = {}, c, k},
    If[(1 <= n && n <= Length[L[[1]])], Return[Defer[Gather1[L, n]]],
    Do[a = Append[a, L[[k]][[n]]], {k, 1, Length[L]}];
    a = Map[List, DeleteDuplicates[a]];
    For[k = 1, k <= Length[a], k++, a[[k]] = Select[L, #[[n]] == a[[k]][[1]] &]; a]

In[2471]:= L = {{42, V, 1}, {47, G, 2}, {67, S, 1}, {68, V, 2}, {63, G, 3}, {43, S, 2}}; Gather1[L, 2]
Out[2471]= {{{42, V, 1}, {68, V, 2}}, {{47, G, 2}, {63, G, 3}}, {{67, S, 1}, {43, S, 2}}}

In[2472]:= L = {{42, V, 1}, {47, G, 2}, {64, S, 1}, {69, V, 2}, {64, G, 3}, {44, S, 2}}; Gather1[L, 3]
Out[2472]= {{{42, V, 1}, {64, S, 1}}, {{47, G, 2}, {69, V, 2}, {44, S, 2}}, {{64, G, 3}}}

In[899]:= Gather2[x_List] := Module[{a = Select[Gather[Flatten[x]], Length[#] > 1 &], b = {}},
    If[a == {}, Return[{}], Do[b = Append[b, {a[[k]][[1]], Length[a[[k]]]}, {k, Length[a]}];
    If[Length[b] > 1, b, First[b]]]

In[900]:= Gather2[{"a", 460, "a", 78, "y", 78, "d", "h", "c", "d", 78, 460}]
Out[900]= {"a", 2}, {460, 2}, {78, 3}, {"d", 2}
```

Следующая группа средств служит для обеспечения сортировок списков различного типа. Фрагмент иллюстрирует достаточно простую функцию **SortNL**[*L*, *p*, *b*], которая производит сортировку *числового* списка *ListList*-типа по элементам в *p*-позиции его подсписков. Сортировка *L* производится по *убыванию* (*Greater*) или *возрастанию* (*Less*) значений элементов в *p*-позиции подсписков. При недопустимом втором аргументе *p* выводится соответствующее сообщение; на недопустимых первом или третьем вызов функции возвращается невычисленным. Следующий фрагмент приводит исходный код функции **SortNL** наряду с наиболее типичными примерами ее использования.

```
In[2039]:= SortNL[L_;/; ListListQ[L] && DeleteDuplicates[Map[NumericQ[#] &,
    Flatten[L]]] == {True}, p_;/; IntegerQ[p], b_;/; MemberQ[{Greater, Less}, b]] :=
    If[p >= 1 && p <= Length[L[[1]]], Sort[L, b[#1[[p]], #2[[p]]] &],
    Print[SortNL::"incorrect second argument, should lay in interval ", {1, Length[L[[1]]}]]

In[2040]:= L := {{42, 71, 460, 78}, {47, 66, 17, 24}, {67, 43, 24, 17}, {62, 48, 17, 24}}
In[2041]:= SortNL[L, 2, Less]
Out[2041]= {{67, 43, 24, 17}, {62, 48, 17, 24}, {47, 66, 17, 24}, {42, 71, 460, 78}}
In[2042]:= SortNL[L, 2, Greater]
Out[2042]= {{42, 71, 460, 78}, {47, 66, 17, 24}, {62, 48, 17, 24}, {67, 43, 24, 17}}
In[2043]:= SortNL[L, 17, Less]
SortNL::incorrect second argument, should lay in interval {1, 4}
In[2044]:= SortNL[{{42, 71}, 460, {47, 66}}, 2, Greater]
Out[2044]= SortNL[{{42, 71}, 460, {47, 66}}, 2, Greater]
In[2045]:= SortNL[{{42, 71}, {460, 78}, {47, 66}}, 2, Art_Kr]
Out[2045]= SortNL[{{42, 71}, {460, 78}, {47, 66}}, 2, Art_Kr]

In[1062]:= SortNL1[L_;/; ListListQ[L], p_Integer, b_;/; MemberQ[{Greater, Less}, b]] :=
    If[p >= 1 && p <= Length[L[[1]]], Sort[L, b[GC[#1[[p]]], GC[#2[[p]]]] &],
    Print[SortNL::"incorrect second argument, should lay in interval ", {1, Length[L[[1]]}]]

In[2022]:= GC[x_] := ToExpression[StringReplace[ToString[ToCharacterCode[
    ToString[InputForm[x]]], {"{" -> "", "-" -> "", "}" -> ""}]]

In[2028]:= Map[GC, {78, Avz, 460, Agn, (a + b), Vsv, Art_Kr}]
Out[2028]= {3080, 935740, 134784, 736450, 774939766720, 1167020, 698177610000}
In[2025]:= SortNL1[{{42, Gs, Ps}, {17, 24, 78}, {17, Sv, 71}}, 2, Greater]
Out[2025]= {{17, Sv, 71}, {42, Gs, Ps}, {17, 24, 78}}
In[2026]:= SortNL1[{{Avz, 42, 71}, {Agn, 47, 66}, {Vsv, 67, 46}}, 1, Less]
Out[2026]= {{Agn, 47, 66}, {Avz, 42, 71}, {Vsv, 67, 46}}

In[1123]:= SortLpos[L_;/; NestListQ[L], n_;/; IntegerQ[n], SF_] :=
    If[! 1 <= n <= Min[Map[Length, L]], $Failed,
    If[! MemberQ[{Greater, Less}, SF], $Failed,
    If[DeleteDuplicates[Map[NumericQ[#] &, Flatten[L]]] != {True}, $Failed,
    Sort[L, SF[#2[[n]], #1[[n]]] &]]]
```

```
In[1124]:= L = {{66, 71, 46}, {17, 24, 48}, {42, 47, 67}, {99, 78, 460}}; L1 = {{42, 47}, {17, 24, 78}};
In[1125]:= SortLpos[L, 2, Greater]
Out[1125]= {{17, 24, 48}, {42, 47, 67}, {66, 71, 46}, {99, 78, 460}}
In[2036]:= SortLpos[L1, 2, Greater]
Out[2036]= {{17, 24, 78}, {42, 47}}
In[1127]:= {SortLpos[L, 2, GreaterEqual], SortLpos[L, 7, Less], SortLpos[L1, 1, Greater]}
Out[1127]= {$Failed, $Failed, {{17, 24, 78}, {42, 47}}}
```

Довольно полезной модификацией функции **SortNL** является функция **SortNL1**, чей вызов **SortNL1[x, p, b]** возвращает результат сортировки списка *x* типа *ListList* согласно элементов в *p*-позиции его подсписков на основе их уникальных десятичных кодов, определяемых функцией **GC**, исходный код которой представлен во фрагменте; при этом, *b* = {*Greater* | *Less*} и все сказанное по **SortNL** в полной мере относится к **SortNL1**. В определенном отношении функция **SortNL1** расширяет функцию **SortNL**, позволяя в качестве элементов сортируемого списка *x* произвольные выражения. В отличие от процедуры **SortNL** процедура **SortLpos[L, n, SF]**, представленная в конце фрагмента, обеспечивает сортировку числового *вложенного списка* типа *NestList* по элементам в *n*-позиции его подсписков. Сортировка списка *L* производится согласно сортирующей функции *SF* = {*Greater* | *Less*} соответственно по возрастанию и убыванию элементов в *n*-позициях его подсписков. Процедура **SortLpos** программно обрабатывает основные возникающие особые и ошибочные ситуации, возвращая на них значение **\$Failed**.

Вызов процедуры **SortNestList[x, p, y]** возвращает результат сортировки вложенного числового или символьного списка *x* по *p*-му элементу его подсписков в соответствии с сортирующей функцией *Greater* (>), *Less* (<) для числовых списков и *SymbolGreater* (>), *SymbolLess* (<) для символьных списков. В обоих случаях в качестве фактического аргумента *x* предполагается *вложенный* список с уровнем вложенности *единица*, иначе возвращается исходный список *x*. Тогда как в случае символьных списков сравнение элементов производится на основе их кодов. В следующем фрагменте представлены исходный код процедуры с примерами ее наиболее типичного использования.

```
In[2038]:= SortNestList[x_/, NestListQ[x], p_/, PosIntQ[p], y_] :=
Module[{a = DeleteDuplicates[Map[Length, x]], b},
b = If[SameQ[DeleteDuplicates[Map[ListNumericQ, x]], {True}] &&
MemberQ[{Greater, Less}, y], y,
If[SameQ[DeleteDuplicates[Map[ListSymbolQ, x]], {True}] &&
MemberQ[{SymbolGreater, SymbolLess}, y], y],
Return[Defer[SortNestList[x, p, y]]];
If[Min[a] <= p <= Max[a], Sort[x, b[#1[[p]], #2[[p]]] &],
Defer[SortNestList[x, p, y]]]]

In[2039]:= SortNestList[{{42, 47, 67}, {71, 66, 46}, {24, 17}}, 17, Greater]
Out[2039]= SortNestList[{{42, 47, 67}, {71, 66, 46}, {24, 17}}, 17, Greater]
In[2040]:= SortNestList[{{42, 47, 67}, {71, 66, 46}, {24, 17}}, 2, Greater]
Out[2040]= {{71, 66, 46}, {42, 47, 67}, {24, 17}}
```



```
In[2041]:= SortNestList[{"a", Avz, b1}, {x4, Agn66, y3}, {V71, G66}], 2, SymbolGreater]
Out[2041]= {{x4, Agn66, y3}, {"a", Avz, b1}, {V71, G66}}
In[2042]:= SortNestList[{{42, 47, 67}, {71, {67, 89}, 66, 46}, {24, {17, 6}}}, 2, Greater]
Out[2042]= {{42, 47, 67}, {71, {67, 89}, 66, 46}, {24, {17, 6}}}
In[2043]:= SortNestList[{"a", Avz, b1}, {x4, {x, y}, Agn65, y3}, {V70, G65}], 2, SymbolLess]
Out[2043]= {"a", Avz, b1}, {x4, {x, y}, Agn65, y3}, {V70, G65}}
```

При программировании процедуры **SortNestList** для расширения ее применимости как на числовые, так и на символьные вложенные списки оказалось целесообразным определить три новые функции, а именно: **SymbolGreater** и **SymbolLess**, как аналоги операций **Greater** и **Less** соответственно, и функцию **ListSymbolQ[x]**, возвращающую **True**, если все элементы списка *x*, включая его подсписки *любого* уровня вложенности, имеют тип *Symbol*, иначе вызов функции **ListSymbolQ** возвращает **False**. Упомянутые средства были рассмотрены нами несколько выше.

**Sort[x]** или **Sort[x, y]** – результат сортировки списка строк *x* в стандартном словарном порядке либо в порядке, определенном сортирующей функцией *y* соответственно. В качестве довольно полезного расширения функции **Sort** на отдельные строки можно предложить функцию **SortLS**, вызов которой **SortLS[x, y]** в качестве *первого* аргумента *x* допускает список из элементов типа *{Numeric, Symbol, String}*, строку и целое число; второй *необязательный* аргумент *y* имеет тот же смысл, что и у функции **Sort**. Пример представляет исходный код функции **SortLS** наряду с примерами ее применения. А в завершение приведен результат применения функции **Sort** к отдельной строке.

```
In[2013]:= SortLS[x_;/; ListQ[x] || StringQ[x] || IntegerQ[x], y___] := If[ListQ[x], Sort[x, y],
    If[StringQ[x], Evaluate, ToExpression][StringJoin[Sort[Characters[If[StringQ[x],
        x, ToString[x]]], y]]]]
In[2014]:= Map[SortLS, {"Kr17Art24", {"9", "7", "5", "3"}, "avzransian", {78, 460, 71, 24, 17}}]
Out[2014]= {"1247AKrrt", {"3", "5", "7", "9"}, "aaainnrsvz", {17, 24, 71, 78, 460}}
In[2015]:= Map[SortLS, {"Kr17", {Art24, "9", 7, "5"}, "avzrans", {ian, "A"}, {78, 4, "71", 24, 17}}]
Out[2015]= {"17Kr", {7, "5", "9", Art24}, "aanrsvz", {"A", ian}, {4, 17, 24, 78, "71"}}
In[2016]:= Map[SortLS, {"Kr17", {Art24, "9", 7, "5"}, "avzrans", {ian, "A"}, 460, 7654, -63976}]
Out[2016]= {"17Kr", {7, "5", "9", Art24}, "aanrsvz", {"A", ian}, 46, 4567, -36679}
In[2017]:= Sort["avzransia"]
Sort::normal: Nonatomic expression expected at position 1 in Sort[avzransia]. >>
Out[2017]= Sort["avzransia"]
```

Одноименная с *Maple*-процедурой процедура **PartialSums[x]** аналогично возвращает список частичных сумм элементов списка *x* с одним отличием, что при кодировании *x* в строчном формате вызов **PartialSums[x]** обновляет исходный список *x* на месте.

```
In[2217]:= PartialSums[L_;/; ListQ[L] || StringQ[L] && ListQ[ToExpression[L]]] :=
    Module[{a = {}, b = ToExpression[L], k = 1, j},
    For[k, k <= Length[b], k++, a = Append[a, Sum[b[[j]], {j, k}]]];
    If[StringQ[L], ToExpression[L <> " = " <> ToString[a]], a]]
```

```
In[2218]:= PartialSums[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}]
Out[2218]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153}
In[2219]:= GS = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}; PartialSums["GS"]; GS
Out[2219]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153}
In[2220]:= SV = {a, b, c, d, e, f}; PartialSums["SV"]; SV
Out[2220]= {a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
```

Предыдущий фрагмент представляет исходный код процедуры **PartialSums** наряду с некоторыми типичными примерами ее использования.

В целом ряде случаев возникает необходимость генерации списка переменных в виде  $Jk$  ( $k=1..n$ ), где  $J$  – имя и  $n$  – целое число. Стандартные средства **CharacterRange**, **Range** системы не решают такой задачи, поэтому для этих целей достаточно успешно можно использовать процедуры **Range1**, **Range2** или **Range3**, исходные коды которых наряду с типичными примерами использования представляет нижеследующий фрагмент.

```
In[2320]:= Range1[x_, y_] := Module[{a, b, c, h}, {a, b, c} = {Characters[ToString[x]],
    Characters[ToString[y]], Join[CharacterRange["a", "z"],
    CharacterRange["A", "Z"], {"$", "_"}]};
    h[z_] := Module[{t = Length[z], n, m, d}, For[t, t >= 1, t--, d = z[[t]];
    If[! MemberQ[c, d], Next[], n = StringJoin[z[[1 ;; t]]];
    m = StringJoin[z[[t + 1 ;; -1]]]; Break[[]]; {n, m}];
    a = Flatten[{h[a], h[b]}]; If[a[[1]] != a[[3]] || ! HowAct[a[[1]]] ||
    ! HowAct[a[[3]]] || a[[2]] == "" || a[[4]] == "" ||
    ToExpression[a[[2]] <> ">" <> a[[4]]], Return[Defer[Range1[x, y]]],
    b = Range[ToExpression[a[[2]]], ToExpression[a[[4]]]];
    ToExpression[Map3[StringJoin, a[[1]], Map[ToString, b]]]]

In[2320]:= Range1[$Kr_Art1, $Kr_Art7]
Out[2320]= {$Kr_Art1, $Kr_Art2, $Kr_Art3, $Kr_Art4, $Kr_Art5, $Kr_Art6, $Kr_Art7}
In[2320]:= Range1[$Rans_Ian1, $Rans_Ian6]
Out[2320]= {$Rans_Ian1, $Rans_Ian2, $Rans_Ian3, $Rans_Ian4, $Rans_Ian5, $Rans_Ian6}

In[2320]:= Range2[x_, y_ /; IntegerQ[y] /; y >= 1] := Module[{a = {}, b = Range[1, y], k = 1},
    For[k, k <= Length[b], k++, a = Append[a, ToString[x] <>
    ToString[b[[k]]]]]; ToExpression[a]]

In[2320]:= Range2[Kr, 14]
Out[2320]= {Kr1, Kr2, Kr3, Kr4, Kr5, Kr6, Kr7, Kr8, Kr9, Kr10, Kr11, Kr12, Kr13, Kr14}

In[2320]:= Range3[x_, y_ /; IntegerQ[y] /; y >= 1] := Module[{a = {}, b = Range[1, y], k = 1},
    For[k, k <= Length[b], k++, a = Append[a, ToString[x] <>
    ToString[b[[k]]] <> "_"]]; ToExpression[a]]

In[2320]:= Agn[Range3[z, 10]] := Module[{}, (z1 + z3 + z5 + z7 + z9)/(z2 + z4 + z6 + z8 + z10)]
In[2320]:= Range3[h, 15]
Out[2320]= {h1_, h2_, h3_, h4_, h5_, h6_, h7_, h8_, h9_, h10_, h11_, h12_, h13_, h14_, h15_}
```

```

In[2320]:= Agn[Range[24]]
Out[2320]= Agn[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}]

In[2320]:= Range4[x_, y_] := Module[{a = Select[Flatten[{x, If[{y} != {}, {y}, Null]}],
    # != "Null" &], b}, If[SameQ[DeleteDuplicates[Map[NumberQ, a]], {True}],
    Range[Sequences[a]], b = Map[FromCharacterCode, Range[32, 4096]];
    If[Length[{x, y}] == 2 && MemberQ3[b, {x, y}], CharacterRange[x, y],
        If[Length[{x, y}] == 1 && StringQ[x],
            Select[b, ToCharacterCode[#][[1]] <= ToCharacterCode[x][[1]] &], $Failed]]]]

In[2320]:= Range4["2"]
Out[2320]= {" ", "!", "\"", "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", "0", "1", "2"}
In[2320]:= Range4["9", "H"]
Out[2320]= {"9", ":", ";", "<", "=", ">", "?", "@", "A", "B", "C", "D", "E", "F", "G", "H"}
In[2320]:= Range4[42, 72, 2]
Out[2320]= {42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72}
In[2320]:= Range4[52, 72]
Out[2320]= {52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72}
In[2320]:= Range4[24]
Out[2320]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}
    
```

Вызов **Range1**[*J1, Jp*] возвращает список переменных в форме {*J1, J2, J3, ..., Jp*}, при этом, фактические аргументы кодируются в формате \$xxx\_yyyN (*N* = {0 .. *p* | 1 .. *p*}), тогда как вызов **Range2**[*J, p*] возвращает список переменных в стандартной форме, обеспечивая произвол в выборе идентификатора переменной *J*, а именно: {*J1, J2, J3, ..., Jp*}; с другой стороны, вызов процедуры **Range3**[*J, p*] возвращает список в форме {*J1\_, J2\_, J3\_, ..., Jp\_*}, где *J* – идентификатор и *p* – целое число. При этом, процедура **Range3** в ряде случаев создает достаточно хорошие предпосылки удобного определения блоков, функций и процедур с большим количеством *формальных* аргументов, как иллюстрирует пример предыдущего фрагмента. Завершает фрагмент процедура **Range4**, совмещающая обе стандартные функции **Range** и **CharacterRange** с расширением возможностей второй. Вызовы **Range4**[*x*], **Range4**[*x, y*], **Range4**[*x, y, z*] на числовых аргументах эквивалентны вызовам **Range**[*x*], **Range**[*x, y*], **Range**[*x, y, z*] стандартной функции соответственно, в то же время как вызов **Range4**[*x, y*] на строчных символах с кодами 32 ÷ 4096 эквивалентен вызову **CharacterRange**[*x, y*], тогда как вызов **Range4**[*x*] возвращает список символов с кодами, меньшими или равными **ToCharacterCode**[*x*]. Фрагмент приводит исходный код процедуры **Range4** наряду с наиболее типичными примерами ее применения. Все представленные процедуры наряду с другими процедурами настоящей книги можно найти в свободно распространяемом пакете "AVZ\_Package\_1" [120], содержащем более 580 программных средств (блоков, функций, процедур, глобальных переменных).

Следующая группа средств служит для расширения стандартной функции **MemberQ** и ее средства довольно полезны в работе со списочными структурами. Так, процедура **MemberQ1** в определенной степени расширяет стандартную функцию **MemberQ** на вложенные списки, тогда как процедура **MemberQ2**[*L, x, y*] расширяет эту же функцию

с учетом числа вхождений  $x$ -выражения в список  $L$ . Вызов процедуры **MemberQ1**[ $L, x, y$ ] возвращает значение **True**, если  $x$  является элементом любого уровня вложенности  $L$ -списка (при условии, что невложенный список имеет уровень вложенности 0); в противном случае возвращается **False**. В случае возврата **True** через третий аргумент  $y$  процедура возвращает список уровней списка  $L$ , которые содержат вхождения  $x$ -значения. Тогда как вызов процедуры **MemberQ2**[ $L, x, y$ ] возвращает значение **True**, если  $x$  – элемент  $L$  списка; в противном случае возвращается **False**. При этом, при возврате значения **True** через 3-й аргумент  $y$  возвращается число вхождений  $x$ -значения в список  $L$ . С целью упрощения алгоритма процедуры **Range4** использовалась функция **MemberQ3**, вызов которой **MemberQ3**[ $x, y$ ] возвращает **True**, если все элементы списка  $y$  входят в состав списка  $x$ , и **False** в противном случае, однако без учета вложенности. Тогда как вызов **MemberQ3**[ $x, y, t$ ] с третьим необязательным аргументом – произвольным выражением – возвращает **True**, если список  $y$  – подсписок списка  $x$  на любом уровне вложенности, и **False** в противном случае. Вызов функции **MemberQ4**[ $x, y$ ] возвращает **True**, если по меньшей мере один элемент списка  $y$  либо элемент  $y$  входит в состав списка  $x$ , и **False** в противном случае. При наличии 3-го необязательного аргумента  $z$  **True** возвращается только при числе вхождений элементов  $y$  не меньшим, чем  $z$ . Фрагмент представляет исходные коды всех 4-х упомянутых средств и типичные примеры их использования.

```
In[2027]:= MemberQ1[L_List, x_, y_ /; ! HowAct[y]] := Module[{a = Flatten[L], b = L, c = 0,
    p = {}}, While[! B == {}, If[MemberQ[b, x], p = Append[p, c], Null];
    b = Select[b, ListQ[#] &]; b = Flatten[b, 1]; c = c + 1;
    If[p == {}, False, y = p; True]]

In[2028]:= L = {42, 47, 66, 71, {{66}}, 71, {{{71}}}, {71}, {66, {{71}}, {{{{71}}}}}, 460, 71}; x = 42;
In[2029]:= {{MemberQ1[L, x, y], y}, {MemberQ1[L, 71, z], z}, {MemberQ1[L, 66, h], h}}
Out[2029]= {{True, {0}}, {True, {0, 1, 3, 6}}, {True, {0, 1, 2}}}

In[2030]:= MemberQ2[L_ /; ListQ[L], x_, y_ /; ! HowAct[y]] := Module[{b = Flatten[L], c = 0,
    k = 1}, If[MemberQ[b, x], For[k, k <= Length[b], k++,
    If[b[[k]] === x, c = c + 1, Next[]]]; y = c; True, False]]

In[2031]:= {MemberQ2[L, 71, g], g}
Out[2031]= {True, 7}

In[2032]:= MemberQ3[x_List, y_List, z_] := If[{z} != {},
    MemberQ[SubsStrLim[ToString[x], "{", "}"], ToString[y]],
    SameQ[DeleteDuplicates[Map3[MemberQ, Flatten[x], Flatten[y]]], {True}]]

In[2033]:= MemberQ3[{{c, a}, {a, b, c}, {c, d, {x, y}}, {m, n}}, {a, b, c}]
Out[2033]= True

In[2039]:= MemberQ4[x_List, y_, z_] := If[ListQ[y], Count[(MemberQ[x, #1] &) /@ y,
    True] >= If[{z} === {}, 1, If[IntegerQ[z], z, 1]], MemberQ[x, y]]

In[2040]:= MemberQ4[{a, {b, a}, c, {d, z}, h}, {n, {b, a}, y, z, {d, z}, x}, 2]
Out[2040]= True
```

В принципе, процедуры предыдущего фрагмента допускают ряд весьма интересных модификаций, существенно расширяющих *сферу* применения этих средств. Оставим это заинтересованному читателю в качестве достаточно полезного упражнения.

Следующий фрагмент представляет 2 простых средства, первое из которых функция **MemberT**[*L*, *x*], возвращающая общее число вхождений выражения *x* в список *L*, тогда как процедура **MemberLN**[*L*, *x*] возвращает список *ListList*-типа, каждый из *подписков* которого определяет номер уровня вложенности вложенного списка *L* первым своим элементом и число вхождений выражения *x* в данный уровень вторым элементом.

```
In[2326]:= MemberT[L_/, ListQ[L], x_] := Length[Select[Flatten[L], SameQ[#, x] &]]
In[2327]:= MemberLN[L_/, NestQL[L], x_] := Module[{a = L, b = {}, c = 0, d, k, p = 0, h = {}},
  While[a != {}, c = c + 1; For[k = 1, k <= Length[a], k++, d = a[[k]];
    If[d === x, p = p + 1, If[ListQ[d], AppendTo[b, d], Null]]];
  h = Append[h, {c, p}]; a = Flatten[b, 1]; b = {}; p = 0; h]
In[2328]:= L = {a, b, {c, d, {d, x, d}, x}, {d, x, h, d}, p, {{d, {d, {a, d, c, d}}}}, d, w}
Out[2328]= {a, b, {c, d, {d, x, d}, x}, {d, x, h, d}, p, {{d, {d, {a, d, c, d}}}}, d, w}
In[2329]:= {MemberLN[L, d], MemberLN[L, t], MemberT[{a, h, {c, h, g}, s}, h]}
Out[2329]= {{{1, 1}, {2, 3}, {3, 3}, {4, 1}, {5, 2}}, {{1, 0}, {2, 0}, {3, 0}, {4, 0}, {5, 0}}, 2}
In[2330]:= {MemberLN[{{{{{x, {{{y}}}}}}}, 2], MemberT[{a, h, {c, {{{{y}}}}}, h, v}, s], 4]}
Out[2330]= {{{1, 0}, {2, 0}, {3, 0}, {4, 0}, {5, 0}, {6, 0}, {7, 0}, {8, 0}}, 0}
```

Фрагмент представляет исходные коды обеих средств с примерами их использования. Отметим, что средства **MemberT** и **MemberLN** предполагают, что *невложенный* список *x* имеет уровень вложенности 1. Итак, наряду с **MemberQ1**, **MemberQ2**, **MemberQ3** и **MemberQ4** средства **MemberT** и **MemberLN** достаточно полезны в обработке списков.

Система рассматривает список, как объект, допускающий кратные вхождения в него элементов и сохраняющий порядок элементов, заданный при его определении. Для определения кратности входящих в список элементов можно использовать довольно простую функцию **MultEntryList**, вызов **MultEntryList**[*x*] которой возвращает *ListList*-список, *первый* элемент подписков которого определяет элемент списка *x*, тогда как *второй* определяет его кратность в списке *x* независимо от его вложенности. Фрагмент представляет исходный код функции **MultEntryList** с примерами ее использования.

```
In[2018]:= MultEntryList[x_/, ListQ[x]] := Map[#[[1]], Length[#] &, Gather[Flatten[x]]]
In[2019]:= MultEntryList[{"a", b, "a", c, h, 71, 71, {"a", b, c, h, g, 71}, g, h, {71, h, g, h, 71}}]
Out[2019]= {"a", 3}, {b, 2}, {c, 2}, {h, 5}, {71, 5}, {g, 3}
In[2020]:= MultEntryList[{}]
Out[2020]= {}
```

В отличие от двух стандартных функций **Split** и **SplitBy** вызов процедуры **Split1**[*x*, *y*] разбивает список *x* на подписки, состоящие из его элементов, расположенных между вхождениями элемента либо элементов списка *y*. Если *y* не принадлежат списку *x*, то возвращается исходный список *x*. Следующий фрагмент представляет исходный код процедуры **Split1** с наиболее типичными примерами ее использования.

```

In[2046]:= Split1[x_List, y_] := Module[{a, b, c = {}, d, h, k = 1},
      If[MemberQ3[x, y] || MemberQ[x, y], a = If[ListQ[y],
Sort[Flatten[Map[Position[x, #] &, y]], Flatten[Position[x, y]]]; h = a;
      If[a[[1]] != 1, PrependTo[a, 1]]; If[a[[-1]] != Length[x], AppendTo[a, Length[x]]];
      d = Length[a];
      While[k <= d - 1, AppendTo[c, x[[a[[k]]] ;; If[k == d - 1, a[[k + 1]], a[[k + 1]] - 1]]]; k++;
      If[h[[-1]] == Length[x], AppendTo[c, {x[[-1]]}]; c, x]]

In[2047]:= Split1[{a, a, a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}, a]
Out[2047]= {{a}, {a}, {a, b}, {a, b, c, d}, {a, b}, {a, b, c, d}, {a, b, d}}
In[2048]:= Split1[{a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}, {a, c, d}]
Out[2048]= {{a, b}, {a, b}, {c}, {d}, {a, b}, {a, b}, {c}, {d}, {a, b, d}, {d}}
In[2049]:= Split1[{a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}, {x, y, z}]
Out[2049]= {a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}

```

Следующая достаточно полезная процедура работы со списками носит структурный характер, прежде всего, для *вложенных* списков. В общем случае вызов **ElemOnLevels[x]** возвращает вложенный список, элементами которого являются подсписки, первыми элементами которых являются *уровни* вложенного списка *x*, в то время как остальные – элементы данного уровня. В отсутствие элементов на *w*-уровне подсписок имеет вид {*w*}; вызов **ElemOnLevels[x]** на простом списке *x* возвращает {**1**, *x*}, т.е. простой список имеет уровень вложенности **1**. Следующий фрагмент представляет как исходный код процедуры **ElemOnLevels**, так и наиболее типичные примеры ее использования.

```

In[2136]:= ElemOnLevels[x_List] := Module[{a, b, c, d, p = 1, k, j = 1},
      If[! NestListQ1[x], Flatten[{1, x}], {a, c, d} = {x, {}, {}};
      While[NestListQ1[a], b = {p++}; For[k = 1, k <= Length[a], k++,
      If[! ListQ[a[[k]]], AppendTo[b, a[[k]]]; AppendTo[c, k]]; AppendTo[d, b];
      a = Flatten[Delete[a, Map[List, c], 1]; {b, c} = {{}, {}}; j++];
      AppendTo[d, Flatten[{p++, a}]]]

In[2137]:= ElemOnLevels[{a, b, {c, d, {f, h, d}, s, {p, w, {n, m, r, u}, t}}, x, y, z]}
Out[2137]= {{1, a, b, x, y, z}, {2, c, d, s}, {3, f, h, d, p, w, t}, {4, n, m, r, u}}
In[2138]:= ElemOnLevels[{a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z]}
Out[2138]= {1, a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}
In[2139]:= ElemOnLevels[{{{a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}}}]
Out[2139]= {{1}, {2}, {3, a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}}
In[2140]:= Map[ElemOnLevels, {{{{}}, {}, {{{{}}}}}]
Out[2140]= {{{1}, {2}, {3}}, {1}, {{1}, {2}, {3}, {4}, {5}}}

```

Если при работе в *интерактивном* режиме диагностические сообщения имеют вполне определенный смысл, в программном (*непрерывном*) режиме выполнения, например, процедур сообщения по особым ситуациям смысла не имеют, затрудняя *программную* обработку таких ситуаций. И в этом контексте более естественно идентифицировать особую ситуацию возвратом удобно обрабатываемого выражения, например, **\$Failed**.

```

In[3378]:= ElemsList[x_List, y_List] := Module[{a = Select[y, ! ListQ[#] &], c = "", k = 1,
                                                b = Select[y, ListQ[#] &]},
    If[a == {} && b == {}, x,
    If[a == {}, Quiet[Check[ToExpression[ToString[x] <> "[" <>
        StringTake[ToString[b], {3, -3}] <> "]", $Failed]],
    If[b == {}, c = ToString[x];
    While[k <= Length[a], c = c <> "[" <> ToString[a[[k]]] <> "]; k++];
        Quiet[Check[ToExpression[c], $Failed]],
        c = ToString[x];
    While[k <= Length[a], c = c <> "[" <> ToString[a[[k]]] <> "]; k++];
        Quiet[Check[ToExpression[c <> "[" <>
            StringTake[ToString[b], {3, -3}] <> "]", $Failed]]]]]]

In[3379]:= L = {{avz, agn, vsv, art, kr}, {d, e, f, g, h, {17, 24, 46, 50, 66, 71}}, {g, h, j}};
In[3380]:= ElemsList[{}, {}]
Out[3380]= {}
In[3381]:= ElemsList[L, {}]
Out[3381]= {{avz, agn, vsv, art, kr}, {d, e, f, g, h, {17, 24, 46, 50, 66, 71}}, {g, h, j}}
In[3382]:= ElemsList[L, {{1 ;; 3}}]
Out[3382]= {{avz, agn, vsv, art, kr}, {d, e, f, g, h, {17, 24, 46, 50, 66, 71}}, {g, h, j}}
In[3383]:= ElemsList[L, {2, 6, {3 ;; -1}}]
Out[3383]= {46, 50, 66, 71}
In[3384]:= ElemsList[L, {2, 6, 5}]
Out[3384]= 66
In[3385]:= ElemsList[L, {2, 78.460, 5}]
Out[3385]= $Failed
In[3386]:= L[[2]][[6]][[3 ;; 0]]
Part::take: Cannot take positions 3 through 0 in {17, 24, 46, 50, 66, 71}. >>
Out[3386]= {17, 24, 46, 50, 66, 71}[[3 ;; 0]]
In[3387]:= ElemsList[L, {2, 6, {3 ;; 0}}]
Out[3387]= $Failed

```

Примером тому служит и процедура, чей успешный вызов **ElemsList** $[x, y]$  возвращает элементы списка  $x$  в зависимости от их списка позиций, заданных списком  $y$ . Формат списка  $y$  в общем случае имеет вид  $\{n1, \dots, nt, \{m1; \dots; mp\}\}$ , возвращая элементы списка  $x$  согласно стандартному соотношению  $x[[n1]] \dots [[nt]][[m1; \dots; mp]]$ . Аргумент  $y$ , между тем, допускает следующие форматы кодировки  $\{n1, \dots, nt\}$ ,  $\{\{m1; \dots; mp\}\}$ ,  $\{\}$ ; результаты использования которых приведены во фрагменте с исходным кодом процедуры.

Следующие 2 процедуры расширяют средства системы, ориентированные на работу со списочными структурами, давая возможность достаточно существенно упрощать программирование целого ряда задач, использующих списки. Следующий фрагмент представляет исходные коды процедур с типичными примерами их применения. Так, вызов **ReduceList** $[L, x]$  возвращает результат сведения к кратности 1 элементов списка

$L$ , которые определены отдельным элементом либо их списком  $x$ . Если же элементы  $x$  не принадлежат списку  $L$ , то вызов процедуры возвращает исходный список  $L$ . Тогда как вызов процедуры **SplitList**[ $L, x$ ] возвращает результат разбиения исходного списка  $L$  на подсписки элементом либо элементами  $x$ ; при этом, разделители  $x$  из результата удаляются. Если элементы  $x$  не принадлежат списку  $L$ , вызов процедуры возвращает исходный список  $L$ . В целом ряде случаев обе процедуры достаточно востребованы.

```
In[2540]:= ReduceList[L_List, x_] := Module[{a = {}, b, c = Flatten[{x}], d, k = 1},
                                         d = MinusList[c, MinusList[c, L]];
                                         If[d == {}, L, For[k, k <= Length[L] - 1, k++, b = L[[k]];
                                         If[! MemberQ[d, b] || (MemberQ[d, b] && ! MemberQ[d, L[[k + 1]]]),
                                         AppendTo[a, b]]]; If[MemberQ[d, L[[-1]]], a = AppendTo[a, L[[-1]]], a]]

In[2541]:= ReduceList[{a, b, c, d, d, f, f, f, g}, f]
Out[2541]= {a, b, c, d, d, f}
In[2542]:= ReduceList[{f, f, a, b, c, d, d, f, f, f, g, f}, f]
Out[2542]= {f, a, b, c, d, d, f, g, f}
In[2543]:= ReduceList[{f, f, a, b, c, d, d, f, f, f, g, f}, {f, d}]
Out[2543]= {f, a, b, c, f, g, f}
In[2544]:= ReduceList[{f, f, a, b, c, d, d, f, f, f, g, f, d, d, d}, {f, d}]
Out[2544]= {f, a, b, c, f, g, d}
In[2545]:= ReduceList[{f, f, a, b, c, d, d, f, f, f, g, f, d, d, d}, {t, h}]
Out[2545]= {f, f, a, b, c, d, d, f, f, f, g, f, d, d, d}

In[9]:= SplitList[L_List, x_] := Module[{a = Flatten[{x}], b = ToString[Unique["$a"]], c, d, h},
                                         c = Map[ToString[#] <> b &, a];
                                         d = StringJoin[Map[ToString[#] <> b &, L]];
                                         h = Select[StringSplit[d, c], # != "" &];
                                         h = Map[StringReplace[#, b -> ","] &, h];
                                         h = ToExpression[Map["{" <> StringTake[#, {1, -2}] <> "} " &, h]];
                                         If[Length[h] == 1, h[[1]], h]]

In[10]:= SplitList[{f, f, a, b, c, d, p, p, d, p, d, f, f, f, g, f}, {d, f}]
Out[10]= {{a, b, c}, {p, p}, {p}, {g}}
In[11]:= SplitList[{f, f, a, b, c, d, p, d, f, f, f, g, f}, {h, f}]
Out[11]= {{a, b, c, d, p, d}, {g}}
In[12]:= SplitList[{f, f, a, b, c, d, p, d, f, f, f, g, f}, {h, t}]
Out[12]= {f, f, a, b, c, d, p, d, f, f, f, g, f}
```

Глава представляет целый ряд средств, расширяющих программную среду системы, в частности, для довольно эффективного программирования задач манипулирования со списочными структур различной организации. В целом же, система предоставляет массу полезных и эффективных средств обработки (кроме уже упомянутых) списочных структур и объектов, базирующихся на данного типа структурах.



## Глава 7. Дополнительные средства, расширяющие стандартные функции *Mathematica* либо ее программную среду в целом

*Строчная* и *списочная* структуры – одни из важнейших в системе *Mathematica*, обе они рассмотрены в предыдущих двух главах в контексте средств, дополнительных к системным средствам, не взирая на большое число стандартных функций обработки такого типа структур. Естественно, весь спектр системных функций этого типа здесь рассмотреть не представляется возможным, отправляя заинтересованного читателя к справочной информации по системе или к соответствующей весьма многочисленной и зачастую превосходной литературе, и, прежде всего, англоязычной. В последней на сегодня мы имеем целый ряд довольно квалифицированных как с писательской, так и профессионально-ориентированной точки зрения авторов. Со многими из данных изданий можно ознакомиться на сайте разработчиков <http://www.wolfram.com/books>.

Представив средства, расширяющие стандартную программную среду *Mathematica* в контексте обработки строчной и списочной структур, в настоящей главе представим средства, расширяющие данную среду и ориентированные на обработку иных типов объектов. Прежде всего представим средства побитовой обработки символов.

Процедура **Bits**[*x*, *P*] довольно существенно использует **BinaryListQ**, обеспечивая ряд полезных функций при работе с символами. На кортеже фактических аргументов <*x*, *P*>, где *x* – односимвольная строка (*character*) и *P* – целое число в диапазоне 0 .. 8, вызов процедуры **Bits**[*x*, *P*] возвращает бинарное представление *x* в форме списка, если *P*=0, и *P*-й бит такого представления символа *x* в противном случае. Тогда как на кортеже фактических аргументов <*x*, *P*>, где *x* – непустой бинарный список длины не более 8 и *P*=0, процедура возвращает символ, соответствующий данному бинарному списку *x*; в остальных случаях вызов процедуры **Bits**[*x*, *P*] возвращается невычисленным. Ниже представлен исходный код процедуры **Bits** наряду с примерами ее использования.

```
In[2019]:= Bits[x_, P_Integer] := Module[{a, k}, If[StringQ[x] && StringLength[x] == 1,
    If[1 <= P <= 8, PadLeft[IntegerDigits[ToCharacterCode[x][[1]], 2], 8][[P]],
    If[P == 0, PadLeft[IntegerDigits[ToCharacterCode[x][[1]], 2], 8], Defer[Bits[x, P]]],
    If[BinaryListQ[x] && 1 <= Length[Flatten[x]] <= 8, a = Length[x];
    FromCharacterCode[Sum[x[[k]]*2^(a - k), {k, 1, a}], Defer[Bits[x, P]]]]]
In[2020]:= Map9[Bits, {"A", "A", {1, 0, 0, 0, 0, 0, 1}, "A", {1, 1, 1, 1, 0, 1}}, {0, 2, 0, 9, 0}]
Out[2020]= {{0, 1, 0, 0, 0, 0, 0, 1}, 1, "A", Bits["A", 9], "="}
```

Итак, если предыдущая процедура **Bits** обеспечивает достаточно простую обработку символов, то следующие 2 процедуры **BitSet1** и **BitGet1** обеспечивают расширенную побитовую обработку информации подобно нашим процедурам пакета *Maple* [45].

В *Maple* нами был создан ряд процедур (**Bit**, **Bit1**, **xbyte**, **xbyte1**, **xNB**), обеспечивающих побитовую обработку информации [45]; *Mathematica* также располагает подобными средствами, в частности, функцией **BitSet**[*n*, *k*], возвращающей результат установки в *k*-ю позицию бинарного представления числа *n* единицы. Нижеследующий фрагмент представляет процедуру, вызов **BitSet1**[*n*, *p*] которой возвращает результат установки

в позиции бинарного представления целого числа  $n$ , которые определяются первыми элементами подписков вложенного списка  $p$ ,  $\{0 | 1\}$ -значений; в случае невложенного списка  $p$  производится замена значения только в единственной позиции числа  $n$ .

```
In[2038]:= BitSet1[n_ /; IntegerQ[n] && n >= 0, p_ /; ListQ[p]] := Module[{b = 1, c, d,
    a = ToExpression[Characters[IntegerString[n, 2]]], h = If[ListListQ[p], p, {p}]},
    If[ListListQ[h] && Length[Select[h, Length[#] == 2 &&
        IntegerQ#[[1]] && IntegerQ#[[2]] &&
        MemberQ[{0, 1}, #[[2]]] &]] == Length[h], Null, Return[Defer[BitSet1[n, p]]];
    For[b, b <= Length[h], b++, {c, d} = {h[[b]][[1]], h[[b]][[2]]};
    If[c <= Length[a], a[[c]] = d, Null]; Sum[a[[k]]*2^(Length[a] - k), {k, Length[a]}]]

In[2039]:= {BitSet1[460, {{3, 1}, {6, 0}, {9, 1}}], BitSet1[78, {4, 0}], BitSet1[78, {7, 1}]}
Out[2039]= {453, 70, 79}
In[2040]:= BitSet1[460, {{3, 1}, {6, 0}, {9, 2}}]
Out[2040]= BitSet1[460, {{3, 1}, {6, 0}, {9, 2}}]

In[9]:= BitGet1[x___, n_ /; IntegerQ[n] && n >= 0, p_ /; IntegerQ[p] && p > 0 | | ListQ[p]]:=
    Module[{b = 1, c = {}, d, a = ToExpression[Characters[IntegerString[n, 2]]],
        h = If[ListQ[p], p, {p}]},
    For[b, b <= Length[a], b++, c = Append[c, If[MemberQ[h, b], a[[b]], Null]];
    If[! HowAct[x], x = Length[a], Null]; Select[c, ToString[#] != "Null" &]]

In[10]:= {BitGet1[h, 78, {1, 5, 7}], h, BitGet1[78, {1, 5, 7}], BitGet1[p, 460, {1, 3, 5, 7, 9}], p}
Out[10]= {{1, 1, 0}, 7, {1, 1, 0}, {1, 1, 0, 1, 0}, 9}
```

Примеры применения процедуры **BitSet1** весьма наглядно иллюстрируют сказанное. Следует отметить, что функционально процедура **BitSet1** перекрывает стандартные функции **BitSet** и **BitClear** системы *Mathematica*. В то время как вызов **BitGet1** $[x, n, p]$  функционально перекрывает стандартные функции **BitGet**, **BitLength** системы; вызов **BitGet1** $[n, p]$  возвращает список битов в позициях бинарного представления целого  $n$  числа, определяемых списком  $p$ ; в случае целого  $p$  возвращается бит в позиции  $p$  числа  $n$ . Тогда как вызов **BitGet1** $[x, n, p]$  через символ  $x$  дополнительно возвращает число бит в бинарном представлении целого числа  $n$ . В предыдущем фрагменте представлены исходные коды процедур **BitSet1**, **BitGet1** и примеры их применения, которые весьма наглядно иллюстрируют вышесказанное без каких-либо дополнительных пояснений.

В среде *Mathematica* правила преобразования в общем случае определяются функцией **Rule**, вызов которой **Rule** $[a, b]$  возвращает правило преобразования в формате  $a \rightarrow b$ . Эти правила используются в преобразованиях выражений следующими функциями: **ReplaceAll** (**/.**), **Replace**, **ReplaceRepeated** (**/>.**), **ReplacePart**, **StringReplaceList**, **StringCases**, **StringReplace**, которые используют или одно правило, или их список как простой, так и **ListList**-типа. Для динамической генерации таких правил может оказаться довольно полезной процедура **GenRules**, вызов которой **GenRules** $[x, y]$  в зависимости от вида ее аргументов возвращает правило или список правил; вызов **GenRules** $[x, y, z]$  с третьим необязательным аргументом  $z$ , в качестве которого допустимо произвольное выражение,

возвращает список с одним правилом или вложенный список *ListList*-типа. А именно, в зависимости от формата кодирования вызова процедуры возвращается результат:

```
GenRules[{x, y, z, ...}, a] => {x -> a, y -> a, z -> a, ...}
GenRules[{x, y, z, ...}, a, h] => {{x -> a}, {y -> a}, {z -> a}, ...}
GenRules[{x, y, z, ...}, {a, b, c, ...}] => {x -> a, y -> b, z -> c, ...}
GenRules[{x, y, z, ...}, {a, b, c, ...}, h] => {{x -> a}, {y -> b}, {z -> c}, ...}
GenRules[x, {a, b, c, ...}] => {x -> a}
GenRules[x, {a, b, c, ...}, h] => {x -> a}
GenRules[x, a] => {x -> a}
GenRules[x, a, h] => {x -> a}
```

Следующий фрагмент представляет *исходный* код процедуры **GenRules** с типичными примерами ее применения на все вышеуказанные случаи кодирования ее вызова.

```
In[4040]:= GenRules[x_, y_, z_] := Module[{a, b = Flatten[{x}],
      c = Flatten[If[ListQ /@ {x, y} == {True, False},
        PadLeft[{}, Length[x], y], {y}]]], a = Min[Length /@ {b, c}];
      b = Map9[Rule, b[[1 ;; a]], c[[1 ;; a]]];
      If[{z} == {}, b, b = List /@ b; If[Length[b] == 1, Flatten[b], b]]]
In[4041]:= {GenRules[{x, y, z}, {a, b, c}], GenRules[x, {a, b, c}], GenRules[{x, y}, {a, b, c}],
      GenRules[x, a], GenRules[{x, y}, a]}
Out[4041]= {{x -> a, y -> b, z -> c}, {x -> a}, {x -> a, y -> b}, {x -> a}, {x -> a, y -> a}}
In[4042]:= {GenRules[{x, y, z}, {a, b, c}, 71], GenRules[x, {a, b, c}, 42], GenRules[x, a, 6],
      GenRules[{x, y}, {a, b, c}, 47], GenRules[{x, y}, a, 66]}
Out[4042]= {{{x -> a}, {y -> b}, {z -> c}}, {x -> a}, {{x -> a}, {y -> b}}, {x -> a}, {{x -> a}, {y -> a}}}
In[4043]:= GenRules[x_, y_, z_] := Module[{a, b}, b = If[ListQ[x] && ! ListQ[y],
      a = Map[Rule[#, y] &, x], If[ListQ[x] && ListQ[y],
      a = Map9[Rule, x, y], {x -> Flatten[{y}][[1]]}];
      b = If[{z} != {}, SplitBy[b, Head[#] & == Rule], b];
      If[NestListQ[b] && Length[b] == 1, b[[1]], b]]
In[4044]:= {GenRules[{x, y, z}, {a, b, c}], GenRules[x, {a, b, c}], GenRules[{x, y}, {a, b, c}],
      GenRules[x, a], GenRules[{x, y}, a]}
Out[4044]= {{x -> a, y -> b, z -> c}, {x -> a}, Map9[Rule, {x, y}, {a, b, c}], {x -> a}, {x -> a, y -> a}}
In[4045]:= {GenRules[{x, y, z}, {a, b, c}], GenRules[x, {a, b, c}], GenRules[{x, y, z}, {a, b, c}],
      GenRules[x, a], GenRules[{x, y}, a]}
Out[4045]= {{x -> a, y -> b, z -> c}, {x -> a}, {x -> a, y -> b, z -> c}, {x -> a}, {x -> a, y -> a}}
In[4046]:= {GenRules[{x, y, z}, {a, b, c}, 71], GenRules[x, {a, b, c}, 42], GenRules[x, a, 6],
      GenRules[{x, y}, {a, b, c}, 47], GenRules[{x, y}, a, 66]}
Out[4046]= {{{x -> a}, {y -> b}, {z -> c}}, {x -> a}, {{x -> a}, {y -> b}}, {x -> a}, {{x -> a}, {y -> a}}}
```

Процедура **GenRules** полезна, в частности, когда в некоторой процедуре необходимо

динамически генерировать правила преобразований в зависимости от условий. Более того, в ряде случаев она позволяет довольно существенно сокращать исходный код. В качестве одной полезной модификации процедуры приводится процедура **GenRules** того же имени, которая функционально эквивалентна первоначальной при условии, что списки  $x$  и  $y$  в качестве 2 первых аргументов имеют одинаковую длину. Исходный код второго варианта процедуры с примерами применения завершают фрагмент. Со средствами, обеспечивающими работу с правилами преобразования, детально можно ознакомиться в литературе, приведенной в книге, либо по справочной базе системы.

Наряду с рассмотренными правилами типа  $a \rightarrow b$  пакет допускает применение также *отложенных* правил (*RuleDelayed*) типа  $a \rightarrow b$  или  $a \rightarrow b$ , которые реализуются только в момент их применения. В остальном они аналогичны уже рассмотренным. С целью генерации списков данного типа правил преобразования может быть использована и представленная выше процедура **GenRules**, для которой функция **Rule** заменяется на функцию **RuleDelayed**, или используется ее модификация **GenRules1**, адаптируемая на использование той или иной функции путем соответствующего кодирования 3-го фактического аргумента в вызове **GenRules1**[ $x, y, h, z$ ], где  $x, y, z$  – аргументы, которые полностью аналогичны одноименным аргументам процедуры **GenRules**, тогда как  $h$  аргумент определяет режим генерации списка обычных либо отложенных правил на основе полученного им значения "r" (*простое правило*) либо "rd" (*отложенное правило*). Следующий фрагмент представляет исходный код процедуры **GenRules1** совместно с наиболее типичными примерами ее использования.

```
In[2023]:= GenRules1[x_, y_, h_ /; h == "r" | | h == "rd", z_] := Module[{a, b = Flatten[{x}],
                                c = Flatten[If[Map[ListQ, {x, y}] == {True, False},
                                PadLeft[{}, Length[x], y], {y}]]], a = Min[Map[Length, {b, c}]];
                                b = Map9[If[h == "r", Rule, RuleDelayed], b[[1 ;; a]], c[[1 ;; a]]];
                                If[{z} == {}, b, b = Map[List, b]; If[Length[b] == 1, Flatten[b], b]]]

In[2024]:= GenRules1[{x, y, z}, {a, b, c}, "r", 78]
Out[2024]= {{x -> a}, {y -> b}, {z -> c}}

In[2025]:= GenRules1[{x, y, z}, {a, b, c}, "rd", 78]
Out[2025]= {{x :-> a}, {y :-> b}, {z :-> c}}

In[2026]:= GenRules1[{x, y, z}, a, "r"]
Out[2026]= {x -> a, y -> a, z -> a}

In[2027]:= GenRules1[{x, y, z}, a, "rd"]
Out[2027]= {x :-> a, y :-> a, z :-> a}

In[2028]:= GenRules1[{x, y}, {a, b, c, d}, "rd", 460]
Out[2028]= {{x :-> a}, {y :-> b}}

In[2029]:= GenRules1[x, a, "rd", 78]
Out[2029]= {x :-> a}
```

Вполне определенный интерес для практического программирования в *Mathematica* может представить также прием, примененный для настройки процедуры на режим выполнения; он использован в целом ряде процедур из пакета *AVZ\_Package\_1* [120].

Учитывая важность однотипной *map*-функции, начиная с *Maple 10*, была определена опция *inplace*, допустимая только в использовании этой функции с прямоугольными *rtable*-объектами и обновляющая эти объекты *на месте*. Тогда как для объектов *другого* типа этот механизм не поддерживается, как иллюстрируют примеры из [99]. С целью устранения данного недостатка нами была предложена довольно простая процедура *MapInSitu* [45,99]. Наряду с этим нами были предложены аналогичные средства и для *Mathematica* в виде двух функций *MapInSitu*, *MapInSitu1*, и процедуры *MapInSitu2*. Фрагмент приводит исходные коды указанных средств с примерами их применения.

```
In[2050]:= MapInSitu[x_, y_ /; StringQ[y]] := ToExpression[y <> "=" <>
                                     ToString[Map[x, ToExpression[y]]]]

In[2051]:= y = {a, b, c}; h = {{4.2, 7.1}, {4.7, 6.6}}; {MapInSitu[G, "y"], MapInSitu[Sin, "h"]}
Out[2051]= {{G[a], G[b], G[c]}, {{-0.871576, 0.728969}, {-0.999923, 0.311541}}}
In[2052]:= {y, h}
Out[2052]= {{G[a], G[b], G[c]}, {{-0.871576, 0.728969}, {-0.999923, 0.311541}}}
In[2053]:= {H, G} = {{7.8, 47.66, 17.24}, {7.8, 47.66, 17.24}}
Out[2053]= {{7.8, 47.66, 17.24}, {7.8, 47.66, 17.24}}

In[2054]:= MapInSitu1[x_, y_] := ToExpression[ToString[Args[MapInSitu, 78]] <> "=" <>
                                     ToString[Map[x, y]]]

In[2055]:= y = {{78.42, 24.56}, {78.45, 78.89}}; MapInSitu1[Sin, y]
Out[2055]= {{0.11953, -0.541938}, {0.0896956, -0.34307}}
In[2056]:= y
Out[2056]= {0.0896956, -0.34307}

In[2057]:= MapInSitu2[x_, y_] := Module[{a = Map[x, y], b = ToString[y], d = {}, k = 1, h,
                                     c = Select[Names["`*"], StringFreeQ[#, "$"] &]],
                                     For[k, k <= Length[c], k++, h = c[[k]];
                                     If[ToString[ToExpression[h]] == b, d = Append[d, h], Null]];
                                     For[k = 1, k <= Length[d], k++, h = d[[k]];
                                     ToExpression[h <> " = " <> ToString[a]]]; a]

In[2058]:= MapInSitu2[Sin, {7.8, 47.66, 17.24}]
Out[2058]= {0.998543, -0.510796, -0.999249}
In[2059]:= {H, G}
Out[2059]= {{0.998543, -0.510796, -0.999249}, {0.998543, -0.510796, -0.999249}}
```

Механизм, реализуемый алгоритмом процедуры *MapInSitu(x, y)*, использует условие, состоящее в кодировании переменной *y*, определяющей выражение, к которому *map*-процедура применяется, в *строчном* формате, как иллюстрирует пример применения процедуры *MapInSitu* к списочной структуре. Аналогичный механизм реализован и в среде пакета *Mathematica* одноименной функцией *MapInSitu[x, y]*, предполагая при вызове процедуры кодирования фактического аргумента *y* в строчном формате. Этот подход позволяет обновлять на месте и целый ряд других результатов вычислений. В

качестве еще одного полезного примера фрагмент представляет еще один механизм, реализованный функцией **MapInSitu1**[*x*, *y*] в предположении, что для нее аргумент *y*, определяющий выражение, обрабатываемое функцией **Map**, остается постоянным.

Средство **MapInSitu** для обеих систем характеризуется той предпосылкой, что второй аргумент при его вызове указывает на идентификатор в *строчном* формате, которому ранее было присвоено некоторое значение, обновляемое *на месте* после его обработки функцией **{map | Map}**. Тогда как, во многих случаях в качестве второго фактического аргумента выступает непосредственно *само* обрабатываемое выражение. Как правило, в такой ситуации результат вызова либо присваивается некоторому идентификатору, или непосредственно входит в состав некоторого выражения. Следующая процедура **MapInSitu1**[*x*, *y*] обеспечивает присвоение *всем* идентификаторам, которым в текущем сеансе пользователем были присвоены значения, совпадающие со значением второго фактического аргумента *y*, результат вызова **Map**[*x*, *y*], обновляя их значения *на месте*. Вторая часть фрагмента представляет процедуру **MapInSitu2**, являющуюся аналогом функции **MapInSitu1** в среде системы *Mathematica*, с примерами ее применения. При этом, достаточно близки также и реализуемые обеими средствами алгоритмы. Между тем, вызовы этих процедур возвращают в качестве результата **Map**[*x*, *y*]. Предыдущий фрагмент представляет исходные коды всех процедур и примеры их использования.

Довольно полезной при анализе и обработке выражений дополнительно к функции **Head** представляется функция **Part**, допускающая шесть форматов кодирования [101]. Между функциями **Head**, **Level**, **Part** существует ряд полезных соотношений, которые могут использоваться для задач тестирования выражений, в частности, **Part**[*Expr*, 0]  $\equiv$  **Head**[*Expr*], **Level**[*Expr*, 1][[1]]  $\equiv$  **Part**[*Expr*, 1], **Level**[*Expr*, *Infinity*]  $\equiv$  **Level**[*Expr*, -1], где *Expr* – произвольное выражение, и др. Указанные средства могут не только довольно успешно использоваться для тестирования выражений, но и для их обработки. Так, в следующем фрагменте представлен исходный код процедуры, чей вызов **Decomp**[*x*] возвращает список атомарных компонент выражения *x*, включая имена переменных, функций, процедур, операций и констант. Процедура существенно использует ранее упомянутые функции **Level** и **Head**; использование функций **Head**, **Level** и **Part** в ряде наших функций и процедур из *AVZ\_Package\_1* оказывается довольно эффективным.

```
In[2017]:= Decomp[x_] := Module[{b = {}, c = DeleteDuplicates[Flatten[Level[x, Infinity]],
    Abs[#1] === Abs[#2] &], k}, Label[ArtKr];
    For[k = 1, k <= Length[c], k++, b = Append[b,
    If[AtomQ[c[[k]]], c[[k]], {Level[c[[k]], -1], Head[c[[k]]}]]];
    b = DeleteDuplicates[Flatten[b], Abs[#1] === Abs[#2] &];
    If[c == b, Return[b], c = b; b = {}; Goto[ArtKr]]

In[2018]:= Decomp[{a*Cos[x] - n*Sin[y]/(Log[h] - b), ProcQ[c, d]]
Out[2018]= {a, x, Cos, Times, -1, n, b, h, Log, Plus, Power, y, Sin, c, d, ProcQ}
```

Следующая процедура производит *группировку* выражений, заданных аргументом *L*, согласно их типам, определяемым функцией **Head1**; при этом, в качестве аргумента *L* кодируется отдельное выражение либо их список. Вызов **GroupNames**[*L*] возвращает

список или вложенный список, элементы которого – списки, чей *первый* элемент – тип объекта согласно функции **Head1**, тогда как остальные – выражения данного типа. Во фрагменте представлен *исходный* код **GroupNames** с примерами, из которых довольно прозрачно прослеживается формат возвращаемого процедурой результата.

```
In[2186]:= GroupNames[L_] := Module[{a = If[ListQ[L], L, {}], b = {"Null", "Null"}, c, d,
    p, t, k = 1}, For[k, k <= Length[a], k++, c = a[[k]]; d = Head1[c];
    t = Flatten[Select[b, #[[1]] == d &]];
    If[t == {}, b = Append[b, {d, c}], p = Flatten[Position[b, t]]][[1]];
    b[[p]] = Append[b[[p]], c]]; b = b[[2 ;; -1]]; If[Length[b] == 1, Flatten[b], b]]

In[2187]:= GroupNames[{Sin, Cos, ProcQ, Locals2, 78, Map1, StrStr, 67/42, Nvalue1, a + b}]
Out[2187]= {{System, Sin, Cos}, {Module, ProcQ, Locals2, Map1, Nvalue1}, {Integer, 78},
    {Function, StrStr}, {Rational, 67/42}, {Plus, a + b}}

In[2188]:= GroupNames[Head1]
Out[2188]= {Module, Head1}

In[2189]:= L = GroupNames[Names["*"]]
Out[2189]= {{Global`System, "\[FormalA]", ..., "CallPacket"}, {Function, "AcNb", ...,
    "$ProcName"}, {String, "ActionMenu", ..., "GroebnerBasis"}, {Module, "ActiveProcess", ...,
    "WhichN"}, {System, "CanberraDistance", ..., "$VersionNumber"}}

In[2190]:= Map[Length, L] - 1
Out[2190]= {394, 148, 28, 260, 4823}
```

В частности, из 2-х последних примеров применения **GroupNames** следует, что *имена* текущего сеанса отнесены к пяти группам, а именно: **Global`System**, **Function**, **String**, **Module** и **System**, число элементов в которых **394**, **148**, **28**, **260** и **4823** соответственно. Между тем, получение данного результата требует значительных временных затрат в связи с необходимостью тестирования большого числа средств текущего сеанса.

В дополнение к процедуре **GroupNames** определенный интерес может представить и достаточно несложная процедура, чей вызов **LocObj[x]** возвращает *трех*-элементный список, первый элемент которого определяет сам объект *x*, второй определяет его тип в разрезе {"Module", "SFunction" (*системная функция*), "Function", "Expression"}, тогда как третий – его местоположение в разрезе {"Global" – *текущий сеанс*, "System" – *ядро* либо библиотека **Mathematica**, и "Context" – *загруженный в текущий сеанс системный или пользовательский пакет, содержащий определение объекта x*}. Фрагмент представляет как исходный код процедуры, так и наиболее типичные примеры ее использования.

```
In[2244]:= LocObj[x_] := Module[{a = Head1[x], b}, b[y_] := StringTake[Context[y], {1, -2}];
    If[a == "Module", {x, "Module", b[x]}, If[a == "Function", {x, "Function", b[x]},
    If[SystemQ[x], {x, "SFunction", b[x]}, {x, "Expression", "Global"}]]]]

In[2245]:= Map[LocObj, {PureDefinition, ProcQ, StrStr, Sin, a + b, 460}]
Out[2245]= {{PureDefinition, "Module", "AladjevProcedures"}, {ProcQ, "Module",
    "AladjevProcedures"}, {StrStr, "Function", "AladjevProcedures"}, {Sin, "SFunction",
    "System"}, {a + b, "Expression", "Global"}, {460, "Expression", "Global"}}
```

Тогда как вызов **Names1[]** возвращает вложенный 4-элементный список, чей первый элемент определяет список *имен* процедур, второй – список *имен* функций, третий – список имен, определения которых были вычислены в текущем сеансе пакета, тогда как четвертый элемент определяет список других имен, ассоциированных с текущим сеансом. Фрагмент представляет исходный код процедуры с примером применения.

```
In[2045]:= Names1[x_ /; {x} == {}] := Block[{a = Names["*"], b = {}, { }, { }, c = 1, d, h},
      While[c <= Length[a], d = a[[c]]; If[ProcQ[d], AppendTo[b[[1]], d],
      If[Quiet[Check[QFunction[d], False]], AppendTo[b[[2]], d],
      h = ToString[Quiet[DefFunc[d]]];
      If[! SameQ[h, "Null"] && h == "Attributes[" <> d <> "] = {Temporary}",
      AppendTo[b[[3]], d], AppendTo[b[[4]], d]]]; c++]; b]
In[2046]:= Names1[]
Out[2046]= {"Bt", "Mas", "Names1", "W"}, {"F", "G"}, {"Art24$", "Kr17", "$NameProcCS"}, {}]
```

Процедура **Names1** достаточно полезна в ряде приложений, в частности, в некоторых вопросах процедурного программирования, в определенных отношениях расширяя стандартную функцию **Names** программной среды системы *Mathematica*. Между тем, с течением работы в текущем сеансе системы выполнение процедуры **Names1** требует все больших временных затрат, предполагая ее осмотнительное использование.

Вызов процедуры **RemoveNames[]** обеспечивает *удаление* из текущего сеанса тех имен, чьи типы отличны от процедур и функций, определения которых были вычислены в текущем сеансе; более того, имена удаляются так, что более не распознаются пакетом. Вызов **RemoveNames[]** наряду с *удалением* вышеотмеченных имен из текущего сеанса возвращает вложенный 2-элементный список, чей *первый* элемент определяет список имен процедур, тогда как *второй* – список имен функций, определения которых были вычислены в текущем сеансе пакета. Следующий фрагмент представляет исходный код процедуры **RemoveNames** наряду с типичными примерами ее использования.

```
In[2352]:= RemoveNames[x_ /; {x} == {}] := Module[{a = Names["*"], b, c},
      ToExpression["Remove[" <> StringTake[ToString[MinusList[a, Select[a, ProcQ[#] | |
      ! SameQ[ToString[Quiet[DefFunc[#]]], "Null"] | |
      Quiet[Check[QFunction[#], False]] &]], {2, -2}] <> "]"];
      Remove["RemoveNames"]; c = Names["*"];
      b = Select[c, ProcQ[#] &]; {b, MinusList[c, b]}]
In[2353]:= {Length[Names["*"]], RemoveNames[], Names["*"]}
Out[2353]= {656, {"Art", "Kr", "Rans"}, {"Rac", "Rea"}}, {"Art", "Kr", "Rans", "Rac", "Rea"}}
In[2354]:= RemoveNames[]
Out[2354]= {"Art", "Kr", "Rans"}, {"Rac", "Rea"}}
```

Процедура **RemoveNames** достаточно полезна в целом ряде приложений, связанных с освобождением рабочей области *Mathematica* от неиспользуемых символов. Данная процедура подтвердила свою эффективность в управлении оперативной памятью.



Используя теперь ранее рассмотренные процедуры и функции **ToString1**, **DefFunc3**, **HeadPF**, **SymbolQ** и **PrefixQ**, можно получить и более развитое средство тестирования объектов программной среды *Mathematica*, в качестве которого выступает процедура **ObjType**. Вызов процедуры **ObjType[x]** возвращает тип объекта *x* в разрезе {*Function*, *Module*, *Block* или *DynamicModule*}, в остальных случаях возвращается тип выражения, присвоенного в текущем сеансе символу *x* операторами {:=, =}. Следующий фрагмент представляет исходный код процедуры **ObjType** наряду с примерами применения.

```
In[2220]:= ObjType[x_] := Module[{a, b, c, d = {}, h},
    If[ToString1[HeadPF[x]] === "HeadPF[" <> ToString1[x] <> "]" | |
    SymbolQ[HeadPF[x]], Return[Head[x]], b = {ToString1[DefFunc[x]]}; c = Length[b]];
    Do[AppendTo[d, h = StringSplit[b[[k]], " := "]; {h[[1]],
        If[PrefixQ["Module[", h[[2]]], Module,
        If[PrefixQ["Block[", h[[2]]], Block,
        If[PrefixQ["Function[", h[[2]]], Function,
        If[PrefixQ["DynamicModule[", h[[2]]], DynamicModule,
        {Function, Head[ToExpression[h[[2]]]}]}]; Flatten[d, 1]]
In[2221]:= Sv[x_, y_] := x + y; G[x_] := Block[{x}, x^2]; V[x_] := If[EvenQ[x], x, 2*x];
    V[x_, y_] := Block[{a = If[PrimeQ[x], NextPrime[y]], a*(x + y)};
In[2222]:= Map[ObjType, {ObjType, 78, a + b, ProcQ}]
Out[2222]= {"ObjType[x_]", Module}, Integer, Plus, {"ProcQ[x_]", Module}}
In[2223]:= Map[ObjType, {Sv, G, V}]
Out[2223]= {"Sv[x_, y_]", {Function, Plus}}, {"G[x_]", Block}, {"V[x_, y_]", Block},
    {"V[x_]", {Function, Times}}}
In[2224]:= ObjType[DefFunc3]
Out[2224]= {"DefFunc3[x_ /; BlockFuncModQ[x]]", Module}
In[2225]:= F := Function[{x, y}, x + y]; {F[78, 460], ObjType[F]}
Out[2225]= {538, Function}
In[2226]:= F1 := #1 * #2 &; {F1[78, 460], ObjType[F1]}
Out[2226]= {35880, Function}
In[2227]:= Map[ObjType, {HeadPF, StrStr}]
Out[2227]= {"Head1[x_]", Module}, {"StrStr[x_]", {Function, String}}}
In[2228]:= Agn := "4247679886"; Avz = 2013; Map[ObjType, {Agn, Avz}]
Out[2228]= {String, Integer}
```

Здесь же вполне уместно сделать одно пояснение: к типу *Function* процедура **ObjType** относит не только сугубо функциональные объекты, но и определения следующего формата **Name[x\_, y\_, z\_, ...] := Выражение**; в данном случае вызов возвращает список следующего формата, а именно: {"Name[x\_, y\_, z\_, ...]", {Function, Head[Выражение]}}, т.е. {Заголовок, {Function, Тип выражения}}. В связи с вышесказанным процедура **ObjType** представляется нам достаточно полезной при тестировании *Math*-объектов текущего сеанса различного типа в задачах процедурного программирования в *Mathematica*.

В целом ряде случаев возникает настоятельная необходимость определения объектов и их типов, активированных непосредственно в текущем сеансе, т.е. чьи определения были вычислены в *Input*-параграфах. Данную задачу решает процедура **TypeActObj**, вызов которой **TypeActObj[]** возвращает вложенный список, чьи подписки в строчном формате *первым* элементом содержат *типы* активных объектов текущего сеанса, тогда как остальные элементы подписка *имена*, соответствующие данному типу; при этом, в качестве *типа* выступают типы, распознаваемые системой или определенные нами, в частности, определенные нами типы выражений {"Procedure", "Function"} [90,99,120].

```
In[2187]:= TypeActObj[] := Module[{a = Names["*"], b = {}, c, d, h, p, k = 1},
    Quiet[For[k, k <= Length[a], k++, h = a[[k]]; c = ToExpression[h];
        p = StringJoin["0", ToString[Head[c]]];
        If[! StringFreeQ[h, "$"] || (p === Symbol &&
            "Definition"[c] === Null), Continue[],
            b = Append[b, {h, If[ProcQ[c], "0Procedure",
                If[Head1[c] === Function, "0Function", p]]}]];
        a = Quiet[Gather1[Select[b, ! #1[[2]] === Symbol & ], 2]];
    a = ToExpression[StringReplace[ToString1[DeleteDuplicates/@ Sort/@ Flatten/@ a],
        "AladjevProcedures`TypeActObj`" -> "]];
    Append[{}, Do[a[[k]][[1]] = StringTake[a[[k]][[1]], {2, -1}], {k, Length[a]}]]; a]
In[2188]:= TypeActObj[]
Out[2188]= {"Symbol", "A", "B", "g", "H3", "m", "n", "PacletFind", "Procedure", "System",
    {"Procedure", "Art", "Asv", "Kr"}, {"Function", "G", "V"}, {"List", "xyz"}}
```

В определенном смысле процедура **TypeActObj** дополняет процедуру **ObjType**.

*Mathematica* допускает ряд эквивалентных способов определения *чистых* функций; в любом случае такой объект, примененный к списку фактических аргументов, должен в результате давать результат вычисления *некоторой* функции. Например, можно для определения *новых* чистых функций на основе имеющихся использовать стандартные функции **Nest**, **Map**. При этом, если предполагается использовать какую-то функцию многократно, то целесообразно определять функцию в формате **G[x<sub>1</sub> y<sub>1</sub> ...]:=<Тело>**, обращая впоследствии к ней по имени **G**. Тогда как при однократном применении либо в составе выражений чистую функцию можно задавать в *короткой* форме, т.е. в форме **<Тело> &**. Примеры, иллюстрирующие сказанное, можно найти в [98–101].

В контексте использования стандартных функций **Nest** и **Map** для определения *новых* чистых функций на основе имеющихся можно предложить в качестве одного весьма полезного обобщения стандартной функции **Map** процедуру, чей вызов **Mapp[F, E, x]** возвращает результат применения функции/процедуры **F** к выражению **E** с *передачей* ей фактических аргументов, определяемых кортежем выражений **x**, который может быть и пустым. При пустом кортеже **x** имеет место тождество **Map[F, E] ≡ Mapp[F, E]**. В качестве формальных аргументов стандартной функции **Map[F, g]** выступают имя процедуры/функции **F**, тогда как в качестве *второго* – выражение **g**, к чьим операндам

первого уровня применяется **F**. В завершение приведем фрагмент с исходным кодом процедуры **Мapp**, обобщающей стандартную функцию **Map** пакета *Mathematica* на большее число аргументов, наряду с типичными примерами ее использования.

```
In[2034]:= Mapp[F_;/; ProcQ[F] || SysFuncQ[F] || SymbolQ[F], Expr_, x_] :=
      Module[{a = Level[Expr, 1], b = {x}, c = {}, h, g = Head[Expr], k = 1},
        If[b == {}, Map[F, Expr], h = Length[a];
        For[k, k <= h, k++, c = Append[c, ToString[F] <> "[" <> ToString1[a[[k]]] <> ", " <>
          ListStrToStr[Map[ToString1, {x}]] <> "]"]; g[Sequences[ToExpression[c]]]]]

In[2035]:= Mapp[F, {a, b, c}, x, y, z]
Out[2035]= {F[a, x, y, z], F[b, x, y, z], F[c, x, y, z]}
In[2036]:= Mapp[F, a + b + c, x, y, z]
Out[2036]= F[a, x, y, z] + F[b, x, y, z] + F[c, x, y, z]
In[2037]:= Mapp[F, (m + n)/(g + h) + Sin[x], a, b, c]
Out[2037]= F[(m + n)/(g + h), a, b, c] + F[Sin[x], a, b, c]
In[2038]:= Mapp[StringPosition, {"11123", "33234"}, {"2", "3", "23"}]
Out[2038]= {{{4, 4}, {4, 5}, {5, 5}}, {{1, 1}, {2, 2}, {3, 3}, {3, 4}, {4, 4}}}
In[2039]:= Mapp[StringReplace, {"812345265", "72345957"}, {"2" -> "V", "5" -> "G"}]
Out[2039]= {"81V34GV6G", "7V34G9G7"}
In[2040]:= Map[F, {{a, b}, {c, d, e}}]
Out[2040]= {F[{a, b}], F[{c, d, e}]}
In[2041]:= Mapp[F, {{a, b}, {c, d, e}}, x, y, z]
Out[2041]= {F[{a, b}, x, y, z], F[{c, d, e}, x, y, z]}
In[2042]:= Mapp[ProcQ, {Sin, ProcQ, Mapp, PureDefinition, SysFuncQ}]
Out[2042]= {False, True, True, True, True}

In[2053]:= Mapp1[F_;/; SymbolQ[F], L_;/; ListQ[L]] := Module[{a = Attributes[F], b},
      SetAttributes[F, Listable]; b = Map[F, L]; ClearAllAttributes[F]; SetAttributes[F, a]; b]

In[2054]:= Map[F, {{a, b, c}, {x, y, {c, d, {h, k, t}}}}]
Out[2054]= {F[{a, b, c}], F[{x, y, {c, d, {h, k, t}}}}]
In[2055]:= Mapp[F, {{a, b, c}, {x, y, {c, d, {h, k, t}}}}]
Out[2055]= {F[{a, b, c}], F[{x, y, {c, d, {h, k, t}}}}]
In[2056]:= Mapp1[F, {{a, b, c}, {x, y, {c, d, {h, k, t}}}}]
Out[2056]= {{F[a], F[b], F[c]}, {F[x], F[y], {F[c], F[d], {F[h], F[k], F[t]}}}}
```

Отметим, что реализация алгоритма процедуры **Мapp** базируется на соотношении

$$\text{Map}[F, \text{Expr}] \equiv \text{Head}[\text{Expr}][\text{Sequences}[\text{Map}[F, \text{Level}[\text{Expr}, 1]]]]$$

Справедливость которого следует из определения системных функций **Head**, **Level** и **Map**, а также процедуры **Sequences**, представленной в настоящей книге. Следующий простой пример достаточно наглядно иллюстрирует вышесказанное, а именно:

```
In[1942]:= Map[F, (m + n)/(g + h) + Sin[x]] == Head[(m + n)/(g + h) + Sin[x]]
      [Sequences[Map[F, Level[(m + n)/(g + h) + Sin[x], 1]]]]
Out[1942]= True
```

Приведенное соотношение может быть использовано и при реализации циклических структур для решения задач другой направленности, включая программирование на основе использования механизма чистых функций. Тогда как процедура **Map** в ряде случаев достаточно существенно упрощает программирование различных задач. По атрибуту *Listable* для функции *F* определяется, что функция *F* будет автоматически применяться к элементам списка, который выступает в качестве ее аргумента. Такой подход может достаточно успешно использоваться в целом ряде случаев организации блоков, функций и модулей. В частности, в этом же контексте представляет интерес и несложная процедура **Map1**, чей вызов **Map1**[*x*, *y*] в отличие от вызова стандартной функции **Map**[*x*, *y*] возвращает результат применения блока, функции либо модуля *x* ко всем элементам списка *y*, вне зависимости от их расположения по уровням списка. Предыдущий фрагмент представляет исходный код процедуры **Map1** с примерами сравнения относительно системной функции **Map**.

Между тем, для целого ряда функций и выражений *Listable*-атрибут не работает, и в этом случае система предоставляет 2 специальные функции **Map** и **Thread**, которые в определенном отношении вполне можно отнести к структурным средствам, которые обеспечивают применение функций к частям выражений. В данном плане нами была создана группа довольно простых и вместе с тем полезных процедур и функций, т.н. **Map**-средств, довольно существенно расширяющих системную функцию **Map**. Выше уже представлено одно средство из данной группы – процедура **Map**, имеющая ряд приложений в программировании средств пакета *AVZ\_Package\_1* [120], представим и другие средства данной **Map**-группы. Представим еще *три* несложные функции из данной группы. В следующем фрагменте представлены как исходные коды функций, так и примеры их применения, которые на формальном уровне достаточно наглядно иллюстрируют результаты вызовов данных функций на корректных аргументах.

```
In[2025]:= Map1[x_ /; ListQ[x] && SameQ[DeleteDuplicates[Map[SymbolQ[#] &, x]],
               {True}], y_List] := Map[Symbol[ToString[#]]][Sequences[y]] &, x]

In[2026]:= Map1[{F, G, H, V}, {x, y, z, h, p, t}]
Out[2026]= {F[x, y, z, h, p, t], G[x, y, z, h, p, t], H[x, y, z, h, p, t], V[x, y, z, h, p, t]}

In[2027]:= Map2[F_ /; SymbolQ[F], c_ /; ListQ[c], d_ /; ListQ[d]] :=
               Map[Symbol[ToString[F]][#, Sequences[d]] &, c]

In[2028]:= Map2[F, {a, b, c, d, e, g}, {x, y, z, p, q, h}]
Out[2028]= {F[a, x, y, z, p, q, h], F[b, x, y, z, p, q, h], F[c, x, y, z, p, q, h],
               F[d, x, y, z, p, q, h], F[e, x, y, z, p, q, h], F[g, x, y, z, p, q, h]}

In[2029]:= Map3[f_ /; SymbolQ[f], g_ /; ListQ[g]] := Map[Symbol[ToString[f]][g, #] &, g]

In[2030]:= Map3[F, H, {x, y, z, h, p, h, m, n, t}]
Out[2030]= {F[H, x], F[H, y], F[H, z], F[H, h], F[H, p], F[H, h], F[H, m], F[H, n], F[H, t]}
```

Ниже, не усложняя справочной информации по функциям, отметим, что их вызовы

**Map1**[[*F*, *G*, *H*, ...], {*x*, *y*, *z*, ...}], **Map2**[*F*, {*a*, *b*, *c*, ...}, {*x*, *y*, *z*, ...}], **Map3**[*F*, *H*, {*x*, *y*, *z*, *h*, ...}]

возвращают соответственно списки следующего формата, а именно:

$\{F[x, y, z, \dots], G[x, y, z, \dots], H[x, y, z, \dots], \dots\}; \{F[a, x, y, z, \dots], F[b, x, y, z, \dots], F[c, x, y, z, \dots], \dots\};$   
 $\{F[H, x], F[H, y], F[H, z], F[H, h], \dots, F[H, g], F[H, m], F[H, n], \dots\}.$

Из представленной информации довольно прозрачен вид возвращаемых функциями результатов их вызова, не требуя каких-либо дополнительных разъяснений. Тогда как в следующем фрагменте приведена функция, вызов которой **Map4**[*x*, *y*, *z*] возвращает по логике реализуемого ею алгоритма результат в формате  $\{x[a1, z], x[a2, z], x[a3, z], \dots\}$ , где  $y = \{a1, a2, a3, \dots\}$ . В то время как две процедуры **Map5** и **Map6** расширяют действие системной функции **Map** на случаи классической и чистой функций с произвольным числом аргументов. Следующий фрагмент представляет исходные коды процедур и иллюстрирует их использование на примере функции **S**[*x*, *y*, *z*, *h*] и ее &-аналоге.

```
In[2012]:= Map4[F_ /; SymbolQ[F], L_List, x_] := Map[Symbol[ToString[F]][#, x] &, L]
In[2013]:= Map4[F, {a, b, c, d, h, g, m, n}, x]
Out[2013]= {F[a, x], F[b, x], F[c, x], F[d, x], F[h, x], F[g, x], F[m, x], F[n, x]}

In[2014]:= Map5[F_, L_ /; ListListQ[L]] := Module[{a, b = Length[L], c, d = {}, k = 1,
             h = ToString[F]}, a = Map[ToString, Map[F, L]];
             For[k, k <= b, k++, c = StringTake[a[[k]], {1, -3}] <> ""];
             AppendTo[d, StringReplace[c, h <> "[" -> h <> ""]]; ToExpression[d]]

In[2015]:= Map6[F_ /; PureFuncQ[F], L_ /; ListListQ[L]] := Module[{a, b = Length[L], k = 1,
             c = Length[L[[1]]], d = {}, h, p}, h = StringTake[ToString[F], {1, -4}];
             For[k, k <= b, k++, a = {}; AppendTo[d, StringReplace[h,
             Flatten[{For[p = 1, p <= c, p++, AppendTo[a, "#" <> ToString[p] ->
             ToString[L[[k]]][[p]]], a][[2 ;; -1]]]; ToExpression[d]]

In[2016]:= S[x_, y_, z_, h_] := a[x]*b[y]*d[z]*g[z] - c[x, y, z]
In[2017]:= Map5[S, {{x1, y1, z1, t1}, {x2, y2, z2, t2}, {x3, y3, z3, t3}, {x4, y4, z4, t4}}]
Out[2017]= {-c[x1, y1, z1] + a[x1] b[y1] d[z1] g[t1], -c[x2, y2, z2] + a[x2] b[y2] d[z2] g[t2],
             -c[x3, y3, z3] + a[x3] b[y3] d[z3] g[t3] -c[x4, y4, z4] + a[x4] b[y4] d[z4] g[t4]}
In[2018]:= Map6[a[#1]*b[#2]*d[#3]*g[#4z] - c[#1, #2, #3] &, {{x1, y1, z1, t1}, {x2, y2, z2, t2},
             {x3, y3, z3, t3}, {x4, y4, z4, t4}}]
Out[2018]= {-c[x1, y1, z1] + a[x1] b[y1] d[z1] g[t1], -c[x2, y2, z2] + a[x2] b[y2] d[z2] g[t2],
             -c[x3, y3, z3] + a[x3] b[y3] d[z3] g[t3] -c[x4, y4, z4] + a[x4] b[y4] d[z4] g[t4]}

In[2019]:= Map7[x_ /; DeleteDuplicates[Map[SymbolQ, {x}]] === {True}, y_ /; ListQ[y] :=
             Map[FunCompose[Reverse[Map[Symbol, Map[ToString, {x}]]], #] &, y]

In[2020]:= Map7[F, G, H, {a, b, c, d, h, t}]
Out[2020]= {F[G[H[a]]], F[G[H[b]]], F[G[H[c]]], F[G[H[d]]], H[G[F[h]]], H[G[F[t]]]}
In[2021]:= Map7[Sin, Sqrt, N, {17, 24, 46, 66, 71, 460}]
Out[2021]= {-0.831339, -0.982642, 0.478675, 0.963756, 0.840729, 0.517167}
In[2022]:= Map7[Sin, Sqrt, 78, {17, 24, 46, 66, 71, 460}]
Out[2022]= Map7[Sin, Sqrt, 78, {17, 24, 46, 66, 71, 460}]
```

Вызов функции **Map7**[*F, G, H, ..., V, {a, b, c, ..., v}*], где *F, G, H, ..., V* – символы и *{a, b, c, ..., v}* – список произвольных выражений, возвращает результат формата

$\{F[G[H[ \dots V[a]]]] \dots ], F[G[H[ \dots V[b]]]] \dots ], F[G[H[ \dots V[c]]]] \dots ], \dots, F[G[H[ \dots V[v]]]] \dots ]\}$

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Предыдущий фрагмент представляет исходный код функции **Map7** с примерами ее применения. В целом ряде случаев краткий формат *чистых* функций является весьма эффективным приемом программирования и определения блоков, функций и модулей, и, в первую очередь, в тех случаях, где целесообразно использовать непоименованные функции. В частности, краткий формат чистых функций позволяет, порой, весьма существенно упрощать определения многих важных объектов (*блоков, функций, модулей*).

Выше рассмотрены полезные средства работы со *списочными* структурами, в качестве которых выступают **Mapp**, **Mapp1** и **Map1, ..., Map7**. В качестве еще одного подобного средства определенный интерес представляет довольно простая функция **Map8**, чей вызов **Map8**[*F, G, H, ..., V, {a, b, ..., v}*], где *F, G, ..., V* – символы, тогда как *{a, b, c, ..., v}* – список произвольных выражений, возвращает результат следующего формата:

$\{F[a, b, c, \dots, v], G[a, b, c, \dots, v], H[a, b, c, \dots, v], \dots, V[a, b, c, \dots, v]\}$

не требуя каких-либо дополнительных пояснений ввиду его прозрачности. Функция **Map8** оказывается весьма полезной, например, в организации сравнений результатов вызовов *функционально* подобных блоков/функций/модулей на идентичных кортежах фактических аргументов. Тогда как функция **Map9**, вызов которой **Map9**[*x, {a, b, ..., v}, {a1, b1, ..., v1}*], где *x* – символ, а *{a, b, c, ..., v}* и *{a1, b1, c1, ..., v1}* – списки произвольных выражений одинаковой длины, возвращает результат следующего формата:

$\{x[a, a1], x[b, b1], x[c, c1], x[d, d1], \dots, x[v, v1]\}$

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Процедура **Map10**, чей вызов **Map10**[*F, x, {a, b, ..., v}, c1, c2, ..., cn*], где *F* – символ, тогда как *x* и *{a, b, c, ..., v}* – произвольное выражение и списки выражений соответственно, *c1, c2, ..., cn* – необязательные аргументы, возвращает результат следующего формата, а именно:

$\{F[x, a, c1, c2, \dots], F[x, b, c1, c2, \dots], F[x, c, c1, c2, \dots], \dots, F[x, v, c1, c2, \dots]\}$

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Следующий фрагмент представляет исходные коды **Map8**, **Map9** и **Map10** наряду с их примерами.

```
In[2018]:= Map8[x_ /; DeleteDuplicates[Map[SymbolQ, {x}]] == {True},
           y_ /; ListQ[y]] := Map[Symbol[ToString[#]][Sequences[y]] &, {x}]

In[2019]:= Map8[x, y, z, h, g, s, {a, b, c, d}]
Out[2019]= {x[a, b, c, d], y[a, b, c, d], z[a, b, c, d], h[a, b, c, d], g[a, b, c, d], s[a, b, c, d]}

In[2020]:= Map9[F_ /; SymbolQ[F], x_ /; ListQ[x], y_ /; ListQ[y]] := Module[{c = {}, k = 1,
           a = Map[ToString1, x], b = Map[ToString1, y], f = ToString[F],
           If[Length[x] != Length[y], Defer[Map9[F, x, y]],
           While[k <= Length[a], AppendTo[c, f <> "[" <> a[[k]] <> "," <> b[[k]] <> ""]; k++];
           ToExpression[c]]]
```

```

In[2021]:= Map9[F, {a, b, c, d, g, p}, {x, y, z, h, s, w}]
Out[2021]= {F[a, x], F[b, y], F[c, z], F[d, h], F[g, s], F[p, w]}
In[2022]:= Map9[Rule, {"71a", "66g", "46s", "78b"}, {"a", "b", "c", "d"}]
Out[2022]= {"71a" -> "a", "66g" -> "b", "46s" -> "c", "78b" -> "d"}
In[2023]:= Map9[Rule, {a, b, c, d, m}, {x, y, z, t, n}]
Out[2023]= {a -> x, b -> y, c -> z, d -> t, m -> n}
In[2024]:= Map10[F_;/; SymbolQ[F], x_/, L_;/; ListQ[L], y___] :=
                Map[Symbol[ToString[F]][x, #, Sequences[{y}]] &, L]
In[2025]:= Map10[F, x, {a, "b", c, d}, y, "z", h, w]
Out[2025]= {F[x, a, y, "z", h, w], F[x, "b", y, "z", h, w], F[x, c, y, "z", h, w], F[x, d, y, "z", h, w]}
In[2026]:= Map10[F, "x", {a, "b", c, d, f, g}]
Out[2026]= {F["x", a], F["x", "b"], F["x", c], F["x", d], F["x", f], F["x", g]}
In[2027]:= Map10[SuffPref, "C:\89b8fcf17cbdce3\ mxdwdrv.dll", {".nb", ".m", ".dll"}, 2]
Out[2027]= {False, False, True}

```

Следующие три процедуры обеспечивают дальнейшее расширение системной **Map** функции, использующие функцию **ToExpression** для генерации выражений из строк. В качестве иллюстрации можно привести достаточно несложные процедуры **Map12**, **Map13** и **Map14**. Следующий фрагмент представляет исходные коды этих процедур наряду с типичными примерами использования данных процедур.

```

In[2000]:= Map12[F_;/; SymbolQ[F], x_;/; NestListQ1[x]] := Module[{a = ToString1[x],
                b = ToString[F] <> "@", c},
                c = StringReplace[a, {"{" -> "{" <> b, "}" -> "}" <> b}];
                c = StringReplace[c, b <> "{" -> "{"; ToExpression[c]]
In[2001]:= Map12[F, {{a, b, c}, {x, y, z}, h, {m, {"p"}}, n, p, {{{x, "y"}}}}}
Out[2001]= {{F[a], F[b], F[c]}, {F[x], F[y], F[z]}, F[h], {F[m], {F["p"]}}, F[n], F[p], {{{F[x], F[y]]}}}
In[2002]:= Map12[ToString1, {{a, b, c}, {x, y, z}, "h", {m, {"x"}}, n, p}]
Out[2002]= {"a", "b", "c"}, {"x", "y", "z"}, "\"h\"", {"m", {"\"x\""}}, "n", "p"}
In[2003]:= Map13[x_;/; SymbolQ[x], y_;/; ListListQ[y]] := Module[{a = Length[y[[1]]],
                b = {}, k = 1},
                While[k <= a, AppendTo[b, Map[y[[#]]][[k]] &, Range[1, Length[y]]]; k++];
                Quiet[Map[x[Sequences[#]] &, b]]
In[2004]:= Map13[F, {{a, b, c, s}, {x, y, z, g}, {m, n, p, w}}]
Out[2004]= {F[a, x, m], F[b, y, n], F[c, z, p], F[s, g, w]}
In[2005]:= Map13[F, {{a, b, c}, {x, y, z}, {m, n, p}, {p, q, h}}]
Out[2005]= {F[a, x, m, p], F[b, y, n, q], F[c, z, p, h]}
In[2006]:= Map13[ProcQ, {ProcQ}]
Out[2006]= {True}
In[2007]:= Map13[Plus, {{a, b, c, g, t}, {x, y, z, g, t}, {m, n, p, h, g}}]
Out[2007]= {a + m + x, b + n + y, c + p + z, 2 g + h, g + 2 t}

```

```

In[2008]:= G[x_, y_] := x + y; Map13[G, {{a, b, c}, {x, y, z}, {m, n, p}}]
Out[2008]= {G[a, x, m], G[b, y, n], G[c, z, p]}
In[2009]:= Map13[G, {{a, b, c, g, h}, {x, y, z, t, v}}]
Out[2009]= {a + x, b + y, c + z, g + t, h + v}
In[2010]:= Map13[Sin, {{a, b, c, g, t}, {x, y, z, g, t}, {m, n, p, h, g}}]
Out[2010]= {Sin[a, x, m], Sin[b, y, n], Sin[c, z, p], Sin[g, g, h], Sin[t, t, g]}
In[2011]:= Map14[x_ /; SymbolQ[x], y_ /; ListQ[y], z_, t_] :=
Module[{a = Map[x[#, z] &, y]}, If[{t} == {}, a, Map[ToString, a]]]
In[2012]:= Map14[G, {a, b, c, d, f, g, h, p}, Kr]
Out[2012]= {G[a, Kr], G[b, Kr], G[c, Kr], G[d, Kr], G[f, Kr], G[g, Kr], G[h, Kr], G[p, Kr]}
In[2013]:= Map14[G, {a, b, c, d, f, g, h}, Kr, 78]
Out[2013]= {"G[a, Kr]", "G[b, Kr]", "G[c, Kr]", "G[d, Kr]", "G[f, Kr]", "G[g, Kr]", "G[h, Kr]"}
In[2014]:= Map14[G, {}, Kr, 78]
Out[2014]= {}

```

Процедура **Map12** обобщает стандартную функцию **Map** на случай *вложенного* списка в качестве ее второго фактического аргумента. Вызов процедуры **Map12**[*F*, {{*a*, *b*, *c*, ..., *v*}, {*a1*, *b1*, *c1*, ..., *v1*}, ..., {*p*, ..., {*ap*, *bp*, *h*, *cp*, ..., *vp*}}], где *F* – символ, а второй аргумент – вложенным списком произвольных выражений, возвращает результат формата:

{**Map**[*F*, {*a*, *b*, *c*, ..., *v*}], **Map**[*F*, {*a1*, *b1*, *c1*, ..., *v1*}], ..., *F*[*p*], ..., **Map**[*F*, {*ap*, *bp*, *F*[*h*], *cp*, ..., *vp*}]}

не требуя каких-либо дополнительных пояснений ввиду ее прозрачности. Тогда как процедура **Map13** обобщает стандартную функцию **Map** на случай списка *ListList*-типа в качестве ее второго фактического аргумента. Вызов процедуры **Map13**[*F*, {{*a*, *b*, *c*, ..., *v*}, {*a1*, *b1*, *c1*, ..., *v1*}, ..., {*ap*, *bp*, *cp*, ..., *vp*}}], где *F* – символ, а второй аргумент – список *ListList*-типа произвольных выражений, возвращает результат формата:

{*F*[*a*, *a1*, *a2*, ..., *ap*], *F*[*b*, *b1*, *b2*, ..., *bp*], *F*[*c*, *c1*, *c2*, ..., *cp*], ..., *F*[*v*, *v1*, *v2*, ..., *vp*]}

не требуя каких-либо дополнительных разъяснений из-за его прозрачности. В случае, неопределенного символа *x* понятие *арности* игнорируется; между тем, в случае, если фактический аргумент *x* определяет процедуру либо функцию пользователя, вызов **Map13** возвращается невычисленным, если арность *x* отлична от длины подсписков *y*. Наконец, вызов процедуры **Map14**[*F*, {*a*, *b*, *c*, ..., *v*}, *y*], где *F* – символ, второй аргумент – список произвольных выражений и *y* – выражение, возвращает результат формата:

{*F*[*a*, *y*], *F*[*b*, *y*], *F*[*c*, *y*], *F*[*d*, *y*], ..., *F*[*v*, *y*]}

При этом, использование при вызове **Map14**[*F*, {*a*, *b*, *c*, ..., *v*}, *y*, *t*] необязательного 4-го фактического аргумента, в качестве которого допускается произвольное выражение, возвращает результат следующего формата, а именно:

{"*F*[*a*, *y*]", "*F*[*b*, *y*]", "*F*[*c*, *y*]", "*F*[*d*, *y*]", ..., "*F*[*v*, *y*]"}

Процедура не требует каких-либо дополнительных пояснений ввиду прозрачности. Предыдущий фрагмент представляет *исходные* коды процедур **Map12** ÷ **Map14** наряду с наиболее типичными примерами их использования.



Ряд полезных средств система *Mathematica* предоставляет для работы с табличными объектами, базирующимися на списочных структурах. В частности, для табличного представления списка может использоваться функция **TableForm**, как иллюстрирует следующий достаточно простой пример, а именно:

```
In[2088]:= TableForm[{{1, Vic, 71}, {2, Gal, 66}, {3, Sv, 46}, {4, Art, 24}, {5, Kr, 17}}]
```

```
Out[2088]/TableForm =
```

```
1 Vic 71
2 Gal 66
3 Sv 46
4 Art 24
5 Kr 17
```

Аналогичным образом дело обстоит с матрицами/векторами, также базирующимися на простых и вложенных списках *ListList*-типа. Фрагмент представляет функцию для представления списка *ListList*-типа в формате таблицы и матрицы, а также функцию **Map16**, смысл которой вполне очевиден из приведенного примера ее применения:

```
In[2144]:= Map15[x_ /; SameQ[DeleteDuplicates[Map[SymbolQ, {x}]], {True}], y_] :=
Map[# [y] &, {x}]
```

```
In[2145]:= Map15[TableForm, MatrixForm, {{1, V, 71}, {2, G, 66}, {3, S, 46}, {4, A, 24}, {5, K, 17}}]
```

```
1 V 71
2 G 66
3 S 46
4 A 24
5 K 17
```

```
Out[2145]=
```

```
In[2146]:= Map15[F, G, H, P, Q, X, Y, Z, (a + b)]
```

```
Out[2146]= {F[a + b], G[a + b], H[a + b], P[a + b], Q[a + b], X[a + b], Y[a + b], Z[a + b]}
```

```
In[2147]:= Map16[f_Symbol, l_List, x_] := Quiet[(f[#1, FromCharacterCode[6]] &)/@ l /
FromCharacterCode[6] -> Sequence[x]]
```

```
In[2148]:= Map16[F, {x, y, z, t}, h, m, p]
```

```
Out[2148]= {F[x, h, m, p], F[y, h, m, p], F[z, h, m, p], F[t, h, m, p]}
```

С целью компактного представления списка *ListList*-типа в *двух* указанных форматах используется полезное обобщение для функции **Map** в виде функции **Map15**, которая представлена последним предыдущим фрагментом. Функция **Map15**, вызов которой **Map15[x1, x2, x3, ..., xp, t]**, где *xj* – символы, а *t* – произвольное допустимое выражение, возвращает результат формата: {x1[t], x2[t], x3[t], x4[t], ..., xp[t]}, не требуя каких-либо дополнительных пояснений ввиду его вполне достаточной прозрачности. Средства, представленные выше, образуют т.н. *Map*-группу, которая достаточно существенно расширяет функциональные возможности стандартного средства **Map** системы.

Все возникающие в процессе вычисления *особые* и *ошибочные* ситуации целесообразно программно обрабатывать, для чего *Mathematica* имеет вполне достаточно средств в режиме выполнения *документа*, тогда как с процедурной обработкой таких ситуаций вопрос оказывается несколько сложнее. Поэтому предложена процедура **Try**, которая представляет собой некий аналог **try**-предложения пакета *Maple*, механизм которого

весьма эффективен и в режиме выполнения документов, и в процедурной обработке особых и ошибочных ситуаций в управляемой программной среде, когда ошибочные и особые ситуации без какой-либо серьезной причины не приводят к завершению выполнения процедуры без возврата соответствующих диагностических сообщений.

Следующий фрагмент представляет исходный код процедуры **Try**[*x*, *y*] с примерами ее применения; **Try** использует средства **ListListQ**, **StringMultipleD**, **ToString1** из [120].

```
In[2158]:= G::norational = "actual argument `1` is not rational"
Out[2158]= "actual argument `1` is not rational"
In[2159]:= G[x_] := If[Head[x] === Rational, Numerator[x]^9 + Denominator[x]^9,
                        Message[G::norational, x]; Defer[G[x]]]
In[2160]:= G[42.71]
Intersection::normal: Nonatomic expression expected at position 2 in
{StandardForm,TraditionalForm} ∩ FormatType. >>
G::norational: actual argument 42.71` is not rational
Out[2160]= G[42.71]
In[2161]:= Try[x_/, StringQ[x], y_/, ListListQ[y]] :=
Module[{a = Length[y], b = "If[Art24Kr17 === ", c = "", d, k = 1},
  For[k, k <= a, k++, c = c <> b <> ToString1[y[[k]][[1]]] <> ", " <>
    ToString1[y[[k]][[2]]] <> ","; c = c <> x <> StringMultiple["", a];
  ToExpression[x <> "; Art24Kr17 = Quiet[{MessageList[-1][[-1]]]; " <> c]]
In[2162]:= Try["G[42/71]", {{b, b1}, {{G::norational}, "Res"}, {c, c1}, {d, d1}}]
Out[2162]= 46 255 172 102 298 503
In[2163]:= Try["G[42.71]", {{b, b1}, {{G::norational}, "Res"}, {c, c1}, {d, d1}}]
G::norational: actual argument 42.71` is not rational
G::norational: actual argument 42.71` is not rational
Out[2163]= "Res"
In[2164]:= Quiet[Try["G[42.71]", {{b, b1}, {{G::norational}, "Res"}, {c, c1}, {d, d1}}]]
Out[2164]= "Res"
```

Прежде всего, для иллюстрации работы процедуры **Try**[*x*, *y*] определяется сообщение с именем **G::norational**, используемое простой функцией **G**[*x*] в случае ее вызова на *x*, отличным от рационального числа. Такой вызов выводит данное сообщение с возвратом вызова невычисленным (могут производиться лишь упрощения выражения *x*). Процедура **Try** в определенной степени аналогична **try**-предложению, обеспечивая обработку *x* в зависимости от инициированных вычислением *x* сообщений. При этом, полагается, что все иницируемые таким вычислением *x* сообщения активизированы в текущем сеансе системы. Вызов процедуры имеет следующий формат, а именно:

**Try**["*x-expression*", {{{*a*::*a1*}, *Res1*}, {{*b*::*b1*}, *Res2*}, ..., {{*h*::*hk*}, *Resk*}}]

где в качестве первого фактического аргумента выступает *x*-выражение в строчном формате, тогда как второй фактический аргумент – вложенный список **ListList**-типа. Каждый элемент списка имеет списочный формат {{*h*::*hk*}, *Resk*}, чей первый элемент

представляет имя сообщения в формате  $\{h::hk\}$ , тогда как второй *Resk* – выражение, возвращаемое при генерации данного сообщения *последним* в процессе вычисления *x*-выражения. В случае, если в *y*-списке не находится сгенерированного сообщения либо вычисление *x*-выражения не генерирует сообщений, то возвращается результат вычисления *x*-выражения. Процедура *Try* оказалось достаточно удобным средством для обработки особых и ошибочных ситуаций при программировании целого ряда прикладных и системных задач. Для достаточно продвинутых пользователей системы *Mathematica* коды иллюстрирующих примеров и процедур данного раздела довольно прозрачны и каких-либо особых дополнительных пояснений не требуют.

**Управляющие структуры ветвления и циклические структуры в среде Mathematica.**

Достаточно сложные алгоритмы вычислений и/или управляющие (в первую очередь) не могут обойтись сугубо *последовательными* схемами, включая различные конструкции, изменяющие *последовательностный* порядок выполнения алгоритма в зависимости от наступления тех или иных условий: *условные* и *безусловные* переходы, *циклы*, *ветвления* (данного типа структуры в целом ряде случаев называются *управляющими*). В частности, для организации управляющих структур ветвящегося типа язык системы располагает достаточно эффективным средством, обеспечиваемым *If-предложением* или *функцией*, имеющим три формата кодирования [98–101,102].

В целом ряде случаев довольно полезной оказывается простая процедура *Iff* от числа аргументов от 1 до *n*, которая обобщает стандартную функцию *If*; она весьма удобна при числе аргументов, начиная с *одного*, что удобно в случаях, когда вызовы функции *Iff* генерируются в процедуре автоматически, облегчая обработку особых ситуаций, возникающих при вызове такой процедуры на числе аргументов вне диапазона 2..4. В следующем фрагменте представлен исходный код процедуры и пример применения. При этом следует иметь в виду, что *все* фактические аргументы *y*, начиная со *второго*, кодируются в *строчном* формате во избежание их преждевременного вычисления при вызове процедуры *Iff[x, ...]*, когда вычисляются/упрощаются фактические аргументы.

```
In[2045]:= Iff[x_, y_ /; StringQ[y]] := Module[{a = {x, y}, b}, b = Length[a];
          If[b == 1 || b >= 5, Defer[Iff[x, y]],
          If[b == 2, If[x, ToExpression[y]],
          If[b == 3, If[x, ToExpression[y], ToExpression[a[[3]]],
          If[b == 4, If[x, ToExpression[a[[2]]], ToExpression[a[[3]]],
          ToExpression[a[[4]]], Null]]]]
In[2046]:= a = {}; For[k = 1, k <= 100, k++, Iff[PrimeQ[k], "a = Append[a, k]"]; a
Out[2046]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

Итак, *If*-предложение представляет собой наиболее типичное средство обеспечения ветвящихся алгоритмов. В данном контексте следует отметить, что *If*-средства *Maple* и *Mathematica* представляются в значительной мере *эквивалентными*, однако в смысле читабельности сложные ветвящиеся алгоритмы, реализованные *if*-предложениями из системы *Maple* воспринимаются несколько нагляднее. В частности, *Maple* допускает условное *If*-предложение следующего формата, а именно:

**if ЛУ1 then V1 elif ЛУ2 then V2 elif ЛУ3 then V3 elif ЛУ4 then V4 ... else Vk end if**

где  $j$ -е ЛУ $j$  – логическое условие и  $V_j$  – выражение, чей смысл достаточно прозрачен и рассмотрен, например, в книгах [8,42–44,99]. Данное предложение весьма удобно при программировании целого ряда условных структур. В целях определения *аналогичной* структуры может быть использована процедура **IFk**, исходный код которой наряду с примерами использования представляет следующий фрагмент, а именно:

```
In[2040]:= IFk[x_] := Module[{a = {x}, b, c = "", d = "If", e = "]", h = {}, k = 1, b = Length[a];
    If[For[k, k <= b - 1, k++, h = Append[h, b >= 2 && ListQ[a[[k]]] &&
        Length[a[[k]]] == 2]];
    DeleteDuplicates[h] != {True}, Return[Defer[Ifk[x]], k = 1];
    For[k, k <= b - 1, k++, c = c <> d <> ToString[a[[k]][[1]]] <>
        "," <> ToString[a[[k]][[2]]] <> ","; c = c <> ToString[a[[b]]] <>
        StringMultiple[e, b - 1]; ToExpression[c]]

In[2041]:= IFk[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]
Out[2041]= If[a, b, If[c, d, If[g, s, If[m, n, If[q, p, h]]]]]
In[2042]:= IFk[{False, b}, {False, d}, {False, s}, {True, n}, {False, p}, h]
Out[2042]= n
In[2043]:= IFk[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]
Out[2043]= If[g, p, h]

In[2060]:= IFk1[x_] := Module[{a = {x}, b, c = "", d = "If", e = "]", h = {}, k = 1, b = Length[a];
    If[For[k, k <= b - 1, k++, h = Append[h, b >= 2 && ListQ[a[[k]]] &&
        Length[a[[k]]] == 2]];
    DeleteDuplicates[h] != {True}, Return[Defer[Ifk1[x]], {h, k} = {{}, 1}];
    If[For[k, k <= b - 1, k++, h = Append[h, a[[k]][[1]]]];
    Select[h, ! MemberQ[{True, False}, #] &] != {}, Return[Defer[Ifk1[x]], k = 1];
    For[k = 1, k <= b - 1, k++, c = c <> d <> ToString[a[[k]][[1]]] <> "," <>
        ToString[a[[k]][[2]]] <> ","; c = c <> ToString[a[[b]]] <>
        StringMultiple[e, b - 1]; ToExpression[c]]

In[2061]:= IFk1[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]
Out[2061]= IFk1[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]
In[2062]:= IFk1[{False, b}, {False, d}, {False, s}, {True, n}, {False, p}, h]
Out[2062]= n
In[2063]:= IFk1[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]
Out[2063]= IFk1[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]
In[2064]:= IFk1[{a, b, g}, {c, d}, {g, s}, {m, n}, {q, p}, h]
Out[2064]= IFk1[{a, b, g}, {c, d}, {g, s}, {m, n}, {q, p}, h]
In[2065]:= IFk1[{True, b}]
Out[2065]= IFk1[{True, b}]
In[2066]:= IFk1[{False, b}, agn]
Out[2066]= agn
```

Вызов процедуры **IFk** использует произвольное число фактических аргументов более одного, в качестве которых выступают 2-элементные списки формата  $\{LUj, Vj\}$ , кроме последнего. В качестве последнего аргумента допускается произвольное корректное выражение языка; при этом, проверка  $LUj$  на булевый тип не производится. Вызов на кортеже корректных фактических аргументов процедуры **IFk** возвращает результат, эквивалентный выполнению соответствующего *Maple*-предложения [98]. При этом, в качестве определенного полезного расширения является процедура **IFk1**, которая в отличие от предыдущей **IFk** в качестве фактических аргументов  $LUj$  допускает лишь булевы выражения, иначе возвращая вызов невычисленным. В остальном процедуры **IFk** и **IFk1** функционально идентичны. С учетом сказанного процедуры и **IFk**, и **IFk1** обеспечены довольно развитым механизмом тестирования передаваемых при вызове фактических аргументов, алгоритм которого легко усматривается из исходного кода. Таким образом, аналогично *if*-предложению системы *Maple* процедуры как **IFk**, так и **IFk1** оказываются довольно полезными в программировании *ветвящихся* алгоритмов. Используя описанный подход, достаточно несложно реализовать в среде *Math*-языка любую конструкцию *Maple*-языка, описывающую ветвящийся алгоритм [98–101].

В определенной мере к *If*-конструкциям можно отнести и **Which**-функцию формата

**Which**[ $LU1, V1, LU2, V2, LU3, V3, \dots, LUK, VK$ ]

которая возвращает результат вычисления первого  $Vj$ -выражения, булево выражение  $LUj$  ( $j=1..k$ ) для которого принимает значение **True**, например:

```
In[2456]:= G[x_] := Which[-Infinity <= x < 78, Sin[x], 78 <= x < 460, Cos[x],
                                460 <= x <= Infinity, x^2]
```

```
In[2457]:= {G[66], G[78.460], G[460], G[2013], G[-18.06]}
Out[2457]= {Sin[66], -0.996816, 211600, 4052169, 0.710041}
```

Пример иллюстрирует определение через **Which** кусочно-определенной функции. Если какое-нибудь из вычисляемых условий  $LUj$  не возвращает  $\{True | False\}$ , то вызов функции возвращается *невычисленным*, тогда как в случае значения *False* для всех  $LUj$  ( $j=1..k$ ) вызов функции возвращает **Null**-значение, т.е. ничего. В случае динамической генерации **Which**-объекта полезной оказывается простая процедура **WhichN**, которая допускает произвольное *четное* число аргументов, аналогичных **Which**-функции, в противном случае возвращая результат вызова невычисленным. В остальном **WhichN** аналогична функции **Which**; следующий простой фрагмент представляет исходный код процедуры наряду с некоторыми типичными примерами ее использования.

```
In[2521]:= WhichN[x_] := Module[{a = {x}, c = "Which[" , d, k = 1}, d = Length[a];
                                If[OddQ[d], Defer[WhichN[x]], ToExpression[For[k, k <= d, k++,
                                c = c <> ToString[a[[k]]] <> ","; StringTake[c, {1, -2}] <> "]" ]]]

In[2522]:= WhichN[a, b, c, d, f, g, h, r]
Out[2522]= Which[a, b, c, d, f, g, h, r]
In[2523]:= f = 78; WhichN[False, b, f == 78, SV, g, h, r, t]
Out[2523]= SV
```

Циклические управляющие структуры системы Mathematica. Так, одна из основных циклических структур системы основывается на **For**-функции, имеющей следующий общий формат кодирования, а именно:

**For**[*A*, <ЛУ>, *B*, <Тело циклической конструкции>]

Начиная с заданного *A*, циклически вычисляется *тело* конструкции, которая содержит предложения языка, с циклическим наращением переменной *цикла* на величину *B* до тех пор, пока логическое условие (ЛУ) не примет значения **True**. Вот простой пример на использование данной функции, а именно:

```
In[2000]:= For[k = 1; h = 1, k < 10000, k = k + 1, h = h^3 + h^2 + h + k; If[k < 5, Continue[],
Print[h]; Break[]]]
18931630251093012373677688241080557562741843863681284
```

Для продолжения **For**-цикла и выхода из него служат управляющие слова **Continue**[] и **Break**[] соответственно, как это весьма наглядно иллюстрирует простой пример. В то время как в качестве другого довольно широко используемого средства в *Mathematica* для организации циклических вычислений является функция **Do**[...], которая имеет пять форматов кодирования, описания которых с примерами можно найти в [98-102]. Между тем, в отличие от системы *Maple* система *Mathematica* не имеет аналога весьма полезных циклических конструкций типов (1.b) и (1.d) [99], позволяющих выполнять циклические вычисления по подвыражениям заданного выражения, что обеспечивает возможность на их основе программировать довольно интересные конструкции, как иллюстрирует простой фрагмент [99,101]. В данном контексте представим процедуру **DO**[*x*, *y*, *k*], возвращающую список значений циклического вычисления выражения *x* по переменной цикла *k*, которая принимает значения из списка **Op**[*y*]. Конструкция в определенном отношении является аналогом циклической **for\_in**-конструкции для программной среды системы *Maple* [8-14,24,28-38,42-45].

```
In[2004]:= DO[x_, y_, k_] := Module[{a = x, b = Op[y], c, d = 1, R = {}, c := Length[b] + 1;
While[d < c, R = Insert[R, a /. k -> b[[d]], -1]; a := x; d++]; R]
In[2005]:= DO[k^2 + Log[k], f[g[a, b], h[c, d, e, j, k, l]], k]
Out[2005]= {g[a, b]^2 + Log[g[a, b]], h[c, d, e, j, k, l]^2 + Log[h[c, d, e, j, k, l]]}
```

Наконец, функция **While**[ЛУ, Тело] по результату проверки ЛУ циклически вычисляет тело цикла до тех пор, пока ЛУ принимает значение **True**, например:

```
In[2027]:= t := 0; b := {}; While[t < 100, t = t + 1; If[PrimeQ[t], AppendTo[b, t], Null]]; b
Out[2027]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

Механизм данной циклической конструкции весьма прост и пояснений не требует. В контексте выше приведенных примеров еще раз довольно наглядно иллюстрируется взаимная функциональная эквивалентность данных систем, когда наиболее важные вычислительные конструкции системы *Mathematica* с той или иной эффективностью симулируются конструкциями *Maple*, и наоборот. И действительно, в принципе, это вполне ожидаемый результат, ибо встроенные языки обоих систем универсальны и в этой связи с той либо иной эффективностью реализуют любой алгоритм. Однако во временном отношении это далеко не так и при использовании циклических структур

достаточно большого уровня вложенности *Maple* может иметь весьма существенные преимущества перед *Mathematica*. Для подтверждения сказанного приведем простой пример циклической конструкции, реализованной как в *Maple*, так и в *Mathematica*.

Результаты говорят сами за себя – если в *Maple 11* для выполнения требуется 22.642 с., то для *Mathematica 9* для этой же конструкции требуется уже 243.344 с., т.е. более чем на порядок больше. Более того, с ростом *глубины вложенности* и *диапазона переменной цикла* у циклических конструкций данная разница довольно существенно растет.

<pre>&gt; t := time(): for k1 to 10 do for k2 to 10 do for k3 to 10 do for k4 to 10 do for k5 to 10 do for k6 to 10 do for k7 to 10 do for k8 to 10 do 78 end do end do end do end do end do end do do end do end do: time() - t;                                     # (Maple 11)   22.642 In[2092]:= n = 10; t = TimeUsed[]; For[k1 = 1, k1 &lt;= n, k1++,     For[k2 = 1, k2 &lt;= n, k2++,         For[k3 = 1, k3 &lt;= n, k3++,             For[k4 = 1, k4 &lt;= n, k4++,                 For[k5 = 1, k5 &lt;= n, k5++,                     For[k6 = 1, k6 &lt;= n, k6++,                         For[k7 = 1, k7 &lt;= n, k7++,                             For[k8 = 1, k8 &lt;= n, k8++, 78]]]]]]]; TimeUsed[] - t Out[2093]= 243.344</pre>
---

Естественно, полученные значения определяются основными ресурсами компьютера, однако на идентичных ресурсах данное принципиальное соотношение сохраняется. Итак, из приведенного примера следует, что *Maple* использует более эффективные во *временном* отношении алгоритмы для реализации циклических конструкций *большой* глубины вложенности и диапазонов переменных циклов, чем это имеет место для его основного конкурента – системы *Mathematica* даже ее последней версии **9.0.1.0**. Среди специальных типов циклических управляющих структур в среде *Mathematica* можно отметить ряд довольно интересных, некоторые из которых имеют различного уровня аналогии с подобными средствами системы *Maple* [98-101]. Однако в целом, средства системы *Mathematica* более предпочтительно при генерации вложенных выражений и, прежде всего, *чистых* функций, что играет особенно существенную роль для задач функционального программирования в среде системы *Mathematica*.

При сравнительном рассмотрении *управляющих* структур *ветвления* и *цикла*, которые поддерживаются обоими системами, были выделены две основные группы, а именно: базовые и дополнительные средства обеспечения указанных управляющих структур. Предложение *if* системы *Maple* и *If*-функция системы *Mathematica*, представляющие собой наиболее типичные средства обеспечения ветвящихся алгоритмов. В процессе работы с обоими средствами сформировалась точка зрения, средства представляются в значительной степени эквивалентными, однако в смысле читабельности несколько проще воспринимаются достаточно сложные ветвящиеся алгоритмы, реализованные именно *if*-предложениями пакета *Maple*. В других отношениях весьма сложно отдать предпочтение какому-либо из данных управляющих средств и в данном отношении обе лидирующие системы вполне можно рассматривать эквивалентными средствами.

Следующая простая функция **Nconcat** довольно наглядно иллюстрирует расширение арифметических функций путем определения *числовой* конкатенации списка *PosInt*-типа любого уровня вложенности подобно случаю стандартной функции **StringJoin**. Фрагмент представляет исходный код функции с примерами ее использования.

```
In[2712]:= Nconcat[h_;/ PosIntQ[h] || PosIntListQ[Flatten[h]]] :=
           If[PosIntQ[h], h, ToExpression[StringJoin[ToString2[Flatten[h]]]]]

In[2713]:= Nconcat[{71, 66, 46, 51, 24, 17}]
Out[2713]= 716646512417

In[2714]:= Nconcat[78]
Out[2714]= 78

In[2715]:= Nconcat[{{71, 66}, {46, 51}, {24, 17}}]
Out[2715]= 716646512417
```

Вызов простой функции **MaxParts[x]** возвращает количество всех частей выражения *x* относительно функции **Op**. Фрагмент представляет исходный код с примерами.

```
In[2719]:= MaxParts[x_] := Length[Op[x]]

In[2720]:= Map[MaxParts, {{a, b, c, d, g, h, f, s, k}, (a + b)/(c + d) + Sin[x]}]
Out[2720]= {9, 2}
```

Как уже неоднократно отмечалось, в отличие от традиционных программных систем *Math*-язык поддерживает механизм т.н. *одноименных* объектов, т.е. таких объектов, в которых при идентичном *имени* определены различные *заголовки*, по которым именно и производится идентификация объектов в точке их вызова. В данной книге довольно широко представлены средства работы с одноименными объектами. Между тем, этот механизм позволяет достаточно эффективно как доопределять, так и переопределять стандартные функции, используя следующие предпосылки, а именно.

Прежде всего, система не позволяет переопределять либо доопределять те функции, которые используются в определениях других системных функций. В частности, при попытке этой процедуры относительно системной функции **Rule** из текущего сеанса удаляются определения всех средств из пакетов пользователя, активированных в нем, но переопределения либо доопределения функции **Rule** не производится. Поэтому в случае необходимости расширить функциональность стандартного средства данного типа следует наделять его другим именем, тогда как его заголовок в качестве одного из вариантов должен допускать кортежи фактических аргументов, которые присущи стандартному заменяемому средству. В этом случае вместо стандартного можно везде использовать его замену. Следующий фрагмент представляет исходный код наряду с примерами применения процедуры **Rule1**, доопределяющей функцию **Rule**.

Вызов процедуры **Rule1[x, y]** эквивалентен вызову **Rule[x, y]**, тогда как вызовы формы **Rule1[a, b, c, d, ...]** и **Rule1[{a, b, c, d, ...}]** возвращают список формата  $\{a \rightarrow b, c \rightarrow d, \dots\}$  в случае, если количество элементов *a, b, c, d, ...* четно, в остальных случаях вызов **Rule1** возвращается невычисленным, как иллюстрируют примеры следующего фрагмента. В целом ряде случаев процедура **Rule1** оказывается предпочтительнее системной **Rule**.



```

In[2000]:= Rule1[x_] := Module[{a = {x}, b = Length[{x}]}, If[b == 2, a[[1]] -> a[[2]],
    If[b > 2 && EvenQ[b], Map[a[[#]] -> a[[# + 1]] &, Select[Range[1, b], OddQ[#] &]],
    If[b == 1 && ListQ[x] && EvenQ[Length[x]], Map[x[[#]] -> x[[# + 1]] &,
        Select[Range[1, Length[x]], OddQ[#] &]], Defer[Rule1[x]]]]]]
In[2001]:= Rule1[a]
Out[2001]= Rule1[a]
In[2002]:= Rule1[a, b]
Out[2002]= a -> b
In[2003]:= Rule1[a, b, c, d, h, t]
Out[2003]= {a -> b, c -> d, h -> t}
In[2004]:= Rule1[a, b, c, d, h]
Out[2004]= Rule1[a, b, c, d, h]
In[2005]:= Rule1[{a, b}, {c, d}, x, y]
Out[2005]= {{a, b} -> {c, d}, x -> y}
In[2006]:= Rule1[{a, b}, {c, d}, x, y, m]
Out[2006]= {{a, b}, {c, d}, x, y} -> m
In[2021]:= ClearAttributes[Sin, Protected]
In[2022]:= Sin[x_] := If[Length[{x}] == 1 && ! ListQ[x], Sqrt[1 - Cos[x]^2],
    If[Length[Flatten[{x}]] > 1, Plus[Sequences[Map[Sqrt[1 - Cos[#]^2] &,
        Flatten[{x}]]], $Failed]]
In[2023]:= SetAttributes[Sin, Protected]
In[2024]:= Definition[Sin]
Out[2024]= Attributes[Sin] = {Listable, NumericFunction, Protected}
    Sin[x_] := If[Length[{x}] == 1 && ! ListQ[x], Sqrt[1 - Cos[x]^2],
    If[Length[Flatten[{x}]] > 1, +Sequences[(Sqrt[1 - Cos[#1]^2] &)/@ Flatten[{x}]], $Failed]
In[2025]:= Sin[460.78]
Out[2025]= 0.859413
In[2026]:= Sin[a, b, c]
Out[2026]= Sqrt[1 - Cos[a]^2] + Sqrt[1 - Cos[b]^2] + Sqrt[1 - Cos[c]^2]
In[2027]:= Sin[{a, b, c}]
Out[2027]= {Sqrt[1 - Cos[a]^2], Sqrt[1 - Cos[b]^2], Sqrt[1 - Cos[c]^2]}
In[2028]:= Sin[]
Out[2028]= $Failed
    
```

Тогда как для случая, например, системных функций с атрибутом *NumericFunction* в возможности пользователя *доопределять* функции такого типа своими определениями, как иллюстрирует вторая часть предыдущего фрагмента на примере доопределения стандартной функции **Sin**. Вызов **Sin[x]** эквивалентен стандартному, тогда как вызов **Sin[x, y, z, ...]** возвращает **Sin[x] + Sin[y] + Sin[z] + ...**, а вызов **Sin[{x, y, z, ...}]** возвращает **{Sin[x], Sin[y], Sin[z], ...}**; вызов **Sin[]** возвращает **\$Failed**. При этом, отсутствует также негативная реакция системы на загруженные средства пакетов пользователя наряду с выводом сообщений об ошибках, что важно в практическом программировании.

## Глава 8. Задачи процедурного программирования в *Mathematica*

*Процедурное* программирование является одной из базовых парадигм *Mathematica*, в довольно существенной степени отличающейся от аналогичной парадигмы хорошо известных традиционных процедурных языков программирования. Именно данное обстоятельство лежит в основе целого ряда системных задач, относящихся к вопросу процедурного программирования в среде системы *Mathematica*. Прежде всего, такие задачи возникают в области различий в реализации упомянутых парадигм в системе как *Mathematica*, так и в среде традиционных процедурных языков. Наряду с этим, в отличие от ряда как традиционных, так и встроенных пакетных языков (*в частности, в среде пакета Maple*) встроенный *Math*-язык не располагает рядом полезных средств для работы с процедурными объектами. Немало таких средств представлен в наших книгах [100,101] и нашим пакетом *AVZ\_Package* [90]. Целый ряд связанных с такими средствами задач и рассматривается в настоящей главе, предварительно обсудив само понятие *процедуры* в среде *Mathematica*, как основы ее процедурной парадигмы. При этом, средства анализа данного раздела касаются лишь пользовательских процедур и функций, ибо определения всех системных функций (*в отличие, скажем, от Maple*) от пользователя скрыты, т.е. недоступны стандартными средствами *Mathematica*.

### 8.1. Определение процедур в программной среде *Mathematica*

*Процедуры* в программной среде системы *Mathematica* формально представляют собой функциональные объекты двух следующих простых форматов, а именно:

$$M[x\_;/ Test_x, y\_;/ Test_y, ...] \{:= | =\} \text{Module}[\{locals\}, \text{Body of Procedure}]$$

$$B[x\_;/ Test_x, y\_;/ Test_y, ...] \{:= | =\} \text{Block}[\{locals\}, \text{Body of Procedure}]$$

т.е., процедуры обоих типов представляют собой функции от двух аргументов – *тела процедуры (Body)* и *локальных переменных (locals)*. Локальные переменные – список символов (*имен*), возможно, с приписанными им *начальными* значениями. Переменные носят относительно процедуры *локальный* характер, т.е. их значения не пересекаются со значениями одноименных символов вне процедуры. Все остальные переменные в процедуре носят *глобальный* характер, разделяя область переменных текущего сеанса.

Таким образом, в определении процедур можно выделить пять компонент, а именно:

- *имя процедуры* (*M* в первом определении процедуры)
- *заголовок процедуры* ( $M[x\_;/ Test_x, y\_;/ Test_y, ...]$  в первом определении процедуры)
- *процедурные скобки* ( $\text{Module}[...]$  либо  $\text{Block}[...]$ )
- *локальные переменные* (список локальных переменных  $\{locals\}$ ; может быть пустым)
- *тело процедуры* (тело процедуры – *Body of Procedure*; может быть пустым)

Прежде всего, следует отметить следующее весьма важное обстоятельство, а именно. Если в традиционных языках программирования идентификация *процедуры/функции* производится по ее имени, то в случае *Math*-языка идентификация производится по ее заголовку. Обусловлено это тем, что определение *процедуры/функции* в *Math*-языке

производится способом, указанным выше и отличным от традиционных способов. В данной ситуации допустимо одновременное существование одноименных процедур/функций с различными заголовками, как это иллюстрирует следующий фрагмент:

```
In[2034]:= M[x_, y_] := Module[{}, x + y]; M[x_] := Module[{}, x^2];
           M[y_] := Module[{}, y^3]; M[x___] := Module[{}, {x}]
In[2035]:= Definition[M]
Out[2035]= M[x_, y_] := Module[{}, x + y]
           M[y_] := Module[{}, y^3]
           M[x___] := Module[{}, {x}]
In[2036]:= {M[480, 78], M[78], M[42, 47, 67, 23, 16]}
Out[2036]= {558, 474552, {42, 47, 67, 23, 16}}
In[2037]:= G[x_;/ IntegerQ[x]] := Module[{}, x]; G[x_] := Module[{}, x^2]; G[480]
Out[2037]= 480
```

При вызове процедуры/функции из списка одноименных выбирается та, у которой формальные аргументы заголовка соответствуют фактическим аргументам вызова, в противном случае вызов возвращается *невычисленным*, за исключением упрощений фактических аргументов согласно стандартным соглашениям пакета. Более того, при соответствии формальных аргументов заголовка фактическим вызывается процедура, чье определение стоит *выше* в списке, выводимом по функции **Definition**; в частности, определение которой было вычислено в текущем сеансе системы первым.

В дальнейшем довольно часто будет идти речь о возврате результата вызова функции либо процедуры *невычисленным*, это относится как к стандартным средствам системы, так и к пользовательским. Так, стандартной реакцией на получение процедурой или функцией недопустимого кортежа фактических аргументов при ее вызове является возврат вызова *невычисленным*, за исключением стандартных *упрощений* фактических аргументов. В данной связи была создана процедура **UnevaluatedQ**, обеспечивающая тестирование процедуры/функции на предмет возврата ее вызова невычисленным на конкретном кортеже фактических аргументов. Вызов процедуры **UnevaluatedQ[F, x]** возвращает **True**, если вызов **F[x]** возвращается невычисленным, и **False** в противном случае; при этом, на ошибочном вызове **F[x]** возвращается "ErrorInNumArgs".

```
In[2246]:= UnevaluatedQ[F_;/ SymbolQ[F], x___] :=
           Module[{a = Quiet[Check[F[x], "error", F::argx]]},
           If[a === "error", "ErrorInNumArgs", If[ToString1[a] === ToString[F] <> "[" <>
           If[{x} == {}, "", ListStrToStr[Map[ToString1, {x}]] <> "], True, False]]]
In[2247]:= {UnevaluatedQ[F, x, y, z], UnevaluatedQ[Sin, x, y, z]}
Out[2247]= {True, "ErrorInArgs"}
In[2248]:= {UnevaluatedQ[Sin, 78], UnevaluatedQ[Sin, 460.78], UnevaluatedQ[Sin]}
Out[2248]= {True, False, "ErrorInNumArgs"}
```

Предыдущий фрагмент представляет *исходный* код процедуры **UnevaluatedQ** наряду с примерами ее использования. Процедура представляет определенный интерес при программной обработке результатов вызовов процедур и функций.

Между тем, стандартная функция **Definition** при наличии одноименных процедур/функций в целом ряде случаев малопригодна для решения задач, которые связаны с обработкой определений таких объектов. Прежде всего, это относится к процедурам, определения которых находятся в пользовательских пакетах, загруженных в текущий сеанс *Mathematica*, как иллюстрирует фрагмент получения определения процедуры **SystemQ** из ранее упоминавшегося нашего пакета *AVZ\_Package\_1* [120], а именно:

```
In[3247]:= Definition["SystemQ"]
Out[3247]= SystemQ[AladjevProcedures`SystemQ`S_] :=
      If[! SymbolQ[AladjevProcedures`SystemQ`S], False,
      If[ToString[HeadPF[AladjevProcedures`SystemQ`S]] ==
      ToString[AladjevProcedures`SystemQ`S], True, False]]
```

Стандартная функция **Definition[x]** в целом ряде случаев возвращает определение *x*-объекта с соответствующим ему контекстом, что при довольно больших *определениях* становится плохо обозримым и менее приемлемым для последующей программной обработки, как наглядно иллюстрирует предыдущий пример. Более того, в качестве фактического аргумента *x* может выступать имя объекта либо его строчный формат. Для устранения данного недостатка нами был определен ряд средств, позволяющих получать определения процедур/функций в оптимизированном формате. В качестве таких средств можно отметить такие как **Definition1**, **Definition2**, **Definition3**, **DefOpt**, **DefFunc**, **DefFunc1**, **DefFunc2**, **DefFunc3** и **DefOptimum**. Средства эти наряду с рядом других представлены в книге [101] и включены в пакет *AVZ\_Package* [90]. Следующий фрагмент представляет исходные коды наиболее употребительных из них, а именно.

```
In[2050]:= Definition1[x_] := Module[{a, b, c}, If[Quiet[SystemQ[x]], Null,
      b = Quiet[{Attributes[x], ClearAllAttributes[x]}][[1]];
      Quiet[Off["Definition"::"notfound"]];
      a = ToString1[ToExpression["Definition[" <> ToString1[x] <> "]"]];
      c = Flatten[{HeadPF[x]}]; a = StringSplit[a, c]; a = Partition[Riffle[a, c], 2];
      a = Map[StringJoin[#[[2]], #[[1]]] &, a];
      Quiet[On["Definition"::"notfound"]]; SetAttributes[x, b];
      If[a === "", $Failed, If[Length[a] == 1, a[[1]], a]]]

In[2051]:= Map[Definition1, {Tan, SystemQ}]
Out[2051]= {"Null", "SystemQ[AladjevProcedures`SystemQ`S_] :=
      If[Off[MessageName[Definition, \"ssle\"]];
      ! ToString[Definition[AladjevProcedures`SystemQ`S]] === Null &&
      SysFuncQ1[AladjevProcedures`SystemQ`S],
      On[MessageName[Definition, \"ssle\"]]; True,
      On[MessageName[Definition, \"ssle\"]]; False]"}

In[2066]:= g = 78; L = {x, y, z, t, h, g, w, d, n, m}; Definition[L[[6]]]
      Definition::ssle: Symbol, string, or HoldPattern[symbol] expected at position 1
      in Definition[L[[6]]]. >>
Out[2066]=
```

```

In[2067]:= Map[Definition1, {L[[6]], L[[8]], 78, "g", Sin}]
Out[2067]= {$Failed, "Null", $Failed, "g = 78", "Null"}

In[2070]:= Definition2[x_ /; SameQ[SymbolQ[x], HowAct[x]]] :=
Module[{a, b = Attributes[x], c},
  If[SystemQ[x], Return[{"System", Attributes[x]}, Off[Part::partw]];
  ClearAttributes[x, b]; Quiet[a = ToString[InputForm[Definition[x]]];
  Mapp[SetAttributes, {Rule, StringJoin}, Listable];
  c = StringReplace[a, Flatten[{Rule[StringJoin[Contexts1[], ToString[x] <> ""], ""]}]];
  c = StringSplit[c, "\n \n"]; Mapp[ClearAttributes, {Rule, StringJoin}, Listable];
  SetAttributes[x, b]; a = AppendTo[c, b];
  If[SameQ[a[[1]], "Null"] && a[[2]] == {},
    On[Part::partw]; {"Undefined", Attributes[x]},
  If[SameQ[a[[1]], "Null"] && a[[2]] != {} && ! SystemQ[x],
    On[Part::partw]; {"Undefined", Attributes[x]},
  If[SameQ[a[[1]], "Null"] && a[[2]] != {} && a[[2]] != {},
    On[Part::partw]; {"System", Attributes[x]},
  On[Part::partw]; a]]]]

In[2071]:= Definition2[SystemQ]
Out[2071]= {"SystemQ[S_] := If[Off[MessageName[Definition, \ "ssle\"];
  ! ToString[Definition[S]] === Null && SysFuncQ1[S],
  On[MessageName[Definition, \ "ssle\"]; True,
  On[MessageName[Definition, \ "ssle\"]; False]}, {}]

In[2072]:= Definition2[Tan]
Out[2072]= {"System", {Listable, NumericFunction, Protected}}

In[2073]:= Definition2[(a + b)/(c + d)]
Out[2073]= Definition2[(a + b)/(c + d)]

In[2144]:= Definition3[x_ /; SymbolQ[x], y_ /; ! HowAct[y]] := Module[{a = Attributes[x],
  b = Definition2[x]}, If[b[[1]] == "System", Null, b = Definition2[x][[1 ;; -2]];
  ClearAttributes[x, a]; If[BlockFuncModQ[x, y], ToExpression[b]; SetAttributes[x, a];
  Definition[x], SetAttributes[x, a]; Definition[x]]]

In[2145]:= Definition3[SystemQ, y]
Out[2145]= SystemQ[S_] := If[Off["Definition"::"ssle"]; ! ToString["Definition"][S]] === Null
  && SysFuncQ1[S], On["Definition"::"ssle"]; True, On["Definition"::"ssle"]; False]

In[2146]:= y
Out[2146]= "Function"

In[2147]:= Definition3[Sin, y]
Out[2147]= Attributes[Sin] = {Listable, NumericFunction, Protected}

```

Целый ряд функциональных средств *Math*-языка в качестве фактических аргументов допускают лишь объекты типов  $\{Symbol, String, HoldPattern[Symbol]\}$ , что в ряде случаев довольно неудобно при программировании задач различного назначения. К данным

средствам относится, в частности, и функция **Definition**, как наглядно иллюстрирует пример предыдущего фрагмента. С целью расширения функции **Definition** на типы, отличные от упомянутых, может быть использована процедура **Definition1**, чей вызов **Definition1[x]** возвращает определение объекта *x* в строчном формате, "Null", если *x* не определен, в противном случае возвращается **\$Failed**. Фрагмент представляет код процедуры с типичными примерами ее применения, из которых довольно наглядно видны определенные преимущества процедуры **Definition1** относительно **Definition**. Процедура **Definition1** реализована с использованием процедуры **ToString1**, которая в отличие от функции **ToString** обеспечивает корректное *конвертирование* выражений в строчный формат. Процедура **ToString1** достаточно проста и используется в наших средствах весьма широко, однако отнесена к средствам манипулирования строчными конструкциями. Процедура **Definition1** обрабатывает основные особые и ошибочные ситуации. Между тем, процедура **Definition1** не избавляет возвращаемые определения от контекстов и корректна только для объектов с уникальными именами. Более того, для случая множественных контекстов данная процедура возвращает определения с контекстом, который отвечает последнему пользовательскому пакету, загруженному в текущий сеанс. На системных функциях вызов **Definition1** возвращает "Null".

Расширением процедуры **Definition1** служит процедура **Definition2**, представленная вторым примером предыдущего фрагмента. Процедура использует наши процедуры **Contexts1**, **HowAct**, **Mapp**, **SymbolQ** и **SystemQ**, рассмотренные нами несколько ниже. Данные процедуры достаточно просты и используются в наших средствах довольно широко. В отличие от предыдущей, процедура **Definition2** избавляет возвращаемые определения от контекстов и корректна для объектов с уникальными именами. Вызов процедуры на системных функциях возвращает вложенный список, первый элемент которого – "**System**", тогда как второй – список атрибутов функции. На функции или процедуре пользователя *x* вызов **Definition2[x]** также возвращает вложенный список, первый элемент которого – оптимизированное определение *x* (*в смысле отсутствия в нем контекстов*), тогда как второй – список атрибутов, приписанных *x*; в их отсутствие в качестве второго элемента списка выступает пустой список. При значении **False** на тесте, приписанном формальному аргументу *x*, вызов процедуры **Definition2[x]** будет возвращен *невычисленным*. Подобно предыдущей процедуре **Definition2** обрабатывает основные особые и ошибочные ситуации. При этом, как **Definition1**, так и **Definition2** возвращают определения объектов в строчном формате.

Большинство средств, представленных в данной книге и нашем пакете *AVZ\_Package* и предназначенных для работы с процедурами/функциями пользователя, полагают, что такие средства не имеют атрибутов, что в подавляющем большинстве случаев так и есть на самом деле. Используя для обработки определения таких пользовательских средств с применением стандартной функции **Definition**, однако, соответствующие алгоритмы требуют учитывать наличие атрибутов у обрабатываемого средства, ввиду структуры возвращаемого **Definition** результата. С этой целью нами была определена процедура **Definition2**, рассмотренная выше и возвращающая список, чей последний элемент определяет список приписанных объекту атрибутов, тогда как предыдущие элементы – оптимальные определения в строчном формате одноименных средств. В

то же время ряд алгоритмов указанного пакета не базируется на этой процедуре, что требует их адаптации под условия наличия у средств атрибутов. При этом, подобная адаптация не вызывает каких-либо затруднений у довольно опытного пользователя. Ее механизм прекрасно иллюстрируется третьим примером предыдущего фрагмента в виде процедуры **Definition3**. Данная процедура использует упомянутые процедуры **Definition2**, **HowAct** и процедуру **ProcFuncQ**, результат вызова которой не зависит от наличия у объекта  $x$  атрибутов, одновременно определяя и тип самого объекта  $x$ .

Вызов **Definition3**[ $x$ ,  $y$ ] возвращает оптимальное определение функции/процедуры  $x$ ; при этом, через *второй* аргумент  $y$  – неопределенную переменную – возвращается тип процедуры/функции, а именно: "Procedure&Function", "Function" или "Procedure", если  $x$  – процедура или функция. Поскольку же в текущем сеансе возможно существование одноименных процедур/функций и их сочетаний, то тип "Procedure&Function" служит для идентификации факта, что существуют и процедуры, и функции с именем  $x$ . При этом, на системных функциях вызов процедуры **Definition3**[ $x$ ,  $y$ ] возвращает значение **Null**, не изменяя значения 2-го аргумента  $y$ ; при недопустимости первого аргумента  $x$  вызов возвращается невычисленным. Смотри примеры предыдущего фрагмента.

Для целого ряда приложений, включая системного характера, довольно важной нам представляется стандартная функция **Definition**, чей вызов **Definition**[ $x$ ] возвращает *определение* объекта  $x$  с приписанными ему атрибутами; при отсутствии определения возвращается **Null**, т.е. *ничего*, или приписанные *неопределенному* символу  $x$  атрибуты следующего формата, а именно:

$$\text{Attributes}[x] = \{\text{Список атрибутов, приписанных символу } x\}$$

Как весьма наглядно иллюстрирует следующий простой пример, а именно:

```
In[2039]:= SetAttributes[h, Listable]; Definition[h]
```

```
Out[2039]= Attributes[h] = {Listable}
```

```
In[2040]:= Definition[Sin]
```

```
Out[2040]= Attributes[Sin] = {Listable, NumericFunction, Protected}
```

Как видим, на системных функциях получаем аналогичный результат. Между тем, с другой стороны, многие задачи обработки объектов основываются собственно на их определениях. Поэтому и выделение определения объекта  $x$  в чистом виде возможно обеспечить, в частности, двумя механизмами, суть которых поясняется примерами в [101] процедурами **Def** и **Def1**, определение второй приводит следующий пример:

```
In[2656]:= Def1[x_] := Module[{a = Attributes[x]}, ClearAllAttributes[x];
                                     {ToString[Definition[x]], ToString[SetAttributes[x, a]]}][[1]]
In[2657]:= Def1[Gs]
Out[2657]= "Gs[x_] := x"
In[2658]:= Definition[Gs]
Out[2658]= Attributes[Gs] = {Listable, Protected}
Gs[x_] := x
```

Вызов процедуры **Def1**[ $x$ ] использует отмену атрибутов объекта  $x$  с возвратом *чистого* определения  $x$  в строчном формате с одновременным восстановлением для него всех

атрибутов. Этот механизм представляется нам наиболее предпочтительным. Именно он использовался нами для ряда полезных модификаций функции **Definition**. Более того, так как наряду с собственно самим определением возвращаются и приписанные объекту атрибуты в приведенном выше формате отдельной заголовочной строкой, то целесообразно представить одно полезное расширение стандартной функции, вызов которой **Attributes[x]** возвращает список атрибутов, приписанных  $x$ -объекту. Данное расширение представляется простой процедурой, вызов которой **Attributes1[x, y, z, ...]** в отличие от стандартного на объектах  $x, y, z, \dots$ , отличных от допустимых функцией **Attributes**, возвращает пустой список, т.е. {}, без вывода сообщений об ошибках, что в целом ряде случаев более предпочтительно с точки зрения обработки ошибочных и особых ситуаций. В то время как на допустимых объектах  $x, y, z, \dots$  вызов процедуры **Attributes1[x, y, z, ...]** возвращает список атрибутов, приписанных объектам  $x, y, z, \dots$ . Следующий фрагмент представляет исходный код процедуры с примерами.

```
In[2096]:= Attributes1[x_] := Module[{a = Map[Quiet[Check[Attributes[#], {}]] &, {x}],
                                     If[Length[a] == 1, a[[1]], a]]
In[2097]:= L := {42, 47, 67, 24, 17, 6}; SetAttributes[L, {Flat, Protected, Listable}]
In[2098]:= Attributes1[L[[5]], Sin, ProcQ]
Out[2098]= {}, {Listable, NumericFunction, Protected}, {}
In[2099]:= Attributes1[5, a + b, Attributes1, While, If]
Out[2099]= {}, {}, {}, {HoldAll, Protected}, {HoldRest, Protected}}
```

Процедура использует нашу процедуру **ToString1**, которая в отличие от стандартной **ToString**, в общем случае обеспечивает более корректное конвертирование *выражения* в *строчный* формат. Эта процедура рассматривается ниже в соответствующем разделе, тогда как процедура **Attributes1** достаточно полезна в целом ряде приложений. Ниже в данной главе будут представлены средства обработки атрибутов, специфичных для процедур/функций, в виде двух процедур **AttributesH** и **DefAttributesH**.

Прежде всего, для возврата определения процедуры/функции в *компактном* формате предлагается процедура **DefFunc[x]**, обеспечивающая возврат определения  $x$ -объекта, содержащегося в пакете либо *nb*-документе, загруженном в текущий сеанс. Фрагмент представляет исходный код процедуры **DefFunc** с примерами ее использования.

```
In[2047]:= DefFunc[x_ /; SymbolQ[x] || StringQ[x]] := Module[{
    a = GenRules[Mapp[StringJoin, {"Global`", Context[x]}, ToString[x] <> "", ""],
    b = StringSplit[ToString[InputForm[Definition[x]]], "\n \n"],
    ToExpression[Map[StringReplace[#, a] &, b]]; Definition[x]]
In[2048]:= DefFunc[ProcQ]
Out[2048]= ProcQ[x_] := Module[{a, atr = Quiet[Attributes[x]], b, c, d, h},
    If[ListQ[atr] && atr != {}, ClearAllAttributes[x]];
    a = Quiet[SubsDel[ToString["InputForm"["Definition"["x"]],
    "" <> ToString[x] <> "", {"[", ",", " ", -1}];
    Quiet[b = StringTake[a, {1, First[First[StringPosition[a, {" := Block["", " :=Block[""] - 1]}]}];
```



```

c = StringTake[a, {1, First[First[StringPosition[a, {" := Module[{", " :=Module[{"] - 1}]]];
d = StringTake[a, {1, First[First[StringPosition[a, {" := DynamicModule[{",
                                " :=DynamicModule[{"] - 1}]]];
If[b === ToString[HeadPF[x]],
If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; True,
If[c === ToString[HeadPF[x]],
If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; True,
If[d === ToString[HeadPF[x]],
If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; True,
If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; False]]]]

```

In[2049]:= **Definition[ListListQ]**

```

Out[2049]= ListListQ[AladjevProceduresAndFunctions`ListListQ`L_] :=
If[AladjevProceduresAndFunctions`ListListQ`L != {} &&
ListQ[AladjevProceduresAndFunctions`ListListQ`L] &&
Length[Select[AladjevProceduresAndFunctions`ListListQ`L, ListQ[#1] &&
Length[#1] == Length[AladjevProceduresAndFunctions`ListListQ`L[[1]]] &]] ==
Length[AladjevProceduresAndFunctions`ListListQ`L], True, False]

```

In[2050]:= **DefFunc[ListListQ]**

```

Out[2050]= ListListQ[L_] := If[L != {} && ListQ[L] && Length[Select[L, ListQ[#1] &&
Length[#1] == Length[L[[1]]] &]] == Length[L], True, False]

```

Естественно, для получения определения объекта, определенного и активированного в текущем сеансе вполне подходит и стандартная функция **Definition**, но в случае *m*-файла или *nb*-документа имеется существенное различие, как хорошо иллюстрирует возврат определения функции **ListListQ** из пакета *AVZ\_Package* [90] посредством как функции **Definition**, так и нашей процедуры **DefFunc**. Естественно, во втором случае определение существенно более читаемо, прежде всего, для значительных по объему исходных кодов процедур и функций. Ниже приведены другие подобные средства.

В связи с представленной выше процедурой **DefFunc** следует иметь в виду следующее немаловажное обстоятельство. В отличие от большинства систем программирования, в которых переопределение процедуры заменяет предшествующее без возможности доступа к нему, т.е. в текущем сеансе история *определений* процедуры не сохраняется, в среде *Mathematica* в этом отношении ситуация совершенно иная. Так, следующий фрагмент довольно наглядно представляет возможность просмотра *всех* определений одноименной процедуры.

```

In[2060]:= P[x_] := Module[{}, x]; P[y_] := Module[{}, y]; P[x_] := Plus[Sequences[{x}]];
P[y_ /; PrimeQ[y]] := Module[{a = "agn"}, y]; P[x_] := Plus[Sequences[{x}]];
P[x_ /; StringQ[x]] := Module[{}, x]; P[x_ /; ListQ[x]] := Module[{}, x];
P[x_, y_] := Module[{}, x + y]

```

In[2061]:= **Definition[P]**

```

Out[2061]= P[y_ /; PrimeQ[y]] := Module[{a = "agn"}, y]
P[x_ /; StringQ[x]] := Module[{}, x]

```

```
P[x_ /; ListQ[x]] := Module[{}, x]
=====
P[x___] := + Sequences[{x}]
```

Из приведенного фрагмента несложно усмотреть, что переопределение процедуры *P* при изменении ее заголовка сохраняет в текущем сеансе все такие переопределения, в то время как модификации тела процедуры без изменения ее заголовка оставляют в текущем сеансе доступным только ее последний вариант. Таким образом, обработке доступна *история* активации в текущем сеансе одноименных процедур с различными заголовками. Для удобства обработки такой истории оказывается довольно полезной процедура **DefFunc3**, являющаяся вариантом процедур **DefFunc**, **DefFunc1**, **DefFunc2** (две последние рассматриваются в [101]). Следующий фрагмент представляет исходный код процедуры **DefFunc3[x]** наряду с типичными примерами ее использования.

```
In[1103]:= DefFunc3[x_ /; BlockFuncModQ[x]] := Module[{a = Attributes[x], b},
                                                    ClearAllAttributes[x];
                                                    b = StringSplit[StrStr[InputForm[DefFunc[x]]], "\n \n"];
                                                    SetAttributes[x, a]; If[Length[b] == 1, b[[1]], b]

In[1104]:= DefFunc3[P]
Out[1104]= {"P[y_ /; PrimeQ[y]] := Module[{a = \"agn\"}, y]",
            "P[x_ /; StringQ[x]] := Module[{}, x]", "P[x_ /; ListQ[x]] := Module[{}, x]", ...
            "P[x___] := Plus[Sequences[{x}]]", "P[x___] := Plus[Sequences[{x}]]"}

```

Вызов **DefFunc3[x]** процедуры возвращает список определений в строчном формате *одноименных x-процедур* с различными заголовками, активизированными в текущем сеансе пакета; при этом, порядок определений в возвращаемом списке соответствует порядку их вывода стандартной функцией **Definition** системы *Mathematica*.

Вполне определенный интерес представляет и следующая процедура **DefOpt**, в целом ряде случаев более приемлемая, чем функция **Definition** и наши процедуры **DefFunc**, **DefFunc1**, **DefFunc2** и **DefFunc3**, рассмотренные в [90,101] и также предназначенные для получения определений процедур/функций в приемлемом для обработки виде. Следующий фрагмент представляет исходный код с примерами ее использования.

```
In[2341]:= DefOpt[x_ /; StringQ[x]] := Module[{a = Definition1[x], b = StringJoin["", x, ""],
                                                    c, d = t}, If[a === Null || a === $Failed, Null,
                                                    c = Flatten[StringPosition[a, b]];
                                                    If[c == {}, Definition1[x], ClearAll[t];
                                                    ToExpression[StringReplace[a, StringJoin[StringReplace[
                                                    StringJoin[SubStr[a, c[[1]], {"[", " ", "\\", t, "\\",
                                                    b -> ""], b] -> ""]]; ClearAll[t]; t = d; Definition1[x]]]]

In[2343]:= DefOpt["SystemQ"]
Out[2343]= SystemQ[S_] := If[! SymbolQ[S], False, If[ToString[HeadPF[S]] == ToString[S],
                                                    True, False]]

In[2344]:= DefFunc[$TypeProc]
Out[2344]= Attributes[$Failed] = {HoldAll, Protected}
```

```

In[2345]:= DefOpt["$TypeProc"]
Out[2345]= $TypeProc := CheckAbort[If[$$$Art23$Kr15$$$ = Select[{Stack[Module],
    Stack[Block], Stack[DynamicModule]}, #1 != {} &];
    If[$$$Art23$Kr15$$$ == {}, Clear[$$$Art23$Kr15$$$];
    Abort[],[$$$Art23$Kr15$$$ = ToString[$$$Art23$Kr15$$$[[1]][[1]]];
    SuffPref[$$$Art23$Kr15$$$, "Block[{", 1],
    Clear[$$$Art23$Kr15$$$]; "Block",
    If[SuffPref[$$$Art23$Kr15$$$, "Module[{", 1] &&
    !StringFreeQ[$$$Art23$Kr15$$$, "DynamicModule"],
    Clear[$$$Art23$Kr15$$$]; "DynamicModule",
    Clear[$$$Art23$Kr15$$$]; "Module"]], $Failed]
In[2346]:= Map[DefOpt, {"If", "Sin", "Goto", "a + b", "78", 460}]
Out[2346]= {Null, Null, Null, Null, Null, DefOpt[460]}
    
```

С другой стороны, некоторые наши процедуры оказываются непригодными в случае необходимости получения определений ряда процедурных переменных, в частности, **\$TypeProc**, как иллюстрирует 3-й пример предыдущего фрагмента. И только вызов процедуры **DefOpt[x]** возвращает определение *x*-объекта в оптимальном формате в независимости от типа объекта *x* пользователя. При этом, вызов **DefOpt[x]** не только возвращает *оптимальный* формат *определения* объекта *x*, но и вычисляет его в текущем сеансе, что в целом ряде случаев является весьма полезным; при вызове имя объекта *x* кодируется в *строчном* формате; тогда как на *системных* функциях и *других* строчных выражениях вызов **DefOpt[x]** возвращает **Null**. В то же время следует иметь ввиду, что **DefOpt** неприменима к *одноименным* процедурам/функциям, т.е. имеющим несколько определений, и дополнительно с рассмотренной процедурой **Definition1** использует достаточно полезную процедуру, чей вызов **SubStr[S, p, a, b, r]** возвращает подстроку строки *S*, которая слева ограничена первым символом, отличным от символа *a* или от символов из списка *a*, и справа ограничена символом, отличным от символа *b* либо от символов из списка *b*. Процедура **SubStr** рассматривается в соответствующем разделе. Предыдущий фрагмент представляет исходный код процедуры **DefOpt** с примерами. Достаточно интересной модификацией предыдущей процедуры является процедура **OptDefinition**, чей исходный код с примерами применения представляет следующий фрагмент. Вызов **OptDefinition[x]** возвращает определение процедуры или функции *x*, оптимизированное в вышеотмеченном смысле, т.е. без контекста, ассоциированного с пользовательским пакетом, содержащим процедуру либо функцию *x*.

```

In[298]:= OptDefinition[x_;/ Quiet[ProcQ[x] || FunctionQ[x]]] := Module[{c = $Packages,
    a, b, d, h = Definition2[x]}, {a, b} = {h[[1 ;; -2]], h[[-1]]};
    ClearAllAttributes[x]; d = Map[StringJoin[#, ToString[x] <> "" &], c];
    ToExpression[Map[StringReplace[#, GenRules[d, ""]] &, a]];
    SetAttributes[x, b]; Definition[x]]
In[299]:= SetAttributes[ToString1, {Listable, Protected}]; Definition[ToString1]
Out[299]= Attributes[ToString1] = {Listable, Protected}
    
```

```
ToString1[AladjevProcedures`ToString1`x_] :=
Module[{AladjevProcedures`ToString1`a = "$Art23Kr15$.txt",
AladjevProcedures`ToString1`b = "", AladjevProcedures`ToString1`c,
AladjevProcedures`ToString1`k = 1},
Write[AladjevProcedures`ToString1`a,
AladjevProcedures`ToString1`x];
Close[AladjevProcedures`ToString1`a];
For[AladjevProcedures`ToString1`k,
AladjevProcedures`ToString1`k < \[Infinity],
AladjevProcedures`ToString1`k++,
AladjevProcedures`ToString1`c =
Read[AladjevProcedures`ToString1`a, String];
If[AladjevProcedures`ToString1`c ===
EndOfFile, Return[DeleteFile[
Close[AladjevProcedures`ToString1`a]];
AladjevProcedures`ToString1`b],
AladjevProcedures`ToString1`b =
AladjevProcedures`ToString1`b <>
StrDelEnds[AladjevProcedures`ToString1`c, " ", 1]]]]
In[300]:= OptDefinition[ToString1]
Out[300]= Attributes[ToString1] = {Listable, Protected}
ToString1[x_] := Module[{a = "$Art23Kr15$.txt", b = "", c, k = 1}, Write[a, x];
Close[a]; For[k, k <  $\infty$ , k++, c = Read[a, String];
If[c === EndOfFile, Return[DeleteFile[Close[a]]; b, b = b <> StrDelEnds[c, " ", 1]]]]
```

Следует обратить внимание на использование процедуры *GenRules*, обеспечивающей генерацию списка правил для выполнения замен в строке-определении средства *x*. В целом ряде случаев данный подход достаточно эффективен при обработке строк.

Полным аналогом предыдущей процедуры является реализованная другим способом процедура **DefOptimum**, чей вызов **DefOptimum[x]** возвращает определение *функции* или *процедуры* *x*, оптимизированное в том отношении, что оно не содержит контекста пакета пользователя, содержащего определение процедуры/функции *x*. Следующий фрагмент представляет исходный код процедуры с примером ее использования.

```
In[2245]:= SetAttributes[OptDefinition, {Listable, Protected}]; Definition[OptDefinition]
Out[2245]= Attributes[OptDefinition] = {Listable, Protected}
OptDefinition[x_ /; ProcQ[x] || FunctionQ[x]] :=
Module[{a = Definition2[x][[1 ;; -2]], b = Definition2[x][[-1]],
AladjevProcedures`OptDefinition`c = $Packages,
AladjevProcedures`OptDefinition`d,
AladjevProcedures`OptDefinition`h},
ClearAllAttributes[ToString1];
```

```

AladjevProcedures`OptDefinition`d = (#1 <> (ToString[x] <> "") &)/@
AladjevProcedures`OptDefinition`c;
ToExpression[(StringReplace[#1,
GenRules[AladjevProcedures`OptDefinition`d, ""]] &)/@ a];
SetAttributes[x, b]; "Definition"[x]]
In[2246]:= DefOptimum[x_ /; Quiet[ProcQ[x] || FunctionQ[x]]] := Module[{a, c, k = 1,
b = "Art$Kr.txt", d = Context[x], f = Attributes[x]},
ClearAttributes[x, f]; Save[a = ToString[x], x];
For[k, k < Infinity, k++, c = Read[a, String];
If[SameQ[c, EndOfFile], Break[]],
Write[b, StringReplace[c, d <> ToString[x] <> "" -> ""]]];
Map[Close, {a, b}]; Get[b]; Map[DeleteFile, {a, b}]; SetAttributes[x, f]; Definition[x]]
In[2247]:= DefOptimum[OptDefinition]
Out[2247]= Attributes[OptDefinition] = {Listable, Protected}
OptDefinition[x_ /; Quiet[ProcQ[x] || FunctionQ[x]]] :=
Module[{c = $Packages, a, b, d, h = Definition2[x]}, {a, b} = {h[[1 ;; -2]], h[[-1]]};
ClearAllAttributes[x]; d = (#1 <> (ToString[x] <> "") &)/@ c;
ToExpression[(StringReplace[#1, GenRules[d, ""]] &)/@ a];
SetAttributes[x, b]; "Definition"[x]]

```

Алгоритм процедуры **DefOptimum** базируется на сохранении текущего определения блока/функции/модуля  $x$  в файле *ASCII*-формата с последующей его конвертацией в *txt*-файл, содержащий определение процедуры  $x$  без вхождений контекста пакета, в котором находится определение данного средства. После чего результатный файл функцией **Get** загружается в текущий сеанс системы с возвратом оптимизированного определения объекта  $x$ .

Как уже отмечалось, в результате загрузки пользовательского пакета в текущий сеанс определения части его средств, возвращаемые стандартной функцией **Definition[x]**, содержат конструкции формата **Context[x] <> ToString[x] <> ""**, порой, существенно, затрудняя как перлюстрацию *определений*, так и их обработку. В качестве еще одного средства получения оптимальных для обработки определений средств пользователя можно предложить и процедуру **ToDefOptPF**, чей вызов **ToDefOptPF[x]** возвращает реально конвертируемое в текущем сеансе определение процедуры либо функции  $x$  в оптимальный формат, не содержащий упомянутых контекстных конструкций. При этом, атрибуты, приписанные объекту (*процедура либо функция*)  $x$ , сохраняются. Более того, процедура **ToDefOptPF** корректно обрабатывает и т.н. *одноименные* объекты, т.е. объекты, которые под одним именем имеют более одного определения с различными заголовками. Следующий фрагмент приводит исходный код процедуры **ToDefOptPF** с наиболее типичными примерами ее использования.

```

In[2026]:= ToDefOptPF[x_ /; BlockFuncModQ[x]] := Module[{a, b, c, k = 1, p = ToString[x]},
a = ToExpression["Attributes[" <> p <> ""];
b = ToExpression["ClearAllAttributes[" <> p <> ""];

```

```

c = StringSplit[ToString[InputForm[Definition[x]]], "\n \n"];
While[k <= Length[c], ToExpression[StringReplace[c[[k]],
Context[x] <> p <> "" -> ""]]; k++;
ToExpression["SetAttributes[" <> p <> "," <> ToString[Sequence[a]] <> ""];
If[Length[c] == 1, c[[1]], c]]

In[2027]:= Definition[SystemQ]
Out[2027]= SystemQ[AladjevProcedures`SystemQ`S_] := If[Off["Definition"::"ssle"];
! ToString["Definition"[AladjevProcedures`SystemQ`S]] === Null &&
SysFuncQ1[AladjevProcedures`SystemQ`S], On["Definition"::"ssle"];
True, On["Definition"::"ssle"]; False]

In[2028]:= ToDefOptPF[SystemQ]
Out[2028]= SystemQ[S_] := If[Off["Definition"::"ssle"]; ! ToString["Definition"[S]] === Null
&& SysFuncQ1[S], On["Definition"::"ssle"]; True, On["Definition"::"ssle"]; False]

In[2039]:= Definition[ToString1]
Out[2039]= Attributes[ToString1] = {Listable, Protected}
ToString1[AladjevProcedures`ToString1`x_] :=
Module[{AladjevProcedures`ToString1`a = "$Art23Kr15$.txt",
AladjevProcedures`ToString1`b = "", AladjevProcedures`ToString1`c,
AladjevProcedures`ToString1`k = 1}, Write[AladjevProcedures`ToString1`a,
AladjevProcedures`ToString1`x]; Close[AladjevProcedures`ToString1`a];
For[AladjevProcedures`ToString1`k, AladjevProcedures`ToString1`k < ∞,
AladjevProcedures`ToString1`k++, AladjevProcedures`ToString1`c =
Read[AladjevProcedures`ToString1`a, String];
If[AladjevProcedures`ToString1`c === EndOfFile,
Return[DeleteFile[Close[AladjevProcedures`ToString1`a]];
AladjevProcedures`ToString1`b], AladjevProcedures`ToString1`b =
AladjevProcedures`ToString1`b <> StrDelEnds[AladjevProcedures`ToString1`c, " ", 1]]]]

In[2040]:= ToDefOptPF[ToString1]
Out[2040]= Attributes[ToString1] = {Listable, Protected}
ToString1[x_] := Module[{a = "$Art23Kr15$.txt", b = "", c, k = 1}, Write[a, x];
Close[a]; For[k, k < ∞, k++, c = Read[a, String];
If[c === EndOfFile, Return[DeleteFile[Close[a]]; b], b = b <> StrDelEnds[c, " ", 1]]]]

```

Между тем, в целом ряде случаев функция или процедура вполне может возвращать результат, ассоциированный с контекстом пользовательского пакета. Для устранения такой ситуации довольно полезной может оказаться функция, чей вызов **OptRes[x, y]** возвращает *оптимизированный* для последующей обработки результат **y**, который был возвращен функцией/процедурой **x**. Следующий пример представляет исходный код функции **OptRes** наряду с примерами ее наиболее типичного использования.

```

In[3322]:= OptRes[x_Symbol, y_] := If[Mapp[SetAttributes, {Rule, StringJoin}, Listable];
StringQ[y] && StringFreeQ[y, "" <> ToString[x] <> ""],
Mapp[ClearAttributes, {Rule, StringJoin}, Listable]; y,

```

```

{If[StringStringQ[y], ToExpression, Evaluate][StringReplace[ToString1[y],
  Flatten[{Rule[StringJoin[Contexts1[]], ToString[x] <> ""], ""}]]],
  Mapp[ClearAttributes, {Rule, StringJoin}, Listable]][[1]]

In[3323]:= OptRes[ProcQ, AladjevProcedures`ProcQ`vsvartkr]
Out[3323]= vsvartkr
In[3324]:= OptRes[ProcQ, "AladjevProcedures`ProcQ`vsvartkr"]
Out[3324]= "vsvartkr"
In[3325]:= Definition[Map1]
Out[3325]= Map1[AladjevProcedures`Map1`F_;/
  ListQ[AladjevProcedures`Map1`F],
  AladjevProcedures`Map1`args_;/
  ListQ[AladjevProcedures`Map1`args]] :=
  Module[{a, b, c, k = 1}, {a, b, c} = {ToString /@ AladjevProcedures`Map1`F,
    {}, ToString[AladjevProcedures`Map1`args]}; For[k, k <= Length[a], k++,
    b = Append[b, ToExpression[StringReplace[StringJoin[a[[k]], c], {"{" -> "[", "}" -> "]" }]]]; b]
In[3326]:= OptRes[Map1, Definition[Map1]]
Out[3326]= "Map1[F_;/ ListQ[F], args_;/ ListQ[args]] := Module[{a, b, c, k = 1},
  {a, b, c} = {ToString /@ F, {}, ToString[args]}; For[k, k <= Length[a], k++, b = Append[b,
    ToExpression[StringReplace[StringJoin[a[[k]], c], {"{" -> "[", "}" -> "]" }]]]; b]"

```

Из последних примеров фрагмента вполне наглядна разница применения функции **Definition** с целью получения определения процедуры **Map1**, находящейся в пакете **AVZ\_Package\_1**, и функции **OptRes**, которая применялась к полученному результату **Definition[Map1]**. При этом, строчный формат результата позволяет либо вычислять определение **Map1** по вызову функции **ToExpression**, или производить его обработку строчными средствами. Отметим, что функция наряду со стандартными средствами использует и наши процедуры **Contexts1** и **Mapp**, рассматриваемые несколько ниже. Между тем, процедура **ToDefOptPF** неэффективна для случая, когда в текущий сеанс загружаются пакеты с одинаковыми контекстами **"Kiev"**, как иллюстрирует пример:

```

In[2134]:= Get["C:\\AVZ_Package\\Kiev.mx"]; Get["C:\\AVZ_Package\\Kiev_1.mx"]
In[2135]:= CNames["Kiev"]
Out[2135]= {"Ga", "Gs", "GSV", "Vgs"}
In[2136]:= ToDefOptPF[Gs]
Out[2136]= "Gs[Kiev`Gs`x_Integer, Kiev`Gs`y_Integer] := Kiev`Gs`x^2 + Kiev`Gs`y^2"
In[2137]:= Definition[Gs]
Out[2137]= Gs[x_Integer, y_Integer] := x^2 + y^2

```

В то же время, обеспечив загрузку пакета пользователя **UPackage** вызовом процедуры **LoadMyPackage[".... \\UPackage.mx", Context]**, все определения, содержащиеся в нем, будут в оптимальном формате, т.е. не будут содержать контекста, ассоциированного с пакетом. При этом, обработка средств из загруженного таким образом пакета будет существенно проще. Данная процедура рассматривается в соответствующем разделе. В последующих разделах рассматриваются средства манипулирования с основными составляющими определения процедур и функций.

## 8.2. Определение функций и чистых функций пользователя в программной среде системы *Mathematica*

Прежде всего, отметим, что т.н. *функциональное программирование* не является каким-то открытием *Mathematica*, а восходит к целому ряду программных средств, которые появились задолго до указанного пакета. В этом контексте вполне уместно несколько подробнее остановиться на концепции *функционального* программирования. Основы *функционального* программирования были заложены примерно в одно и то же время, что и *императивного* программирования (которое наиболее распространено в настоящее время), т.е. в 30-е годы прошлого века. Основными создателями математических основ *функционального* программирования с полным основанием можно считать А. Чёрча (США), автора  $\lambda$ -исчисления и одного из основоположников концепции однородных структур (*Cellular Automata*) в связи с его работами в области бесконечных абстрактных автоматов и математической логики [85], а также Х. Карри (Англия) и М. Шёнфинкеля (Германия), разработавших математическую теорию комбинаторов.

Тогда как *первым*, практически, функциональным языком программирования можно считать язык *Lisp*, созданный в начале 50-х Дж. МакКарти, который довольно долгое время оставался единственным языком данного типа. В процессе использования *Lisp* в практическом программировании все большую роль начинает играть *типизация*. В конце 70-х – начале 80-х интенсивно разрабатываются модели типизации, пригодные для функциональных языков. Большинство этих моделей включали поддержку таких мощных механизмов как абстракция данных и полиморфизм. Появляется множество *типизированных функциональных* языков, например, *Scheme*, *Hope*, *Miranda*, *Clean*, *ML* и др., растет число их диалектов. Между тем, как правило, все группы разработчиков в области *функционального* программирования, использовали собственный язык, что не способствовало широкому распространению этих языков, создавая многочисленные мелкие проблемы. Для исправления такой ситуации объединенная группа ведущих исследователей в области *функционального* программирования решила объединить достоинства различных языков в новом *универсальном функциональном* языке. Первая реализация такого языка, названного *Haskell* в честь Хаскелла Карри, была создана в начале 90-х годов прошлого века. На сегодня действует стандарт языка *Haskell 98*.

Следуя традициям языка *Lisp*, большинство *функциональных* языков реализуются как *интерпретаторы*, наиболее удобные для быстрой отладки программ. Однако с другой стороны, интерпретаторы относительно *компиляторов*, как правило, проигрывают по скорости выполнения программ в разы. По этой причине наряду с интерпретаторами существуют компиляторы, генерирующие достаточно эффективный машинный код (например, *Objective Caml*) или код на языке C++ (например, *Glasgow Haskell Compiler*). Из достаточно большого числа языков *функционального* программирования можно отметить следующие, оказавшие большое влияние на прогресс в данной области:

– *Lisp* – первый функциональный нетипизированный язык программирования; язык включает немало императивных свойств, однако в целом поощряет *функциональный* стиль программирования. Существует объектно-ориентированный диалект *CLOS*;



- *Scheme* – диалект языка *Lisp*, ориентированный на исследования в компьютерных науках. При разработке *Scheme* был сделан акцент на элегантности и простоте языка, что позволило сделать его намного меньшим, чем собственно язык *Common Lisp*;
- *ISWIM* – абстрактный язык, предложенный *P.J. Landin* в 1966 в качестве *прототипа функциональных* языков. И хотя в прямом виде язык реализован не был, *Landin* вместе с языком разработал и специальную *виртуальную* машину для выполнения программ на *ISWIM (SECD-машина)*. *ISWIM* является *императивным* языком с функциональным ядром, чей синтаксис навеян  *$\lambda$ -исчислением* с включением изменяющихся переменных и присвоением наряду с мощным управляющим механизмом – *J-оператором*. Сегодня на синтаксисе языка *ISWIM* базируется синтаксис многих функциональных языков, а сам язык оказал достаточно сильное влияние на развитие языков программирования, особенно функциональных языков типа *SASL*, *KRC (Kent Recursive Calculator)*, *Miranda*, *Haskell*, *Hope*, *Clean* и целого ряда их прямых либо опосредствованных приемников;
- *ML* – семейство языков с развитой *полиморфной* системой типов и *параметризуемыми* модулями. Концепция данного семейства изучается во многих университетах. Одним из первых *типизированных* языков *функционального* программирования вполне можно рассматривать *Standard ML*, однако он содержит ряд императивных черт, например, ссылки на *изменяемые* значения. С другой стороны, в языке имеется очень интересная реализация модульности и мощная полиморфная система типов. К семейству языков *ML* в полной мере относятся языки *Caml Light* и *Objective Caml*; более того, *второй* от *первого* отличается, в основном, поддержкой классического *объектно-ориентированного* программирования. Между тем, подобно *Standard ML*, язык, являясь строгим, имеет определенную встроенную поддержку отложенных вычислений;
- *Miranda* – разработан *Д. Тернером* в качестве стандартного функционального языка, который использует *отложенные* вычисления и имеет строгую полиморфную систему типов. Язык *Miranda* оказал очень большое влияние на разработчиков языка *Haskell*;
- *Haskell* – один из самых распространенных современных стандартных, нестрогих и *чисто-функциональных* языков программирования. Он поддерживает все черты языков этого типа, включая полиморфную типизацию, отложенные вычисления и функции более высокого порядка. Язык *Haskell* имеет также передовую систему типов, которая поддерживает систематический режим *перезагрузки* и систему модулей. Данный язык предназначен для программирования широкого круга приложений от численных до символьных; к тому же, язык имеет выразительный синтаксис и богатое разнообразие встроенных типов данных. *Haskell* стал языком быстрой разработки очень надежных, кратких и корректных программ. Имеются средства взаимодействия с кодом на ряде других языков программирования. Есть также встроенная поддержка *многозадачного* и *параллельного* программирования, существует развитый инструментарий (*средства для автоматического тестирования, отладки и профилирования, в том числе для параллельных программ*), существует очень много библиотек с открытым исходным кодом (*более 1800 пакетов в одном только архиве Hackage*). В настоящее время для этого языка существует множество компиляторов и интерпретаторов, доступных на бесплатной основе. При этом, начинающие пользователи могут начать освоение *Haskell* с *Hugs* – небольшого портативного *Haskell-интерпретатора*;

- *Gofer* – упрощенный диалект языка *Haskell*, ориентированный на задачи обучения в университетах парадигме функционального программирования;
- *Hugs 98* – система функционального программирования, базирующаяся на *Haskell* и фактически представляющая собой стандарт для нестрогих функциональных языков программирования. По сути дела это интерпретатор для *Haskell*, выросший из *Gofer* и в отличие от *Haskell* являющийся переносимым с простой инсталляцией, делая его привлекательным для начинающих в функциональном программировании;
- *Clean* – специально ориентирован на параллельные и распределенные вычисления. По синтаксису напоминает язык *Haskell*; использует отложенные вычисления. Более того, с компилятором поставляется набор библиотек (*I/O libraries*), которые позволяют программировать графический пользовательский интерфейс под *Win32* или *MacOS*.

Кратко уместно отметить как принципиальные отличия функциональных языков (выше отмечены некоторые из них) от императивных языков программирования (*Pascal*, *C++*, *Ada*, *Java* и др.), так и преимущества и недостатки языков обоих типов. Неймановская архитектура компьютера, исповедующая последовательный принцип вычислений, в основе предполагает, что программа должна состоять из последовательности команд, выполняемых процессором и модифицирующих память. В этой связи и архитектура языков программирования максимально приближалась к архитектуре компьютера. И по этой причине был создан т.н. императивный стиль программирования, основными признаками которого являются ориентированность, прежде всего, на последовательное выполнение команд, оперирующих с памятью и итеративные циклы. И хотя почти за полвека было создано немало довольно развитых императивных языков, существенно усовершенствованы механизмы и методы императивного программирования, однако базовая идея, лежащая в его основе, остается без изменений – программы, написанные на таких языках, описывают процесс последовательного, пошагового решения задач.

Каждое вычисление можно представить себе в виде некоторого процесса, имеющего вход и выход, а так как функциональная зависимость определяет взаимосвязь между аргументами функции и результатом ее вызова, то понятие *функции* является вполне адекватным для описания вычислительного процесса. Данное простое соображение и положено в основу функционального стиля программирования. При таком подходе программа вычислений представляет собой систему определений функций. Функции определяются через другие функции либо рекурсивно. Таким образом, программист на функциональном языке не описывает порядок вычислений, а описывает нужный результат в виде системы функций. В процессе выполнения такой функциональной программы входящие в нее функции при вызове получают значения для аргументов, вычисляются, при необходимости вычисляя другие функции, и возвращают нужный результат. По большому счету, если императивные языки строятся на присваиваниях и циклах, то функциональные языки на рекурсиях. Из преимуществ функциональных языков следует отметить следующие наиболее важные, а именно:

- программы на функциональных языках, как правило, намного короче и проще, чем те же самые программы на императивных языках;
- почти все современные функциональные строго типизированные, что обеспечивает безопасность программ, т.к. программа, не прошедшая проверку типов, завершается

с выводом соответствующей диагностики. Данное обстоятельство позволяет выявлять и устранять большую часть ошибок уже на стадии компиляции, позволяя сокращать время отладки наряду с общим временем разработки программы. Более того, строгая *типизация* позволяет генерировать более эффективный код, ускоряя его выполнение;

- в функциональном языке *функции* могут передаваться в качестве аргумента другим функциям либо возвращаться в качестве результата их вызова;

- в чистых функциональных языках (*не допускающих побочных эффектов для функций*) отсутствует оператор присваивания, объекты такого языка нельзя модифицировать и удалять, можно лишь создавать *новые* путем декомпозиции и синтеза существующих. Ненужными объектами занимается встроенный сборщик «мусора». Поэтому в чистых функциональных языках все функции свободны от побочных эффектов. Между тем, функциональные языки могут имитировать полезные императивные свойства. Не все функциональные языки являются чистыми, поскольку во многих случаях допущение *побочных* эффектов позволяет существенно упрощать программирование. Однако на сегодня наиболее развитые функциональные языки являются, как правило, чистыми.

Наряду с рассмотренными чертами *функциональных* языков, которые в значительной степени определяют их преимущества перед языками императивного типа, подобно вторым первые также используют различные механизмы поддержки модулей, давая возможность разбивать программы на ряд относительно независимых *блоков (модулей)* с четко определенными связями между ними, облегчая процесс программирования и поддержки больших программных систем. Наряду с этим, функциональные языки в определенной мере могут поддерживать и т.н. *отложенные* вычисления. В этом случае язык называется *ленивым* (например, *Gofer, Haskell, Miranda*), в противном случае он называется *строгим* (например, *Caml, Standard ML*). Более того, многие *ленивые* языки являются *чистыми* функциональными языками. Между тем, *чистые* функциональные языки имеют весьма существенные преимущества – наряду с более простым *анализом* программ они хорошо приспособлены к распараллеливанию; параллелизм возможно организовать как на уровне компилятора с языка, так и на уровне архитектуры самой вычислительной системы. Со многими интересными вопросами, связанными с темой *функционального* программирования, читатель может ознакомиться, например, в [86]. Тогда как с довольно интересными критическими замечаниями по функциональным языкам и возможными путями их устранения возможно ознакомиться в статье [87].

Функциональные языки программирования, особенно *чисто* функциональные, были в значительной степени в ходу в академической среде, но не в области коммерческой разработки программного обеспечения. Между тем, такие известные *функциональные* языки как *Erlang, Objective Caml, Haskell* и *Scheme* начали использоваться достаточно широким кругом организаций в промышленных и коммерческих приложениях. При этом, функциональное программирование находит применение в промышленности через проблемно-ориентированные языки программирования, например, *K (анализ в финансовой сфере), R (статистика), CAS Mathematica* и др. В качестве *функциональных* языков программирования можно рассматривать электронные таблицы (*SpreadSheets*). Более того, программирование в функциональном стиле может быть реализовано на императивных языках. Так, императивный язык *Perl* допускает также использование

понятий функционального программирования [88], в то время как один из наиболее широко используемых языков *JavaScript* включает и функциональные возможности. Язык *F#* (для платформы *.NET*) является языком программирования, охватывающим как *функциональное* программирование, так и *императивное* программирование вместе с *объектно-ориентированным*. Язык *F#* является диалектом *ML*, в значительной степени совместимым с реализацией языка *CamL*, также являющегося диалектом языка *ML*.

Действительно, ряд концепций и парадигм специфичны именно функциональному программированию и отсутствуют в императивном программировании. Между тем, многие языки программирования, как правило, базируются на *нескольких* парадигмах программирования, поэтому императивные языки программирования вполне могут использовать концепции функционального программирования. Из этих концепций можно выделить такие, как использование функций высших порядков, т.е. функций, допускающих в качестве *аргументов* функции и возвращающих в качестве результата другие функции, в частности, функции дифференцирования и интегрирования. Не менее важной концепцией являются т.н. *чистые* функции, чей результат выполнения зависит только от их фактических аргументов при вызове. Такие функции обладают рядом полезных свойств, часть из которых возможно использовать для оптимизации кода и распараллеливания вычислений. Между тем, следует отметить, что и в случае *императивных* языков программирования вполне можно создавать *чистые* функции и функции *высших* порядков. При этом, некоторые компиляторы (например, *Fortran 2003* и *gcc*) с императивных языков программирования для обеспечения *оптимизации* кода располагают ключевыми словами для обозначения *чистых* функций. Итак, проблема состоит не в возможности средствами языка запрограммировать *чистую* функцию, а в том, чтобы *компилятор* с языка распознавал ее как чистую функцию и обрабатывал ее соответствующим образом, в частности, в режиме распараллеливания.

Наряду с отмеченными концепциями *функциональные* языки для организации циклов используют рекурсивные конструкции. Строго говоря, в функциональной парадигме программирования нет такого понятия как «*цикл*». Рекурсивные функции вызывают сами себя, обеспечивая повторное выполнение операции. Между тем, использование рекурсий может потребовать большой стек; для решения данной задачи существуют специальные приемы. Так, рекурсивные функции можно обобщать на базе функций высших порядков, используя, например «*свертку-развертку*». Функции данного типа играют роль циклов в императивных языках программирования.

В принципе, нет особых затруднений для программирования в *функциональном* стиле на языках, не являющихся функциональными. Те же императивные языки, которые поддерживают типичные конструкции функциональных языков такие, как функции высшего порядка и дополнение списков, оказываются более удобными в применении функционального стиля. В частности, в языке *C* указатели на функцию вполне могут быть использованы для обеспечения эффекта функций высшего порядка. Более того, функции высшего порядка и *отложенная* списочная структура реализованы также и в библиотеках языка *C++*. В языке *C#* возможно использовать *λ-функции* для создания программ в функциональном стиле. Некоторые языки, имеющие довольно развитые средства метапрограммирования, позволяют создать объекты данных и программные

конструкции, специфичные для функционального стиля, предоставляя возможность на их основе писать достаточно эффективные функциональные программы.

И если *Mathematica*-язык, исповедующий смешанную концепцию *функционального* и *процедурного* программирования, поддерживает функциональное программирование, то *Maple*-язык, исповедующий концепцию сугубо *процедурного* программирования, в то же время допускает целый ряд элементов функционального программирования.

Однако, прежде всего несколько слов о *системных* функции, т.е. функциях собственно самого *Math*-языка и его окружения. Вообще говоря, эти системные средства называть функциями не совсем правильно, поскольку реализация многих из них базируется на процедурной организации, однако мы остановились именно на такой терминологии, присущей собственно самой системе. Итак, система *Mathematica* располагает весьма большим количеством встроенных функций, в то же время она предоставляет весьма простые механизмы для определения *собственных* функций пользователя. В наиболее простом случае функция  $F$  с несколькими формальными аргументами  $x, y, z, \dots$  имеет следующий весьма простой формат, а именно:

$F[x\_ y\_ z\_ \dots] \{:= | =\}$  *Выражение (как правило, зависящее от переменных  $x, y, z, \dots$ )*

Так,  $F[x_] := x^3 + 78$  определяет функцию  $F(x) = x^3 + 78$  в стандартной математической нотации; вызов такой функции на конкретном фактическом аргументе определяется как  $F[x]$ , в частности, как иллюстрирует следующий простой пример, а именно:

```
In[2142]:= F[x_] := x^3 + 78; F[460]
Out[2142]= 97336078
```

Для получения определения *любой* функции (и не только функции, а любого определения по оператору соответственно отложенного " $:=$ " или немедленного " $=$ " присвоения), исключая *встроенные* в пакет функции, служат как встроенная функция **Definition**, так и наши средства, рассмотренные в предыдущем разделе, позволяя получать *оптимальные* в вышерассмотренном отношении определения как процедур, так и функций.

Для присвоения *атрибута* символу, в частности определяющему имя процедуры или функции, система располагает функцией **SetAttributes**, чей вызов **SetAttributes** $[x, a]$  присваивает символу  $x$  атрибут  $a$ , в частности, *Protected*, который защищает символ  $x$  от модификации. Именно данным атрибутом наделены все встроенные функции. В случае обладания символом  $x$  атрибутами, они по вызову **Definition** $[x]$  возвращаются вместе с определением  $x$ , если оно имеется; в противном случае возвращаются только атрибуты, приписанные символу  $x$ . Система *Mathematica* располагает *встроенными* 19 атрибутами, определяющими различные свойства объекта, которому они приписаны по функции **SetAttributes**, полный список которых с их назначением можно получить в справочной базе системы по фразе "*tutorial/Attributes*". Среди данных 19 атрибутов можно отметить такие, как *Flat*, *Orderless*, *Listable*, *OneIdentity*, *NumericFunction* и др. В частности, кроме защиты от модификаций (*Protected*) атрибут *Listable* определяет, что функция/процедура с *Listable*-атрибутом должна автоматически применяться к каждому из элементов кортежа в качестве ее фактических аргументов.

Между тем, стандартные средства работы с атрибутами имеют ряд весьма серьезных недостатков, оперируя непосредственно с символами, как иллюстрирует фрагмент:

```

In[3405]:= a = {m, n, p}; Attributes[n]
Out[3405]= {}
In[3406]:= Attributes[a[[2]]]; SetAttributes[a[[2]], Protected]
Attributes::ssle: Symbol, string, or HoldPattern[symbol] expected
at position 1 in Attributes[a[[2]]]. >>
SetAttributes::sym: Argument a[[2]] at position 1 is expected to be a symbol. >>
Out[3406]= SetAttributes[a[[2]], Protected]
In[3407]:= Attributes[n]
Out[3407]= {}
In[3408]:= SetAttributes1[a[[2]], {Protected, Listable}]
In[3409]:= Attributes1[a[[2]]]
Out[3409]= {Listable, Protected}
In[3410]:= Attributes[78]
Attributes::ssle: Symbol, string, or HoldPattern[symbol] expected
at position 1 in Attributes[78]. >>
Out[3410]= Attributes[78]
In[3411]:= Attributes1[78]
Out[3411]= $Failed

```

С целью устранения данных недостатков и были созданы модификации *стандартных* средств для работы с атрибутами, не только расширяющие стандартные функции на аргументы, в качестве которых допустимы любые выражения, но и обеспечивающие более удобно непрерывные вычисления в *процедурном* режиме выполнения программ. Конец предыдущего фрагмента иллюстрирует сравнение результатов использования как стандартных, так и наших средств на одном и том же весьма простом выражении. Данные средства рассматриваются в настоящей книге и представлены в пакете [120].

Рассмотрим теперь вкратце элементы функционального программирования в среде пакета *Mathematica*, основу которого составляет понятие *чистой функции*. *Чистыми* называют функции, которые не имеют побочных эффектов ввода-вывода и памяти (они зависят *только* от своих аргументов и возвращают *только* свой результат). Чистые функции обладают рядом полезных свойств, которые, например, можно использовать для оптимизации кода либо распараллеливания. В [101] обсуждались свойства такого типа функций. Итак, *чистые функции* – одно из основных понятий функционального программирования, которое является *составляющей* всей системы программирования в *Mathematica* в целом. В дальнейшем вопросы, относящиеся к данной составляющей будут нами рассмотрены более детально, здесь определим лишь основные понятия. В *Mathematica* чистые функции определяются следующим образом. *Чистая* функция в среде *Mathematica* имеет следующие три формата кодирования, а именно:

**Function**[*x*, *Тело функции*] – чистая функция с одним формальным аргументом *x*;

**Function**[*{x1, x2, ..., xp}*, *Тело функции*] – чистая функция с формальными аргументами *x1, x2, ..., xp*;

(*Тело функции*) & – чистая функция с формальными аргументами #, #1, #2, ..., #*n*.

При этом, при использовании третьего формата, часто называемого *краткой формой* чистой функции, его идентификатором служит символ *амперсанда (&)*, чье отсутствие вызывает или ошибочные ситуации, или неверные результаты в виду невозможности идентифицировать требуемую чистую функцию. Читатель, знакомый с формальной логикой либо языком программирования *Lisp*, несложно может отождествить чистые функции с  $\lambda$ -выражениями или *непоименованными* функциями. Более того, чистые функции достаточно близки к математическому понятию *операторов*. В определении чистой функции используются т.н. заменители (*#*) переменных, а именно:

- #** – первая переменная чистой функции;
- #*n*** – *n*-я первая переменная чистой функции;
- ##** – последовательность всех переменных чистой функции;
- ##*n*** – последовательность переменных чистой функции, начиная с *n*-й.

В отличие от ранее рассмотренных объектов таких, как *функции* и *процедуры*, при их применении необходимо явно кодировать их имена с указанием аргументов либо без оных, тогда как в случае применения *чистых* функций нет необходимости указывать их имена, позволяя кодировать их определения непосредственно в точке вызова, что обусловлено тем, что результаты вызова чистых функций зависят только от значений получаемых ими *фактических* аргументов. Например, выборка из списка *L* элементов, удовлетворяющих определенным условиям, и поэлементное применение функции к элементам списка может выполняться по конструкциям формата **Select**[*L*, **Test**[*#*] &] и **Map**[**F**[*#*] &, *L*] соответственно, как иллюстрирует следующий простой пример:

```
In[2341]:= L = {a, 70, c, 75, h, 42, g, Art, Kr, 2012, s, 450}; Select [L, IntegerQ[#] &]
Out[2341]= {70, 75, 42, 2012, 450}
In[2342]:= Map[(#^2 + #) &, {1, 2, 3, 4, 5, 6, 7, 8, 9}]
Out[2342]= {2, 6, 12, 20, 30, 42, 56, 72, 90}
```

При использовании краткой формы чистой функции следует быть осторожным при ее кодировании, т.к. нотация амперсанда имеет довольно низкий приоритет. Поэтому выражение **#1 + #2 - #3 + #2 &** без круглых скобок корректно, тогда как, вообще говоря, они обязательны, в частности, при использовании чистой функции в качестве правой части правила преобразования, как иллюстрирует весьма простой пример, а именно:

```
In[2392]:= {a /. a -> #1 + #2 + #3 &, a /. a -> (#1 + #2 + #3 &)}
Out[2392]= {a /. a -> #1 + #2 + #3 &, #1 + #2 + #3 &}
In[2393]:= {Replace[a, a -> #1*#2*#3 &], a /. a -> (#1*#2*#3 &)}
Replace::reps: {a -> #1 #2 #3 &} is neither a list of replacement rules nor a valid
dispatch table, and so cannot be used for replacing. >>
Out[2393]= {Replace[a, a -> #1 #2 #3 &], #1 #2 #3 &}
```

В сочетании с рядом функций, например, **Map**, **Select** и некоторых других достаточно удобно использовать чистые функции, поэтому вопрос *конвертации* из традиционных функций в *чистые* представляется довольно актуальным; для его решения могут быть использованы различные подходы, включая создание процедур-конверторов. Нами чистые функции использовались достаточно широко при программировании целого ряда как прикладных, так и различного типа системных задач [90,98,100,101,120].

Следующая процедура обеспечивает *конвертацию* функции, определяемой форматом  $G[x\_y\_ \dots] := W(x, y, \dots)$ , в чистую функцию любого допустимого формата, а именно: вызов **FuncToPure**[*x*] возвращает чистую функцию, являющуюся *аналогом* функции *x*, третьего формата, тогда как вызов **FuncToPure**[*x*, *p*], где *p* – произвольное выражение, возвращает чистую функцию 2-х первых форматов. Следующий фрагмент приводит исходный код процедуры **FuncToPure** с типичными примерами ее использования.

```
In[2822]:= FuncToPure[x_ /; QFunction[ToString[x]], y___] := Module[{a = HeadPF[x],
    b = Map[ToString, Args[x]], c = Definition2[x][[1]], d, k = 1, h, t, g = {}, p},
    d = Map[First, Mapp[StringSplit, b, "_"]; p = StringTrim[c, a <> " := "];
    h = "Hold[" <> p <> "]; {t, h} = {Length[b], ToExpression[h]};
    While[k <= t, g = Append[g, d[[k]] <> " -> #" <> ToString[k]]; k++;
    h = ToString1[ReplaceAll[h, ToExpression[g]]]; g = StringTake[h, {6, -2}];
    ToExpression[If[{y} != {}, "Function[" <> If[Length[b] == 1,
    StringTake[ToString[d], {2, -2}], ToString[d]] <> ", " <> p <> "]", g <> "&"]]]

In[2823]:= G[x_Integer, y_Integer, z_Real] := z*(x + y) + Sin[x*y*z]; FuncToPure[G]
Out[2823]= #3 (#1 + #2) + Sin[#1 #2 #3] &
In[2824]:= G[x_Integer, y_Integer, z_Real] := z*(x + y) + Sin[x*y*z]; FuncToPure[G, 78]
Out[2824]= Function[{x, y, z}, z (x + y) + Sin[x y z]]
In[2825]:= V[x_ /; IntegerQ[x]] := If[PrimeQ[x], True, False];
    Select[{47, 70, 23, 16, 450, 13, 6, 42, 567, 2, 123, 321, 17, 29}, FuncToPure[V]]
Out[2825]= {47, 23, 13, 2, 17, 29}
In[2826]:= {S[x_] := x^2 + 23*x + 16; FuncToPure[S, 460], FuncToPure[S][78]}
Out[2826]= {Function[x, x^2 + 23 x + 16], 7894}
```

Между тем, применяя данную процедуру для конвертации традиционной функции в чистую функцию, следует иметь в виду ряд существенных моментов, а именно. Во-первых, результирующая чистая функция не будет обладать атрибутами, опциями и начальными значениями аргументов, а также тестами на допустимость фактических аргументов. Во-вторых, *конвертация* автоматически не делает результирующую функцию чистой, если исходная традиционная функция таковой не являлась, т.е. результат ее вызова зависит только от получаемых фактических аргументов. Ряд полезных средств работы с чистыми функциями будет рассмотрен в книге несколько ниже.

Поскольку процедуры обоих типов (*Module*, *Block*) наряду с функциями пользователя являются базовыми объектами *процедурного* программирования в среде *Mathematica*, в качестве весьма важной задачи является создание средств для тестирования объекта на предмет принадлежности его типу {*Module*, *Block*, *Function*}. Данную задачу вполне успешно решает процедура **TestBFM**, представленная нижеследующим фрагментом.

```
In[2210]:= TestBFM[x_] := Module[{a = Flatten[{PureDefinition[x]}], b, d, h, p, k, j, t = {}},
    If[MemberQ[{$Failed, "System"}, a[[1]]], Return[$Failed],
    b = Flatten[{HeadPF[x]}];
    For[k = 1, k <= Length[a], k++, d = a[[k]];
    p = Map[b[[k]] <> # &, {" := ", " = "}]];
```



```

h = StringReplace[d, {p[[1]] -> "", p[[2]] -> ""}, 1];
If[SuffPref[h, "Module[{", 1], t = AppendTo[t, "Module"],
  If[SuffPref[h, "Block[{", 1], t = AppendTo[t, "Block"],
    If[SuffPref[h, "DynamicModule[{", 1],
      t = AppendTo[t, "DynamicModule"],
      t = AppendTo[t, "Function"]]]]]]; If[Length[t] == 1, t[[1]], t]

In[2211]:= M[x_] := x; M[x_, y_] := Module[{}, x + y]; M[x_, y_, z_] := Block[{}, x + y + z];
In[2212]:= PureDefinition[M]
Out[2212]= {"M[x_] := x", "M[x_, y_] := Module[{}, x+y]", "M[x_, y_, z_] := Block[{}, x+y+z]"}
In[2213]:= TestBFM[M]
Out[2213]= {"Function", "Module", "Block"}
In[2214]:= TestBFM[ProcQ]
Out[2214]= "Module"
In[2215]:= Map[TestBFM, {a + b, avz, Sin, SuffPref, For}]
Out[2215]= {$Failed, $Failed, $Failed, "Function", $Failed}
In[2016]:= TestBFM[TestBFM]
Out[2016]= "Module"

```

Процедура **TestBFM** наряду со стандартными функциями системы использует наши процедуры **HeadPF**, **PureDefinition** и **SuffPref**, также рассматриваемые и в настоящей книге. Вызов процедуры **TestBFM[x]** возвращает тип функционального, блочного или модульного объекта *x* в формате *"Module"*, *"Block"*, *"DynamicModule"*, *"Function"*; тогда как на аргументе *x* другого типа вызов процедуры возвращает **\$Failed**. Если аргумент *x* определяет *одноименный* объект, то вызов процедуры **TestBFM[x]** возвращает список типов составляющих его подобъектов, имеющий взаимно-однозначное соответствие со списком определений, возвращаемым вызовом процедуры **PureDefinition[x]**.

Ранее отмечалось, что в *определениях* процедур/функций целесообразно использовать не оператор немедленного присвоения "=", а оператор отложенного присвоения ":=". В первую очередь, это связано с задачами тестирования процедурных объектов. Если при определении процедуры любого из двух типов был использован оператор "=", то как стандартные, так и наши средства идентифицируют такой объект как функцию, а не как процедуру, как это весьма наглядно иллюстрирует следующий фрагмент. И если определения на основе и оператора немедленного присвоения "=", и оператора отложенного присвоения ":=" эквивалентны, то с точки зрения дифференцировки их типов имеет место вырождение в единый тип **"Function"**, что в целом ряде случаев не представляется довольно удобным, например, в задачах отработки объектов, которая в том либо ином контексте связана с типом обрабатываемого системой объекта.

```

In[2052]:= M[x_] = Module[{}, x]; M[x_, y_] = Module[{}, x*y]; M[x___] = Block[{}, {x}];
M[x_, y_, z___] = Module[{a, b, c}, x*y*z];

In[2053]:= TestBFM[M]
Out[2053]= {"Function", "Function", "Function", "Function"}
In[2054]:= Definition[M]

```

```

Out[2054]= M[x_] = x
          M[x_, y_] = x y
          M[x_, y_, z_] = x y z
          M[x_] = {x}
In[2055]:= M1[x_] := Module[{}, x]; M1[x_, y_] := Module[{}, x*y]; M1[x_] := Block[{}, {x}];
          M1[x_, y_, z_] := Module[{a, b, c}, x*y*z];
In[2056]:= PureDefinition[M1]
Out[2056]= {"M1[x_] := Module[{}, x]", "M1[x_, y_] := Module[{a, b}, x*y]",
          "M1[x_, y_, z_] := Module[{a, b, c}, x*y*z]", "M1[x_] := Block[{}, {x}]}"}
In[2057]:= TestBFM[M1]
Out[2057]= {"Module", "Module", "Module", "Block"}

```

Именно по этой причине целесообразен именно оператор *отложенного* присвоения.

В отличие от **FuncToPure** вызов **ModToPureFunc[x]** обеспечивает конвертацию модуля или блока *x* в *чистую* функцию при следующих условиях: (1) модуль/блок *x* не может иметь *локальных* переменных или все *локальные* переменные должны иметь начальные значения; (2) модуль/блок *x* не может иметь активных глобальных переменных, т.е. переменных, которым в объекте *x* производятся *присвоения*; (3) *формальные* аргументы возвращаемой функции не сохраняют тесты на допустимость; (4) функция наследует атрибуты и опции объекта *x*. Фрагмент представляет код процедуры с примерами.

```

In[2128]:= ModToPureFunc[x_ /; QBlockMod[x]] := Module[{a, Atr = Attributes[x], c, d, p,
          O = Options[x], b = Flatten[{PureDefinition[x]}][[1]],
          n = "$$$" <> ToString[x], k = 1, j, t, q = {}},
  ToExpression["$$$$" <> b]; c = LocalsGlobals1[Symbol[n]]; a = Args[Symbol[n], 78];
  d = StringReplace[PureDefinition[n], HeadPF[n] <> " := " -> "", 1];
  ToExpression["ClearAll[" <> n <> "];"];
  If[c[[3]] != {}, Return[{$Failed, "Globals", c[[3]]}]];
  c = Map[#, ToString[Unique[#]] &, Join[a, c[[1]]]];
  While[k <= Length[c], p = c[[k]]; d = StringReplaceS[d, p[[1]], p[[2]]; k++;
    d = ToString[ToExpression[d]];
    t = Map[ToString, UnDefVars[ToExpression[d]]];
    t = Map[StringTake[#, {1, If[StringFreeQ[#, "$"], -1,
      Flatten[StringPosition[#, "$"]][[1]] - 1]}] &, t];
    k = 1; While[k <= Length[t], j = 1; While[j <= Length[c], If[t[[k]] == c[[j]][[2]],
      q = Append[q, c[[j]][[1]]]; j++; k++; k = 1;
    While[k <= Length[c], p = c[[k]]; d = StringReplaceS[d, p[[2]], p[[1]]; k++;
      If[p = MinusList[q, a]; p != {}, {$Failed, "Locals", p},
      ToExpression["ClearAll[" <> n <> "];"]; n = "$$$$" <> ToString[x];
      ToExpression[n <> " := Function[" <> ToString[a] <> ", " <> d <> "];"];
      If[Atr != {}, ToExpression["SetAttributes[" <> n <> ", " <> ToString[Atr] <> "];"];
      If[O != {}, ToExpression["SetOptions[" <> n <> ", " <> ToString[O] <> "];"]; n]]

```

```

In[2129]:= B[x_, y_] := Block[{a, b = 78, c, d}, (a + b + c)*(x + y + d)];
      B1[x_, y_] := Block[{a = 460, b = 78, c = 71}, (a + b + c)*(x + y)];
      SetAttributes[B1, {Protected, Listable}]
      B2[x_, y_] := Block[{a = 460, b = 78}, h = (a + b)*(x + y); t = 42];
      B3[x_, y_] := Block[{a = 460, b, c}, h = (a + b + c)*(x + y); g = 66];
      B4[x_, y_] := Block[{a = 460, b = 78}, h = (a + b)*(x + y); t = z];
      B5[x_, y_] := Module[{a = 460, b, c, d = 78}, (a + b)*(c + d)]
In[2130]:= ModToPureFunc[B]
Out[2130]= {$Failed, "Locals", {"a", "c", "d"}}
In[2131]:= ModToPureFunc[B1]
Out[2131]= "$$$B1"
In[2132]:= Definition["$$$B1"]
Out[2132]= Attributes[$$$B1] = {Listable, Protected}
      $$$B1 := Function[{x, y}, 609*(x + y)]
In[2133]:= ModToPureFunc[B2]
Out[2133]= {$Failed, "Globals", {"h", "t"}}
In[2134]:= ModToPureFunc[B3]
Out[2134]= {$Failed, "Globals", {"h", "g"}}
In[2135]:= ModToPureFunc[B4]
Out[2135]= {$Failed, "Globals", {"h", "t"}}
In[2136]:= ModToPureFunc[B5]
Out[2136]= {$Failed, "Locals", {"b", "c"}}
In[2137]:= A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] := Module[{a = 42.78}, h*(m+n+p)/a]
In[2138]:= ModToPureFunc[A]
Out[2138]= "$$$A"
In[2139]:= Definition["$$$A"]
Out[2139]= $$$A := Function[{m, n, p, h}, 0.0233754*h*(m + n + p)]
In[2140]:= M[x_, y_ /; StringQ[y]] := Module[{a, b = 78, c = 6, d}, a*x + b*y]
In[2141]:= SetAttributes[M, Protected]; ModToPureFunc[M]
Out[2141]= {$Failed, "Locals", {"a", "d"}}
In[2142]:= G[x_] := Module[{a = 78, b = 460}, a + b]; ModToPureFunc[G]; Definition[$$$G]
Out[2142]= $$$G := Function[{x}, 538]

```

Успешный вызов **ModToPureFunc[x]** возвращает *имя* результирующей чистой функции в форме **ToString[Unique[x]]**, в противном случае вызов возвращает вложенный список формата **{ \$Failed, {"Locals" | "Globals"}, {список переменных в строчном формате} }**, первый элемент которого **\$Failed** определяет недопустимость конвертации, второй элемент – тип переменных, явившихся причиной этого, и третий элемент – список переменных данного типа в строчном формате. Наряду со стандартными средствами процедура в весьма существенной степени использует наши процедуры **Args**, **HeadPF**, **MinusList**, **PureDefinition**, **QBlockMod**, **StringReplaceS**, **UnDefVars**, также рассмотренные в этой книге, что позволило значительно упростить программирование процедуры.

### 8.3. Средства тестирования процедур и функций в *Mathematica*

Определив процедуры двух типов (*Module*, *Block*) и функции, включая чистые, мы, в то же время не располагаем стандартными средствами для идентификации объектов данных типов. В этой связи нами был создан целый ряд средств, которые позволяют идентифицировать объекты указанных типов. В настоящем разделе рассматриваются *нестандартные* средства для тестирования процедурных и функциональных объектов. Отметим, что *Mathematica* – достаточно закрытая система в отличие, например, от его основного конкурента – системы *Maple*, в которой допустима перлюстрация исходных кодов его программных средств, находящихся как в *главной*, так и во вспомогательных библиотеках. Тогда как система *Mathematica* подобной возможностью не располагает. В этой связи предлагаемые ниже программные средства относятся лишь к функциям/процедурам пользователя, загруженным в текущий сеанс из пакета (*m-* или *mx-файл*), либо документа (*nb-файл*; также может содержать пакет) и активизированным в нем.

Как известно, для обеспечения *модульности* программного обеспечения достаточно широко используются *процедуры*, которые в условиях среды *Mathematica* реализуют модульные и блочные конструкции, представленные выше. Как модуль (*Module*), так и блок (*Block*) обеспечивают замкнутую область переменных, поддерживаемую через механизм *локальных* переменных. Процедуры на основе как модульной, так и блочной структур поддерживают, в целом, удовлетворительный механизм модульности. Выше к типу процедур мы отнесли модульные (*Module*) и блочные (*Block*) конструкции, но здесь не все так однозначно и вот почему. В *процедурном* программировании *процедура* представляет собой своего рода т.н. «*черный ящик*», содержание которого скрыто от внешней программной среды, с которой он взаимодействует лишь через аргументы и *глобальные* переменные (если они используются телом процедуры). Тогда как *локальные* переменные имеют область своего действия лишь тело процедуры, не пересекаясь с одноименными переменными внешней среды процедуры. Между тем, именно между процедурами *модульного* и *блочного* типов имеется достаточно существенное различие, базирующееся на механизмах *локальных* переменных, которые используются обеими типами процедур, именно. Это, прежде всего, обусловлено тем обстоятельством, что механизм *локальных* переменных в процедурах модульного типа более развит именно с точки зрения обеспечения более высокого уровня принципа *черного ящика*. Вкратце суть такого различия лежит в следующем. Традиционные языки программирования для работы с переменными используют механизм «*лексического просмотра*», который подобен модульному механизму в *Mathematica*, тогда как *блочный* механизм подобен «*динамическому просмотру*», используемому, например, таким *символьным* языком, как *Lisp*. Так, если *лексический* просмотр рассматривает *локальные* переменные связанными с модулем, *динамический* просмотр рассматривает *локальные* переменные, связанными лишь с конкретным отрезком истории выполнения блока. В языках компилирующего типа, например, *C++* имеется четкое различие между кодом и историей выполнения, тогда как доминирующий *символьный* характер *Mathematica* делает такое различие более скрытым, ибо код может генерироваться во время выполнения программы. Так, при вызове процедуры на модульной основе все появляющиеся в ее теле переменные

полагаются локальными на время выполнения процедуры. Тогда как процедура на блочной основе не рассматривает тело процедуры, используя в течение выполнения процедуры локальные значения для ее переменных. Именно поэтому, процедуры на основе *модульной* структуры намного более применимы, чем на основе *блочной*. Между тем, *блочные* процедуры зачастую удобны в организации различных интерактивных вычислений. Но не только лишь вкус программиста определяет значительно *меньшую* применимость процедур на блочной основе. Таким образом, в общем случае, полагая в среде *Mathematica* наличие процедур 2-х упомянутых типов (*модульного и блочного*), для обеспечения надежности рекомендуется использовать процедуры типа *Module*. В качестве примера различий процедур на обеих основах можно привести следующий:

```
In[2054]:= B[x_] := Block[{a, b, c, d}, x*(a + b + c + d)]
In[2055]:= {a, b, c, d} = {42, 47, 67, 6}
Out[2055]= {42, 47, 67, 6}
In[2056]:= {B[100], a, b, c, d}
Out[2056]= {16200, 42, 47, 67, 6}
In[2057]:= B[x_] := Block[{a = 78, b = 460, c, d}, x*(a + b + c + d)]
In[2058]:= {B[100], a, b, c, d}
Out[2058]= {61100, 42, 47, 67, 6}
In[2059]:= M[x_] := Module[{a = 78, b = 460, c, d}, x*(a + b + c + d)]
In[2060]:= {M[100], a, b, c, d}
Out[2060]= {100 (538 + c$11064 + d$11064), 42, 47, 67, 6}
In[2061]:= {a, b, c, d} = {42, 47, 16, 24}
In[2062]:= B2[x_] := Block[{a, b, c, d}, {a, b, c, d} = {71, 65, 78, 460}; Plus[a, b, c, d]]
In[2063]:= {B2[100], {a, b, c, d}}
Out[2063]= {674, {42, 47, 16, 24}}
```

Из представленного фрагмента следует, что если локальные переменные модульного объекта не пересекаются с областью значений *одноименных* переменных, внешних по отношению к нему, то совершенно иная картина имеет место в случае с локальными переменными блочного объекта, а именно: если всем локальным переменным такого объекта присвоены *начальные* значения либо значения присваивались в самом теле, то именно они имеют силу в области тела объекта; те же переменные объекта, которым такие значения не присваивались, принимают значения одноименных переменных, внешних по отношению к блочному объекту. Итак, при выполнении перечисленных условий *модульный и блочный* объекты относительно *локальных* переменных (*и в целом, как процедурные объекты*) вполне можно рассматривать *эквивалентными*. Естественно, сказанное сохраняет силу также для блочных объектов с пустыми списками *локальных* переменных. Именно указанные соображения и были положены в основу алгоритма, реализуемого процедурой **RealProcQ**, представленной следующим фрагментом.

```
In[2046]:= RealProcQ[x_ /; BlockModQ[x]] := Module[{a, b},
    If[ModuleQ[x], True, a = LocalsGlobals1[x];
    If[a[[1]] == {}, True, b = Select[a[[2]], StringFreeQ[#, " = "] &];
    If[b == {} || MinusList[a[[2]], a[[3]]] == {}, True, False]]]
```

```
In[2047]:= B[x_] := Block[{a, b, c, d}, x*(a + b + c + d)]; RealProcQ[B]
Out[2047]= False
In[2048]:= B1[x_] := Block[{a = 78, b = 460, c = 70, d = 42}, x*(a + b + c + d)]; RealProcQ[B1]
Out[2048]= True
In[2049]:= M2[x_] := Module[{a = 78, b = 460, c, d}, x*(a + b + c + d)]; RealProcQ[M2]
Out[2049]= True
In[2050]:= M3[x_] := Module[{a = 78, b = 460, c, h}, h = x*a*b; x + h]; RealProcQ[M3]
Out[2050]= True
```

Опыт использования процедуры **RealProcQ** подтвердит ее эффективность для задачи тестирования объектов типа **"Block"**, рассматриваемых в качестве реальных процедур. При этом, под *реальной* процедурой мы будем понимать объект типа **{Module, Block}**, который в программной среде *Mathematica* функционально эквивалентен **Module**, т. е. является процедурой в ее классическом понимании. Вызов процедуры **RealProcQ[x]** возвращает **True**, если символ  $x$  определяет **Module** или **Block**, который эквивалентен **Module**, и **False** в противном случае. Процедура наряду со стандартными использует также и наши процедуры **BlockModQ**, **LocalsGlobals1** и **MinusList**, рассматриваемые в настоящей книге. В случае фактического аргумента  $x$ , отличного от типа **{Module, Block}**, вызов **RealProcQ[x]** возвращается *невычисленным*. Таким образом, из всех наших средств, решающих задачу тестирования процедурных объектов, именно процедура **RealProcQ** с максимально возможной достоверностью идентифицирует процедуру в классическом ее понимании; при этом, процедура может быть типа **{Module, Block}**.

В ряде случаев дополнительно к вышерассмотренным средствам тестирования *Math*-объектов достаточно полезной может оказаться и довольно простая процедура, вызов **BlockQ[x]** которой возвращает **True**, если символ  $x$  определяет *блочную* конструкцию, и **False** в противном случае. Следующий фрагмент приводит исходный код **BlockQ** с наиболее типичными примерами ее использования.

```
In[2177]:= BlockQ[x_] := Module[{a = If[SymbolQ[x], Flatten[{PureDefinition[x]}][[1]],
                                     $Failed], b}, If[MemberQ[{$Failed, "System"}, a], False,
                                     b = Mapp[StringJoin, {" := ", " = "}, "Block[{"];
                                     If[SuffPref[a, Map3[StringJoin, HeadPF[x], b], 1], True, False]]]

In[2178]:= Sv[x_] := Module[{}, y := 71; z := 66; {y, z}]; Agn[x_] := Block[{a = 78}, a*x]
In[2179]:= Kr[x_] := Block[{y = a, h = b}, (y^2 + h^2)*x]; Art[x_] := Module[{a = 71}, x*a]
In[2180]:= Map[BlockQ, {Sv, Kr, Agn, Art, a + b, 78}]
Out[2180]= {False, True, True, False, False, False}

In[2250]:= BlockQ1[x_Symbol] := If[TestBFM[x] === "Block", True, False]

In[2251]:= Map[BlockQ1, {Sv, Kr, Agn, Art, 78}]
Out[2251]= {False, True, True, False, BlockQ1[78]}

In[2252]:= ModuleQ1[x_Symbol] := If[TestBFM[x] === "Module", True, False]

In[2253]:= Map[ModuleQ1, {Sv, Kr, Agn, Art, 78}]
Out[2253]= {True, False, False, True, ModuleQ1[78]}
```

```
In[2254]:= ModuleQ2[x_] := Module[{a = If[SymbolQ[x], Flatten[{PureDefinition[x]}][[1]],
    $Failed], b}, If[MemberQ[{$Failed, "System"}, a], False,
    b = Mapp[StringJoin, {" := ", " = "}, "Module[{"];
    If[SuffPref[a, Map3[StringJoin, HeadPF[x], b], 1], True, False]]]
```

Завершается фрагмент примером простой функции **BlockQ1**, которая функционально эквивалентна предыдущей процедуре **BlockQ** и базируется на ранее рассмотренной процедуре **TestBFM**, а также подобной ей не менее простой функцией **ModuleQ1**, чей вызов **ModuleQ1[x]** возвращает **True**, если символ  $x$  определяет *модульную* структуру, и **False** в противном случае. Аналогичен и результат вызова процедуры **ModuleQ2[x]**.

Отметим, предыдущие средства тестирования объектов типа *{Function, Module, Block}* поддерживают лишь одинарные, а не одноименные объекты, т.е. для каждого такого объекта в текущем сеансе активировано единственное определение. Поэтому особый интерес представляют средства тестирования объектов в независимости от количества стоящих за ними определений. Именно такую задачу и решает следующая процедура **FuncBlockModQ**, чей результат вызова **FuncBlockModQ[x, y]** возвращает **True**, если  $x$  – символ, определяющий объект типа *{Function, Module, Block}*; при этом, при наличии для символа  $x$  нескольких определений **True** возвращается только тогда, когда все его определения генерируют объект одного и того же типа. Тогда как через 2-й аргумент  $y$  – неопределенную переменную – возвращается тип объекта в разрезах {"Function", "Block", "Module"}. Если символ  $x$  определяет одноименный объект, чьи определения ассоциируются с подобъектами разных типов, вызов процедуры **FuncBlockModQ[x, y]** возвращает **False**, тогда как через 2-й аргумент  $y$  возвращается "Multiple". Следующий фрагмент представляет исходный код процедуры **FuncBlockModQ** наряду с наиболее типичными примерами ее использования.

```
In[2050]:= FuncBlockModQ[x_Symbol, y_ /; ! HowAct[y]] := Module[{b, c, m, n,
    a = PureDefinition[x]},
    If[MemberQ[{"System", $Failed}, a], False, a = Flatten[{a}];
    b = Flatten[{HeadPF[x]}];
    c = Join[Mapp[StringJoin, b, " := "], Mapp[StringJoin, b, " = "]];
    c = GenRules[c, ""]; c = StringReplace[a, c];
    {m, n} = Map[Length, {Select[c, SuffPref[#, "Block[{" , 1] &],
    Select[c, SuffPref[#, "Module[{" , 1] &]}];
    If[Length[a] == m, y = "Block"; True,
    If[Length[a] == n, y = "Module"; True,
    If[m + n == 0, y = "Function"; True, y = "Multiple"; False]]]]]
```

```
In[2051]:= Sv[x_] := Module[{}, y := 71; z := 66; {y, z}]; Agn[x_] := Block[{a = 78}, a*x]
In[2052]:= B[x_] := Block[{a, b, c, d}, x*(a + b + c + d)]; B[x_, y_] := Block[{}, x + y]
In[2053]:= M[x_] := Module[{a, b, c, d}, x*(a + b + c + d)]; M[x_, y_] := Module[{}, x + y]
In[2054]:= V[x_] := Module[{a, b, c, d}, x*(a + b + c + d)]; V[x_, y_] := Block[{}, x + y]
In[2055]:= F[x_, y_] := x + y; F[x_, y_, z_] := x + y + z
In[2056]:= {FuncBlockModQ[Sv, y], y}
```

```

Out[2056]= {True, "Module"}
In[2057]:= {FuncBlockModQ[B, y1], y1}
Out[2057]= {True, "Block"}
In[2058]:= {FuncBlockModQ[M, y2], y2}
Out[2058]= {True, "Module"}
In[2059]:= {FuncBlockModQ[V, y3], y3}
Out[2059]= {False, "Multiple"}
In[2060]:= {FuncBlockModQ[While, y4], y4}
Out[2060]= {False, y4}
In[2061]:= {FuncBlockModQ[F, y4], y4}
Out[2061]= {True, "Function"}

```

Процедура наряду со стандартными использует и наши средства **GenRules**, **HeadPF1**, **HowAct**, **Mapp**, **PureDefinition**, **SuffPref**, **SymbolQ**, рассматриваемые в данной книге. Ниже будут представлены и другие средства тестирования объектов типа {"Function", "Block", "Module"}, однако и уже упомянутые в значительной мере решают задачу.

При этом, достаточно полезным средством тестирования объектов может оказаться и процедура, чей вызов **ProcFuncBlQ[x, y]** возвращает **True**, если *x* является функцией, процедурой либо блоком, и **False** в противном случае. Более того, при возврате **True** через аргумент *y* – *неопределенную переменную* – возвращается тип *x*-объекта {"Module", "DynamicModule", "Function", "PureFunction", "Block"}, иначе второй аргумент остается неопределенным. Следующий фрагмент приводит исходный код процедуры наряду с наиболее типичными примерами ее использования.

```

In[3178]:= ProcFuncBlQ[x_, y_ /; ! HowAct[y]] := Module[{a = ToString[HeadPF[x]],
    b = ToString[y] <> " = ", c = PureDefinition[x]},
    If[ListQ[c], False, If[SuffPref[a, "HeadPF["], 1],
    If[SuffPref[a, " & "], 2], y = "PureFunction"; True, False],
    If[HeadingQ[a],
    If[SuffPref[c, a <> " := Module["], 1], y = "Module"; True,
    If[SuffPref[c, a <> " := Block["], 1], y = "Block"; True,
    If[SuffPref[c, a <> " := DynamicModule["], 1],
    y = "DynamicModule"; True, y = "Function"; True]], False]]]

In[3179]:= Dm[] := DynamicModule[{x}, {Slider[Dynamic[x]], Dynamic[x]}]
In[3180]:= DPOb[] := Module[{a = 78, b = 65, c = 16, d = 23}, Plus[a, b, c, d]]
In[3181]:= B[x_] := Block[{a}, a = x]; G := Function[460 + 78*# &];
In[3182]:= Clear[g, g1, g2, g3, g4, g5]; {ProcFuncBlQ[Dm, g], ProcFuncBlQ[DPOb, g1],
    ProcFuncBlQ[B, g2], ProcFuncBlQ[G, g3], ProcFuncBlQ[460 + 78*# &, g4],
    ProcFuncBlQ[460, g5]}
Out[3182]= {True, True, True, True, False}
In[3183]:= {g, g1, g2, g3, g4, g5}
Out[3183]= {"DynamicModule", "Module", "Block", "PureFunction", "PureFunction", g5}

```



```
In[3184]:= ClearAll[t]; F[x_] := 460 + 78*x; {ProcFuncBlQ[F, t], t}
Out[3184]= {True, "Function"}
```

Следует отметить, что данная процедура корректно выполняется только на объектах указанного типа при условии, что они имеют *единственные* определения, в противном случае возвращая **False**. Процедура **ProcFuncBlQ** наряду со стандартными использует и наши средства **HeadPF**, **HowAct**, **PureDefinition**, **SuffPref**, также рассматриваемые в настоящей книге.

Как уже отмечалось выше, в общем случае между процедурами типов **Module** и **Block** существуют достаточно принципиальные различия, которые не позволяют априори рассматривать *блочную* структуру в качестве полноценных процедур. Такие различия базируются на различных используемых механизмах *локальных* переменных, как было весьма наглядно проиллюстрировано примерами несколько выше. Можно привести и более сложные примеры таких различий. Поэтому тип *процедуры* следует выбирать достаточно осмотрительно, отдавая предпочтение процедурам типа **Module**. В такой ситуации в ряде случаев может оказаться весьма полезной процедура **BlockToModule**, чей вызов **BlockToModule[x]** возвращает **Null**, обеспечивая конвертацию процедуры **Block**-типа в процедуру **Module**-типа. Следующий фрагмент содержит исходный код процедуры **BlockToModule** наряду с типичными примерами ее использования.

```
In[2068]:= BlockToModule[x_Symbol] := Module[{a = Definition2[x], b, c, d, h = {}, k = 1,
                                             n, m}, If[ListQ[a] && a[[1]] == "System" | |
                                             UnevaluatedQ[Definition2, x], $Failed, b = a[[-1]];
                                             ClearAllAttributes[x]; c = a[[1 ;; -2]]; d = Flatten[{HeadPF[x]}];
                                             For[k, k <= Length[d], k++, {n, m} = {c[[k]], d[[k]]};
                                             If[SuffPref[n, {m <> " := Block[{", m <> " = Block[{", 1},
                                             h = Append[h, StringReplace[n, "Block[{ " -> "Module[{", 1]],
                                             h = Append[h, n]]]; Map[ToExpression, h]; SetAttributes[x, b]]]

In[2069]:= V[x_] := Module[{a, b}, x*(a + b)]; V[x_, y_] := Block[{x}, x + y]; V[x_] := {x}
In[2070]:= Options[V] = {agn -> 65, asv -> 45}; SetAttributes[V, {Protected, Listable}]
In[2071]:= Definition2[V]
Out[2071]= {"V[x_] := Module[{a, b}, x*(a + b)]", "V[x_, y_] := Block[{x}, x + y]",
           "V[x_] := {x}", "Options[V] = {agn -> 65, asv -> 45}", {Listable, Protected}}
In[2072]:= BlockToModule[V]
In[2073]:= Definition2[V]
Out[2073]= {"V[x_] := Module[{a, b}, x*(a + b)]", "V[x_, y_] := Module[{x}, x + y]",
           "V[x_] := {x}", "Options[V] = {agn -> 65, asv -> 45}", {Listable, Protected}}
In[2085]:= G[x_] := Block[{x}, x^2]; G[x_, y_] := Block[{x}, x * y]; G[x_] := Block[{x}, {x}]
In[2086]:= Options[G] = {ian -> 78, rans -> 460}; SetAttributes[G, {Protected, Listable}]
In[2087]:= Definition2[G]
Out[2087]= {"G[x_] := Block[{x}, x^2]", "G[x_, y_] = x*y", "G[x_] := Block[{x}, {x}]",
           "Options[G] = {ian -> 78, rans -> 460}", {Listable, Protected}}
```

```
In[2088]:= BlockToModule[G]; Definition2[G]
Out[2088]= {"G[x_] := Module[{}, x^2]", "G[x_, y_] = x*y", "G[x_] := Module[{}, {x}]",
"Options[G] = {ian -> 78, rans -> 460}", {Listable, Protected}}
```

Вызов процедуры **BlockToModule[x]** возвращает **Null**, т.е. ничего, выполняя при этом конвертацию процедуры *x* блочного типа в одноименную процедуру *модульного* типа с сохранением всех *атрибутов* и *опций* исходной процедуры блочного типа. Более того, с объектом *x* может быть ассоциировано несколько определений *модулей*, *блоков* и/или *функций*, однако вызов **BlockToModule[x]** обеспечивает конвертацию только блочных составляющих объекта *x* в модульные структуры. Вышеприведенные примеры весьма наглядно иллюстрируют вышесказанное.

В связи с механизмом глобальных переменных, импользуемых блоками и модулями, следует сделать некоторые пояснения. В данном контексте можно различать два типа глобальных переменных – *пассивные* и *активные*. *Пассивные* глобальные переменные характеризуются тем, что они лишь используются объектом, не изменяя их значений вне объекта. Тогда как *активным* глобальным переменным в теле объекта выполняется присвоение значений по операторам `{":= ", " = "}`, изменяя их значения и вне объекта. В виду сказанного именно *активные* глобальные переменные представляют интерес при обработке *блоков* и *модулей*, и процедур в целом. Ряд наших средств, обрабатывающих объекты данного типа, чьи определения содержат активные глобальные переменные, учитывают указанное обстоятельство, выполняя обработку объектов типа `{"Module", "Block"}` так, чтобы не изменять значений используемых ими активных глобальных переменных вне их области действия. В данном отношении достаточно показательны процедуры **BlockFuncModQ**, **BlockModQ**, **BlockQ** и **ModuleQ**, представленные ниже.

Вызов **BlockFuncModQ[x]** возвращает **True**, если *x* – символ, определяющий типичную (с заголовком) функцию, блок или модуль, и **False** в противном случае. Тогда как вызов **BlockFuncModQ[x, y]** при условии основного возврата **True** через 2-й необязательный аргумент *y* – *неопределенную переменную* – возвращает тип объекта *x* в разрезе `{"Block", "Function", "Module"}`. С другой стороны, вызов **BlockModQ[x]** возвращает **True**, если *x* – символ, определяющий блок либо модуль, и **False** в противном случае. В то время как вызов **BlockModQ[x, y]** при условии основного возврата **True** через *необязательный* аргумент *y* – *неопределенную переменную* – возвращает тип объекта *x* в разрезе `{"Block", "Module"}`. Следующий фрагмент представляет исходные коды процедур **BlockModQ** и **BlockFuncModQ** наряду с наиболее типичными примерами их использования.

```
In[2112]:= BlockFuncModQ[x_, y_] := Module[{a = Flatten[{PureDefinition[x]}][[1]], b,
c, s = FromCharacterCode[6]},
If[MemberQ[{$Failed, "System"}, a], False,
b = StringReplace[a, {" := " -> s, " = " -> s}, 1];
b = StringTake[b, {Flatten[StringPosition[b, s]][[1]] + 1, -1}];
c = If[SuffPref[b, "Module[{" , 1], "Module",
If[SuffPref[b, "Block[{" , 1], "Block", "Function"]];
If[{y} != {} && ! HowAct[y], y = c]; True]]
```

```

In[2113]:= M[x_, y_] := Module[{a = 78, b = 460}, x*y*a*b]; F[x_] := x; B[_] := Block[{}, x]
In[2114]:= {BlockFuncModQ[M, y], y}
Out[2114]= {True, "Module"}
In[2115]:= {BlockFuncModQ[F, y1], y1}
Out[2115]= {True, "Function"}
In[2116]:= {BlockFuncModQ[B, y2], y2}
Out[2116]= {True, "Block"}
In[2139]:= BlockModQ[x_, y___] := Module[{a = Flatten[{PureDefinition[x]}][[1]], b, c,
                                         s = FromCharacterCode[6]},
                                         If[MemberQ[{$Failed, "System"}, a], False,
                                         b = StringReplace[a, {" := " -> s, " = " -> s}, 1];
                                         b = StringTake[b, {Flatten[StringPosition[b, s]}][[1]] + 1, -1];
                                         c = If[SuffPref[b, "Module[{" , 1], "Module", If[SuffPref[b, "Block[{" , 1], "Block"}];
                                         If[{y} != {} && ! HowAct[y], y = c]; If[c === Null, False, True]]]
In[2140]:= {BlockModQ[M, y3], y3}
Out[2140]= {True, "Module"}
In[2141]:= {BlockModQ[F, y4], y4}
Out[2141]= {False, Null}
In[2142]:= {BlockModQ[B, y5], y5}
Out[2142]= {True, "Block"}

```

Из вышесказанного следует, что при программировании средств, манипулирующих с объектами типа *"Block"*, *"Module"* и использующих *глобальные* переменные, нужно учитывать возможность того, что в процессе вызова этих средств на объектах данного типа *глобальным* переменным объектов выполняются присвоения и текущего сеанса, в целом. Так, например, предыдущая версия нашего пакета *AVZ\_Package* [90] содержит несколько подобных средств. Естественно, в целом, это не так уж существенно по той причине, что при вызове таких объектов, используемые ими глобальные переменные и так получают значения, если не предусмотрено противного. Между тем, для чистоты эксперимента все же было бы желательным, чтобы средства, манипулирующие этими объектами, заранее не вычисляли глобальные переменные, используемые ими.

Функции пакета *Mathematica* располагают рядом интересных средств для поддержки работы с динамическими объектами. Так, динамический модуль **DynamicModule**[{*x*, *y*, ...}, *W*] представляет собой объект, который поддерживает тот же самый локальный статус для переменных *x*, *y*, ... в процессе вычисления всех динамических объектов из тела *W*. Указанные в **DynamicModule** переменные по умолчанию имеют значения на протяжении всего текущего сеанса с пакетом. При этом, динамический объект может выступать не только непосредственно в качестве выражения, но также, в частности, в качестве координаты в графическом примитиве, объекта типа «*ползунок*», в качестве установки для опции. Между тем, в отличие от стандартного модуля динамический модуль **DynamicModule** непосредственно не позволяет получать его определение по стандартной функции **Definition**, лишь наши процедуры **Definition2**, **PureDefinition**, рассмотренные выше, решают задачу, как это иллюстрирует следующий фрагмент.

```

In[2060]:= Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 78, b = 460}, a + b*(x + y)]
In[2061]:= Definition[Dm]
Out[2061]= Dm[x_, y_ /; PrimeQ[y]] := a$$ + b$$ (x + y)
In[2062]:= Definition2[Dm]
Out[2062]= {"Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 78, b = 460}, a + b*(x + y)]", {}
In[2063]:= PureDefinition[Dm]
Out[2063]= "Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 78, b = 460}, a + b*(x + y)]"
In[2099]:= ModuleQ[x_Symbol, y___ /; y == Null || SymbolQ[y] && ! HowAct[y]] :=
                Module[{a = PureDefinition[x], b},
                If[ListQ[a] || a == "System" || a === $Failed, False, b = HeadPF[x];
                If[SuffPref[a, b <> " ] := " <> "Module[{" , 1],
                If[{y} != {}, y = "Module"]; True,
                If[SuffPref[a, b <> " ] := " <> "DynamicModule[{" , 1],
                If[{y} != {}, y = "DynamicModule"]; True, False]]]]
In[2100]:= Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 78, b = 460}, a + b*(x + y)]
In[2101]:= {ModuleQ[Dm, t], t}
Out[2101]= {True, "DynamicModule"}
In[2102]:= V[x_] := Module[{a, b}, x*(a + b)]
In[2103]:= {ModuleQ[V, t1], t1}
Out[2103]= {True, "Module"}
In[2104]:= V[x_, y_] := Block[{}, x + y]; V[x_] := {x}; {ModuleQ[V, t2], t2}
Out[2104]= {False, t2}
In[2105]:= {ModuleQ[Sin, t2], t2}
Out[2105]= {False, t2}
In[2106]:= {ModuleQ[460, t2], t2}
Out[2106]= {ModuleQ[460, t2], t2}

```

Завершает настоящий фрагмент достаточно полезная процедура **ModuleQ**, чей вызов **ModuleQ[x]** возвращает **True**, если объект  $x$ , заданный символом, является модулем, и **False** в противном случае; тогда как вызов **ModuleQ[x, y]** со вторым необязательным  $y$ -аргументом – *неопределенной переменной* – через  $y$  возвращает тип модуля  $x$  в разрезах {"**Module**", "**DynamicModule**"}. При этом, вызов процедуры на кортеже некорректных фактических аргументов возвращается *невычисленным*. В остальных же случаях вызов **ModuleQ[x, y]** процедуры возвращает значение **False**. Процедура **ModuleQ** наряду со стандартными использует и наши средства **HeadPF**, **HowAct**, **PureDefinition**, **SuffPref**, **SymbolQ**, также рассматриваемые в настоящей книге. Между тем, следует отметить ряд весьма существенных моментов, учитываемых алгоритмом, реализуемым данной процедурой. Прежде всего, процедура **ModuleQ** ориентирована на *модульный* объект  $x$ , имеющий уникальное определение, возвращая на одноименных объектах значение **False**. Более того, алгоритм процедуры исходит из того, что определение модульного объекта  $x$  базируется на операторе *отложенного* присвоения ":", а не на операторе "=" *немедленного* присвоения, ибо в последнем случае объект  $x$  распознается стандартной функцией **Definition** и нашими тестирующими средствами в качестве *функции*. На

наш взгляд, процедура **ModuleQ**, представленная фрагментом, достаточно полезна в программировании различного типа задач и, прежде всего, системного характера.

Для тестирования объектов на процедурный тип нами был предложен ряд процедур, среди которых можно отметить такие, как **ProcQ**, **ProcQ1** и **ProcQ2**. Вызов процедуры **ProcQ[x]** обеспечивает тестирование *x*-объекта на предмет быть *процедурным* объектом *{Block, Module}*; тогда как процедура **ProcQ1** является весьма полезной модификацией процедуры **ProcQ**, ее вызов **ProcQ1[w, t]** возвращает **True**, если *w* – объект типа **Block**, **Module** или **DynamicModule**, и "Others" или **False** в противном случае; при этом, тип объекта *w* возвращается через фактический *t*-аргумент. В нашей книге [101] и пакете *AVZ\_Package* [90] представлены исходный код упомянутых процедур, их описание и наиболее типичные примеры их применения. Ряд приемов, использованных при их написании, может оказаться довольно полезным в практическом программировании. При этом, процедура **ProcQ** достаточно реактивна, обрабатывает атрибуты и опции, но имеет ряд ограничений, прежде всего, в случае одноименных объектов [101].

```
In[2092]:= ProcQ[x_] := Module[{a, atr = Quiet[Attributes[x]], b, c, d, h},
    If[ListQ[atr] && atr != {}, ClearAllAttributes[x]];
    a = Quiet[SubsDel[ToString[InputForm[Definition[x]]],
        "" <> ToString[x] <> "", {"[", ",", " ", "-1"}];
    Quiet[b = StringTake[a, {1, First[First[StringPosition[a, {" := Block[{", " :=Block[{"] - 1}]}];
    c = StringTake[a, {1, First[First[StringPosition[a, {" := Module[{", " :=Module[{"] - 1}]}];
    d = StringTake[a, {1, First[First[StringPosition[
        a, {" := DynamicModule[{", " :=DynamicModule[{"] - 1}]}];
    If[b === ToString[HeadPF[x]], If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; True,
    If[c === ToString[HeadPF[x]], If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; True,
    If[d === ToString[HeadPF[x]], If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; True,
    If[ListQ[atr] && atr != {}, SetAttributes[x, atr]]; False]]]]

In[2093]:= Map[ProcQ, {Sin, a + b, ProcQ1, ProcQ}]
Out[2093]= {False, False, True, True}

In[2132]:= ProcQ1[x_Symbol, y___/; y == Null || SymbolQ[y] && ! HowAct[y]] :=
    Module[{a = Flatten[{PureDefinition[x]}], b, c, d, g = "$$78", h = {}, p, k = 1, t = {}},
        If[a === "System" || a === $Failed, False, b = Flatten[{TestBFM[x]}];
        c = DeleteDuplicates[b]; If[MemberQ[c, "Function"], False,
        If[MemberQ3[{"Module", "DynamicModule"}, c],
        If[{y} != {}, y = If[Length[b] == 1, b[[1]], b]]; True,
            d = Part[a, Flatten[Position[b, "Block"]]];
            For[k, k <= Length[d], k++, p = g <> ToString[k];
            h = Append[h, p]; t = Append[t, p <> d[[k]]];
            h = Mapp[StringJoin, h, ToString[x]]; Map[ToExpression, t];
            If[DeleteDuplicates[Map[RealProcQ, h]] == {True}, Map[ClearAll, h];
            If[{y} != {}, y = If[Length[b] == 1, b[[1]], b]]; True, Map[ClearAll, h]; False]]]]
```

```
In[2186]:= ProcBMQ[x_ /; BlockModQ[x], y___] := Module[{a, b, c, d},
    If[ModuleQ[x], True, {a, b} = {PureDefinition[x], LocalsGlobals1[x]};
    If[StringQ[a], c = Select[b[[2]], StringFreeQ[#, " = "] &];
    If[c == {}, True, d = MinusList[c, b[[3]]]; If[d == {}, True,
        If[{y} != {} && ! HowAct[y], y = d]; False]],
    "Object <" <> ToString[x] <> "> has multiple definitions""]]]]
In[2187]:= P[x_] := Block[{a = 78, b = 460, c, d, h, g}, h = (a + b)*x; h^2]; {ProcBMQ[P, q], q}
Out[2187]= {False, {"c", "d", "g"}}
In[2188]:= T[x_] := Block[{a = 6, b = 8, c, d, h, g}, {c, d, h, g} = {1, 2, 3, 4}]; {ProcBMQ[T, v], v}
Out[2188]= {True, v}
In[2189]:= G[x_] := Block[{a, b}, x]; G[x_, y_] := Block[{a, b}, x + y]; ProcBMQ[G]
Out[2189]= "Object <G> has multiple definitions"
```

В данном контексте представим процедуру **ProcQ1**, обобщающую процедуры **ProcQ**, **ProcQ1** и **ProcQ2** с применением новых средств, прежде всего, на *одноименный* случай. Предыдущий фрагмент представляет исходный код процедуры **ProcQ1** с примерами ее наиболее типичного применения. Процедура наряду со стандартными средствами существенно использует также и наши средства такие, как **HowAct**, **Mapp**, **MemberQ3**, **PureDefinition**, **RealProcQ**, **SymbolQ**, **TestBFM**, рассматриваемые в настоящей книге. Вызов процедуры **ProcQ1[x]** возвращает **True**, если символ *x* определяет процедурный объект, в качестве которого могут выступать как структуры типа **Block**, **Module** либо **DynamicModule** с уникальными определениями, так и их произвольные сочетания с разными заголовками (*одноименные объекты*). Более того, в случае отдельного блока либо блока в качестве составляющей одноименного объекта *x* возвращается значение **True** лишь тогда, когда блок является процедурным объектом в указанном контексте. В остальных случаях вызов **ProcQ[x]** процедуры возвращает **False**; при недопустимых фактических аргументах вызов процедуры возвращается *невывчисленным*. Между тем, вызов процедуры **ProcQ1[x, y]** со вторым *необязательным аргументом y* – *неопределенной переменной* – возвращает через него тип объекта *x* либо список типов, составляющих объект *x*, если основной возврат **True**; иначе *y* остается неопределенным. Процедура **ProcQ1** достаточно широко используется и довольно полезна во многих приложениях, отличаясь от предыдущей **ProcQ** следующими чертами, а именно: (1) вполне успешно обрабатывает *одноименные* объекты, (2) определяет истинную процедурность в случае *одноименных* объектов, чьи подобъекты являются блоками либо функциями. Наконец, завершает фрагмент процедура **ProcBMQ**, чей вызов **ProcBMQ[x]** с одним аргументом возвращает **True**, если блок или модуль *x* – процедура в указанном контексте, и **False** в противном случае; вызов **ProcBMQ[x, y]** со вторым *необязательным аргументом y* – *неопределенной переменной* – возвращает через нее список локальных переменных блока *x* в строчном формате, которые не имеют начальных значений либо которым в теле блока *x* не производилось присвоений значений. Отметим, процедура **ProcBMQ** ориентирована лишь на *одионочные* объекты, определения которых уникальны, тогда как на *одноименных* возвращается сообщение "*Object <x> has multiple definitions*".

Дополнительно к нашим средствам, тестирующим процедурные объекты, отметим и

процедуру **UprocQ**[*x*], чей вызов возвращает значение **False**, если объект *x* не является процедурой, и 2-элементный список в противном случае; в данном случае его первым элементом возвращается значение **True**, тогда как вторым возвращается тип {"Block" | "Module" | "DynamicModule"} процедуры *x*. Следующий фрагмент представляет код процедуры **UprocQ** с результатами ее применения к процедурам разных типов.

```
In[4183]:= UprocQ[x_;/; SymbolQ[x] && ProcQ[x]] := Module[{a = $$$$$g, b, c,
                                                    d = Attributes[$$$$$g]},
  ToExpression["ClearAttributes["$$$$$g\", Protected]"];
  ToExpression["Clear["$$$$$g\""]; b = ProcQ1[x, $$$$$g];
  c = $$$$$g; $$$$$g = a;
  ToExpression["SetAttributes["$$$$$g\" <> ", \" <> ToString[d] <> \"]"];
  b = {b, c}; If[b[[1]] == False, False, b]]

In[4184]:= Agn[x_, y_] := Module[{a = $HeadProc}, ToExpression[StringTake[a, {1, -2}] <>
  ", z_"] <> " := Module[{x, y, z}, x + 2*y + z]; ToExpression[StringTake[a, {1, -2}] <>
  ", z_;/; PrimeQ[z]]" <> " := Module[{x, y, z}, x + 3*y + z]]

In[4185]:= Kr[x_] := Block[{y = a, h = b}, (y^2 + h^2)*x]; Art[x_] := Module[{a}, a = 78; x + a]
In[4186]:= Sv[y_] := DynamicModule[{x = y}, {Slider[Dynamic[x]], Dynamic[x]}]
In[4187]:= Map[UprocQ, {Agn, Kr, Art, Sv, ProcQ, Subs, Sin, 2013, UprocQ}]
Out[4187]= {{True, "Module"}, {True, "Block"}, {True, "Module"}, {True, "DynamicModule"},
  {True, "Module"}, {True, "Module"}, UprocQ[Sin], UprocQ[2013], {True, "Module"}}
In[4188]:= ArtKr[x_] := Block[{y, h = b}, (y^2 + h^2)*x]; ArtKr[x_] := Module[{x}]
In[4189]:= UprocQ[ArtKr]
Out[4189]= False
```

Следует отметить, что процедура **UprocQ** успешно обрабатывает как уникальные, так и одноименные процедурные объекты, используя процедуры **ProcQ**, **ProcQ1**, которые были рассмотрены в настоящей книге несколько выше.

Рассмотрев основные средства тестирования процедурных объектов, отсутствующие в числе стандартных средств *Mathematica*, вполне резонно рассмотреть подобные им средства для тестирования функциональных объектов, где под *функциональными* мы будем понимать объекты, определения которых имеют следующий формат:

$$F[x_;/; Test_x, y_;/; Test_y, ...] := W(x, y, ...)$$

либо чистые функции одного из следующих форматов, а именно:

**Function**[*Тело*] или краткая форма *Тело* & (*формальные аргументы* # (#1), #2, #3 и т.д.)

**Function**[*x*, *Тело*] – чистая функция с единственным формальным *x*

**Function**[{*x1*, *x2*, ...}, *Тело*] – чистая функция с формальными аргументами {*x1*, *x2*, ...}

Приведем простые примеры на данные типы функций, а именно:

```
In[2125]:= y := Function[{x, y}, x + y]; y1 = Function[{x, y}, x + y]; z := #1 + #2 &;
  z1 = #1 + #2 &; F[x_, y_] := x + y
In[2126]:= {y[78, 460], y1[78, 460], z[78, 460], z1[78, 460], F[78, 460]}
Out[2126]= {538, 538, 538, 538, 538}
```

На объектах данного функционального типа вызовы обеих процедур **ProcQ** и **ProcQ1** возвращают **False**, поэтому для тестирования функционального типа и предлагаются другие средства, рассматриваемые ниже. Однако, прежде всего рассмотрим средства, тестирующие системные функции, т.е. функции самого *Math*-языка и его окружения. Вообще говоря, данные системные средства называть *функциями* не совсем правильно, ибо реализация многих из них базируется на *процедурной* организации, между тем, мы остановились именно на такой терминологии, присущей собственно самой системе. И в этом отношении можно представить средства тестирования системных функций, к тому же в целом ряде довольно важных задач возникает необходимость тестирования объектов на предмет быть стандартными функциями системы *Mathematica*. И в этом отношении достаточно простая процедура **SysFuncQ** решает данную задачу; ее вызов **SysFuncQ[x]** возвращает **True**, если *x*-объект является стандартной функцией системы *Mathematica*, и **False** в противном случае. Следующий фрагмент приводит исходный код процедуры **SysFuncQ** наряду с типичными примерами ее использования.

```
In[2219]:= SysFuncQ[F_] := Module[{a = Quiet[ToString[Definition[F]]], b},
                                b = Flatten[StringPosition[a, "="]; If[b == {}, False,
                                If[StringTake[a, {1, b[[1]] - 1}] == "Attributes[" <> ToString[F] <> "]" &&
                                FindList[FindFile["DocumentationNavigator.nb"],
                                ToString1[ToString[F]]] != {}, True, False]]]
In[2220]:= Map[SysFuncQ, {Sin, Tan, While, If, Do, ProcQ, 78, Length, a + b, Art}]
Out[2220]= {True, True, True, True, True, False, False, True, False, False}
In[3037]:= SysFuncQ1[x_] := MemberQ[Names["System`*"], ToString[x]]
In[3038]:= Map[SysFuncQ1, {Sin, Tan, While, If, Do, ProcQ, 78, Length, a + b, Art}]
Out[3038]= {True, True, True, True, True, False, False, True, False, False}
In[3039]:= t = TimeUsed[]; SysFuncQ[While]; TimeUsed[] - t
Out[3039]= 0.047
In[3040]:= t = TimeUsed[]; SysFuncQ1[While]; TimeUsed[] - t
Out[3040]= 0.015
```

Тогда как достаточно простая функция **SysFuncQ1** – функционально эквивалентная модификация предыдущей процедуры **SysFuncQ**, являясь в то же время и несколько более реактивной, как иллюстрируют последние примеры предыдущего фрагмента.

Рассмотрим теперь средства тестирования функциональных объектов пользователя, в качестве *первого* из которых выступает процедура **QFunction[x]**, являющаяся наиболее общим средством тестирования объектов *x* функционального типа, которая возвращает на традиционной функции *x* и объектах *x*, генерируемых функцией **Compile**, **True**, и **False** в противном случае. При этом, в качестве традиционной понимается функция, определяемая конструкцией формата  $J[x_, y_, \dots] \{:= | =\} J(x, y, \dots)$ . Фрагмент приводит исходный код процедуры **QFunction** с типичными примерами ее использования.

```
In[2093]:= QFunction[x_] := Module[{a = Quiet[Definition2[x]][[1]]},
                                b = ToString[HeadPF[x]]],
                                If[SuffPref[Quiet[ToString1[a]], "CompiledFunction", 1], True,
```



```

If[SuffPref[b, "HeadPF", 1], False,
  b = Map3[StringJoin, b, {" := ", " = " }];
If[MemberQ[{SuffPref[StringReplace[a, b -> ""], "Module", 1],
  SuffPref[StringReplace[a, b -> ""], "Block", 1]}, True], False, True]]]]
In[2094]:= V := Compile[{{x, _Real}, {y, _Real}}, (x + y)^2]; Kr := (#1^2 + #2^4) &;
Art := Function[{x, y}, x*Sin[y]];
GS[x_/_; IntegerQ[x], y_/_; IntegerQ[y]] := Sin[75] + Cos[42];
Sv[x_/_; IntegerQ[x], y_/_; IntegerQ[y]] := x^2 + y^2;
S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
In[2095]:= Map[QFunction, {V, S, Art, Kr, Pi, 42.71, GS, Sv}]
Out[2095]= {True, True, False, False, False, False, True, True}
In[2096]:= G[x_Integer, y_Real, z_Real] := x*y^2 + z
In[2097]:= Map[QFunction, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ]}
Out[2097]= {False, False, True, False}
In[2142]:= QFunction1[x_String] := Module[{atr = Quiet[Attributes[x]], b, d = {}, h = {},
  k = 1, p, c = "$$Art24$Kr16$$"}, Quiet[ClearAllAttributes[x]];
  a = Quiet[StringSplit[ToString[InputForm[Definition[x]]], "\n\n"];
  Quiet[SetAttributes[x, atr]]; If[a === {"Null"} || a === {} || SysFuncQ[x], False,
    For[k, k <= Length[a], k++, p = c <> ToString[k];
    d = Append[d, p <> x]; h = Append[h, p <> a[[k]]];
    Map[ToExpression, h]; c = Map[QFunction, Map[ToExpression, d]];
    Map[Clear, d]; If[DeleteDuplicates[c] == {True}, True, False]]]
In[2143]:= F[x_] := x^2; F[x_, y_] := x + y; F := Compile[{{x, _Real}, {y, _Real}}, (x + y)^2];
F = Compile[{{x, _Real}, {y, _Real}}, (x + y)^2];
In[2144]:= Map[QFunction1, {"Sin", "F", "Art", "V", "Kr", "GS", "Sv", "S"}]
Out[2144]= {False, True, False, True, False, True, True, True}
In[2145]:= G[x_] := x; SetAttributes[G, Protected]; {QFunction[G], QFunction1["G"]}
Out[2145]= {True, True}
In[2146]:= {Map[QFunction, {Art, Kr}], Map[QFunction1, {"Art", "Kr"}]}
Out[2146]= {{False, False}, {False, False}}

```

Однако, процедура **FunctionQ**, успешно тестируя функциональные объекты, которые определяются как традиционными функциями с заголовками, так и генерируемыми стандартной функцией **Compile**, не обрабатывает чистых функций; при этом, данная процедура не обрабатывает и *одноименных* функциональных объектов, как наглядно иллюстрируют последние примеры предыдущего фрагмента. Решает данную задачу процедура **QFunction1**, чей исходный код представлен во второй части предыдущего фрагмента. Вызов **QFunction1[x]** возвращает на традиционной функции *x* и объектах *x*, генерируемых функцией **Compile**, **True**, и **False** в противном случае; более того, на одноименном объекте *x* значение **True** возвращается только в том случае, если все его составляющие являются традиционными функциями и/или генерируемы функцией

**Compile.** Вызов процедуры **QFunction1[x]** предполагает кодирование фактического  $x$  аргумента в строчном формате. Обе процедуры эффективно обрабатывают опции и атрибуты тестируемых объектов. Однако, как процедура **QFunction1**, так и процедура **QFunction** не могут, вообще говоря, корректно тестировать чистые функции, как это довольно наглядно иллюстрирует последний пример предыдущего фрагмента.

Наряду с указанными типами функций система *Mathematica* использует и функцию **Compile**, предназначенную для компиляции функций, которые вычисляют числовые выражения при определенных предположениях. Функция располагает следующими *четырьмя* форматами кодирования, каждый из которых ориентирован на отдельный тип компиляции, а именно:

**Compile[{x1, x2, ...}, expr]** – компилирует функцию для вычисления выражения *expr* в предположении, что все значения аргументов  $x_j$   $\{j = 1, 2, \dots\}$  носят числовой характер;

**Compile[{x1, t1}, {x2, t2}, ..., expr]** – компилирует функцию для вычисления выражения *expr* в предположении, что все значения аргументов  $x_j$  имеют тип  $t_j$   $\{j = 1, 2, \dots\}$ ;

**Compile[{x1, t1, n1}, {x2, t2, n2}, ..., expr]** – компилирует функцию для вычисления *expr* в предположении, что значения аргументов  $x_j$  являются рангами  $n_j$  массива объектов, каждый из которых соответствует типу  $t_j$   $\{j = 1, 2, \dots\}$ ;

**Compile[vars, expr, {{p1, pt1}, {p2, pt2}, ...}]** – компилирует функцию для вычисления выражения *expr* в предположении, что его подвыражения *vars*, которые соответствуют шаблонам  $p_j$ , имеют типы  $pt$   $\{j = 1, 2, \dots\}$ .

Функция **Compile** обрабатывает *числовые* функции, *матричные* операции, *процедурные* и *функциональные* конструкции, функции работы со *списками* и т.д. Функция **Compile** генерирует специальный объект **CompiledFunction**. Вызов **Compile[... , Evaluate[expr]]** определяет, выражение *expr* перед компиляцией следует вычислять алгебраически.

Для тестирования данного типа функций предлагается достаточно простая функция **CompileFuncQ**, вызов которой **CompileFuncQ[x]** возвращает **True**, если  $x$  представляет функцию **Compile**, и **False** в противном случае. Следующий фрагмент представляет исходный код функции с наиболее типичными примерами ее использования.

```
In[2067]:= V := Compile[{x, _Real}, {y, _Real}], (x^3 + y)^2]; Kr := (#1^2 + #2^4) &;
      Art := Function[{x, y}, x*Sin[y]]; H[x_] := Block[{}, x]; H[x_, y_] := x + y;
      SetAttributes["H", Protected]; P[x_] := Plus[Sequences[{x}]];
      GS[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := Sin[78] + Cos[42];
      Sv[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x^2 + y^2;
      Sv = Compile[{x, _Integer}, {y, _Real}], (x + y)^6];
      S := Compile[{x, _Integer}, {y, _Real}], (x + y)^3];
      G = Compile[{x, _Integer}, {y, _Real}], (x + y)]; P[x_] := Module[{}, x]

In[2068]:= CompileFuncQ[x_] := If[SuffPref[ToString[InputForm[Definition2[x]]],
      "Definition2[CompiledFunction[", 1], True, False]

In[2069]:= Map[CompileFuncQ, {Sv, S, G, V, P, Art, Kr, H, GS, ProcQ}]
Out[2069]= {True, True, True, True, False, False, False, False, False, False}
```

```
In[2070]:= Map[CompileFuncQ, {78, avz, a + b, Sin, While, 42.71}]
Out[2070]= {False, False, False, False, False, False}
```

Процедура **CompileFuncQ** расширяет возможности тестирования функциональных объектов в системе *Mathematica*, представляя вполне определенный интерес, в первую очередь, для задач системного программирования.

Ниже приведена функция **PureFuncQ**, вызов которой **PureFuncQ[F]** возвращает **True**, если *F* является чистой функцией, и **False** в противном случае. Пример представляет исходный код функции наряду с примерами ее типичного использования.

```
In[2085]:= PureFuncQ[F_] := Quiet[StringTake[ToString[F], {-3, -1}] == " & " &&
! StringFreeQ[ToString[F], "#"] || SuffPref[ToString[InputForm[F]], "Function", 1]]

In[2086]:= Map[PureFuncQ, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}]
Out[2086]= {True, True, False, False}

In[2087]:= Map[PureFuncQ, {Sin, F, Art, V, Kr, GS, Sv, S}]
Out[2087]= {False, False, True, False, True, False, False, False}

In[2088]:= Z := Function[{x, y, z}, x + y + z]; SetAttributes[Z, Protected]
In[2090]:= {PureFuncQ[Z], Attributes[Z]}
Out[2090]= {True, {Protected}}

In[2091]:= FunctionQ[x_] := If[StringQ[x], PureFuncQ[ToExpression[x]] ||
QFunction1[x], PureFuncQ[x] || QFunction[x]

In[2092]:= Map[FunctionQ, {"G", "ProcQ", "Function[{x, y, z}, x + y*z] ", "#1 + #2*#3 &"}]
Out[2092]= {True, False, True, True}

In[2093]:= Map[FunctionQ, {"V", "S", "Art", "Kr", "Pi", "42.71", "GS", "Sv", "F"}]
Out[2093]= {True, True, True, True, False, False, True, True, True}

In[2094]:= Map[QFunction, {V, S, Art, Kr, Pi, 42.71, GS, Sv, F}]
Out[2094]= {True, True, False, False, False, False, True, True, True}
```

И завершает предыдущий фрагмент простая реализация функции **FunctionQ**, вызов **FunctionQ[x]** которой возвращает **True**, если объект *x* – функция произвольного типа как традиционного, так и чистого, и **False** в противном случае. При этом, имя объекта может кодироваться как в символьном, так и в строчном форматах; во втором случае обеспечивается корректное тестирование объекта *x*, допускающего множественность определений, т.е. объект *x* может быть одноименным в указанном выше смысле. Более того, примеры предыдущего фрагмента позволяют оценивать разницу в получаемых результатах. Между тем отметим, приведенные примеры используют функциональные объекты, определенные в предпоследнем фрагменте. Также следует иметь в виду, что представленные выше средства тестирования относятся к функциям пользователя, и не предназначены для стандартных функций пакета *Mathematica*, возвращая на них, как правило, значение **False**. Итак, определен ряд средств для дифференцированной идентификации как функций пользователя в целом, так и традиционных, и чистых функций в частности – процедуры и функции **FunctionQ**, **QFunction** и **QFunction**, и **PureFuncQ** соответственно. Таким образом, данные средства обеспечивают довольно

строгую дифференциацию такого базового элемента *функционального и процедурного* программирования, как *функция*. Эти и им подобные средства достаточно полезны в прикладном и системном программировании в среде системы *Mathematica*.

Между тем, здесь необходимо еще раз сделать одно весьма существенное замечание. Как уже отмечалось, в отличие от большинства известных языков программирования *Math*-язык идентифицирует процедуры и функции не по их именам, а по *заголовкам*, допуская не только *одноименные* процедуры с разными *заголовками*, но и их сочетания с функциями. Поэтому вопрос тестирования объектов на тип  $\{Procedure, Function\}$  не столь однозначен. Представленные в книге средства тестирования  $\{ProcQ, FunctionQ, QFunction1, PureFuncQ$  и др. $\}$  в качестве аргумента  $x$  допускают объект либо только с единственным заголовком, либо первый, возвращаемый по вызову **Definition**[ $x$ ], как было проиллюстрировано выше. При этом, для одноименных объектов вызовы ряда приведенных выше средств возвращают **True** только в случае, когда составляющие их определения ассоциированы с подобъектами одного и того же типа. В данной связи весьма целесообразно определить процедуру, тестирующую принадлежность объекта  $x$  к группе  $\{Block, CompiledFunction, Function, Module, PureFunction\}$ . В качестве одного из таких подходов можно предложить процедуру, вызов которой **ProcFuncTypeQ**[ $x, y$ ] возвращает **True**, если тип объекта, чье имя  $x$  кодируется в строчном формате, имеет тип  $\{Block, CompiledFunction, Function, Module, PureFunction\}$ , и **False** иначе. Тогда как, через второй аргумент  $y$  – неопределенную переменную – возвращается тип объекта  $x$  в строчном формате. В случае одноименного объекта  $x$  возвращается список типов подобъектов, составляющих  $x$ . Итак, процедура **ProcFuncTypeQ** может применяться в качестве группового теста на принадлежность объекта  $x$  вышеперечисленным типам. Следующий фрагмент приводит исходный код процедуры **ProcFuncTypeQ** наряду с наиболее типичными примерами ее использования.

```
In[2364]:= V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2]; Sv[x_] := Module[{}, x];
  Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &; H[x_] := Block[{}, x];
  GS[x_/, IntegerQ[x], y_/, IntegerQ[y]] := Sin[78] + Cos[42];
  Sv[x_/, IntegerQ[x], y_/, IntegerQ[y]] := x^2 + y^2;
  Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6];
  S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
  G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)];
  H[x_, y_] := x + y; SetAttributes["H", Protected]

In[2365]:= ProcFuncTypeQ[x_, y_/, ! HowAct[y]] := Module[{a1, ArtKr, attrib, b1 = {}, d,
  c = "$Art24Kr16$", k = 1},
  If[Quiet[StringQ[x] && ProcQ[x] || FunctionQ[Symbol[x]]],
    attrib = Attributes[x];
  ArtKr[z_String] := Module[{a, b}, If[TrueQ[ModuleQ[Symbol[z]]], "Module",
    If[TrueQ[BlockQ[Symbol[z]]], "Block",
      a = ToString[InputForm[Definition[z]]];
      b = Map3[StringJoin, ToString[z], {" := ", " = "}]];
```

```

If[SuffPref[a, Mapp[StringJoin, b, "Function[", 1] | |
  StringTake[a, {-3, -1}] == " & ", "PureFunction",
  If[SuffPref[a, Mapp[StringJoin, b, "Compile", 1],
    "CompiledFunction", "Function"]]]];
  ClearAllAttributes[x];
a1 = Flatten[{StringSplit[ToString[InputForm[Definition[x]]], "\n \n"]];
  SetAttributes[x, attrib];
  For[k, k <= Length[a1], k++, d = c <> a1[[k]]; ToExpression[d];
b1 = Append[b1, ArtKr[c <> x]]; ToExpression["Clear[" <> c <> x <> ""]];
  y = If[Length[b1] == 1, b1[[1]], b1]; True, False]]

In[2366]:= {ProcFuncTypeQ["Sv", t], t}
Out[2366]= {True, {"CompiledFunction", "Function", "Module"}}
In[2084]:= Map9[ProcFuncTypeQ, {"G", "V", "Art", "Kr", "GS", "S"}, {g1, g2, g3, g4, g5, g6}]
Out[2084]= {True, True, True, True, True, True}
In[2085]:= {g1, g2, g3, g4, g5, g6}
Out[2085]= {"CompiledFunction", {"CompiledFunction", "PureFunction", "PureFunction",
  "Function", "CompiledFunction"}
In[2368]:= {ProcFuncTypeQ["H", h], h, Attributes[H]}
Out[2368]= {True, {"Block", "Function"}, {Protected}}

```

В целом ряде приложений как прикладного, так и системного характера, связанных с обработкой процедур и функций, **ProcFuncTypeQ** оказывается весьма эффективным групповым тестирующее средство.

Для идентификации функциональных объектов (*традиционных и чистых функций*) в программной среде пакета имеются довольно ограниченные средства, базирующиеся только на вызовах функций **Part[x, 0]** и **Head[x]**, возвращающих заголовки выражения *x*; на чистых функциях возвращающих *Function*, тогда как на традиционных – *Symbol*, как весьма наглядно иллюстрирует следующий простой фрагмент, а именно:

```

In[2905]:= G[x_Integer, y_Real, z_Real] := x*y^2 + z
In[2906]:= Map[Head, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}]
Out[2906]= {Function, Function, Symbol, Symbol}
In[2907]:= Mapp[Part, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}, 0]
Out[2907]= {Function, Function, Symbol, Symbol}
In[2908]:= Map[PureFuncQ, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}]
Out[2908]= {True, True, False, False}
In[2909]:= Map[QFunction, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}]
Out[2909]= {False, False, True, False}
In[2910]:= Map[FunctionQ, {#1 + #2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}]
Out[2910]= {True, True, True, False}
In[2942]:= {m, n} = {#1 + #2*#3 &, Function[{x, y, z}, x + y*z]}; Map[Head, {m, n}]
Out[2942]= {Function, Function}

```

```
In[2943]:= {Map[Part, {m, n}, 0], Map[QFunction, {m, n}]}
Out[2943]= {{Function, Function}, {False, False}}
In[2944]:= {Map[FunctionQ, {m, n}], Map[PureFuncQ, {m, n}]}
Out[2944]= {{True, True}, {True, True}}
```

И в данном контексте достаточно полезной представляется процедура **Head2**, которая является модификацией предыдущей процедуры **Head1**, основываясь на процедуре **ProcFuncTypeQ**, функциях **SymbolQ**, **SystemQ**, и стандартной функции **Head**. Вызов процедуры **Head2[x]** возвращает заголовок либо тип объекта  $x$ , заданного в строчном формате. В принципе, тип объекта вполне можно рассматривать в качестве заголовка в широком его понимании. Именно такой задаче и служит процедура **Head2**, которая обобщает стандартную функцию **Head**, возвращая заголовок выражения  $x$  в разрезах **{Block, CompiledFunction, Function, Module, PureFunction, Symbol, System, Head[x]}**. В качестве сравнения и приводятся примеры использования обеих средств на одном и том же списке тестируемых объектов, которые в целом ряде случаев и подтверждают предпочтение именно процедуры **Head2**. Следующий фрагмент приводит исходный код процедуры **Head2** наряду с наиболее типичными примерами ее использования.

```
In[2151]:= Head2[x_String] := Module[{a, b}, If[Quiet[SystemQ[x]], System, ClearAll[b];
      a = ProcFuncTypeQ[x, b]; If[SameQ[a, False], If[SymbolQ[x], Symbol,
      Head[ToExpression[x]]], Symbol[b]]]
In[2152]:= Map[Head2, {"#1 + #2*#3&", "Function[{x, y, z}, x+y*z]", "G", "ProcQ", "a + b",
      "{x, y, z}", "\"ransian\"", "Avz", "While"}]
Out[2152]= {Function, Function, CompiledFunction, Module, Plus, List, String,
      Symbol, System}
In[2153]:= Map[Head2, {"V", "Art", "G", "ProcQ", "GS", "Sv", "S", "H", "Agn", "78", "42.78"}]
Out[2153]= {CompiledFunction, PureFunction, CompiledFunction, Module, Function,
      {CompiledFunction, Function, Module}, CompiledFunction,
      {Block, Function, Symbol, Integer, Real}}
In[2196]:= Map[Head, {V, Art, G, ProcQ, GS, Sv, S, H, Agn, 78, 42.78}]
Out[2196]= {CompiledFunction, Function, CompiledFunction, Symbol, Symbol,
      CompiledFunction, CompiledFunction, Symbol, Symbol, Integer, Real}
In[2199]:= Head3[x_] := Symbol[If[Part[x, 1] === -1, Head1[-1*x], Head1[x]]]
In[2200]:= {Head[Sin[-a + b]], Head2["Sin[-a + b]"], Head3[Sin[-a + b]]}
Out[2200]= {Times, Times, Sin}
In[2197]:= h[x_] := Module[{b}, ClearAll[b]; HowAct[b]; h[78]
Out[2197]= False
In[2198]:= h1[x_] := Module[{b}, HowAct[b]; h1[78]
Out[2198]= True
```

Тогда как функция **Head3** расширяет стандартную **Head** и наши процедуры **Head1** и **Head2** при условии, тестируемое выражение  $x$  рассматривается с точностью до знака; различие наглядно иллюстрируют результаты вызова данных средств на идентичных фактических аргументах. В целом же, вызов функции **Head3[x]** подобен **Head1[x]**.

При этом, на примере данной процедуры следует акцентировать внимание на одном довольно важном моменте, имеющем существенное значение в программировании и прикладных, и системных задач, а именно. В целом ряде случаев заголовки *вложенных* процедур могут содержать аргументы, в качестве которых выступают *неопределенные* переменные, через которые возвращаются вторичные результаты. В качестве данных переменных-аргументов вполне допустимо использование *локальных* переменных из *главной* процедуры, предварительно очистив их по вызову функции **ClearAll**; данный прием базируется на свойстве *локальных* переменных модулей, проиллюстрированных двумя последними примерами предыдущего фрагмента. Данный прием может быть достаточно полезен при программировании вложенных процедур модульного типа или использовании в процедурах вызовов процедур/функций, кортежи фактических аргументов которых содержат неопределенные переменные.

Наконец, вполне естественный интерес представляет вопрос наличия *активированных* в текущем сеансе процедур и функций пользователя. Решение данного вопроса можно получить вызовом процедуры, чей вызов **ActBFMuserQ[]** возвращает **True**, если такие объекты в текущем сеансе имеются, и **False** в противном случае; вызов **ActBFMuserQ[x]** через необязательный аргумент *x* – *неопределенную переменную* – возвращает *вложенный* список, чьи 2-элементные подписки содержат имена объектов в строчном формате и их типы соответственно. Фрагмент представляет исходный код наряду с примерами.

```
In[2070]:= ActBFMuserQ[x___/; If[{x} == {}, True, If[Length[{x}] == 1 && ! HowAct[x],
True, False]]] := Module[{b = {}, c = 1, d, h,
a = Select[Names["`*"], ProcQ[#] || FunctionQ[#] &]},
For[c, c <= Length[a], c++, ClearAll[d]; h = a[[c]];
If[ProcFuncTypeQ[h, d], b = Append[b, {h, d}]]];
If[b == {}, False, If[{x} != {}, x = If[Length[b] == 1, b[[1]], b]]; True]]

In[2071]:= V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2];
Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &;
GS[x_;/; IntegerQ[x], y_;/; IntegerQ[y]] := Sin[78] + Cos[42];
G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)]; P[x_, y_] := Module[{}, x + y]
H[x_] := Block[{}, x]; H[x_, y_] := x + y; SetAttributes["H", Protected]
P[x_] := Module[{}, x]; P[y_] := Module[{}, y]; P[x_] := Plus[Sequences[{x}]];
P[y_;/; PrimeQ[y]] := Module[{a = "Agn"}, y]; P[x___] := Plus[Sequences[{x}]];
P[x_;/; StringQ[x]] := Module[{}, x]; P[x_;/; ListQ[x]] := Module[{}, x];

In[2072]:= ActBFMuserQ[]
Out[2072]= True
In[2073]:= {ActBFMuserQ[t], t}
Out[2073]= {True, {"Art", "PureFunction"}, {"G", "CompiledFunction"}, {"GS", "Function"},
{"H", {"Block", "Function"}}, {"Kr", "PureFunction"}, {"P", {"Module", "Module",
"Module", "Module", "Module", "Function", "Function"}}, {"V", "CompiledFunction"}}
```

Данная процедура существенно использует **ProcFuncTypeQ**, рассмотренную выше, и представляется достаточно полезным средством в системном программировании.

## 8.4. Заголовки процедур и функций в среде пакета *Mathematica*

В целом ряде случаев нет необходимости знать точное значение выражения – вполне достаточно знать, что выражение принадлежит к некоторому широкому классу, либо группе выражений, которые объединяет несколько общих свойств. Эти классы либо группы известны как *типы*. Если *T* представляет тип, то выражение имеет тип *T*, если оно принадлежит классу, который представляет *T*. Например, некоторое выражение имеет тип *integer*, если оно принадлежит классу выражений, обозначенных типовым именем *integer*, который является множеством всех целых чисел. Многие процедуры используют механизм типов для управления вычислением или проверки выражений на предмет допустимости в качестве фактических аргументов процедур. Более того, результат ряда операций определяется типом их аргументов. *Тип* – фундаментальное понятие теории программирования, определяя допустимое множество значений или операций, которые можно применять к таким значениям и, возможно, также способ реализации хранения значений и выполнения операций. Любые объекты, которыми оперируют программы, относятся к определенным типам. Концепция типа данных в языках программирования высокого уровня появилась как совершенно естественное отражение того факта, что обрабатываемые программой данные и выражения могут иметь различные множества допустимых значений, храниться в оперативной памяти компьютера различным способом, обрабатываться разными командами процессора. При этом, тип произвольного объекта может быть определен 2 способами, а именно: множеством всех значений, принадлежащих данному типу, или некоей предикатной функцией, определяющей принадлежность объекта к данному типу. Преимущества от использования типов объектов сводятся к трем основным моментам: (1) защита от ошибок присваивания, некорректных операций, а также недопустимых фактических аргументов, передаваемых процедуре/функции; (2) стандартизация, обеспечиваемая соглашениями о типах, поддерживаемых большинством систем программирования, (3) документирование программных средств во многом упрощается использованием общепринятой *типизации* используемых в них объектов и данных.

В современных языках программирования существует несколько систем типизации, а исторически наиболее распространенной является *строгая* типизация. Данный тип в каждый момент использования языкового объекта предполагает, для него существует однозначное соответствие с его типом. Например, пожалуй, язык программирования *Ada* является единственным практически используемым языком *строгой* типизации, в то время как классический вариант языка *C* не является строго типизированным. При *строгой* типизации *несоответствия* типов выявляются еще до выполнения программы (*на этапе контроля соответствия типов в ходе трансляции*), что гарантирует отсутствие как семантических, так и логических ошибок и, в целом, определенную безопасность программного кода. Сильная типизация необходима для обеспечения корректности ассоциирования переменных с их значениями до начала выполнения программы. В теории программирования *строгая* типизация является наиболее важным элементом обеспечения робастности разрабатываемых программных средств. При правильном применении *строгая* типизация защищает программиста от достаточно простых, но,



порой, довольно трудно обнаруживаемых ошибок, которые обусловлены совместным использованием логически несовместимых значений. Подобные ошибки выявляются еще на этапе компиляции программы, тогда как в случае нестрогой типизации такие ошибки выявляются лишь при выполнении программы (*а в лучшем случае на стадии ее тестирования*), да и то зачастую не все и не сразу. При отсутствии сильной типизации язык программирования может быть определен как язык со *слабой* типизацией. Еще одним весьма важным *видом* типизации языка является *полиморфная* типизация, этот вид типизации допускает использование выражений переменного типа.

При этом, процедура контроля соответствия типов может быть реализована во время компиляции (*т.н. статическая типизация*) либо во время выполнения программы (*т.н. динамическая типизация*). Статическая типизация означает, что все проверки типов объектов выполняются на этапе компиляции, а не на этапе выполнения программы. Довольно большое число распространенных языков программирования используют *нестрогую* статическую типизацию, например, язык *Java*. В них обязательно описание типов переменных, аргументов и функций, однако допускается и *неявное* приведение типов: если значение одного типа присваивается переменной другого, то компилятор автоматически генерирует код для преобразования значения в нужный тип, но лишь если данное преобразование не приводит к потере данных. Проверка типов, которая выполняется на этапе компиляции, использует только сам код программы, ее важное преимущество состоит в том, что такую проверку достаточно выполнить только один раз. Если такая проверка установила, что типы объектов в программе удовлетворяют определенному условию, то это будет выполняться для всех ее последующих вызовов. Более того, отсутствие проверок, производимых на стадии выполнения программы, и определение *всех* типов на стадии компиляции позволяет создать скомпилированную программу существенно более эффективной. Итак, статическая типизация позволяет находить ошибки типизации на этапе компиляции, которые были допущены в редко используемых ветвях программ. Статическая типизация используется в большинстве языков программирования компилирующего типа. В свою очередь, сама статическая типизация является противоположностью *динамической* типизации объектов.

*Динамическая* типизация весьма широко применяется в языках программирования и языках спецификации, когда переменная ассоциируется с требуемым типом только в момент присвоения значения, а не в момент объявления переменной. Таким образом, в разных областях программы одноименная переменная может принимать значения разных типов. В качестве примера языков, использующих динамическую типизацию, можно привести такие, как *Perl, Lisp, JavaScript, Smalltalk, Python, Object Pascal*. При этом, большинство языков программирования с динамической типизацией являются *интерпретаторами*, а не *компиляторами*. Таким образом, если *статическая* типизация дает возможность уже на этапе компиляции выявить простые ошибки, тогда как при динамической типизации требуется как минимум выполнить данный участок кода. В первую очередь, в динамическом языке программирования особо коварны опечатки: программист может многократно просматривать исходный код программы, не найдя ничего предосудительного до тех пор, пока не сделает первый прогон программы на компьютере. Весьма существенным недостатком динамической типизации является и

относительно низкая скорость, связанная с динамической проверкой типов объектов. Учитывая важность типизации языковых объектов, данный аспект рассмотрен нами и относительно системы *Mathematica* в книгах [98-101]. В частности, в нашей книге [99] довольно детально с точки зрения развитости механизма *типизации* рассматриваются обе системы *Maple* и *Mathematica*, как наиболее развитые и популярные на сегодня универсальные системы компьютерной математики.

В отличие от 209 типов, например, пакета *Maple 11*, тестируемых процедурой *type* (не считая значительного набора пользовательских типов, подключенных к *Maple* посредством библиотеки [45]), *Mathematica 9* располагает только 60 тестирующими Q-функциями, имена которых имеют вид *Имя`Q*, например, *SyntaxQ["string"]* возвращает *True*, если содержимое строки является корректным *Mathematica*-выражением, и значение *False* в противном случае. В определенной мере к ней примыкает и функция *ToExpression*, вычисляющая все выражения, находящиеся в строке-аргументе, с возвратом *Null*. По результатам их выполнения обе эти функции вполне можно считать тестирующими корректность выражений, находящихся в строке-аргументе. При этом, если в первом случае мы получаем значение {*True*, *False*}, то во втором случае корректность можно ассоциировать с возвратом значения *Null*. В данном контексте функция *ToExpression* в определенном отношении аналогична процедуре *parse* системы *Maple* [99]. В случае необходимости пользователь может создавать и собственные функции типа *Имя`Q*, которые позволят существенно расширить набор аналогичных стандартных средств пакета, но непосредственно включить их в состав средств пакета не представляется возможным, учитывая отсутствие средств для создания пользовательских библиотек, логически сцепляемых со стандартными средствами пакета. Ниже данный вопрос в значительной мере детализируется на конкретных примерах подобных средств.

Кодирование определений типов непосредственно в заголовке процедур и функций возможно лишь для системы *Mathematica*, предоставляя возможность в точке вызова процедуры/функции, не выполняя ее и не обращаясь к внешним для нее средствам, проводить тестирование на допустимость получаемых ею фактических аргументов. Такой подход как повышает эффективность выполнения процедуры/функции, так и делает ее более мобильной. Данный подход особенно удобен в том случае, когда тип носит узкоспециальный характер либо его определение описано не очень большим и достаточно прозрачным программным кодом. В целом, данный подход позволяет во многих важных приложениях довольно детально типировать данные; его суть весьма наглядно иллюстрирует следующий простой фрагмент, а именно:

```
In[2124]:= ArtKr[x_ /; {T[z_] := If[z <= 460 && z >= 78, True, False], T[x]}[[2]],
           y_ /; StringQ[y] && ! SuffPref[y, {"avz", "agn", "vsv"}, 1] :=
           Module[{a = 71, b = 66}, y <> " = " <> ToString[x + a + b]]
In[2125]:= {T[6], Map7[ArtKr, Sequences, {{100, "Total"}, {42, "Total"}, {100, "avzagnvsv"}}]}
Out[2125]= {T[6], {"Total = 237", ArtKr[42, "Total"], ArtKr[100, "avzagnvsv"]}}
```

Пример простой процедуры *ArtKr* модульного типа довольно наглядно иллюстрирует возможности по организации типированного тестирования фактических аргументов процедуры, когда определение типа *T* задано непосредственно в заголовке процедуры

и активируется сразу же после первого вызова процедуры **ArtKr**. Ряд средств нашего пакета *AVZ\_Package\_1* [120] использует подобный подход в своей организации.

Исходя из общего определения процедуры, в частности, модульного типа

$M[x\_;/ Test_x, y\_;/ Test_y, \dots] := Module[\{locals\}, Body\ of\ Procedure]$

и того факта, что конкретное определение процедуры идентифицируется не *именем*, а *заголовком*, рассмотрим ряд полезных средств, обеспечивающих манипулирование с заголовками процедур/функций и играющими весьма важную роль в процедурном программировании и, прежде всего, программировании задач системного характера.

Определив такой достаточно полезный во многих приложениях объект, как *заголовок* процедуры/функции в виде конструкции вида «Имя[Список формальных аргументов]», довольно естественно возникает вопрос создания средств для тестирования объектов на предмет отношения их к типу *заголовка* (*heading*). В качестве такого средства можно привести процедуру **HeadingQ**, чей исходный код с примерами использования даны следующим фрагментом. Вызов процедуры **HeadingQ[x]** возвращает *True*, если объект *x*, заданный в строчном формате, может быть рассмотрен синтаксически корректным заголовком; в противном случае возвращается значение *False*; в случае недопустимого аргумента *x* вызов процедуры **HeadingQ[x]** возвращается невычисленным.

```
In[3385]:= HeadingQ[x_;/ StringQ[x]] := Module[{a, b, c, k = 1, m = True, n = True},
    If[StringTake[x, {-1, -1}] == "]" && StringCount[x, {"[", "]"}] == 2 &&
        ! StringFreeQ[StringReplace[x, " " -> "", {"[", "]"}, Return[m],
        If[! StringFreeQ[RedSymbStr[x, "_", "_"], "[_"], Return[! m]]];
        Quiet[Check[ToExpression[x], Return[False]]];
        If[DeleteDuplicates[Map3[StringFreeQ, x, {"[", "]"}]] == {False},
            c = StringPosition[x, "["][[1]][[2]];
            If[c == 1, Return[False], a = StringTake[x, {c, -1}], Return[False]];
            b = StringPosition[a, "["][[1]][[1]]; c = StringPosition[a, "]"][[1]][[1]];
            a = "{" <> StringTake[a, {b + 1, c - 1}] <> "}"; a = Map[ToString, ToExpression[a]];
            If[DeleteDuplicates[Mapp[StringFreeQ, a, "_"]] == {False}, Return[True]];
            If[{c, a} == {2, {}}, Return[True], If[a == {} | |
                StringTake[a[[1]], {1, 1}] == "_", Return[False],
                For[k, k <= Length[a], k++, b = a[[k]]; If[StringReplace[b, "_ " -> "" ] != "" &&
                    StringTake[b, {-1, -1}] == "_" | | ! StringFreeQ[b, "_ "] | |
                    ! StringFreeQ[b, "_."] | | ! StringFreeQ[b, "._"], m = True, n = False]]]; m && n]]
In[3386]:= {HeadingQ["D[x_, y_;/ ListQ[y], z_:75, h_]"], HeadingQ["D[x_, y_, z_:75, h_]"],
    HeadingQ["D[x_, y_;/ ListQ[y], z_:75, _]"]}
Out[3386]= {True, True, True}
In[3387]= {HeadingQ["D[x_, y_;/ ListQ[y], z_:75, h_]"], HeadingQ["[x_, y_;/ ListQ[y], z:75]"]}
Out[3387]= {False, False}
In[3388]:= {HeadingQ["g[]"], HeadingQ["t[x_]"], HeadingQ["p[x_]"], HeadingQ["h[_]"]}
Out[3388]= {True, True, True, False}
```

```
In[3389]:= {HeadingQ["D[_ , x_]"], HeadingQ["Z[x_]"], HeadingQ["Q[x_]"]}
Out[3389]= {True, True, True}
In[3390]:= {HeadingQ["D[x_, y_]; ListQ[y], z_:75, h]"], HeadingQ["V[x_, y_]; ListQ[y], z_."]]}
Out[3390]= {False, True}
```

Дополнительно к стандартным процедура использует наши процедуры **Map3**, **Mapr**, представляющие собой достаточно полезные обобщения стандартной функции **Map**, рассматриваемые ниже. **HeadingQ** достаточно полезна, например, при тестировании типов объектов, подобно использованию в определениях процедур/функций весьма полезных функций **Head1** и **Head2**, также рассматриваемых в настоящей книге.

Следующая процедура **HeadingQ1** представляет собой весьма полезное расширение рассмотренной процедуры **HeadingQ** относительно ее возможностей тестирования *заголовков* на их корректность. Вызов процедуры **HeadingQ1[x]** возвращает **True**, если фактический аргумент *x*, заданный в строчном формате, может быть рассмотрен как *синтаксически* корректный *заголовок*; иначе возвращается **False**. Следующий фрагмент представляет исходный код процедуры **HeadingQ1** с примерами ее использования.

```
In[2512]:= HeadingQ1[x_ /; StringQ[x]] := Module[{b, c = {}, d, h = "F", k = 1,
    a = Quiet[StringTake[x, {Flatten[StringPosition[x, "[", 1]]][1] + 1, -2]}],
    If[StringFreeQ[x, "["], False, b = StringSplit1[a, ","];
    For[k, k <= Length[b], k++, d = b[[k]]; c = Append[c, If[StringFreeQ[d, "_"], False,
    If[MemberQ[ToString/@ {Complex, Integer, List, Rational, Real, String, Symbol},
    StringTake[d, {Flatten[StringPosition[d, "_"]][1] + 1, -1]}], True,
    HeadingQ[h <> "[" <> d <> "]" ]]]]; If[DeleteDuplicates[c] == {True}, True, False]]]
In[2514]:= Map[HeadingQ1, {"H[s_String,x_;/StringQ[x],y_]", "T[x_,y_;/ListQ[y],z_List]",
    "V[x_, y_;/ ListQ[y]&&Length[L] == 78]", "E[x_, y_;/ ListQ[y], z_]"}]
Out[2514]= {True, True, True, True}
In[2515]:= {Map[HeadingQ, {"H[s_Integer]", "G[n_Integer,L_List]", "G[n__Integer]"}],
    Map[HeadingQ1, {"H[s_Integer]", "G[n_Integer,L_List]", "G[n__Integer]"}]}
Out[2515]= {{True, True, True}, {True, True, True}}
```

Дополнительно к стандартным **HeadingQ1** использует нашу процедуру **StringSplit1**, представляющую собой одно полезное обобщение стандартной функции **StringSplit**, рассматриваемую ниже. Следует отметить, что не взирая на *корректное* тестирование довольно широкого типа заголовков, процедуры **HeadingQ** и **HeadingQ1**, между тем, не носят всеобъемлющий характер по причине ряда особенностей синтаксического контроля *Math*-языка. В то же самое время данные процедуры оказываются во многих случаях достаточно полезными. Между тем, на основе нашей процедуры **ArgsTypes**, служащей для тестирования формальных аргументов функции/процедуры (*которая была активирована в текущем сеансе пакета*), рассматриваемой в книге ниже, возможно дальнейшее расширение тестирующих возможностей **HeadingQ1**, позволяя довольно существенно расширять типы корректно тестируемых *заголовков* процедур/функций. Следующий фрагмент представляет исходный код процедуры **HeadingQ2** с наиболее типичными примерами ее использования.

```

In[2942]:= HeadingQ2[x_ /; StringQ[x]] := Module[{b, c = {}, d, h = "F", k = 1,
      a = StringTake[x, {Flatten[StringPosition[x, "[", 1]]][[1]] + 1, -2}],
      Quiet[ToExpression["Clear[AvzRansIn]; AvzRansIn[" <> a <> "] := 75"]];
      If[Quiet[Select[ToExpression["ArgsTypes[AvzRansIn]"], ! StringFreeQ[#, "_"] &]] != {},
          ToExpression["Clear[AvzRansIn]"]; Return[False],
          ToExpression["Clear[AvzRansIn]"]; b = StringSplit1[a, ","];
          For[k, k <= Length[b], k++, d = b[[k]];
              c = Append[c, If[StringFreeQ[d, "_"], False, If[HeadingQ[h <> d <> "]"] | |
                  MemberQ[Map[ToString, {Complex, Integer, List, Rational, Real, String, Symbol}],
                      StringTake[d, {Flatten[StringPosition[d, "_"]][[-1]] + 1, -1]], True, False]]];
          If[DeleteDuplicates[c] == {True}, True, False]]

In[2943]:= Map8[HeadingQ1, HeadingQ2, {"V[x_/_String]"}]
Out[2943]= {True, False}
In[2944]:= Map8[HeadingQ1, HeadingQ2, {"V[x_/_StringQ[x]]"}]
Out[2944]= {True, True}
In[2945]:= F[x_/_StringQ[x]; y_/_IntegerQ[y]; z_/_ListQ[z]] := 78 + Length[z]
In[2946]:= F[{70, 65, 45, 24, 16, 50}]
Out[2946]= 84
In[2947]:= Map[HeadingQ2, {"F[x_/_StringQ[x]]", "F[x/_StringQ[x]]", "F[x; StringQ[x]]",
      "F[x_/_StringQ[x]]", "F[x_/_StringQ[x]]", "F[x_/_StringQ[x]]"}]
Out[2947]= {True, False, False, True, False, True}
In[2948]:= HeadingQ3[x_/_StringQ[x]] := Block[{a = "AvzRansIn", b}, Clear[AvzRansIn];
      b = Quiet[ToExpression[a <> StringTake[x,
          {Flatten[StringPosition[x, "[", 1]]][[1]], -1] <> "] := 78"]];
      If[SameQ[b, Null], Clear[a]; HeadingQ2[x], Clear[a]; False]]
In[2949]:= Map[HeadingQ3, {"F[x_/_StringQ[x]]", "F[x/_StringQ[x]]", "F[x; StringQ[x]]",
      "F[x_/_StringQ[x]]", "F[x_/_StringQ[x]]", "F[x_/_StringQ[x]]"}]
Out[2949]= {True, False, False, True, False, True}

```

Аналогично процедурам **HeadingQ[x]**, **HeadingQ1[x]** вызов процедуры **HeadingQ2[x]** возвращает **True**, если *фактический* аргумент *x*, заданный в строчном формате, может быть рассмотрен как синтаксически корректный *заголовок*; иначе возвращается **False**. Представленные во фрагменте примеры применения обоих процедур **HeadingQ1** и **HeadingQ2** наглядно иллюстрируют различие их функциональных возможностей. В завершение предыдущего фрагмента представлена процедура **HeadingQ3**, которая в функциональном отношении эквивалентна **HeadingQ2**, использует ее, однако в ряде случаев является более реактивной. Вызов процедуры **HeadingQ3[x]** возвращает **True**, если *фактический* аргумент *x*, заданный в строчном формате, может быть рассмотрен как синтаксически корректный *заголовок*; иначе вызов возвращает **False**. В то же время, между парами процедур {**HeadingQ[x]**, **HeadingQ1[x]**} и {**HeadingQ2[x]**, **HeadingQ3[x]**} имеются и принципиальные различия, в частности, на заголовке **F[x\_/\_StringQ[x]]** все четыре процедуры возвращают **True**, тогда как на заголовках **F[x\_/\_StringQ[x]]** вторая

пара возвращает **False**. Между тем, учитывая подобный и ряд других маловероятных форматов кодирования заголовков процедур/функций, представленные процедуры **HeadingQ[x]**, **HeadingQ1[x]**, **HeadingQ2[x]**, **HeadingQ3[x]** вполне можно рассматривать в качестве довольно полезных средств тестирования в модульном программировании. При этом, следует отметить, что в примерах предыдущего фрагмента используется и наша функция **Map8** из группы так называемых **Map**-средств [101], созданных нами в целях расширения часто используемой стандартной функции **Map**; данные средства рассматриваются и в настоящей книге в соответствующем разделе несколько ниже.

В качестве достаточно полезного средства при работе с процедурами и функциями в значительной степени отвечает процедура, чей вызов **HeadPF[x]** возвращает *заголовок* в *строчном* формате активированных в текущем сеансе блока, модуля либо функции с именем *x*, т.е. функции в традиционном ее понимании с заголовком. В то время как на остальных аргументах *x* вызов возвращается *невычисленным*. Между тем, проблема определения *заголовков* актуальна также для случая *одноименных* блоков, функций и модулей, которые имеют более одного заголовка. В данном случае вызов процедуры **HeadPF[x]** возвращает список заголовков в строчном формате составляющих объект *x* подобъектов. Следующий фрагмент представляет исходный код процедуры **HeadPF** наряду с наиболее типичными примерами ее использования.

```
In[2942]:= HeadPF[x_ /; BlockFuncModQ[x]] := Module[{b, c = ToString[x],
               a = Select[Flatten[{PureDefinition[x]}], ! SuffPref[#, "Default[" , 1] &]},
               b = Map[StringTake[#, {1, Flatten[StringPosition[#, {" := ", " = "}]][[1]] - 1] & , a];
               If[Length[b] == 1, b[[1]], b]]

In[2943]:= G[x_, y_] := x*Sin[y] + y*Cos[x]; s[] := 78*x; g := 460
In[2944]:= Map[HeadPF, {G, s, Sin, 2013, g}]
Out[2944]= {"G[x_, y_]", "s[]", HeadPF[Sin], HeadPF[2013], HeadPF[460]}
In[2945]:= Map[HeadPF, {If, Tan, Log, True, Tan, "Infinity", While, Do, InputForm}]
Out[2945]= {HeadPF[If], HeadPF[Tan], HeadPF[Log], HeadPF[True], HeadPF[Tan],
             HeadPF["Infinity"], HeadPF[While], HeadPF[Do], HeadPF[InputForm]}
In[2946]:= M[x_ /; x == "avzagn"] := Module[{a, b, c, x}; M[x_, y_, z_] := x + y + z;
               M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c, x};
               M[x_, y_] := Module[{a, b, c, "abc"; x + y}; M[x_String] := x;
Out[2946]= {"M[x_ /; x == \"avzagn\"] := Module[{a, b, c, x}], "M[x_, y_, z_] := x + y + z",
             "M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c, x}]",
             "M[x_, y_] := Module[{a, b, c, \"abc\"; x + y}", "M[x_String] := x"}]
```

Выше представлена процедура, чей вызов **HeadPF[x]** в строчном формате возвращает *заголовок* активизированного в текущем сеансе объекта с именем *x* типа блока, модуля либо функции. При этом, для объекта *x*, имеющего несколько различных заголовков, вызов **HeadPF[x]** возвращает *список* заголовков, порядок которых взаимно-однозначен порядку определений, возвращаемых вызовом функции **Definition[x]**. Тестирование объекта *x* на предмет быть *одноименным* представляется в данной связи достаточно актуальным и решает эту задачу процедура **QmultiplePF**, чей исходный код наряду с типичными примерами использования представляет нижеследующий фрагмент.

```

In[2080]:= QmultiplePF[x_, y_] := Module[{a = Flatten[{PureDefinition[x]}]},
                                         If[MemberQ[{"System"}, {$Failed}], a], False,
                                         If[{y} != {} && ! HowAct[y], y = If[Length[a] == 1, a[[1]], a]]; True]]

In[2081]:= M[x_ /; x == "avzagn"] := Module[{a, b, c}, x]; M[x_, y_, z_] := x + y + z;
                                         M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x];
                                         M[x_, y_] := Module[{a, b, c}, "abc"; x + y]; M[x_String] := x;
                                         M[x_List, y_] := Block[{a, b, c}, "abc"; Length[x] + y]

In[2082]:= QmultiplePF[M]
Out[2082]= True
In[2083]:= {QmultiplePF[M, s], s}
Out[2083]= {True, {"M[x_ /; x == \"avzagn\"] := Module[{a, b, c}, x]",
                    "M[x_, y_, z_] := x + y + z",
                    "M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]",
                    "M[x_List, y_] := Block[{a, b, c}, \"abc\"; Length[x] + y]",
                    "M[x_, y_] := Module[{a, b, c}, \"abc\"; x + y]",
                    "M[x_String] := x"}}}

In[2084]:= Map[QmultiplePF, {M, 78, avz, Sin, If, a + b}]
Out[2084]= {True, False, False, False, False, False}
    
```

Вызов процедуры **QmultiplePF[x]** возвращает **True**, если *x* – *одноименный* объект (блок, функция или модуль), и **False** в противном случае. Тогда как вызов **QmultiplePF[x, y]** со вторым необязательным аргументом – *независимой переменной y* – через нее возвращает список определений всех объектов с именем *x*. **QmultiplePF** существенно использует ранее рассмотренные процедуру **PureDefinition**, а также функцию **HowAct**. В целом ряде случаев процедура **QmultiplePF**, не взирая на относительную простоту, является достаточно удобным средством при тестировании объектов указанных типов.

При тестировании объектов на тип *Procedure, Function* часто возникает необходимость выделения среди них системных функций; эту задачу решает простая функция, чей вызов **SystemQ[x]** возвращает **True**, если *x*-объект является *системным*, т.е. определен *Math*-языком *Mathematica*, и **False** в противном случае. Функцию довольно несложно определить непосредственно на основе стандартной функции **Definition**. Следующий фрагмент представляет исходный код функции с примерами ее применения. В ряде приложений данная функция представляет вполне определенный интерес и, прежде всего, предоставляя возможность довольно эффективно дифференцировать средства.

```

In[2975]:= SystemQ[S_] := If[Off[Definition::ssle]; ! ToString[Definition[S]] == Null &&
                               SysFuncQ1[S], On[Definition::ssle]; True, On[Definition::ssle]; False]

In[2976]:= Map[SystemQ, {78, G, Sin, Do, While, False, ProcQ, "Infinity", M}]
Out[2976]= {False, False, True, True, True, True, False, True, False}
    
```

Прежде всего, функция **SystemQ** часто используется в заголовках процедур, тестируя фактические аргументы на допустимость. Дополнительно к функции **SystemQ** имеет смысл представить достаточно полезную функцию, вызов которой **LangHoldFuncQ[x]**

возвращает **True**, если  $x$  – базовая функция в *Math*-языке, и **False** в противном случае. Более того, под *базовой* понимается системная функция с одним из приписанных ей атрибутов, а именно: **HoldAll**, **HoldFirst**, **HoldRest**. Функция представляется довольно полезной при необходимости более четкой дифференциации программных средств. Следующий фрагмент представляет исходный код функции **LangHoldFunc** наряду с примерами ее наиболее типичного использования.

```
In[2198]:= LangHoldFuncQ[x_] := If[Intersection[Attributes[x], {HoldAll, HoldFirst,
HoldRest}] == {}, False, True]
In[2199]:= Map[LangHoldFuncQ, {If, Goto, Do, Sin, Rule, Break, While, Switch, Which}]
Out[2199]= {True, False, True, False, False, False, True, True, True}
```

Для ряда задач системного характера данная функция позволяет дифференцировать множество всех системных функций *Math*-языка по указанному признаку.

Тут же уместно отметить еще несколько средств, связанных с процедурой **HeadPF**. В частности, процедура **Headings** – полезное расширение процедуры **HeadPF** на случай *одноименных* блоков/функций/модулей с различными заголовками. Вызов **Headings[x]** в общем случае возвращает *вложенный* список, чьими элементами являются *подписки*, определяющие соответственно заголовки составляющих объект  $x$  подобъектов; *первый* элемент такого подписка определяет тип  $x$ -подобъекта, тогда как другие определяют соответствующие ему заголовки. Следующий фрагмент представляет исходный код процедуры **Headings** наряду с наиболее типичными примерами ее использования.

```
In[2185]:= Headings[x_ /; BlockFuncModQ[x]] := Module[{b, c = {"Block"}, {"Function"},
{"Module"}}, k = 1, a = Flatten[{PureDefinition[x]}], n = "$$$" <> ToString[x], d, h,
While[k <= Length[a], d = a[[k]]; ToExpression["$$$" <> d];
h = HeadPF[n]; h = StringReplace[h, "$$$" -> "", 1];
ClearAll[b]; BlockFuncModQ[n, b];
If[b == "Block", c[[1]] = Append[c[[1]], h],
If[b == "Function", c[[2]] = Append[c[[2]], h],
c[[3]] = Append[c[[3]], h]]];
ToExpression["ClearAll[" <> n <> "]; k++];
c = Select[c, Length[#] > 1 &]; If[Length[c] == 1, c[[1]], c]]
In[2186]:= M[x_ /; SameQ[x, "avz"], y_] := Module[{a, b, c}, y]; M[x_, y_, z_] := x + y + z;
M[x_ /; x == "avz"] := Module[{a, b, c}, x]; L1[x_, y_] := Block[{a, b, c}, x + y]
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x];
M[x_, y_] := Module[{a, b, c}, "agn"; x + y]; M[x_String] := x;
M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y]; L[x_] := x
In[2187]:= Headings[M]
Out[2187]= {"Block", "M[x_List, y_]", "M[x_ /; ListQ[x], y_]"}, {"Function", "M[x_, y_, z_]",
"M[x_String]", "M[x_]"}, {"Module", "M[x_ /; x == \"avzagn\"]",
"M[x_ /; IntegerQ[x], y_String]", "M[x_ /; x === \"avz\", y_]", "M[x_, y_]"},
"M[x_ /; x == \"avz\"]"}]
```



```
In[2188]:= V1[x_] = x; Map[Headings, {L, L1, 78, Sin, agn, V1}]
Out[2188]= {{Function, "L[x_]"}, {Block, "L1[x_, y_]"}, Headings[78], Headings[Sin],
            Headings[agn], {Function, "V1[x_]"}]}
In[2189]:= G[x_] := x; Headings[G]
Out[2189]= {Function, "G[x_]"}
In[2190]:= h = 78; P[x_] := Module[{a = 78, b = 460}, h = (a + b)*x; h^2]; {Headings[P], h}
Out[2190]= {{Module, "P[x_]"}, 78}
```

На аргументах  $x$ , отличных от блока/функции/модуля, вызов процедуры **Headings**[ $x$ ] возвращается невычисленным. Данное средство представляет интерес, прежде всего, с программистской точки зрения. В целом ряде приложений, широко использующих *процедурное* программирование, процедура **Headings** оказывается довольно полезной. Процедура дополнительно к стандартным функциям использует и наши процедуры **BlockFuncModQ**, **PureDefinition** и **HeadPF**. Примеры предыдущего фрагмента весьма наглядно иллюстрируют структуру возвращаемых процедурой результатов.

В целом ряде приложений, широко использующих процедурное программирование, достаточно полезным оказывается расширение **HeadingsPF** предыдущей процедуры. В общем случае вызов процедуры **HeadingsPF**[] возвращает *вложенный* список, чьими элементами являются подсписки, определяющие соответственно заголовки функций, блоков и модулей, определения которых были вычислены в текущем сеансе; первый элемент каждого такого *подсписка* определяет тип объекта в разрезе {"Block", "Module", "Function"}, тогда как остальные определяют заголовки, соответствующие ему. Вызов процедуры возвращает простой список, если какой-либо из подсписков не содержит заголовков; при этом, если в текущем сеансе системы не производилось вычислений определений объектов указанных трех типов, то вызов процедуры возвращает пустой список. Вызов процедуры с аргументами возвращается невычисленным. Следующий фрагмент представляет исходный код процедуры **HeadingsPF** наряду с примерами ее наиболее типичного использования.

```
In[2913]:= HeadingsPF[x___;/; SameQ[x, {}]] := Module[{b, d = t, p, k = 1,
    a = MinusList[Flatten[Names1[{{1 ;; 2}}], {"$ArtKr$", "HeadingsPF"}],
    c = {{Block}, {Function}, {Module}}}, b = Map[Headings, a];
    For[k, k <= Length[b], k++, p = b[[k]]; ClearAll[t]; Levels[p, t];
    If[t == 1, If[p[[1]] == "Module", c[[3]] = Join[c[[3]], p[[1 ;; -1]]],
    If[p[[1]] == "Block", c[[1]] = Join[c[[1]], p[[1 ;; -1]]],
    c[[2]] = Join[c[[2]], p[[1 ;; -1]]],
    p = Sort[p, OrderedQ[{#1[[1]], #2[[1]]}] == True &];
    c[[1]] = Join[c[[1]], p[[1]][[1 ;; -1]];
    c[[2]] = Join[c[[2]], p[[2]][[1 ;; -1]];
    c[[3]] = Join[c[[3]], p[[3]][[1 ;; -1]]];
    c = Select[Map[DeleteDuplicates, c], Length[#] > 1 &];
    ToExpression["t = " <> ToString[d]; If[Length[c] == 1, Flatten[c], c]]]
```

```
In[2914]:= M[x_ /; SameQ[x, "avz"], y_] := Module[{a, b, c}, y]; M[x_, y_, z_] := x + y + z;
M[x_ /; x == "avz"] := Module[{a, b, c}, x]; L[x_] := x; L[x_, y_] := x + y;
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x];
M[x_, y_] := Module[{a, b, c}, "agn"; x + y]; M[x_String] := x;
M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y];
F[x_ /; SameQ[x, "avz"], y_] := {x, y}; F[x_ /; x == "avz"] := x
```

```
In[2915]:= HeadingsPF[]
```

```
Out[2915]= {{ "Function", "M[x_, y_, z_]", "M[x_String]", "F[x_ /; x == \"avz\"]",
              "F[x_]", "F[x_ /; x === \"avz\"]", y_]", "L[x_]", "L[x_, y_]"],
              { "Procedure", "M[x_ /; x === \"avz\"]", y_]", "M[x_ /; x == \"avz\"]",
              "M[x_ /; IntegerQ[x], y_String]", "M[x_ /; ListQ[x], y_]", "M[x_, y_]"]}}
```

*Перезагрузка системы без активации пользовательских средств указанных 3 типов*

```
In[942]:= HeadingsPF[]
```

```
Out[942]= {}
```

```
In[943]:= V[x_, y_] := v + y; F[x_ /; StringQ[x]] := StringLength[x]; G[x_] := x; L[y_] := y
```

```
In[944]:= HeadingsPF[]
```

```
Out[944]= { "Function", "F[x_ /; StringQ[x]]", "G[x_]", "L[y_]", "V[x_, y_]"} 
```

Процедура дополнительно к стандартным функциям использует и наши процедуры **Headings**, **Levels**, **MinusList** и **Names1**, также рассматриваемые в данной монографии. Примеры предыдущего фрагмента весьма наглядно иллюстрируют сказанное. Более того, необходимо иметь ввиду, что реактивность процедуры *напрямую* зависит от того, на каком этапе текущего сеанса процедура была вызвана и сколько определений для средств типа блок/функция/модуль было вычислено в текущем сеансе.

В задачах обработки заголовков зачастую стоит вопрос вычисления *имени* заголовка, в решении которого помогает весьма простая функция, вызов **HeadName[x]** которой возвращает имя заголовка *x* в строчном формате при условии, что данный заголовок распознается процедурой **HeadingQ1** в качестве синтаксически корректного, т.е. на *x* вызов процедуры **HeadingQ1[x]** возвращает **True**. Следующий фрагмент представляет исходный код функции наряду с типичными примерами ее использования.

```
In[2045]:= HeadName[x_ /; HeadingQ[x] | | HeadingQ1[x]] :=
              StringTake[x, {1, StringPosition[x, "[", 1][[1]][[1]] - 1}]
```

```
In[2046]:= Map[HeadName, {"V[x_ /; StringQ[x]]", "G[x_String]", "S[x_ /; IntegerQ[x]]"}]
```

```
Out[2046]= {"V", "G", "S"}
```

```
In[2047]:= Map[HeadName, {"V[x_ /; StringQ[x]]", "G[x_ /; String]", "S[x_ /; IntegerQ[x]]"}]
```

```
Out[2047]= {HeadName["V[x_ /; StringQ[x]]"], "G", HeadName["S[x_ /; IntegerQ[x]]"]}
```

В ряде случаев процедурного программирования, например, в случае необходимости имплементирования вызовов процедур/функций на основе их заголовков в структуры строчного типа довольно полезной представляется процедура **HeadToCall**, чей вызов **HeadToCall[h]** в строчном формате возвращает вызов процедуры/функции на основе ее заголовка на «чистых» формальных аргументах (т.е. без приписанных им тестов на

допустимость), где *h* – допустимый заголовок процедуры/функции. Нижеследующий фрагмент представляет исходный код процедуры с примером ее использования.

```
In[2511]:= HeadToCall[h_/: HeadingQ[h]] := Module[{a = HeadName[h], b},
    b = "{" <> StringTake[StringReplace[h, a <> "[" -> "", 1], {1, -2}] <> "}";
    b = Select[StrToList[b], ! StringFreeQ[#, "_"] &];
    b = Map[StringTake[#, {1, Flatten[StringPosition[#, "_"][[1]] - 1]} &, b];
    a <> "[" <> StringTake[ToString[b], {2, -2}] <> "]"

In[2512]:= HeadToCall["G[x_, y_/: StringQ[y], z_/: MemberQ[{0, 1, 2}, z], t_Symbol]"
Out[2512]= "G[x, y, z, t]"
In[2513]:= HeadToCall["V[x_List, y_/: PrimeQ[y] && y < 78, z_/: !HowAct[z], t_Integer]"
Out[2513]= "V[x, y, z, t]"
```

В свете возможности существования в текущем сеансе пакета одноименных процедур с различными заголовками определенный интерес представляет задача удаления из сеанса процедуры с конкретным заголовком, решаемая процедурой **RemProcOnHead**.

```
In[2437]:= RemProcOnHead[x_/: StringQ[x]] := Module[{b, c},
    a = StringTake[x, {1, StringPosition[x, "["][[1]][[1]] - 1}], If[! MemberQ[Names["*"], a],
    Return[$Failed], b = StringSplit[StringReplace[ToString[InputForm[DefFunc[a]]],
    " " -> " ", "\n"]; c = Select[b, StringFreeQ[#, StringReplace[x, " " -> " "] <> "!="] &&
    # != " " &]; If[Length[b] == Length[c], $Failed, Quiet[ToExpression["Remove[" <>
    ToString1[a] <> ""]]; Quiet[Check[Map[ToExpression, c], Return[$Failed]]]; "Done"]

In[2438]:= G[x_] := Module[{a = 75}, x^2 + a]; G[x_/: PrimeQ[x]] := Module[{a = 75}, x + a];
    G[x_, y_] := Module[{}, x + y]; G[x_, y_/: ListQ[y], z_] := Module[{}, x + Length[y] + z];
    V[x_] := Module[{}, x^2]; V[x_/: ListQ[x]] := Module[{}, Length[x]]

In[2440]:= Definition[G]
Out[2440]= G[x_/: PrimeQ[x]] := Module[{a = 75}, x + a]
    G[x_] := Module[{a = 75}, x^2 + a]
    G[x_, y_/: ListQ[y], z_] := Module[{}, x + Length[y] + z]
    G[x_, y_] := Module[{}, x + y]

In[2441]:= Definition[V]
Out[2441]= V[x_/: ListQ[x]] := Module[{}, Length[x]]
    V[x_] := Module[{}, x^2]

In[2442]:= {RemProcOnHead["G[x_, y_]"], RemProcOnHead["V[x_]"]}
Out[2442]= {"Done", "Done"}

In[2443]:= Definition[G]
Out[2443]= G[x_/: PrimeQ[x]] := Module[{a = 75}, x + a]
    G[x_] := Module[{a = 75}, x^2 + a]
    G[x_, y_/: ListQ[y], z_] := Module[{}, x + Length[y] + z]

In[2444]:= Definition[V]
Out[2444]= V[x_/: ListQ[x]] := Module[{}, Length[x]]
```

Успешный вызов процедуры **RemProcOnHead[x]** возвращает значение **"Done"**, удалив из текущего сеанса пакета процедуру с заголовком **x**, заданном в строчном формате; в противном случае возвращается значение **\$Failed**. Приведенный фрагмент содержит исходный код процедуры с примерами ее использования и с контролем результатов.

Итак, если в *Maple* идентификатором процедуры либо функции является ее *имя*, то в *Mathematica* эту функцию выполняет ее *заголовок*, т.е. конструкция вида «Имя[список формальных аргументов]», что следует учитывать при программировании средств для обработки указанных объектов. Именно поэтому применять функцию **Names** нужно в сочетании с функцией **Definition**, ибо первая возвращает только имена процедур и ничего не говорит о существовании в текущем сеансе пакета одноименных процедур с разными заголовками, как иллюстрирует следующий простой фрагмент, а именно:

```
In[2620]:= G[x_] := Module[{a = 75}, x^2 + a]; G[x_]; PrimeQ[x] := Module[{a = 75}, x + a];
G[x_, y_] := Module[{x, y}, x + y];
G[x_, y_]; ListQ[y], z_] := Module[{x, y}, x + Length[y] + z];
V[x_] := Module[{x}, x^2];
V[x_]; ListQ[x] := Module[{x}, Length[x]]
In[2621]:= Select[Names["*"], Quiet[ProcQ[ToExpression[#]]] &]
Out[2621]= {"G", "V"}
In[2622]:= Definition[G]
Out[2622]= G[x_]; PrimeQ[x] := Module[{a = 75}, x + a]
G[x_] := Module[{a = 75}, x^2 + a]
G[x_, y_] := Module[{x, y}, x + y]
G[x_, y_]; ListQ[y], z_] := Module[{x, y}, x + Length[y] + z]
In[1623]:= Definition[V]
Out[1623]= V[x_]; ListQ[x] := Module[{x}, Length[x]]
V[x_] := Module[{x}, x^2]
In[2064]:= MdP[x___] := Module[{a, b}, a[y_] := If[! BlockFuncModQ[y], $Failed,
{y, Length[Flatten[{PureDefinition[y]}]}]];
If[Length[Flatten[{x}]] >= 1, b = Map[a, Flatten[{x}]],
If[{x} == {}, b = Map[a, Flatten[Names1[[{1 ;; 2}]]]];
If[b == {$Failed}, b[[1]], If[Length[b] == 1, Flatten[b], b]]]
In[2065]:= MdP[]
Out[2065]= {{G, 4}, {MdP1, 1}, {V, 2}}
In[2066]:= MdP[G]
Out[2066]= {G, 4}
In[2067]:= MdP[{G, V}]
Out[2067]= {{G, 4}, {V, 2}}
In[2068]:= MdP[{G, V, Z}]
Out[2068]= {{G, 4}, {V, 2}, $Failed}
In[2069]:= MdP[]
Out[2069]= {"LocalsGlobals1", 1}, {"G", 5}, {"GS", 1}, {"H", 2}, {"H1", 1}, {"P", 7}, {"V", 3}]
```

Завершает данный фрагмент процедура, вызов **MdP[x]** которой возвращает простой 2-элементный список (если *x* определяет активированную в текущем сеансе функцию, блок или модуль; при отсутствии подобного объекта возвращается **\$Failed**), вложенный список, 2-элементные подсписки которого имеют структуру, описанную выше (если объект *x* определяет список блоков/функций/модулей, активированных в текущем сеансе системы), указанного выше вида вложенный список (если *x* пуст, определяя список всех функций/блоков/модулей, активированных в текущем сеансе); при отсутствии функций/модулей или блоков, активированных в текущем сеансе, вызов **MdP[]** возвращает пустой список, т.е. {}. Процедура **MdP** использует ранее упоминавшиеся наши средства **BlockFuncModQ**, **PureDefinition**, **Names1**. Примеры, приведенные в предыдущем фрагменте, довольно наглядно иллюстрируют вышесказанное.

Вызов вышерассмотренной процедуры **RemProcOnHead[x]** вполне корректно удаляет из текущего сеанса единственную процедуру/функцию с заголовком *x* при условии отсутствия для имени процедуры **Protected**-атрибута, в противном случае вызывая **\$Failed**. Тогда как вызов соответствующей модификации **RemProcOnHead1[x]** вполне корректно удаляет из текущего сеанса отдельную процедуру/функцию с заголовком *x* либо процедуры/функции с заголовками, определяемыми списком *x*, независимо от наличия у объекта с именем **HeadName[x]** атрибутов. С такой целью предварительно производится сохранение атрибутов с последующими их отменой и восстановлением после выполнения удаления из текущего сеанса процедур/функций с заголовками *x*, определения которых соответствуют заданным заголовкам *x*. Следующий фрагмент представляет исходный код процедуры с типичными примерами ее использования.

```
In[2738]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[2739]:= RemProcOnHead["M[x_, y_, z_]"]
Out[2739]= "Done"
In[2740]:= Definition[M]
Out[2740]= M[x_, y_] := Module[{}, {x}
           M[x_] := Module[{}, {x}]
In[2741]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[2742]:= SetAttributes[M, {Flat, Protected}]
In[2743]:= RemProcOnHead["M[x_, y_, z_]"]
Out[2743]= $Failed
In[2744]:= Definition[M]
Out[2744]= Attributes[M] = {Flat, Protected}
           M[x_, y_] := Module[{}, {x}]
           M[x_, y_, z_] := x + y + z
           M[x_] := Module[{}, {x}]
In[2278]:= G[x_] := Module[{a = 78}, x^2 + a];
           G[x_;/ PrimeQ[x]] := Module[{a = 78}, x + a];
           G[x_, y_] := Module[{}, x + y];
           G[x_, y_;/ ListQ[y], z_] := Module[{}, x + Length[y] + z];
           V[x_] := Module[{}, x^2]; V[x_;/ ListQ[x]] := Module[{}, Length[x]]
In[2281]:= SetAttributes[G, Protected]
```

```

In[2283]:= RemProcOnHead1[x_ /; HeadingQ[x] || ListQ[x] &&
DeleteDuplicates[Map[HeadingQ, x]] == {True}] :=
Module[{a = HeadName[Flatten[{x}][[1]]], b, c},
c = Flatten[{PureDefinition[a]}];
If[c === {$Failed}, Print["Symbol <" < a < "> is undefined."]; "Done",
b = ToExpression["Attributes[" < a < "]""];
c = Select[c, ! SuffPref[#, Flatten[{x}], 1] &];
ToExpression["ClearAllAttributes[" < a < "]""];
ToExpression["ClearAll[" < a < "]""];
If[c != {}, ToExpression[c];
ToExpression["SetAttributes[" < a < " , " < ToString[b] < "]""]; "Done"]]

In[2284]:= RemProcOnHead1["G[x_]"]
Out[2284]= "Done"
In[2285]:= Definition[G]
Out[2285]= Attributes[G] = {Protected}
G[x_ /; PrimeQ[x]] := Module[{a = 75}, x + a]
G[x_, y_] := Module[{}, x + y]
G[x_, y_ /; ListQ[y], z_] := Module[{}, x + Length[y] + z]
In[2286]:= RemProcOnHead1[{"G[x_, y_]", "G[x_ /; PrimeQ[x]]"}]
Out[2286]= "Done"
In[2287]:= Definition[G]
Out[2287]= Attributes[G] = {Protected}
G[x_, y_ /; ListQ[y], z_] := Module[{}, x + Length[y] + z]
In[2288]:= RemProcOnHead1[{"V[x_]", "V[x_ /; ListQ[x]]"}]
Out[2288]= "Done"
In[2289]:= Definition[V]
Out[2289]= Null
In[2290]:= RemProcOnHead1["Z[x_]"]
Symbol <Z> is undefined.
Out[2290]= "Done"

```

Прежде всего, определение процедуры **RemProcOnHead1** наряду с функциями *Math*-языка использует и наши средства **ClearAllAttributes**, **HeadingQ**, **HeadName**, **SuffPref**, **PureDefinition**, также рассматриваемые в настоящей книге. При этом, в определении процедуры использован искусственных ряд приемов, существенных в практическом программировании. Во-первых, прямое применение нашей и стандартных функций {**Attributes**, **ClearAllAttributes**, **SetAttributes**} обработки атрибутов не дает ожидаемого результата, поэтому приходится применять специальные конструкции, организация которых достаточно прозрачна и особых пояснений не требует. Во-вторых, произвол кодирования *заголовков* при вызове **RemProcOnHead1** в определенной мере ограничен и должен соответствовать формату заголовков в определении блока/функции/модуля

$x$ , получаемым в результате вызова процедуры **Definition2**[ $x$ ]. Именно лишь в случае соблюдения этого условия обеспечивается *корректное* выполнение вызова процедуры **RemProcOnHead1**. Результат корректного вызова возвращает **"Done"**, удаляя объекты с заданными заголовками из текущего сеанса, тогда как вызов процедуры на *заголовке неактивного* объекта тоже возвращает **"Done"** с выводом соответствующего сообщения. При этом, оставшиеся подобъекты с именем обрабатываемого объекта  $x$  сохраняют и опции, и атрибуты, исключая случай, когда объект  $x$  удаляется полностью.

Как уже отмечалось, текущий сеанс может содержать несколько разных определений процедуры/функции с одним и тем же именем, которые различаются лишь на уровне их заголовков. Вызов предыдущей процедуры **Definition2**[ $x$ ] в оптимальном формате возвращает список всех определений в строчном формате процедуры/функции под именем  $x$ , сопровождая его опциями и списком атрибутов приписанных символу  $x$ . В соответствии с соглашениями пакета одноименные процедуры/функции имеют одни и те же как опции, так и атрибуты, как иллюстрирует нижеследующий фрагмент.

```
In[2089]:= G[x_, y_] := x^2 + y^2
In[2090]:= Options[G] = {Art -> 24, Kr -> 17}
Out[2091]= {Art -> 24, Kr -> 17}
In[2092]:= SetOptions[G, Art -> 24, Kr -> 17]
Out[2092]= {Art -> 24, Kr -> 17}
In[2093]:= Definition2[G]
Out[2093]= {"G[x_, y_] := x^2 + y^2", "Options[G] := {Art -> 24, Kr -> 17}", {}}
In[2094]:= G[x_] := x^2; G[x_, y_, z_] := x + y + z; SetAttributes[G, {Protected, Listable}]
In[2095]:= Definition2[G]
Out[2095]= {"G[x_, y_] := x^2 + y^2", "G[x_] := x^2", "G[x_, y_, z_] := x + y + z",
            "Options[G] := {Art -> 24, Kr -> 17}", {Listable, Protected}}

In[2096]:= DefOnHead[x_;/; HeadingQ[x]] := Module[{a, b, c, d,
            h = RedSymbStr[StringReplace[x, "-" -> ".", " ", " ", " "],
            a = HeadName[h]; b = Definition2[ToExpression[a]];
            c = Select[b, SuffPref[#, Map3[StringJoin, h, {" := ", " = "}], 1] &];
            d = Select[b, SuffPref[#, Quiet[Map3[StringJoin,
            "Options[" <> a <> "]", {" = ", " := "}], 1] &];
            If[MemberQ[b, "Undefined"], $Failed,
            If[d == {}, AppendTo[c, b[[-1]]], Join[c, {d[[1]], b[[-1]]}]]]]

In[2097]:= DefOnHead["G[x_, y_, z_]"]
Out[2097]= {"G[x_, y_, z_] := x + y + z", "Options[G] := {Art -> 24, Kr -> 17}", {Listable,
            Protected}}

In[2098]:= DefOnHead["G[x_, y_]"]
Out[2098]= {"G[x_, y_] := x^2 + y^2", "Options[G] := {Art -> 24, Kr -> 17}", {Listable,
            Protected}}

In[2099]:= DefOnHead["G[x_]"]
Out[2099]= {"G[x_] := x^2", "Options[G] := {Art -> 24, Kr -> 17}", {Listable, Protected}}
```

Для получения определения процедуры/функции  $x$  с заданным заголовком (*основным идентификатором*) создан ряд средств, одно из которых и представлено предыдущим фрагментом, а точнее процедурой **DefOnHead**, чей вызов **DefOnHead[h]** возвращает список, первый элемент которого – определение в строчном формате процедуры или функции с заданным заголовком  $h$  (либо список определений для одноименных объектов), тогда как остальными элементами являются *опции* (если они есть) и список атрибутов, приписанных данной процедуре либо функции  $x$ . При этом, имеет место следующее *определяющее* соотношение **HeadName[h] = x**. В целом же, рекомендуется использовать уникальное имя для каждого определения, для обеспечения чего можно использовать функции **Clear**, **ClearAll** при модификациях средств, если изменяются их заголовки.

Таким образом, при вызове процедуры/функции из списка определений выбирается то, которое имеет заголовок, соответствующий фактическим аргументам, т.е. которые оказываются допустимыми для формальных аргументов с приписанными им тестами на допустимость. Более того, заголовок формата **G[x\_, y\_, z\_, ...]** имеет минимальный приоритет среди заголовков других форматов независимо от порядка вычисления в текущем сеансе определений одноименных процедур/функций, как весьма наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[2863]:= G[x_, y_] := StringJoin[x, y] <> "RansIan"
In[2864]:= G[x_Integer, y_Integer] := x + y
In[2865]:= G[x_String, y_Integer] := y*StringLength[x]
In[2866]:= Definition2[G]
Out[2866]= {"G[x_Integer, y_Integer] := x + y", "G[x_String, y_Integer] :=
           y*StringLength[x]", "G[x_, y_] := StringJoin[StringJoin[x, y], \"RansIan\"]", {}}
In[2867]:= {G[70, 78], G["AvzAgnVsvArtKr", 460]}
Out[2867]= {148, 6440}
In[2868]:= G["AvzAgnVsvArtKr", "Tallinn"]
Out[2868]= "AvzAgnVsvArtKrTallinnRansIan"
In[2908]:= G[x_, y_] := If[{y} == {}, x^2, {y} = {x}; x^2]
In[2909]:= G[460]
Out[2909]= 211 600
In[2910]:= {G[78, t], t}
Out[2910]= {6084, 78}
In[2911]:= ClearAll[G, t]
In[2912]:= G[x_] := x^2; G[x_, y_ /; ! HowAct[y] === Null] := {y = x, x^2}[[2]]
In[2913]:= Definition2[G]
Out[2913]= {"G[x_] := x^2", "G[x_, y_ /; !HowAct[y] === Null] := {y = x, x^2}[[2]]", {}}
```

Выше уже отмечалось, что в большинстве случаев целесообразно использовать только одно определение процедуры/функции, что, порой, довольно существенно упрощает его обработку. Между тем, в некоторых случаях вполне уместно использование ряда одноименных процедур/функций, в частности, с целью упрощения их программной реализации. В качестве примера может служить реализация функции **G** второй части



предыдущего фрагмента от неопределенного числа аргументов. Определение двух  $G$  функций, охватывающих все случаи функции  $G$ , в ряде случаев позволяет упростить реализацию. В приведенном примере не столь очевидно такое *упрощение*, т.к. он лишь иллюстрирует сам прием, тогда как в случае достаточно сложных процедур, которые в своем теле должны выполнять обработку получаемых фактических аргументов при неопределенном их количестве, такой подход может оказаться весьма эффективным.

Как отмечалось выше, в общем случае процедура/функция пользователя может иметь как приписанные ей атрибуты, так и опции. При этом, ранее рассмотренные средства базировались, в основном, на нашей процедуре **Definition2** $[x]$ , возвращающей список, последний элемент которого содержит список атрибутов, приписанных  $x$ , тогда как в подписке **Definition2** $[x][[1;;-2]]$  находятся определения процедуры/функции  $x$  вместе с опциями, если таковые имеются. Нижеследующая процедура **PureDefinition** решает задачу получения чистых определений без опций и приписанных им атрибутов.

```
In[2126]:= G[x_] := x^2; G[x_, y_ /; ! HowAct[y] === Null] := {y = x, x^2}[[2]]
In[2127]:= Options[G] = {Art -> 24, Kr -> 17}; SetOptions[G, Art -> 24, Kr -> 17]
Out[2127]= {Art -> 24, Kr -> 17}
In[2128]:= SetAttributes[G, {Listable, Protected}]; Definition2[G]
Out[2128]= {"G[x_] := x^2", "G[x_, y_ /; !HowAct[y] === Null] := {y = x, x^2}[[2]]",
  "Options[G] := {Art -> 24, Kr -> 17}", {Listable, Protected}}
In[2134]:= PureDefinition[x_, t___] := Module[{b, c, d, h = ToString[x] <> "/: Default["},
  a = If[UnevaluatedQ[Definition2, x], $Failed, Definition2[x]],
  If[a === $Failed, Return[$Failed]]; b = a[[1 ;; -2]];
  c = If[SuffPref[b[[-1]], Map3[StringJoin,
  "Options[" <> ToString[x] <> "]", {" = ", " := "}], 1], b[[1 ;; -2]], b];
  If[{t} != {} && ! HowAct[t], d = MinusList[a, c];
  t = Join[If[Length[d] > 1, d, Flatten[d]], Select[a, SuffPref[#, h, 1] &]];
  c = Select[c, ! SuffPref[#, h, 1] &]; If[Length[c] == 1, c[[1]], c]]
In[2135]:= {PureDefinition[G, t], t}
Out[2135]= {"G[x_] := x^2", "G[x_, y_ /; !HowAct[y] === Null] := {y = x, x^2}[[2]]",
  "Options[G] := {Art -> 24, Kr -> 17}", {}}
```

Вызов процедуры **PureDefinition** $[x]$  возвращает определение в строчном формате или их список блока/функции/модуля  $x$  без опций, приписанных атрибутов и значений по умолчанию для формальных аргументов, в то время как вызов **PureDefinition** $[x, t]$  со вторым необязательным аргументом  $t$  – неопределенной переменной – возвращает через него список опций, атрибутов и значений по умолчанию, приписанных  $x$ . При недопустимом аргументе  $x$  вызов процедуры возвращает значение **\$Failed**, включая и вызов на функциях **Compile**. Фрагмент выше представляет исходный код процедуры с типичными примерами ее применения. Процедура **PureDefinition** представляется нам довольно полезной в различного рода обработках определений блоков, функций и модулей. Процедура широко используется средствами пакета *AVZ\_Package\_1* [120].

Концепция *Mathematica* допускает существование нескольких *одноименных* процедур или функций, идентифицируемых их заголовками, а не именами. Работа с данными объектами поддерживается рядом средств, представленных в данной книге. В данной связи определенный интерес представляет процедура, вызов которой **ExtrProcFunc[h]** возвращает уникальное *имя* сгенерированной процедуры/функции, которая в списке определений имеет *заголовок* **h**. В противном случае возвращается **\$Failed**. Процедура характерна тем, что оставляет без изменения все определения символа **HeadName[h]**. Реализация процедуры существенно использует процедуру, чей вызов **StandHead[h]** возвращает *заголовок* процедуры/функции в формате, соответствующем соглашениям пакета при вычислении ее *определения*. Следующий фрагмент представляет исходные коды процедур **StandHead** и **ExtrProcFunc** с типичными примерами их применения.

```
In[2058]:= StandHead[x_;/; HeadingQ[x]] := Module[{a = HeadName[x], b},
                                         b = StringReplace[x, a <> "[" -> "", 1];
                                         b = ToString[ToExpression["{" <> StringTake[b, {1, -2}] <> "}"];
                                         a <> "[" <> StringTake[b, {2, -2}] <> "]"

In[2059]:= StandHead["V[x_, y_Integer, z_;/;StringQ[z]]"]
Out[2059]= "V[x_, y_Integer, z_;/;StringQ[z]]"

In[2060]:= ExtrProcFunc[x_;/; HeadingQ[x]] := Module[{a = StandHead[x], c, d,
                                         b = HeadName[x], c = Definition2[ToExpression[b]];
                                         If[c[[1]] == "Undefined", $Failed, d = Select[c, SuffPref[#, a <> " := ", 1] &];
                                         c = ToString[Unique[b]]; If[d != {}, ToExpression[c <> d[[1]]]; c <> b, $Failed]]

In[2061]:= H[x_] := x^2; H[x_, y_] := x + y; H[x_, y_, z_] := x + y + x; H[x_Integer] := x;
                                         H[x_, y_Integer] := x + y; H[x_String] := x <> "RansIan"

In[2062]:= Definition2[H]
Out[2062]= {"H[x_Integer] := x", "H[x_String] := StringJoin[x, \"RansIan\"], "H[x_] := x^2",
            "H[x_, y_Integer] := x + y", "H[x_, y_] := x + y", "H[x_, y_, z_] := x + y + x", {}}

In[2063]:= ExtrProcFunc["H[x_, y_, z_]"]
Out[2063]= "H3H"

In[2064]:= Definition["H3H"]
Out[2064]= H3H[x_, y_, z_] := x + y + x

In[2065]:= ExtrProcFunc["H[x_, y_, z_String]"]
Out[2065]= $Failed

In[2066]:= ExtrProcFunc["H[x_String]"]
Out[2066]= "H5H"

In[2067]:= Definition["H5H"]
Out[2067]= H5H[x_String] := x <> "RansIan"

In[2068]:= H5H["AvzAgnVsvArtKr"]
Out[2068]= "AvzAgnVsvArtKrRansIan"

In[2069]:= H3H[42, 2013, 71]
Out[2069]= 2097
```

## 8.5. Формальные аргументы процедур и функций, и средства манипулирования ими в среде системы *Mathematica*

Рассмотрев в предыдущих двух разделах средства манипулирования определениями блоков/функций/модулей, а также их заголовков, переходим к рассмотрению средств, в сферу интересов которых входит ряд важных задач, связанных с манипулированием *формальными аргументами*, составляющими заголовки определений как процедур, так и функций пользователя. Данные компоненты являются чрезвычайно важными и их полное отсутствие в заголовках не позволяет системе, в целом, рассматривать объекты с подобными заголовками в качестве процедур или функций. В предыдущем разделе нами рассмотрены средства обработки заголовков процедур и функций, из которых наша процедура **HeadingQ1** наилучшим образом тестирует строку на предмет быть в качестве заголовка, что весьма наглядно иллюстрирует следующий простой пример:

```
In[2046]:= Map[HeadingQ1, {"G[]", "G[ ]", "G[ ]"}]
Out[2046]= {False, False, False}
In[2047]:= G[] := x; {FunctionQ[G], Clear[G]}[[1]]
Out[2047]= False
In[2048]:= G[x_]; SameQ[{x}, {}] := x; FunctionQ[G]
Out[2048]= True
In[2049]:= HeadingQ["G[x_]; SameQ[{x}, {}]"]
Out[2049]= True
```

Из приведенного примера следует, что строки типа "G[]" не могут рассматриваться в качестве синтаксически корректных заголовков и определения на их основе не могут рассматриваться процедурами либо функциями. Между тем, в случае необходимости определять процедуры либо функции, вызовы которых имеют смысл лишь на пустом списке фактических аргументов, можно кодировать их заголовки, как указано выше; в этом случае наши средства идентифицируют их надлежащим образом. Дальнейшее рассмотрение средств манипулирования с формальными аргументами процедур или функций предполагает краткое введение в понятие *шаблонов*; более детально данный вопрос рассматривается в справке по системе и, в частности, в нашей книге [101].

**Шаблоны** используются в среде *Mathematica* для представления классов выражений. Весьма простым примером шаблона является выражение *h[x\_]*, представляющее класс выражений типа *h[Любое выражение]*. Предпосылкой введения понятия «**Шаблон**» в среду *Mathematica* послужил тот факт, что довольно много операций поддерживают работу не только с отдельными выражениями, но и с шаблонами, представляющими целые классы выражений. Так, в частности, можно использовать шаблоны в правилах преобразования для указания того, как следует преобразовывать классы выражений. Шаблоны можно использовать для вычисления позиций *всех* выражений в некотором определенном классе и целого ряда других приложений механизма шаблонов.

Базовым идентификатором, определяющим, практически, все *шаблоны* в *Mathematica* является приписанный некоторому символу справа символ «*\_*» (*символ подчеркивания*). В этом случае *Mathematica* рассматривает такой символ как допускающий в качестве

значения для него произвольное допустимое пакетом выражение. Ранее упомянутая функция **Head[x]** на шаблоне  $x$  возвращает значение **Pattern**, тогда как весьма простая функция **PatternQ[x]** возвращает **True**, если  $x$  – шаблон, и **False** в противном случае:

```
In[2689]:= PatternQ[x_] := If[Head[x] === Pattern, True, False]
```

```
In[2690]:= {Map[PatternQ, {agn_, _, a_, x_, ^_}], Map[Head, {agn_, _, a_, x_, ^_}]}
```

```
Out[2690]= {{True, False, False, True, False}, {Pattern, Blank, Times, Pattern, Power}}
```

Примеры применения функции **PatternQ** с ее исходным кодом представлены выше; в частности,  $x_$  определяет допустимость для  $x$  любого выражения. В принципе, символ « $_$ » возможно помещать в любое место выражения, определяя таким образом шаблон, соответствующий некоторой группе выражений, получаемых путем замены данного символа произвольным выражением. Ниже представлены некоторые весьма простые примеры шаблонов, а именно:

**h[x\_]** – заголовок функции/процедуры с одним формальным аргументом  $x$ , в качестве которого допустимо произвольное выражение;

**h[x\_, y\_]** – заголовок функции/процедуры  $h$  с двумя формальными аргументами  $x$  и  $y$ , в качестве которых допустимы произвольные выражения;

**h[x\_, x\_]** – заголовок функции/процедуры  $h$  с двумя идентичными аргументами  $x$ , в качестве которых допустимы произвольные выражения;

$x^n_$  – определяет выражение  $x$  в произвольной степени  $n$ ;

$x^{}_n$  – определяет произвольное выражение  $x$  в произвольной степени  $n$ ;

$x_ + y_ + z_$  – определяет сумму трех произвольных выражений  $x$ ,  $y$  и  $z$ ;

$\{x_, y_, z_\}$  – определяет список из трех произвольных выражений  $x$ ,  $y$  и  $z$ ;

$78 x^{}_y_ + 460 x_*y_ + z_$  – определяет выражение с пятью шаблонами.

**Базовыми** шаблонами в среде *Mathematica* являются три следующих, а именно:

$_$  или **Blank[]** (*в полной форме*) – шаблон, определяющий произвольное выражение;

$_t$  или **Blank[t]** (*в полной форме*) – шаблон, определяющий произвольное выражение с заголовком  $t$ ;

$__$  (2 символа " $_$ ") или **BlankSequence[]** (*в полной форме*) – шаблон, который определяет одно произвольное выражение либо последовательность произвольных выражений;

$__t$  либо **BlankSequence[t]** (*в полной форме*) – шаблон, который определяет одно либо последовательность произвольных выражений, каждое из которых имеет заголовок  $t$ ;

$___$  (3 символа " $_$ ") или **BlankNullSequence[]** (*в полной форме*) – шаблон, определяющий отсутствие выражений либо последовательность произвольных выражений;

$___t$  либо **BlankNullSequence[t]** (*в полной форме*) – шаблон, определяющий отсутствие выражений либо последовательность произвольных выражений с заголовками  $t$ .

При этом, в полной форме выражений, содержащих шаблоны типов {« $_$ », « $__$ », « $___$ »}, они представляются в форматах, иллюстрируемых следующим простым примером:

```
In[99]:= Map[FullForm, {x_, x__, x___}]
```

```
Out[99]= {Pattern[x, Blank[]], Pattern[x, BlankSequence[]], Pattern[x, BlankNullSequence[]]}
```

Достаточно простая функция **ExprPatternQ** обеспечивает тестирование выражения на

предмет наличия в нем шаблонов типов {«\_», «\_\_», «\_\_\_»}, возвращая при своем вызове **ExprPatternQ**[*x*] значение **True**, если выражение *x* содержит по меньшей мере один из шаблонов {«\_», «\_\_», «\_\_\_»}, и **False** в противном случае. Фрагмент приводит исходный код функции **ExprPatternQ** наряду с типичными примерами ее использования:

```
In[2279]:= ExprPatternQ[x_] := !StringFreeQ[ToString[FullForm[x]], {"BlankSequence[]",  
                                                                    "BlankNullSequence[]", "Blank[]"}]
```

```
In[2280]:= Map[ExprPatternQ, {a + Sin[x], 78 x_^y_ + 460 x_*y_, x_^y_, x_, z_}]
```

```
Out[2280]= {False, True, True, True, True}
```

Возможно создавать шаблоны для выражений с любой структурой, однако наиболее распространенным способом использования шаблонов является именно определение функции/процедуры, когда в ее заголовке указываются формальные аргументы. При этом, задание формальных аргументов без шаблонов не позволяет рассматривать эти объекты в качестве функций либо процедур, как иллюстрирует простой пример:

```
In[2490]:= G[x, y] := x^2 + y^2; G1[x_, y_] := x^2 + y^2;
```

```
In[2491]:= {G[78, 460], G1[78, 460]}
```

```
Out[2491]= {G[78, 460], 217684}
```

Еще раз следует подчеркнуть, что шаблоны в среде *Mathematica* представляют собой классы выражений с заданной структурой, когда один шаблон соответствует некоему выражению и если структура шаблона совпадет со структурой выражения, т.е. путем заполнения шаблонов можно получить выражение. Более того, даже два выражения, математически эквивалентные, не могут быть представлены тем же шаблоном, если у них нет той же самой структуры. Так,  $(a + b)^2$  эквивалентно  $a^2 + 2*a*b + b^2$ , однако выражения не эквивалентны на уровне шаблонов, их представляющих, по причине, что оба имеют различную полную форму, как иллюстрирует простой пример:

```
In[2131]:= FullForm[(a + b)^2]
```

```
Out[2131]/FullForm =
```

```
Power[Plus[a, b], 2]
```

```
In[2132]:= FullForm[a^2 + 2*a*b + b^2]
```

```
Out[2132]/FullForm =
```

```
Plus[Power[a, 2], Times[2, a, b], Power[b, 2]]
```

Тот факт, что шаблоны определяют структуру выражений, очень важен для решения задачи определения правил преобразования, изменяющих структуру выражений без изменения их математической эквивалентности. Другим общим критерием, который позволил бы определить эквивалентность двух выражений, пакет не располагает. Для реализации алгоритма сравнения выражений пакет использует сведение их к полной форме, определяемой функцией **FullForm**. В справке по *Mathematica* обсуждается ряд важных механизмов создания шаблонов для довольно широкого класса выражений, в других справочных руководствах рассматриваются приемы, применяемые пакетом с целью расширения и ограничения классов выражений, представляемых шаблонами.

Для определения выражений, совпадающих с заданным шаблоном, можно применять функцию **Cases**, допускающую пять форматов кодирования, вызов *первого* из которых

**Cases**[*a*, *p*] возвращает элементы-выражения списка *a*, структурно соответствующие шаблону *p*, как весьма наглядно иллюстрирует следующий простой пример:

```
In[3373]:= Cases[{a + b*c^5, 5 + 6*y^7, a + b*p^m, a + b*m^(-p)}, a + b*x_^n_]
Out[3373]= {a + b*c^5, a + b*p^m, a + b*m^(-p)}
```

Между тем, не отвлекаясь на детали, только отметим, *Mathematica* располагает рядом функций, обеспечивающих работу с выражениями на уровне представляющих их шаблонов как в целом, так и на уровне составляющих их подвыражений; с данными средствами читатель может ознакомиться в системной справке или в [100,104,101,116].

Как отмечалось, для определения выражений, совпадающих с указанным шаблоном, можно применять функцию **Cases**, позволяющую возвращать элементы-выражения некоторого списка, структурно совпадающие с заданным шаблоном. При этом, далеко не все задачи *сравнения* выражений с *шаблонами* решаются стандартными средствами. Для решения данной задачи в ее более широком аспекте достаточно полезной может оказаться процедура **EquExprPatt**, чей вызов **EquExprPatt**[*x*, *p*] возвращает **True**, если выражение *x* соответствует заданному шаблону *p*, и **False** в противном случае. Ниже представлен фрагмент с исходным кодом процедуры и с примерами ее применения.

```
In[3395]:= EquExprPatt[x_, y_ /; ExprPatternQ[y]] := Module[{c, d = {}, t, v = {}, k = 1, j, p,
    g = {}, s = {}, a = Map[FullForm, Map[Expand, {x, y}]],
    b = Mapp[MinusList, Map[OP, Map[Expand, {x, y}]], {FullForm}],
    z = SetAttributes[ToString, Listable], w, {b, c} = ToString[{b, a}];
    p = StringPosition[c[[2]], {"Pattern["], "Blank[]"]];
    While[k = 2*k - 1; k <= Length[p],
    d = Append[d, StringTake[c[[2]], {p[[k]][[1]], p[[k + 1]][[2]]}]; k++];
    {t, k} = {ToExpression[d], 1};
    While[k <= Length[t], v = Append[v, StringJoin[ToString[Op[t[[k]]]]]; k++];
    v = ToString[v]; v = Map13[Rule, {d, v}]; v = StringReplace[c[[2]], v];
    b = Quiet[Mapp[Select, b, ! SystemQ[#] || ProcQ[#] || QFunction[ToString[#] &]]];
    {b, k, j} = {ToString[b], 1, 1};
    While[k <= Length[b[[1]]], z = b[[1]][[k]]; g = Append[g, {"[" <> z <> "," -> "[w",
    " " <> z <> "," -> " w", "[" <> z <> "]" -> "[w]", " " <> z <> "]" -> " w"}]; k++];
    While[j <= Length[b[[2]]], z = b[[2]][[j]]; s = Append[s, {"[" <> z <> "," -> "[w",
    " " <> z <> "," -> " w", "[" <> z <> "]" -> "[w]", " " <> z <> "]" -> " w"}]; j++];
    ClearAttributes[ToString, Listable];
    z = Map9[StringReplace, {c[[1]], v}, Map[Flatten, {g, s}]];
    If[z[[1]] === z[[2]], True, False]]

In[3396]:= EquExprPatt[a*Sin[x] - 5*b*c^5, a*Sin[x] - 5*b*x_^n_]
Out[3396]= True
In[3397]:= EquExprPatt[a*Sin[x] - 5*b*c^5, 78*Sin[x] - 460*b*x_^n_]
Out[3397]= True
In[3398]:= EquExprPatt[a^2 + 2*a*b + b^2, (x_ + y_)^2]
Out[3398]= True
```

```

In[3399]:= Cases[{a + b*c^5, 5 + 6*y^7, a + b*p^m, a + b*m^(-p)}, a + b*x_^n_]
Out[3399]= {a + b c^5, a + b p^m, a + b m^- p}
In[3400]:= Mapp[EquExprPatt, {a + b*c^5, 5 + 6*y^7, a + b*p^m, a + b*m^p}, a + b*x_^n_]
Out[3400]= {True, True, True, True}
In[3401]:= Mapp[Cases, {{a + b*c^5}, {5 + 6*y^7}, {a + b*p^m}, {a + b*m^p}}, a + b*x_^n_]
Out[3401]= {{a + b c^5}, {}, {a + b p^m}, {a + b m^p}}
In[3402]:= EquExprPatt1[a^2 + 2*a*b + b^2, (a + b)^2]
Out[3402]= True
    
```

Определение **EquExprPatt** использует ряд наших средств: **ExprPatternQ**, **Map13**, **Map9**, **Mapp**, **MinusList**, **OP**, **Op**, **ProcQ**, **QFunction**, **SystemQ**, также рассматриваемых в книге. Последние примеры фрагмента иллюстрируют также и более широкие возможности процедуры **EquExprPatt** относительно функции **Cases**. А так как алгоритм процедуры базируется на представлении выражений и шаблонов в полной форме (**FullForm**), то, в принципе, в качестве второго аргумента процедуры **EquExprPatt** можно применять любое выражение, закодировав второй аргумент как **y\_** в определении **EquExprPatt**, модифицировав ее в процедуру **EquExprPatt1**, отличающуюся от **EquExprPatt** только этим моментом. В данном случае возможно проводить тестирование 2 произвольных выражений на предмет их *структурной эквивалентности*, что представляется довольно важным в целом ряде задач анализа выражений. Отметим, в реализации процедуры используется довольно полезный прием временного наделения системной функции **ToString** *Listable*-атрибутом, но и ряд средств из нашего пакета **AVZ\_Package\_1** [120]. Необходимо иметь в виду, что при использовании правил преобразования возникает необходимость *именования* шаблонов. Так, согласно соглашению **x\_** будет обозначать произвольное выражение, однако каждый раз присваивая выражению имя **x**. Имеется возможность использования этого имени в правой стороне правила преобразования. Однако пакет ограничивает использование шаблона **x\_** в одном и том же выражении только относительно уникального выражения. В частности, в приведенном примере

```

In[2301]:= G[x_, y_, z_] := x + y + z
In[2302]:= {G[78, 460, 2013], G[78, 78, 78]}
Out[2302]= {2551, 234}
In[2303]:= {G[78, 460], G[78, 78, 78, 78]}
Out[2303]= {G[78, 460], G[78, 78, 78, 78]}
    
```

шаблоны **{x\_, y\_, z\_}** в общем случае определяют различные выражения, определяя в среде пакета уникальный объект – функцию **G[x, y, z]**. При этом, при вызове данной функции ее фактические аргументы могут принимать произвольные значения, в том числе и равные. Однако вызов функции будет корректным лишь при кодировании 3 фактических аргументов, при наличии числа фактических аргументов, отличного от трех, вызов возвращается *невычисленным*, т.е. в том же виде как был определен вызов с точностью до стандартных преобразований фактических аргументов в соответствии с соглашениями пакета. Предыдущий пример иллюстрирует сказанное. Более того, в случае использования выражения **G[a, b, c]**, где **{a, b, c}** – произвольные выражения, оно будет ассоциироваться именно с объектом **G[x, y, z]**. Так, определив простую функцию

```
In[2209]:= V[x_, x_, x_] := Plus[x, x, x]
In[2210]:= {V[78, 78, 78], V[78, 75, 460]}
Out[2210]= {234, V[78, 75, 460]}
In[2211]:= S[x_, y_, x_] := x + y
In[2212]:= {S[45, 67, 45], S[45, 67, 6]}
Out[2212]= {112, S[45, 67, 6]}
```

также с 3 формальными аргументами  $x_$ , в качестве которых выступает один и тот же шаблон, корректно обращаться к такому объекту  $V[x, x, x]$  можно только при условии трех идентичных фактических аргументов, иначе вызов возвращается *невычисленным*. Подобная же картина имеет место также и при указании частичного числа шаблонов идентичными, как иллюстрирует приведенный выше пример объекта  $S[x, y, x]$ . Итак, шаблон "\_" определяет произвольное выражение, а  $x_$  – любое выражение с именем  $x$ ; при этом, если выражение  $G[x_, x_]$  отвечает выражениям, для которых оба аргумента идентичны, то  $G[_ , _]$  отвечает всем выражениям формы  $G[x, y]$ , где  $x$  и  $y$  допускают и различные значения. Более того, конструкция формата " $x:p$ " определяет выражение с именем  $x$ , отвечающее шаблону  $p$ ; вообще говоря, данная конструкция представляет собой шаблон, присвоенный символу  $x$ . Рассмотрим теперь вопрос типирования.

**Определение типов выражения в шаблонах.** С этой целью вполне можно использовать заголовки выражений  $x$  (определяются вызовом функции **Head**[ $x$ ]), которые определяют их основную суть. Так, шаблоны  $_h$  и  $x_h$  будут представлять выражения с заголовком  $h$ , из которых наиболее часто используемыми являются следующие, а именно:

$x_h$  – выражение  $x$  с заголовком  $h$

```
=====
x_Integer  – выражение x с заголовком Integer (целое число)
x_Real     – выражение x с заголовком Real (действительное число)
x_Comple   – выражение x с заголовком Complex (комплексное число)
x_List     – выражение x с заголовком List (список)
x_String   – выражение x с заголовком String (строка)
x_Symbol   – выражение x с заголовком Symbol (символ)
x_Plus     – выражение x с заголовком Plus (сложение, вычитание)
x_Times    – выражение x с заголовком Times (произведение, деление)
x_Power    – выражение x с заголовком Power (степень)
```

В принципе, в качестве заголовка в составе шаблона может выступать произвольный заголовок, допустимый пакетом. Приведем примеры таких шаблонов, а именно:

```
In[2415]:= G[x_Plus] := x^2; S[x_Power] := x^2; {G[78], G[a + b], S[460], S[a^b]}
Out[2415]= {G[78], (a + b)^2, S[460], a^(2 b)}
```

Между тем, в целом ряде случаев стандартно определенных заголовков не хватает для задания шаблонов, вполне естественно поднимая вопрос подключения к их списку и *заголовкой*, определенных пользователем. А так как для вычисления заголовка служит стандартная функция **Head**, естественным представляется модифицировать данную функцию на предмет тестирования ею более широкого класса заголовков. Поэтому, с



этой целью определена процедура **RepStandFunc**, вызов которой **RepStandFunc[x, y, z]** возвращает вызов функции *y*, одноименной со стандартной *y* и определение которой задано в строчном формате аргументом *x*, на списке *z* ее фактических аргументов. В то же время такой вызов процедуры **RepStandFunc** является *разовым* в том смысле, что после него восстанавливается исходное состояние стандартной функции *y*. В начале следующего фрагмента представлен исходный код процедуры **RepStandFunc** наряду с примерами ее применения и тестированием последействия результата ее вызова. В остальной части фрагмента представлены средства, иллюстрирующие сказанное.

```
In[3381]:= RepStandFunc[x_String, y_Symbol, z_List] := Module[{b = Attributes[y], c, d,
    a = ToString[y] <> ".mx"}, DumpSave[a, y]; ClearAllAttributes[y];
    Clear[y]; ToExpression[x]; d = y[Sequences[z]]; Clear[y]; Get[a];
    SetAttributes[y, b]; DeleteFile[a]; d]

In[3382]:= x = "Sin[x_, y_, z_] := x^2 + y^2 + z^2"; RepStandFunc[x, Sin, {71, 78, 460}]
Out[3382]= 222725
In[3395]:= x = "Sin[x_] := x^5"; RepStandFunc[x, Sin, {47}]
Out[3395]= 229345007
In[3383]:= Definition[Sin]
Out[3383]= Attributes[Sin] = {Listable, NumericFunction, Protected}
In[3384]:= Sin[71.46078]
Out[3384]= 0.714459

In[3390]:= Headd := "Head[x_] := Module[{b = {ListListQ, NestListQ, ProcQ, QFunction,
    SystemQ}, c = {ListList, NestList, Procedure, Function, System},
    h = SetAttributes[SetAttributes, Listable], d = 78, k = 1},
    SetAttributes1[c, Protected];
    Quiet[For[k = 1, k <= Length[b], k++, If[b[[k]][x], d = c[[k]]; Break[]]];
    ClearAttributes[SetAttributes, Listable]; If[d === 78, x[[0]], d]]

In[3391]:= RepStandFunc[Headd, Head, {{{a}, {b, c}, {d}}}]
Out[3391]= NestList
In[3392]:= Definition[Head]
Out[3392]= Attributes[Head] = {Protected}
In[3393]:= Head[{{a}, {b, c}, {d}}]
Out[3393]= List
In[3396]:= G[h_NestList] := Length[h]
In[3397]:= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
Out[3397]= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
In[3398]:= G[h_List] := Length[h]
In[3399]:= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
Out[3399]= 7
In[2705]:= ClearAllAttributes[Head]; Clear[Head]; ToExpression[Headd]
In[2706]:= G[h_ListList] := Length[h]
In[2707]:= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
Out[2707]= 7
```

```

In[2795]:= SetAttributes1[x_, y_] := ToExpression["SetAttributes[SetAttributes, Listable];
SetAttributes[" <> ToString[x] <> ", " <> ToString[y] <> "];
ClearAttributes[SetAttributes, Listable]"
In[2796]:= t = {x, y, z}; SetAttributes1[t, Listable]; Map[Attributes, Flatten[{t, SetAttributes}]]
Out[2796]= {{Listable}, {Listable}, {Listable}, {HoldFirst, Protected}}

```

В предыдущем фрагменте представлена строчная структура **Headd**, представляющая собой определение процедуры **Head**, одноименной со стандартной функцией **Head** с расширением функциональных возможностей последней. В качестве примера далее приведен вызов **RepStandFunc[Headd, Head, {{{a}, {b, c}, {d}}}]**, в результате которого к списку *NestList*-типа разово применяется модифицированная версия функции **Head** (**Headd**), возвращая на таком списке его заголовок как *NestList*, тогда как стандартная функция **Head** на данном списке возвращает заголовок *List*.

Модификации процедуры **Head** в строчной структуре **Headd** довольно просты (путем соответствующего расширения списков, представляемых локальными переменными *b* и *c*), в принципе позволяя расширять список заголовков сколь угодно широко. Но в качестве составляющей шаблонов типа "*x\_h*" данные заголовки *Mathematica* не распознаются, как это весьма наглядно иллюстрирует пример фрагмента с функцией **G**. Более того, данный результат имеет место как при использовании процедуры **RepStandFunc**, так и при пролонгированной замене (на весь период текущего сеанса) стандартной функции **Head** ее модификацией, находящейся в строчной структуре **Headd**. В результате этой процедуры требуется перезагрузка пакета для восстановления оригинальной версии функции **Head**, если до того она не была сохранена в файле *mx*-формата, из которого ее можно было бы загрузить в текущий сеанс, как делает процедура **RepStandFunc**.

Наконец, завершает фрагмент функция **SetAttributes1**, чей вызов **SetAttributes1[x, y]** расширяет стандартную функцию **SetAttributes** на форму представления первого *x*-аргумента, в качестве которого могут выступать *индексированные* переменные, *списки* и др., например, обеспечивая установку атрибутов *y* для элементов списка. Между тем, описанный механизм разового применения одноименных *заменителей* стандартных функций в целом ряде случаев достаточно эффективен, тогда как пролонгирование таких заменителей на текущий сеанс пакета может вызвать конфликтные ситуации с его функциями, существенно использующими *оригиналы* замещенных средств. Итак, данный механизм следует использовать достаточно осмотрительно.

Первой задачей, относящейся к вопросу обработки *формальных* аргументов, с полным основанием можно отнести вычисление кортежа формальных аргументов функций/модулей/блоков пользователя, активированных в текущем сеансе явно или на основе загрузки пакетов, содержащих их определения. В предыдущей нашей работе [90,101] был предложен ряд средств для решения данной задачи в виде процедур **Args**, **Args0**, **Args1**, **Args2**, ниже мы представим подобные средства в более узком ассортименте и с улучшенными функциональными характеристиками. Прежде всего, в качестве весьма полезного средства представим процедуру **Args**, чей вызов **Args[x]** возвращает список формальных аргументов модуля/блока/функции *x*. Следующий фрагмент приводит исходный код процедуры **Args** с наиболее типичными примерами ее использования.

```

In[2122]:= V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2]; Kr := (#1^2 + #2^4 - 78*#3) &;
Art := Function[{x, y}, x*Sin[y]]; H[x_] := Block[{}, x]; H[x_, y_] := x + y;
P[x_] := Plus[Sequences[{x}]]; SetAttributes[H, Protected];
GS[x_;/ IntegerQ[x], y_;/ IntegerQ[y]] := Sin[78] + Cos[42];
Sv[x_;/ IntegerQ[x], y_;/ IntegerQ[y]] := x^2 + y^2;
Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6];
S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)];
P[y_] := Module[{}, y]; T := Compile[{{x, _Real}}, (x + y)];

In[2123]:= Args[P_, z___] := Module[{a, b, c, d = {}, k = 1, Vt},
    If[CompileFuncQ[P] || ProcQ[P] || FunctionQ[P],
        Vt[y_List] := Module[{p = 1, q = {}, t},
            While[p <= Length[y], q = Append[q, t = ToString[y[[p]]];
                StringTake[t, {1, StringPosition[t, "_"][[1]][[1]] - 1}]; p++]; q];
        If[CompileFuncQ[P],
            a = StringSplit[ToString[InputForm[Definition2[P]]], "\n \n"][[1]];
            b = Quiet[SubStrSymbolParity1[a, "{", "}"];
            b = Select[b, ! StringFreeQ[#, "_"] || ! StringFreeQ[a, "Function[" <> #] &];
            b = Mapp[StringSplit, b, ", "; b = Mapp[StringReplace, b, {"{" -> "", "}" -> ""}];
            b = Mapp[Select, b, StringFreeQ[#, "Blank$"] &]; c = b[[2]];
            For[k, k <= Length[c], k++, d = Append[d, c[[k]] <> b[[1]][[k]]]; d = ToExpression[d];
            If[{z} == {}, d, Flatten[Map[Vt, {d}]]],
            If[ProcQ[P] || QFunction[P], a = Flatten[{HeadPF[P]}];
            For[k, k <= Length[a], k++, d = Append[d,
                If[{z} != {}, Vt[ToExpression["{" <> StringTake[a[[k]],
                    {StringLength[ToString[P]] + 2, -2}] <> "}"]],
                ToExpression["{" <> StringTake[a[[k]], {StringLength[ToString[P]] + 2, -2}] <> "}"]]]];
            If[Length[d] == 1, d[[1]], d],
            a = StringTake[StringReplace[ToString[InputForm[Definition2[P]]],
                "Definition2[" -> "", 1], {1, -2}];
            If[SuffPref[a, "Function["], 1],
                b = SubStrSymbolParity1[a, "{", "}"];
                b = Select[b, ! StringFreeQ[a, "Function[" <> #] &]][[1]];
                a = StringSplit[StringTake[b, {2, -2}], ", ",
                a = StringReplace[a, {"#" -> "$$$$"}];
                a = Map[ToString, UnDefVars[ToExpression[a]]];
            Map[ToString, ToExpression[Mapp[StringReplace, a, "$$$$" -> "#"]]]], $Failed]]

In[2124]:= Map[Args, {V, S, Sv, T}]
Out[2124]= {{x_Real, y_Real}, {x_Integer, y_Real}, {x_Integer, y_Real}, {x_Real}}
In[2225]:= Mapp[Args, {V, S, Sv, T}, gs]
Out[2225]= {"x", "y"}, {"x", "y"}, {"x", "y"}, {"x"}

```

```
In[2126]:= Map[Args, {H, P, GS}]
Out[2126]= {{{x_}, {x_, y_}}, {{y_}, {x_}}, {x_ /; IntegerQ[x], y_ /; IntegerQ[y]}}
In[2227]:= Mapp[Args, {H, P, GS}, gs]
Out[2227]= {{{"x", {"x", "y"}}, {"y", {"x"}}, {"x", "y"}}
In[2128]:= Map[Args, {Art, Kr}]
Out[2128]= {"x", "y"}, {"#1", "#2", "#3"}
In[2229]:= Mapp[Args, {Art, Kr}, gs]
Out[2229]= {"x", "y"}, {"#1", "#2", "#3"}
In[2230]:= Map[Args, {avz, 78, a + b}]
Out[2230]= {$Failed, $Failed, $Failed}
```

При этом, формат возвращаемого результата определяется типом объекта  $x$ , а именно:

- на функции **Compile** возвращается список *формальных* аргументов с приписанными им типами;
- на {*модуле, блоке, типичной функции*} возвращается список *формальных* аргументов с приписанными им тестами на допустимость фактических аргументов или без них; при этом, процедура **Args** обрабатывает ситуацию «одноименные объекты с различными заголовками», возвращая вложенный список формальных аргументов относительно всех составляющих объект подобъектов в порядке, определенном процедурой **Definition2**;
- на *чистой* функции в кратком формате возвращается список *заместителей* { $\#1, \dots, \#n$ } в строчном формате формальных аргументов, в то время как для стандартной чистой функции возвращается список формальных аргументов в строчном формате.

Более того, вызов процедуры **Args**[ $W, h$ ] со вторым необязательным аргументом  $h$ , в качестве которого может выступать произвольное допустимое выражение или любая их последовательность, возвращает результат, подобный вызову с *первым* аргументом, с тем отличием, что все формальные аргументы закодированы в строчном формате, но без приписанных аргументам типов и тестов на допустимость. Из представленных в предыдущем фрагменте примеров все просматривается весьма наглядно.

На недопустимом фактическом аргументе  $W$  вызов процедуры **Args**[ $W$ ] возвращает значение **\$Failed**. Процедура наряду со стандартными использует ряд наших средств таких, как **CompileFuncQ**, **Definition2**, **FunctionQ**, **HeadPF**, **Mapp**, **ProcQ**, **QFunction**, **SubStrSymbolParity1**, **SuffPref**, **UnDefVars**. Данная процедура используется довольно широко, прежде всего, в задачах системного программирования в среде *Mathematica*, существенно расширяя вышеупомянутые процедуры **Args**, **Args0**, **Args1**, **Args2**. Более того, следует отметить, везде по тексту данной книги вновь представляемые средства замещают одноименные средства из нашей книги [101] и пакета *AVZ\_Package* [90]. В этом же контексте отметим, что ряд средств, представленных в [90,101], отсутствуют в этой книге по причине замены их более реактивными и функционально развитыми, как например, процедура **ArgsProc**, чьи функции перекрываются процедурой **Args**.

В дополнение к процедуре **Args** довольно полезной оказывается процедура **ArgsBFM**, предназначенная для вычисления формальных аргументов блока/ функции/ модуля. Фрагмент представляет исходный код процедуры с примерами ее использования.



```

In[2076]:= ArgsTypes[x_/: CompileFuncQ[x] || ProcQ[x] || FunctionQ[x]] :=
Module[{a = Args[x], c = {}, d = {}, k = 1},
  If[CompileFuncQ[x], a = Mapp[StringSplit, Map[ToString, a], "-"];
  If[Length[a] == 1, a[[1]], a],
  If[PureFuncQ[x], a = Map[{#, "Arbitrary"} &, a]; If[Length[a] == 1, a[[1]], a],
  SetAttributes[ToString, Listable]; a = Map[ToString, a];
  ClearAttributes[ToString, Listable]; a = If[NestListQ[a], a, {a}];
  For[k, k <= Length[a], k++, c = Append[c,
    Mapp[StringSplit, Mapp[StringSplit, a[[k]], "-/:"], {"__", "_", "-"}]]; c;
  For[k = 1, k <= Length[c], k++, d = Append[d, Map[Flatten, c[[k]]]]; c = {}];
  For[k = 1, k <= Length[d], k++, c = Append[c,
    Map[If[Length[#] == 1, {#[[1]], "Arbitrary"}, #] &, d[[k]]]];
  c = Map[If[Length[#] == 1, {#[[1]], #] &, c]; If[Length[c] == 1, c[[1]], c]]]

In[2077]:= Map[ArgsTypes, {GS, Args}]
Out[2077]= {{{{"x", "IntegerQ[x]"}, {"y", "IntegerQ[y]"}}, {"x", "Arbitrary"}, {"y", "String"}},
  {"x", "Arbitrary"}}, {"P", "CompileFuncQ[P] || ProcQ[P] || FunctionQ[P]"},
  {"z", "Arbitrary"}}}

In[2078]:= ArgsTypes[P]
Out[2078]= {{{{"x", "StringQ[x]"}, {"y", "Arbitrary"}}, {"y", "Arbitrary"}, {"x", "Arbitrary"}}

In[2079]:= Map[ArgsTypes, {Art, Kr}]
Out[2079]= {{{{"x", "Arbitrary"}, {"y", "Arbitrary"}}, {"#1", "Arbitrary"}, {"#2", "Arbitrary"},
  {"#3", "Arbitrary"}}}

In[2080]:= Map[ArgsTypes, {V, Sv, S, G, T}]
Out[2080]= {{{{"x", "Real"}, {"y", "Real"}}, {"x", "Integer"}, {"y", "Real"}}, {"x", "Integer"},
  {"y", "Real"}}, {"x", "Integer"}, {"y", "Real"}, {"x", "Real"}]}

```

Предыдущий фрагмент представляет исходный код процедуры **ArgsTypes** наряду с типичными примерами ее использования. Более того, процедура **ArgsTypes** успешно обрабатывает упомянутую ситуацию «одноименные объекты с различными заголовками», возвращая вложенные 2-элементные списки формальных аргументов относительно составляющих объект подобъектов в порядке, определенном функцией **Definition**. И в этом случае 2-элементные списки имеют представленный выше формат, тогда как для объектов с пустым списком формальных аргументов возвращается пустой список, т.е. {}. В отличие от одноименной **ArgsTypes** [90,101] данная процедура обрабатывает блоки/функции/модули, включая чистые функции и **Compile** функции. При этом, вызов **ArgsTypes[x]** на недопустимом аргументе *x* возвращается невычисленным.

Множественные шаблоны форматов *x\_\_* и *x\_\_\_* позволяют определять произвольное количество допустимых фактических аргументов блока/функции/модуля; при этом, если первый определяет не менее одного аргумента, то второй допускает отсутствие фактического аргумента. Эти форматы шаблонов *формальных* аргументов позволяют определять объекты указанного типа, допускающие произвольное число *фактических* аргументов при их вызове. Это обстоятельство положено в основу программирования

средств, определяющих *арность* блока/функции/модуля пользователя, т.е. количество фактических аргументов, допускаемых при вызове указанного типа объекта, которое не вызывает особых (*невычисленный вызов*) или ошибочных ситуаций, обусловленных несоответствием количества полученных фактических аргументов допустимому при определении объекта. Вопрос вычисления *арности* блока/функции/модуля во многих приложениях и, прежде всего, системного характера является достаточно важным, но *Mathematica* не располагает средствами для ее определения, поэтому был создан ряд средств для решения данного вопроса в составе процедур **Arity**, **Arity1**, **Arity2**, **ArityM**, **ArityPF**, решающих данную задачу с той или иной степенью общности [90,101]. Ниже фрагмент представляет исходный код процедуры **Arity** с примерами ее применения, которая обобщает все вышеперечисленные средства, решающие задачу арности.

```
In[2065]:= V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2]; Kr := (#1^2 + #2^4 - 78*#3) &;
Art := Function[{x, y}, x*Sin[y]]; H[x_] := Block[{}, x]; H[x_, y_] := x + y;
P[x_] := Plus[Sequences[{x}]]; SetAttributes[H, Protected];
GS[x_/_IntegerQ[x], y_/_IntegerQ[y]] := Sin[78] + Cos[42];
Sv[x_/_IntegerQ[x], y_/_IntegerQ[y]] := x^2 + y^2;
Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6];
S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)]; P[y_] := Module[{}, y];
T := Compile[{{x, _Real}}, (x + y)]; Vs[x_/_SameQ[{x}, {}]] := {x}

In[2066]:= Arity[P_/_SystemQ[P] || CompileFuncQ[P] || ProcQ[P] || FunctionQ[P]] :=
Module[{a}, If[SystemQ[P], "System", a = Args[P];
Mapp[SetAttributes, {ToString, StringFreeQ}, Listable]; a = Map[ToString, a];
a = Map[If[DeleteDuplicates[StringFreeQ[#, "___"]] === {True}, Length[#],
"Undefined"] &, If[NestListQ[a, a, {a}]]];
Mapp[ClearAttributes, {ToString, StringFreeQ}, Listable]; If[Length[a] == 1, a[[1]], a]]

In[2067]:= Map[Arity, {V, S, Sv, T}]
Out[2067]= {2, 2, 2, 1}
In[2068]:= Map[Arity, {H, P, GS}]
Out[2068]= {{1, 2}, {1, "Undefined"}, 2, 1}
In[2069]:= Map[Arity, {Art, Kr, ProcQ, Sin, For}]
Out[2069]= {2, 3, 1, "System", "System"}
In[2070]:= Map[Arity, {avz, 78, a + b}]
Out[2070]= {Arity[avz], Arity[78], Arity[a + b]}

In[2166]:= Arity[P_/_SystemQ[P] || CompileFuncQ[P] || ProcQ[P] || FunctionQ[P]] :=
Module[{a}, If[SystemQ[P], "System", a = Args[P]; a = Mapp1[ToString, a];
a = Map[If[DeleteDuplicates[StringFreeQ[#, "___"]] === {True}, Length[#],
"Undefined"] &, If[NestListQ[a, a, {a}]]]; If[Length[a] == 1, a[[1]], a]]

In[2067]:= Map[Arity, {V, S, Sv, T}]
Out[2067]= {2, 2, 2, 1}
```

```
In[2068]:= Map[Arity, {H, P, GS}]
Out[2068]= {{1, 2}, {1, "Undefined"}, 2, 1}
In[2069]:= Map[Arity, {Art, Kr, ProcQ, Sin, For}]
Out[2069]= {2, 3, 1, "System", "System"}
In[2070]:= Map[Arity, {avz, 78, a + b}]
Out[2070]= {Arity[avz], Arity[78], Arity[a + b]}
```

На блоках/функциях/модулях с неопределенным числом аргументов вызов **Arity**[*x*] возвращает значение **"Undefined"**, на системных функциях возвращается **"System"**, в то время, как на объектах, имеющих фиксированное число фактических аргументов, возвращается их количество, в остальных случаях вызов возвращает невычисленным. Отметим, что процедура **Arity** обрабатывает особую ситуацию «одноименные процедуры с различными заголовками», возвращая список арностей подобъектов, составляющих *x* объект. При этом, между этим списком и списком определений подобъектов, который возвращается по вызову **Definition**[*x*] существует взаимно-однозначное соответствие.

Наряду со стандартными процедура **Arity** использует средства **Args**, **CompileFuncQ**, **FunctionQ**, **Mapp**, **NestListQ**, **ProcQ**, **SystemQ**. Более того, при ее программировании в свете упрощения алгоритма оказалось целесообразным на время вызова процедуры наделять системные функции **ToString** и **StringFreeQ** атрибутом **Listable**, позволяя в значительной мере уменьшать исходный код программируемых средств.

Довольно полезным дополнением к процедуре **Arity** служит и процедура **ArityBFM**, определяющая арность объектов типа блок/классическая функция/модуль. Фрагмент представляет исходный код процедуры наряду с примерами ее использования.

```
In[2074]:= ArityBFM[x_;/ BlockFuncModQ[x]] := Module[{a = Flatten[{HeadPF[x]], b},
    b = Map[If[! StringFreeQ[#, {"_", ""}], "Undefined",
    Length[ToExpression["{" <> StringTake[StringReplace[#,
    ToString[x] <> "[" -> "", 1], {1, -2}] <> ""]}]] &, a];
    If[Length[b] == 1, b[[1]], b]]
In[2075]:= G[x_;/ IntegerQ[x], y_;/ IntegerQ[y]] := x + y; G[x_Integer, y_] := x + y;
    G[x_, y_;/ IntegerQ[y], z_] := x + y + z; G[x_, y_] := x + y; G[x_;/ IntegerQ[x]] := x;
    G[x_;/ x == {42, 47, 67}, y_;/ IntegerQ[y]] := Length[x] + y;
In[2076]:= ArityBFM[G]
Out[2076]= {2, 3, "Undefined", 2, "Undefined", 1}
In[2077]:= S[x_, y_] := x*y; V[x_] := {x}; Map[ArityBFM, {S, V}]
Out[2077]= {2, "Undefined"}
In[2078]:= V[a_, b_, c_, d_, h_] := N[h*(3*a*b + (c - b)*d + (d - a)*c)/3000]
In[2079]:= {V[25, 36, 40, 30, 60], ArityBFM[V]}
Out[2079]= {60.4, 5}
```

Вызов процедуры **ArityBFM**[*x*] возвращает арность (*количество аргументов*) объекта *x*, в качестве которого может выступать блок/функция/модуль; при этом, под функцией понимается функция классического типа, т.е. с заголовком. При наличии в заголовке *формальных* аргументов с шаблонами {"\_", ""} арность определяется неопределенной,



("Undefined"), т.к. под ней понимается реальное количество фактических аргументов, допустимых при вызове  $x$ , которое в таком случае может быть неопределенным. При этом, на объектах  $x$  типа, отличного от указанных, вызов возвращается *невычисленным*.

Подобно *Maple* система *Mathematica* не дает возможности тестировать в точке вызова блока/функции/модуля недопустимость всех ее фактических аргументов, прерывая вызов уже на первом встреченном *недопустимом* фактическом аргументе. Между тем, принимая во внимание важность установления всех недопустимых аргументов всего за *один* проход, была создана процедура **TestArgsTypes**, решающая данную довольно важную задачу и представленная в нашей книге [101] и пакете *AVZ\_Package* [90]. При этом, появление новых и обновление существующих наших функциональных средств позволило обновить и данное средство, представляемое одноименной процедурой. В следующем фрагменте представлен исходный код процедуры **TestArgsTypes** наряду с одной ее полезной модификацией **TestArgsTypes1** и примерами их использования.

```
In[2060]:= TestArgsTypes[P_;/; ModuleQ[P] || BlockQ[P] || QFunction[P], y_] :=
Module[{a = Map[ToString, Args[P]], b = ToString[InputForm[y]], c = {}, d = {}, h, k = 1},
ClearAll["$TestArgsTypes"];
If[! SuffPref[b, ToString[P] <> "[", 1], Return[y], c = Map[ToString1,
ToExpression["{" <> StringTake[b, {StringLength[ToString[P]] + 2, -2}] <> "}"]];
If[Length[a] != Length[c], $TestArgsTypes =
"Number of formal and actual arguments is different"; $Failed,
For[k, k <= Length[a], k++, d = Append[d,
ToExpression["{" <> c[[k]] <> "}"] <> "/. " <> a[[k]] <> " -> True"]];
d = Map[If[ListQ[#], #[[1]], #] &, d];
h = Flatten[Map3[Position, d, Cases[d, Except[True]]];
h = Map[{#, If[ListQ[d[[#]]], Flatten[d[[#]], 1], d[[#]]}] &, h];
$TestArgsTypes = If[Length[h] == 1, h[[1]], h]; $Failed]]

In[2061]:= P[x_, y_String, z_;/; If[z === 78, True, False]] := {x, y, z}
In[2062]:= TestArgsTypes[P, P[agn, "ArtKr", 78]]
Out[2062]= {agn, "ArtKr", 78}
In[2063]:= TestArgsTypes[P, P[x, y, z]]
Out[2063]= $Failed
In[2064]:= $TestArgsTypes
Out[2064]= {{2, y}, {3, z}}
In[2065]:= TestArgsTypes[P, P[x, y, z, h]]
Out[2065]= $Failed
In[2066]:= $TestArgsTypes
Out[2066]= "Number of formal and actual arguments is different"
In[2067]:= TestArgsTypes[P, P[x, "y", {460}]]
Out[2067]= $Failed
In[2068]:= $TestArgsTypes
Out[2068]= {3, {460}}
```

```

In[2069]:= TestArgsTypes[P, P[x, a + b, {460}]]
Out[2069]= $Failed
In[2070]:= $TestArgsTypes
Out[2070]= {{2, a + b}, {3, {460}}}
In[2071]:= VS[x_, n_;/ IntegerQ[n], y_, z_;/ StringQ[z], L_;/ ListQ[L] &&
            MemberQ[{{0}, {1}, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]] :=
            Block[{}, L[[StringLength[y <> z] + n]]]
In[2072]:= VS[6, -4, "A", "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[2072]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
In[2073]:= VS[6, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[2073]= VS[6, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
In[2074]:= TestArgsTypes[VS[9, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[2074]= $Failed
In[2075]:= $TestArgsTypes
Out[2075]= {2, 7.2`}
In[2076]:= TestArgsTypes[VS, VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[2076]= $Failed
In[2077]:= $TestArgsTypes
Out[2077]= {{2, 7.2`}, {4, vz}, {5, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}}}
In[2078]:= TestArgsTypes[VS, VS[9, 0, "A", "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[2078]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
In[2079]:= $TestArgsTypes
Out[2079]= $TestArgsTypes
In[2162]:= TestArgsTypes1[P_;/ ModuleQ[P] || BlockQ[P] || QFunction[P], y_] :=
            Module[{a = Quiet[ArgsTypes[P]], b = ToString[InputForm[y]],
                    g = Map[ToString1, Args[P]], c, d = {}, h, k = 1, n, p, w, w1],
                    a = Map[#[[1]], StringReplace[#[[2]], "\\\\" -> "\\"] &, a];
                    ClearAll["$TestArgsTypes", "$$Art$Kr$$"];
                    If[! SuffPref[b, ToString[P] <> "[", 1], Return[y], c = Map[ToString1,
                    ToExpression["{" <> StringTake[b, {StringLength[ToString[P]] + 2, -2}] <> "}"]];
                    If[Length[a] != Length[c], Return[$TestArgsTypes =
                    "Number of formal and actual arguments is different"; $Failed],
                    w = Map[StringTake[#, {1, StringPosition[#, "_"][[1]][[1]] - 1}] &, g];
                    w1 = Map[ToString, Unique[w]];
                    While[k <= Length[w], ToExpression[w1[[k]] <> " = " <> w[[k]]; k++];
                    Map[ClearAll, w]; For[k = 1, k <= Length[a], k++, p = a[[k]];
                    If[p[[2]] === "Arbitrary", d = Append[d, True],
                    If[StringFreeQ[g[[k]], "/;"],
                    If[ToExpression["Head[" <> c[[k]] <> "]"] === " <> p[[2]],
                    d = Append[d, True], d = Append[d, False]], $$Art$Kr$$ = ToExpression[p[[1]]];
                    n = ToExpression[{p[[1]] <> " = " <> c[[k]], p[[2]]];

```

```

ToExpression[p[[1]] <> " = " <> "$$Art$Kr$$"];
If[n[[-1]], d = Append[d, True], d = Append[d, False]]];];
h = DeleteDuplicates[Flatten[Map3[Position, d, Cases[d, Except[True]]]]];
h = Map[{#, If[ListQ[c[[#]]], Flatten[c[[#]], 1], c[[#]]] &, h];
$TestArgsTypes = If[Length[h] == 1, h[[1]], h]; k = 1;
While[k <= Length[w], ToExpression[w[[k]] <> " = " <> w1[[k]]; k++]; $Failed]

In[2163]:= TestArgsTypes1[P, P[x, a + b, {460}]]
Out[2163]= $Failed
In[2164]:= $TestArgsTypes
Out[2164]= {{2, "a + b"}, {3, "{460}"}}
In[2165]:= TestArgsTypes1[P, P[agn, "ArtKr", 78]]
Out[2165]= {agn, "ArtKr", 78}
In[2166]:= TestArgsTypes1[P, P[x, y, z, h]]
Out[2166]= $Failed
In[2167]:= $TestArgsTypes
Out[2167]= "Number of formal and actual arguments is different"
In[2168]:= TestArgsTypes1[P, P[x, y, z]]
Out[2168]= $Failed
In[2169]:= $TestArgsTypes
Out[2169]= {{2, "y"}, {3, "z"}}
In[2170]:= TestArgsTypes1[VS, VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 78, 2}}]]
Out[2170]= $Failed
In[2171]:= $TestArgsTypes
Out[2171]= {{2, "7.2"}, {4, "vz"}, {5, "{0, {1, 0, 1}, {2, 0, 0, 0, 78, 2}"}}}

```

Вызов приведенной процедуры **TestArgsTypes**[*x*, *x*[...]] обрабатывает *вызов* процедуры *x* таким образом, что возвращает результат вызова процедуры *x*[...] в случае отсутствия недопустимых фактических аргументов и равного числа фактических и формальных аргументов в точке вызова процедуры. В противном случае возвращается **\$Failed**. При этом, через глобальную переменную **\$TestArgsTypes** возвращается *вложенный* список, двухэлементные подспски которого определяют набор недопустимых фактических аргументов, а именно: первый элемент подспска определяет номер недопустимого фактического аргумента, второй – его значение. При этом, при несоответствии числа *формальных* аргументов *фактическим* через переменную **\$TestArgsTypes** возвращается соответствующее диагностическое сообщение. Между тем, для упрощения алгоритма тестирования, реализуемого процедурой, предполагается, что *формальные* аргументы процедуры *P* типированы шаблоном «\_» (*подчеркивание*) либо конструкцией формата «*Аргумент\_* /; *Тест*». Более того, предполагается, вызов процедуры *P* невычисленным обусловлен лишь несоответствием типов фактических аргументов формальным либо несоответствием их количеств. Итак, вопрос тестирования фактических аргументов рассматривается на уровне заголовка блока/функции/модуля лишь для случая, когда их количество фиксировано. Если же процедура/функция допускает необязательные аргументы, их типирование предполагает корректным использование произвольных

выражений в качестве фактических значений, т.е. предполагается тип формата «*x\_*». В этой связи при необходимости их тестирование должно производиться в самом теле процедуры/функции, как это иллюстрируют полезные примеры в [101]. Итак, при сложных алгоритмах проверки получаемых *фактических* аргументов на *допустимость* рекомендуется программировать их в теле блоков/процедур, точнее, целесообразнее. Возвращаясь вновь к процедуре **TestArgsTypes**[*x*, *x*[...]], обеспечивающей диагностику одновременно всех полученных на вход процедуры *x* фактических аргументов, лишь отметим, дальнейшее развитие процедуры **TestArgsTypes** вполне может завершиться созданием во многих отношениях достаточно полезного средства манипулирования объектами процедурного типа, однако в задачу книги этот вопрос не входил. Между тем, в качестве расширения процедуры **TestArgsTypes** можно отметить возможность тестирования на допустимость фактических аргументов при условии существования в заголовках формальных аргументов типов {«*x\_*», «*x*\_\_»}. Именно использованный в процедуре **TestArgsTypes1**, являющейся одной полезной модификацией процедуры **TestArgsTypes**, является к этому достаточно перспективной предпосылкой. При этом, результат вызова **TestArgsTypes1**[*x*, *x*[...]] аналогичен вызову **TestArgsTypes**[*x*, *x*[...]] с тем только отличием, что значения недопустимых фактических аргументов даются в строчном формате. Не взирая на меньшую реактивность второй процедуры против первой, использованный при ее реализации алгоритм достаточно интересен для ряда приложений, прежде всего, системного характера; его анализ может быть достаточно полезен заинтересованному читателю, который желает глубже освоить программную среду *Mathematica*. Данное замечание относится и к ряду средств настоящей книги.

Между тем, следует иметь в виду, что использование процедурами **TestArgsTypes1** и **TestArgsTypes** глобальной переменной **\$TestArgsTypes**, через которую возвращается информация о полученных тестируемой процедурой при ее вызовах недопустимых *фактических* аргументов, должна быть определена в пакете пользователя, содержащем определения данных процедур, т.е. являться предопределенной, в противном случае через нее не передается диагностическая информация. Сделать это можно, например, включением в пользовательский пакет *AVZ\_Package\_1* следующего блока, а именно:

```
Begin["$TestArgsTypes`"]
$TestArgsTypes = 78
End[]
```

с обязательным обеспечением по данной переменной *справки (usage)*, например, вида

```
$TestArgsTypes::usage = "The variable $TestArgsType defined by procedures
  TestArgsTypes and TestArgsTypes1."
```

Данное замечание следует учитывать при организации процедур, использующих для дополнительного возврата результата *глобальные \$-переменные*, т.е. такие переменные должны инициализироваться пакетом, содержащим средства с их использованием.

К процедурам **TestArgsTypes** и **TestArgsTypes1** в определенной степени примыкает и процедура **TestArgsCall**, позволяющая выделять определения блока/функции или модуля, на которых вызов с заданными фактическими аргументами является вполне корректным. Фрагмент представляет исходный код процедуры с примерами.

```
In[2089]:= TestArgsCall[x_;/ BlockFuncModQ[x], y_] := Module[{d, p, h = {}, k = 1,
    a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}],
    c = "$$$", n = ToString[x]},
    While[k <= Length[b], d = c <> n; ToExpression[c <> b[[k]] <> ":=78"];
    p = Symbol[d][y]; ToExpression["Clear[" <> d <> ""];
    If[p === 78, h = Append[h, a[[k]]]; k++]; If[Length[h] == 1, h[[1]], h]
In[2090]:= G[x_;/ IntegerQ[x], y_;/ IntegerQ[y]] := x + y; G[x_Integer, y_] := x + y;
G[x_, y_;/ IntegerQ[y], z_] := x+y+z; G[x_, y_] := x+y; G[x_;/ IntegerQ[x]] := x;
G[x_;/ x == {42, 47, 67}, y_;/ IntegerQ[y]] := Length[x] + y;
In[2091]:= TestArgsCall[G, 19.42, 78]
Out[2091]= "G[x_, y_] := x + y"
In[2092]:= V[x_;/ IntegerQ[x], y_;/ IntegerQ[y]] := x + y; TestArgsCall[V, 19.42, 78]
Out[2092]= {}
In[2093]:= TestArgsCall[Avz, 19.42, 78]
Out[2093]= TestArgsCall[Avz, 19.42, 78]
```

Вызов **TestArgsCall**[*x*, *y*] возвращает определение либо список определений модуля/ блока/ функции *x*, на которых вызов с кортежем фактических аргументов *y* является корректным, т.е. их типы соответствуют допустимым типам формальных аргументов. В противном случае вызов возвращает пустой список; на недопустимом аргументе *x*, отличном от блока/функции/модуля, вызов возвращается невычисленным. Тогда как вызов процедуры **TestFactArgs**[*x*, *y*] возвращает список из *True* и *False*, определяющих какой фактических аргумент из списка *y* будет допустимым в вызове *x*[**Sequences**[*y*]], где *x* – имя объекта с заголовком. Процедура предполагает равное число формальных и фактических аргументов, определяемых *y*, и наличие у объекта *x* фиксированного числа аргументов, в противном случае вызов возвращает **\$Failed**. Фрагмент приводит исходный код процедуры **TestFactArgs** с типичными примерами ее использования.

```
In[762]:= TestFactArgs[x_;/ ProcQ[x] || QFunction[x], y_List] := Module[{b, c = {}, d, k = 1,
    a = Flatten[{HeadPF[x]}][[1]]},
    b = StrToList["{" <> StringTake[a, {StringLength[ToString[x]] + 2, -2}] <> "}"];
    b = Map[StringSplit[#, "_"] &, b]; If[Length[b] == Length[y] && StringFreeQ[a, "_"],
    While[k <= Length[b], d = b[[k]]; If[Length[d] == 1, AppendTo[c, True],
    If[Length[d] == 2 && SymbolQ[d[[2]]], AppendTo[c, Head[y[[k]]] === Symbol[d[[2]]],
    If[SuffPref[d[[2]], "/; ", 1], AppendTo[c, ToExpression[StringReplace3[
    StringTake[d[[2]], {5, -1}], d[[1]], ToString[y[[k]]]]]]]; k++]; c, $Failed]]
In[763]:= VGS[x_, y_Integer, z_;/ ListQ[z]] := Flatten[{x, y, z}]
In[764]:= TestFactArgs[VGS, {avz, 71, {g, s, a, k}}]
Out[764]= {True, True, True}
In[765]:= TestFactArgs[VGS, {42, ag, a + b}]
Out[765]= {True, False, False}
```

Процедура **TestFactArgs** является обобщением процедуры **CheckArgs** системы *Maple*.

## 8.6. Локальные переменные процедурных и блочных объектов, и средства манипулирования с ними в системе *Mathematica*

Рассмотрев в предыдущих двух разделах средства манипулирования определениями процедур/функций, а также с их заголовками, переходим к рассмотрению средств, в сферу интересов которых входят задачи, связанные с манипулированием следующей важнейшей компонентой определения процедур – *локальными переменными*. Данная компонента определяет *первую* ведущую переменную в определении процедуры, как функции от двух переменных – списка локальных переменных и ее тела. Локальные переменные имеют место лишь для процедурных объектов *Math*-языка типа *Module* или *Block*, тогда как для функций такое понятие отсутствует. *Локальные* переменные имеют областью своего действия лишь тело модуля, не пересекаясь с одноименными переменными его внешней среды. Между тем, именно между процедурами *блочного* и *модульного* типов имеется весьма существенное *различие*, базирующееся на механизмах локальных переменных, которые используются обоими типами процедур и которые рассмотрены несколько выше и более детализированы ниже. Ввиду важности данной компоненты, для манипулирования которой *Mathematica* не располагает средствами, весьма желательно располагать подобными средствами. Между тем, предварительно имеет смысл более детально рассмотреть данную компоненту модулей и блоков.

Процедуры (*в терминологии пакета «модули»*) в среде *Mathematica* функционируют следующим образом. При каждом вызове процедуры для ее локальных переменных генерируются новые символы, определяющие их имена, *уникальные* в текущем сеансе. Каждая локальная переменная идентифицируется символом формы *Имя\$Номер*, где *Имя* – имя локальной переменной, определенное в процедуре, и *Номер* – ее текущий номер в сеансе. При этом номер определяется пакетной переменной *\$ModuleNumber*, как это иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[2172]:= G[x_, y_, z_] := Module[{a, b, c}, h = a*x + b*y + c*z; {h, a, b,
      Symbol["a" <> "$" <> ToString[$ModuleNumber - 1]]}]
In[2173]:= {$ModuleNumber, G[71, 66, 45], $ModuleNumber}
Out[2173]= {4477, {71 a$4477 + 66 b$4477 + 45 c$4477, a$4477, b$4477, a$4477}, 4478}
In[2174]:= {$ModuleNumber, G[71, 66, 45], $ModuleNumber}
Out[2174]= {4479, {71 a$4479 + 66 b$4479 + 45 c$4479, a$4479, b$4479, a$4479}, 4480}
In[2175]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[2175]= {4481, {71 a$4481 + 66 b$4481 + 45 c$4481, a$4481, b$4481, a$4481}, 4482}
In[2176]:= n = 1; While[n < 6, Print[$ModuleNumber]; n++]
      4489
      4489
      4489
      4489
      4489
In[2177]:= {$ModuleNumber, $ModuleNumber}
Out[2177]= {4490, 4490}
```

Из приведенного примера довольно четко прослеживается сам принцип присвоения текущих номеров локальным переменным при каждом новом обращении к модулю, их содержащему. Из фрагмента также следует, что *наращивание* текущих номеров для локальных переменных при отсутствии вызовов модулей производится лишь в новых *Input*-параграфах документа. При этом, при условии знания текущей нумерации для локальных переменных процедуры появляется возможность динамически получать их значения вне процедуры после каждого ее вызова. Как иллюстрирует следующий достаточно простой и весьма наглядный фрагмент, а именно:

```
In[2079]:= S[x_, y_] := Module[{a = $ModuleNumber - 1, b = $ModuleNumber - 1,
                                c = $ModuleNumber - 1}, h := a*x + b*y + c;
                                {h, Symbol["a$" <> ToString[$ModuleNumber - 1]],
                                Symbol["b$" <> ToString[$ModuleNumber - 1]],
                                Symbol["c$" <> ToString[$ModuleNumber - 1]], a b, c}]

In[2080]:= S[71, 66]
Out[2080]= {627900, 4550, 4550, 4550, 20702500, 4550}
In[2081]:= g := {a$4551, b$4551, c$4551}
In[2082]:= S[71, 66]
Out[2082]= {629004, 4558, 4558, 4558, 20775364, 4558}
In[2083]:= d := {g, {a$4559, b$4559, c$4559}}
In[2084]:= S[71, 66]
Out[2084]= {629832, 4564, 4564, 4564, 20830096, 4564}
In[2085]:= {d, {a$4565, b$4565, c$4565}}
Out[2085]= {{{4550, 4550, 4550}, {4558, 4558, 4558}}, {4564, 4564, 4564}}
```

Следовательно, пользователь имеет возможность работать с *локальными* переменными и вне самой процедуры, содержащей их, т.е. по сути на уровне *глобальных переменных*, что в ряде случаев может довольно эффективно использоваться в программировании различного рода задач и, прежде всего, задач системного характера.

```
In[2087]:= Kr[x_, y_] := Module[{a, b}, h := a*x + b*y; {{a, b, h}, h}]
In[2088]:= Kr[16, 24]
Out[2088]= {a$4576, b$4576, 16 a$4576 + 24 b$4576, 16 a$4576 + 24 b$4576}
In[2089]:= %[[1]][[1]]^2 + Take[%[[1]], {2, 2}]^2
Out[2089]= {a$4576^2 + b$4576^2}
In[2090]:= Kr[x_, y_] := Module[{a, b}, a = 96; b = 89; h := a*x + b*y;
                                Print[{"a$" <> ToString[$ModuleNumber - 1],
                                "b$" <> ToString[$ModuleNumber - 1]};
                                {Symbol["a$" <> ToString[$ModuleNumber - 1]],
                                Symbol["b$" <> ToString[$ModuleNumber - 1]]}]

In[2091]:= Kr[16, 24]
                                {a$4586, b$4586}
Out[2091]= {96, 89}
In[2092]:= %[[1]]^2 + %[[1]]^2
Out[2092]= 17137
```

Предыдущий простой фрагмент достаточно наглядно иллюстрирует вышесказанное. Как правило, пользователь не должен работать со значениями *локальных* переменных вне самой процедуры (*модуля*); между тем, в случае работы с процедурой в *диалоговом* режиме или при использовании для мониторинга выполнения процедуры функции, например, **Trace** данные локальные переменные визуализируются. Более того, такая возможность может быть использована для нестандартных вычислений, но эффект от этого полностью определяется опытом и навыками пользователя, его знанием пакета. В пакете *Maple* подобного явного механизма работы с локальными переменными вне процедур не имеется, хотя специальными ухищрениями подобный механизм вполне может быть реализован, в частности, на основе метода «*дисковых транзитов*». Однако, именно такой подход делает переменные процедур реально *локальными* с областью действия, ограниченной самой процедурой. В данном случае локальные переменные недоступны извне процедуры, что в некотором отношении можно рассматривать как определенную предпосылку для определения «*черного ящика*» и вполне естественного перехода к парадигме модульной организации в программировании.

В ряде случаев возникает необходимость генерации уникальных для текущего сеанса имен объектов. Для этой цели предназначена функция **Unique**, достаточно детально рассмотренная в книге [101] и генерирующая имена без приписанных им атрибутов. При этом, для обеспечения символам уникальности каждый вызов функции **Unique** обеспечивает приращение для значения переменной **\$ModuleNumber**, как это хорошо иллюстрируют приведенные там же примеры. Механизм функционирования **Unique** подобен механизму генерации имен для локальных переменных модуля (*процедуры*). Простой пример иллюстрирует один из вариантов программной реализации **Unique** посредством процедуры **Un**, чей исходный код с примерами приводятся ниже, тогда как завершает фрагмент полезная процедура, вызов которой **Unique1[x,y]** возвращает уникальное имя в строчном формате, которое зависит от второго аргумента либо его отсутствия, одновременно присваивая данному имени значение *x*.

```
In[855]:= Un[x_] := Module[{a, If[{x} == {}, Symbol["$" <> ToString[$ModuleNumber]],
    a[y_] := If[StringQ[y], Symbol[y <> ToString[$ModuleNumber]],
    If[Head[y] == Symbol, Symbol[ToString[y] <> "$" <> ToString[$ModuleNumber]], y]];
    If[ListQ[x], Map[a, Flatten[x]], a[x]]]

In[856]:= {Un[], Un[S], Un["G"], Un[{x, y, z}], Un[V]}
Out[856]= {$4599, S$4600, G4601, {x$4602, y$4602, z$4602}, V$4603}

In[870]:= Unique1[x_, y_] := Module[{a = Unique[y], b, b = ToString[a];
    ToExpression[ToString[a] <> "=" <> ToString1[x]]; b]

In[871]:= {Unique1[78, a], Unique1[460]}
Out[871]= {"a$5999", "$3"}
In[872]:= ToExpression[{"a$5999", "$3"}]
Out[872]= {78, 460}
```

По стандартной конструкции *?имя* можно получать информацию по всем символам с заданным *именем*, которые были сгенерированы внутри процедур (*модулей*) либо по



функции *Unique*, как иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[2099]:= n = 1; While[n < 5, Unique[{x, y, z}]; n++]
```

```
In[2100]:= ?x*
```

```
▼ Global`
```

```
x$4602
```

```
▼ AladjevProcedures`
```

```
x
```

```
x$
```

```
x$6012
```

```
x$6013
```

```
x$6014
```

```
x$6015
```

Таким образом, имена, сгенерированные процедурой (*модулем*) ведут себя точно так же, как другие имена относительно вычислений. Однако, эти имена носят *временный* характер, который определяет, что они должны быть полностью удалены из системы при отсутствии в них надобности. Поэтому, большинство имен, сгенерированных в модулях (*процедурах*), удаляется, если выполнение этих модулей закончено. Остаются только имена, возвращаемые процедурами (*модулями*) явно. Более того, вне модулей (*процедур*) их *локальные* переменные остаются неопределенными вне самих процедур, даже если в модулях они и получали начальные значения. Между тем, следует иметь ввиду, что использование имен формы *name\$nnnn* является соглашением *Mathematica* для генерируемых модулями (*процедурами*) локальных переменных. Вследствие чего во избежание конфликтных ситуаций с генерируемыми пакетом именами указанной формы пользователю не рекомендуется использовать такого формата имена в своих программах. При этом, необходимо иметь ввиду, что переменные, сгенерированные процедурами (*модулями*), являются уникальными только в течение текущего сеанса и определяющая их *нумерацию* переменная **\$ModuleNumber** в начале каждого сеанса с системой устанавливается в свое изначальное значение. В частности, при сохранении пользовательского выражения, именнованного именем формата *name\$nnnn* в файле с последующим его чтением, даже в другом сеансе с системой могут вполне возникать конфликтные ситуации упомянутого типа. Естественно, в случае необходимости из такой ситуации имеется выход путем *переопределения* переменной **\$ModuleNumber** в текущем сеансе *Mathematica*. Между тем, использовать переопределения переменной **\$ModuleNumber** следует крайне осторожно во избежание возможных конфликтных ситуаций в текущем сеансе [101]. Механизм использования *локальных* переменных при вызове модуля в целом сводится к следующему. После генерации символов *name\$nnnn*, представляющих данные *локальные* переменные, посредством функции **With** в самом теле модуля (*процедуры*) делается замена *локальных* переменных представляющими их сгенерированными символами после чего тело модуля (*процедуры*) вычисляется [101]. Между тем, следует иметь в виду, что указанного типа подстановок не производится, если тело не кодируется явно, а вызывается при обращении к модулю (*процедуре*). И в этом случае требуется применение к телу модуля функции **Evaluate** системы. Так как функция **Module** имеет атрибут **HoldAll**, то тело модуля, как правило, сохраняется в *невычисленном* виде до тех пор, пока *определение* модуля не будет вычислено. В любом случае генерация символов, отвечающих *локальным* переменным, и их вставка в тело модуля производятся лишь при *обращении* к модулю, но не при *определении* модуля (*процедуры*) в *Input*-параграфе текущего сеанса системы *Mathematica*.

*Локальные* переменные *Math*-процедур допускают присвоение им *начальных* значений

в форме произвольных выражений, включая выражения, чьи значения могут зависеть от фактических аргументов, получаемых при вызове, либо от внешних переменных:

```
In[2043]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], If[PrimeQ[y],  
NextPrime[x], z]]}, a*(x + y)]
```

```
In[2044]:= z = 78; {G[7, 460], G[6, 13], G[70, 65]}
```

```
Out[2044]= {215287, 133, 10530}
```

Между тем, на уровне *локальных* переменных отсутствует возможность немедленного (без выполнения предложений тела процедуры) выхода из процедуры, например, в случае вычисления присваиваемых локальным переменным начальных выражений, как это иллюстрирует достаточно простой фрагмент, исключение составляет использование вызова **Abort[]**, который инициирует возврат процедурой значения **\$Aborted**:

```
In[2306]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], Return[x]]}, a*(x + y)]
```

```
In[2307]:= G[78, 460]
```

```
Out[2307]= 41964
```

```
In[2308]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], Defer[G[x]]]}, a*(x + y)]
```

```
In[2309]:= G[78, 460]
```

```
Out[2309]= 538 G[78]
```

```
In[2310]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], Abort[]]}, a*(x + y)]
```

```
In[2311]:= G[78, 460]
```

```
Out[2311]= $Aborted
```

К понятию *модулей* (процедур) в контексте механизма *локальных* переменных довольно тесно примыкают объекты типа *блоков* (*blocks*), чья организация имеет следующий вид

**Block[{a, b, c, ...}, Тело]** – вычисляется *Тело*, используя *локальные* значения для {a, b, c, ...}

**Block[{a = a0, b = b0, c = c0, ...}, Тело]** – вычисляется *Тело* при начальных значениях для локализованных в блоке переменных {a, b, c, ...}

В модульной структуре локальные переменные являются таковыми по определению, тогда как в блочной структуре переменные, определенные локальными, действуют в пределах только блока. При этом, если им в блоке не присваивается значений, то они принимают значения *одноименных* внешних по отношению блока переменных, тогда как в случае присвоения им значений в блоке, значения *одноименных* переменных вне блока остаются без изменения. Этим механизмы локальных переменных процедур и блоков довольно существенно разнятся. Именно поэтому, говоря о процедурах типов {"Module", "Block"}, мы должны анализировать *блочные* объекты на предмет характера их *локальных* переменных, а именно: отсутствие *локальных* переменных или наличие у каждой локальной переменной *начального* значения говорит о том, что объект может быть рассмотрен в качестве процедуры, получающей при вызове информацию лишь от своих фактических аргументов. Именно на таком принципе строятся наши тесты, рассмотренные выше, на проверку блока быть в качестве фактической процедуры.

Итак, общим правилом для блочной структуры является принцип – расположенные вне блока переменные, до блока и после блока сохраняют свои значения, выступая по отношению к блоку *глобальными*, тогда как в самом блоке одноименные переменные

могут весьма произвольно изменять свои значения согласно требуемому алгоритму. Как уже отмечалось выше, локальная переменная **a** в структуре типа **Module**[{**a**}, *Тело*] соотносится с *уникальным* символом, модифицируемым всякий раз, когда этот модуль используется; этот символ отличен от глобального имени **a**. Тогда как переменная **a** в структуре типа **Block**[{**a**}, *Тело*] является *глобальной* вне блока, в процессе выполнения блока может принимать любые значения, а по выходе из блока восстанавливает свое значение, которое она имела на входе в блок.

Если в случае **Module**-конструкции локальные переменные в теле процедуры сугубо временные, то в **Block**-конструкции они таковыми не рассматриваются. Не вдаваясь в детали, только отметим, что для обеспечения робастности процедур рекомендуется программировать их, вообще говоря, на основе именно **Module**-конструкций, или на основе **Block**-конструкций, для которых отсутствуют локальные переменные или все локальные переменные имеют начальные значения. Между тем, следует отметить, и в рамках модульной структуры вполне может быть реализован механизм *локализации* глобальных переменных, подобно используемому в блочной структуре; два варианта предложены в [101]. Более того, возможны и некоторые другие варианты реализации механизма *локализации* глобальных переменных, используемого как блоками, так и на основе других подходов, один из которых рассматривается в [101]. В этой связи вполне уместно отметить, существует довольно простой и универсальный механизм работы с глобальными переменными в теле процедур, сохраняющий их значения на момент входа в процедуру и выхода из нее. Формально его суть можно наглядно представить на основе следующей достаточно простой схемы, а именно:

**Q = Ex; P[x\_, y\_ ...] := Module[{a = Q, ...}, Тело[Q, ...]; Return[{Q = a, Res}][[-1]]]**

Пусть вне тела процедуры **P** некоторая используемая ею *глобальная* относительно нее переменная **Q** получила значение **Ex**. Локальная переменная **a** процедуры получает в качестве начального значение **Ex**, сохраняя его до каждого потенциального выхода из процедуры. Далее алгоритм, реализуемый телом процедуры, может использовать **Q**-переменную произвольным образом, и только каждый возможный выход процедуры **P** наряду с возвратом результата (**Res**) должен обеспечить присвоение переменной **Q** ее исходного значения **Ex** до входа в процедуру. Довольно простой пример наглядно иллюстрирует описанный механизм использования глобальных переменных в теле процедуры на локальном уровне:

```
In[2054]:= Q = 78; P[x_] := Module[{a = Q, Res}, Q = 460; Res = Q + x; {Q = a, Res}][[-1]]
In[2055]:= {Q, P[70]}
Out[2055]= {78, 530}
```

Таким образом блочные конструкции позволяют достаточно эффективно определять «*окружения*», в которых можно временно изменять значения глобальных переменных. Выражения, содержащиеся в теле блочной конструкции или сгенерированные в ней, в течение выполнения конструкции будут использовать значения, определенные для локальных переменных в блоке. И только при выходе из блочной конструкции такие локализованные переменные вновь восстанавливают свои значения до входа в блок. В общем понимании, блочные структуры служат в качестве неких «*областей*» текущего

сеанса, в которых допустимо изменение значений переменных, находящихся *вне* этих областей, *без изменения* их значений вне таких областей, т.е. производится своего рода *локализация* глобальных переменных сеанса в некоторых областях вычислительного пространства в целом. Данная возможность использовалась достаточно эффективно в целом ряде средств, составляющих наш пакет *AVZ\_Package\_1* [120]. При этом, *блочная* структура неявно используется в реализации целого ряда функций *Mathematica* – **Do**, **Table**, **Product**, **Sum** и др., главным образом, итеративного типа в целях локализации переменных индексирования, как весьма наглядно иллюстрирует простой пример:

```
In[2057]:= n := 70; {{Sum[n^2, {n, 16}], n}, {Product[n, {n, 24}], n}}
Out[2057]= {{1496, 70}, {620448401733239439360000, 70}}
```

Как правило, любая переменная, определенная пользователем в текущем сеансе, если не указано противного, рассматривается пакетом в качестве глобальной переменной. Однако, в некоторых случаях требуется на какой-то период локализовать глобальную переменную, что вполне успешно позволяет сделать *блочная* структура. В то же время следует еще раз отметить, *локализованные* в блоке переменные только тогда являются таковыми, если им в блоке присваиваются значения, иначе их значения в блоке будут совпадать со значениями одноименных с ними переменных, внешних относительно блока. Таким образом, если локализованной переменной в блоке не было определено начальных значений, то *реальной локализации* для такой переменной не производится. Теперь вновь имеет смысл обратиться к сравнению *блоков* и *модулей*, имея ввиду, что в целом ряде случаев имеет смысл создавать *процедурные* объекты на основе *блочной* и *модульной* организации. Итак, в общем смысле под *процедурными* объектами в системе *Mathematica* вполне можно полагать созданные как на основе *модульной*, так и *блочной* организаций при условии выполнения *блочной* организацией указанных условий – отсутствие у нее *локальных* переменных или наличие у *каждой* локальной переменной начального значения. Представим несколько средств манипулирования локальными переменными модулей и блоков, играющие достаточно существенную роль в задачах продвинутого *процедурного* программирования. В качестве таких объектов выступают процедуры модульного и блочного типов, и функции традиционного типа.

Прежде всего, весьма желательно располагать средством для определения локальных переменных блоков и модулей, отсутствующим в числе стандартных средств системы *Mathematica*. В качестве такого средства можно рассматривать процедуру **Locals**, чьи исходный код с примерами применения представляет следующий фрагмент. Данная процедура предназначена для вычисления *локальных* переменных блоков и модулей.

```
In[2218]:= M[x_, y_] := Module[{a = 78, b = 460}, (x + y)*(a + b)]; M[x_] := x;
M[x_Integer] := Block[{}, x^2]; M[x_Integer, y_ /; ListQ[y]] := Block[{a, b = 78}, x^2];
P[x_] := Module[{a = Q, Res}, Q = 460; Res = Q + x; {Q = a, Res}][[-1]]

In[2219]:= Locals[x_ /; ModuleQ[x] || BlockQ[x] || ProcQ[x] || QFunction[x], R_] :=
Module[{a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], c, d = {}, k = 1, p, t, Sg},
Sg[y_String] := Module[{h = 1, v = {}, j, s = "", z = StringLength[y] - 1},
Label[avz]; For[j = h, j <= z, j++, s = s <> StringTake[y, {j, j}];
If[! SameQ[Quiet[ToExpression[s]], $Failed] &&
```

```

StringTake[y, {j + 1, j + 1}] == ",", v = Append[v, s]; h = j + 3; s = ""; Goto[avz]]];
v = Append[v, s <> StringTake[y, {-1, -1}]];
Map[If[Quiet[StringTake[#, {1, 2}]] === " ", StringTake[#, {3, -1}], #] &, v]];
c = Flatten[Map[Map3[StringJoin, #, {" := ", " = "}] &, b]];
c = Map[StringReplace[#, Map[Rule[#, ""]] &, c]] &, a];
For[k, k <= Length[a], k++, p = c[[k]]; If[SuffPref[p, "Module[{", 1], t = 8,
If[SuffPref[p, "Block[{", 1], t = 7, t = 0; d = Append[d, "Function"]]];
If[t != 0, d = Append[d, SubStrSymbolParity1[StringTake[p, {t, -1}], "{", "}"]][[1]]];
Continue[]];
d = Map[StringReplace[#, {"{" -> {"$$$78$$$"}, " -> ", $$$78$$$"}, " -> " = $$$78$$$"}] &, d];
d = Map[If[MemberQ[{"Function", "{}"}, #], #, Sg[StringTake[#, {2, -2}]]] &, d];
d = Map[If[FreeQ[Quiet[ToExpression[#]], $Failed], #, StringJoin1[#]] &, d];
d = Map[If[# === {"", "{}", #} &, Mapp[StringReplace, d, {"$$$78$$$" -> "", "\\" -> ""}]]];
Map[Remove, Names["$$$78$$$*"]]; d = If[Length[d] == 1, Flatten[d], d];
If[{R} != {} && ! HowAct[R], If[d === {"{}"}, R = {}, {b, k, R} = {d, 1, {}];
While[k <= Length[b], p = b[[k]]; R = Append[R, If[MemberQ[{"{}", "Function"}, p], p,
If[StringQ[p], If[StringFreeQ[p, "="], {p, "No"}, StringSplit[p, "="]],
Map[If[StringFreeQ[#, "="], {#, "No"}, StringSplit[#, "="]] &, p]]]; k++];
R = If[NestListQ[R] && Length[R] == 1, R[[1]], R]]; If[d === {"{}"}, {}, d]]

In[2220]:= Locals[M]
Out[2220]= {"a", "b = 78"}, {"a = 78", "b = 460"}, {"", "Function"}
In[2221]:= Locals[P]
Out[2221]= {"a = Q", "Res"}
In[2222]:= Map[Locals, {ModuleQ, Definition2, Locals}]
Out[2222]= {"a", "atr = Quiet[Attributes[x]]", "b", "c", "d", "h"}, {"a = PureDefinition[x]", "b"},
{"a", "b = Attributes[x]", "c"}, {"a = Flatten[{PureDefinition[x]}]",
"b = Flatten[{HeadPF1[x]}]", "c", "d = {}", "k = 1", "p", "t", "Sg"}]
In[2223]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], If[PrimeQ[y],
NextPrime[x], z]], a*(x + y)}; Locals[G]
Out[2223]= {"a = If[PrimeQ[x], NextPrime[y], If[PrimeQ[y], NextPrime[x], z]]"}
In[2288]:= Locals[P, t]
Out[2288]= {"a = Q", "Res"}
In[2289]:= t
Out[2289]= {"a", "Q"}, {"Res", "No"}]
In[2290]:= Locals[M, t1]
Out[2290]= {"a", "b = 78"}, {"a = 78", "b = 460"}, {"", "Function", "Function"}
In[2291]:= t1
Out[2291]= {{{"a", "No"}, {"b", "78"}}, {"a", "78"}, {"b", "460"}}, {"", "Function", "Function"}
In[2292]:= Z[x_] := x; Z[x_, y_] := x + y; Z[x_, y_, z_] := x + y + z; Locals[Z, t2]
Out[2292]= {"Function", "Function", "Function"}

```

```

In[2293]:= t2
Out[2293]= {"Function", "Function", "Function"}
In[2294]:= Locals[G, t3]
Out[2294]= {"a" = If[PrimeQ[x], NextPrime[y], If[PrimeQ[y], NextPrime[x], z]]"}
In[2295]:= t3
Out[2295]= {"a", "If[PrimeQ[x], NextPrime[y], If[PrimeQ[y], NextPrime[x], z]]"}
In[2296]:= B[x_] := Module[{a}, x]; Locals[B, t4]
Out[2296]= {"a"}
In[2297]:= t4
Out[2297]= {"a", "No"}
In[2298]:= V[x_] := Module[{a, b, c, d}, x]; Locals[V, t5]
Out[2298]= {"a", "b", "c", "d"}
In[2299]:= t5
Out[2299]= {"a", "No"}, {"b", "No"}, {"c", "No"}, {"d", "No"}
In[2300]:= B1[x_] := Module[{}, x]; Locals[B1, t6]
Out[2300]= {}
In[2301]:= t6
Out[2301]= {}

```

Представленная процедура **Locals** по функциональным возможностям как одноименную процедуру, так и процедуру **Locals1**, рассмотренные в [101] и пакете [90]. Вызов процедуры **Locals[x]** возвращает список, элементы которого в строчной форме представляют локальные переменные вместе с их начальными значениями блока или модуля *x*. В то время как вызов **Locals[x, y]** со вторым необязательным аргументом *y* – неопределенной переменной – обеспечивает возврат через *y* дополнительно или простой 2-элементный список, либо список *ListList*-типа с 2-элементными подсписками, чьи первые элементы определяют имена локальных переменных блока/модуля *x* в строчном формате, тогда как вторые – приписанные им начальные значения в строчном формате; отсутствие начальных значений определяется символом "No". Если объект *x* не имеет локальных переменных, вызов процедуры **Locals[x, y]** возвращает пустой список, т.е. {}, такой же результат возвращается через второй аргумент. Более того, на типичной функции вызов процедуры возвращает значение "Function". Данная процедура очень весьма широко используется в задачах манипулирования локальными переменными.

```

In[2152]:= StringJoin1[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}] :=
Module[{a = x, b = Length[x], c = "", k = 1},
While[k <= b - 1, c = c <> a[[k]] <> ", "; k++]; c <> a[[-1]]]

In[2153]:= StringJoin1[{"Avz", "Agn", "Vsv", "Art", "Kr"}]
Out[2153]= "Avz, Agn, Vsv, Art, Kr"
In[2154]:= StringJoin1[{"Avz", 78, x + y, "Art", Sin}]
Out[2154]= StringJoin1[{"Avz", 78, x + y, "Art", Sin}]

```

При этом, для реализации данной процедуры наряду с нашими средствами **BlockQ**, **GenRules**, **Map3**, **HeadPF**, **HowAct**, **Mapp**, **ModuleQ**, **NestListQ**, **ProcQ**, **PureDefinition**, **QFunction**, **SubStrSymbolParity1** и **SuffPref**, рассматриваемыми и в настоящей книге,

выяснилась целесообразность определить одну несложную процедуру **StringJoin1**, с целью специальной обработки списков строк, которая является одной модификацией стандартной функции **StringJoin**. При этом, вызов **StringJoin1[x]** возвращает результат последовательной конкатенации элементов-строк списка *x*, разделенных запятой, как весьма наглядно иллюстрирует пример предыдущего фрагмента наряду с исходным кодом данной процедуры. Процедура относится к группе средств, манипулирующих со строчными структурами, однако рассмотрена именно здесь в контексте процедуры **StringJoin1**. Процедура **StringJoin1** имеет достаточно широкий спектр приложений.

Между тем, в целом ряде случаев требуется определить лишь список имен локальных переменных безотносительно приписанных им *начальных* значений. Для этого может быть использована процедура **Locals1**, замещающая две прежние процедуры **Locals1** и **Locals2** [90,101]. Вызов **Locals1[x]** возвращает список имен локальных переменных в строчном формате блока либо модуля *x*; в случае отсутствия локальных переменных для объекта *x* вызов процедуры возвращает пустой список, т.е. {}. Более того, в случае одноименного объекта *x* возвращается *вложенный* список, элементы которого взаимно однозначны с подьобъектами *x*, согласно их порядку при применении к *x* процедуры **PureDefinition**. Следующий фрагмент представляет исходный код процедуры **Locals1** наряду с наиболее типичными примерами ее использования.

```
In[2187]:= Locals1[x_;/; ModuleQ[x] || BlockQ[x] || ProcQ[x] || QFunction[x]] :=
      Module[{a, b = {}, c, k = 1, Kr},
        Kr[y_List] := Module[{d = {}, v = Flatten[y], j = 1},
          While[j <= Length[v]/2, d = Append[d, v[[2*j - 1]]]; j++]; d];
        ClearAll[a]; Locals[x, a];
        If[NestListQ1[a], For[k, k <= Length[a], k++, c = a[[k]];
          b = Append[b, If[MemberQ[{"", "Function"}, c], c, Kr[c]]];
          If[StringQ[PureDefinition[x]], Flatten[b], b], Kr[a]]]

In[2188]:= Locals1[P]
Out[2188]= {"a", "Res"}
In[2189]:= Locals1[G]
Out[2189]= {"a"}
In[2190]:= Locals1[M]
Out[2190]= {"a", "b"}, {"a", "b"}, {"", "Function", "Function"}
```

Для примеров использовались средства, приведенные во фрагменте несколько выше. Тут же отметим, что процедура **Locals1** представляет одно из наиболее эффективных средств в задачах обработки *локальных* переменных блоков и модулей. В качестве еще одного полезного средства такого типа можно отметить процедуру **Locals2**, которую следует предварить во многих случаях полезной процедурой, предназначенной для тестирования одноименных объектов, имеющих несколько определений различного типа. Вызов процедуры **QBlockMod[x]** возвращает **True**, если определения объекта *x* имеют тип {*Module*, *Block*}, и **False** в противном случае. Процедура предполагает, что при наличии среди определений *x* определений другого типа, такой объект в целом не может рассматриваться объектом типа {*Module*, *Block*}. Это позволяет выделять из

одноименных объектов объекты именно типа *{Module, Block}*. Фрагмент представляет исходный код процедуры **QBlockMod** с примерами ее типичного использования.

```
In[2086]:= M[x_, y_] := Module[{a = 78, b = 460}, (x + y)*(a + b)]; M[x_] := x;
      A[m_, n_] := Module[{a = 42.78, b = {m, n = 78}}, h*(m + n + p)/(a + b)];
      A[m_] := Block[{a = 42.78, b = {m, n = 78}, q, t}, h*(m + n + p)/(a + b)]

In[2106]:= QBlockMod[x_] := Module[{a = Flatten[{PureDefinition[x]}], b, c = True, k = 1},
      If[MemberQ[{"System", $Failed}, a[[1]]], False,
      b = Flatten[{HeadPF[x]}]; While[k <= Length[a],
      If[! SuffPref[StringReplace[a[[k]], b[[k]] <> " := " -> "", 1],
      {"Module[{", "Block[{", 1}, c = False; Break[]]; k++]; c]]

In[2090]:= QBlockMod[M]
Out[2090]= False
In[2105]:= Map[QBlockMod, {ProcQ, A, ToString1, StrStr, 78, Sin}]
Out[2105]= {True, True, True, False, False, False}
```

В контексте предыдущей процедуры **QBlockMod** нижеследующая процедура **Locals2** на объектах *x* типа *{Module, Block}* возвращает списки их локальных переменных без приписанных им начальных значений в строчном формате. На объектах *x* отличного от указанных типа вызов возвращается невычисленным. Фрагмент представляет код процедуры с примерами ее применения; примеры взяты из предыдущего фрагмента.

```
In[2109]:= Locals2[x_ /; QBlockMod[x]] := Module[{a = Flatten[{PureDefinition[x]}],
      b = Flatten[{HeadPF[x]}], c = {}, d, p, h = {}, k = 1, j = 1},
      While[k <= Length[a], c = Append[c, d = StringReplace[a[[k]],
      Mapp[Rule, Map[b[[k]] <> " := " <> # &, {"Module[{", "Block[{", ""}, 1];
      Quiet[SubStrSymbolParity[d, {"{", "}", 1}][[-1]]]; k++];
      While[j <= Length[c], p = c[[j]]; p = Map[ToString, ToExpression[
      StringReplace[p, {" := " -> "$$78$", " := " -> "$$78$ -> "}]];
      p = If[Length[p] == 1, p[[1]], p]; p = Map[If[StringFreeQ[#, " := "],
      StringReplace[#, "$$78$" -> ""], StringReplace[StringTake[#,
      {1, Flatten[StringPosition[#, " := "][[1]] - 1}], "$$78$" -> ""]] &, p];
      h = Append[h, p]; j++]; If[Length[h] == 1, h[[1]], h]]

In[2110]:= Locals2[A]
Out[2110]= {"a", "b"}, {"a", "b"}, {"a", "b", "q", "t"}
In[2111]:= Locals2[M]
Out[2111]= Locals2[M]
```

В целом ряде случаев появляется необходимость динамического расширения списка локальных переменных у блока/модуля, который активирован в текущем сеансе, без изменения самого объекта на носителе. Данную задачу решает процедура **ExpLocals**, представленная следующим фрагментом. Успешный вызов процедуры **ExpLocals[x, y]** возвращает список *локальных* переменных с приписанными начальными значениями в строчном формате, на которые расширены локальные переменные объекта *x*. При



этом, в общем случае этот список может быть меньше заданного при вызове списка *y* (либо и вообще пустым), так как из него исключаются переменные, имеющиеся в объекте *x* в качестве формальных аргументов либо локальных переменных.

```
In[2057]:= ExpLocals[P_ /; ModuleQ[P] || BlockQ[P], L_ /; ListQ[L] &&
DeleteDuplicates[Map[StringQ, L]] == {True}] :=
Module[{a = Flatten[{PureDefinition[P]}][[1]], b = Locals1[P],
c = Args[P, 78], d, p, p1, h, Op = Options[P], Atr = Attributes[P]},
d = Map[If[StringFreeQ[#, {" = ", "="}], #, StringSplit[#, {" = ", "="}][[1]]] &, L];
p = Locals[P]; h = MinusList1[d, Flatten[{b, c}]]; If[h == {}, Return[{}]];
h = Flatten[Map[Position[d, #] &, h]]; d = Join[p, c = Map[L[[#]] &, h]];
ToExpression["ClearAllAttributes[" <> ToString[P] <> "]; ClearAll[P];
ToExpression[StringReplace[a, ToString[p] -> ToString[d], 1]];
If[Op != {}, SetOptions[P, Op]]; SetAttributes[P, Atr]; c]

In[2058]:= Avz[x_] := Module[{a = 75, b, c}, a + x^2]; SetAttributes[Avz, Protected];
Agn[x_] := Module[{}, {x}]; Z[x_ /; IntegerQ[x]] := Module[{a, b, c, d}, {a, b, c, d}][[x]]

In[2059]:= ExpLocals[Agn, {"x", "a = c + d", "b", "Art = 24", "Sv", "Kr = 16"}]
Out[2060] = {"a = c + d", "b", "Art = 24", "Sv", "Kr = 16"}

In[2061]:= Definition[Agn]
Out[2061] = Agn[x_] := Module[{a = c + d, b, Art = 24, Sv, Kr = 16}, {x}]

In[2062]:= ExpLocals[Avz, {"x", "a = c + d", "b", "Art = 24", "Sv", "Kr = 16"}]
Out[2062] = {"Art = 24", "Sv", "Kr = 16"}

In[2063]:= Definition[Avz]
Out[2063] = Attributes[Avz] = {Protected}
Avz[x_] := Module[{a = 75, b, c, Art = 24, Sv, Kr = 16}, a + x^2]

In[2064]:= ExpLocals[Avz, {"x", "a = c + d", "b", "Art = 24", "Sv", "Kr = 16"}]
Out[2064] = {}

In[2065]:= ExpLocals[Z, {"m = 78", "n = 460", "p = 71"}]
Out[2065] = {"m = 78", "n = 460", "p = 71"}

In[2066]:= Definition[Z]
Out[2066] = Z[x_ /; IntegerQ[x]] := Module[{a, b, c, d, m = 78, n = 460, p = 71}, {a, b, c, d}][[x]]
```

Фрагмент содержит исходный код процедуры **ExpLocals** с примерами ее применения к весьма простым процедурам **Agn**, **Avz** и **Z** с целью расширения их списка *локальных* переменных. Часть из которых имеет начальные значения; при этом, во второй процедуре список локальных переменных пуст, тогда как для первой имеется непустое пересечение объединенного списка формальных аргументов и локальных переменных со списком переменных, на которые следует расширить список *локальных* переменных процедуры. Если объединенный список совпадает со списком *y*, то вызов процедуры возвращает пустой список, т.е. {}, не меняя исходного объекта *x* в текущем сеансе. Следует иметь в виду, элементы списка *y* необходимо кодировать в строчном формате во избежание присвоения им значений одноименных переменных текущего сеанса и/или вычислений согласно приписанных им *начальным* значениям. Результат модификации исходного объекта *x* сохраняет опции и атрибуты исходного объекта.

Известно, что активация в текущем сеансе модуля или блока в поле *имен* переменных системы добавляет все их *локальные* переменные, как иллюстрирует простой пример:

```
In[2050]:= Mb[x_] := Block[{Art = 24, Kr = 16, Sv = 45}, x]; Mb[78];
In[2051]:= B[x_] := Block[{Art1 = 24, Kr1 = 16, Sv1 = 45}, x]; B[78];
In[2052]:= Names["*"]
Out[2052]= {"Art", "Art1", "B", "Kr", "Kr1", "Mb", "Sv", "Sv1"}
```

Поэтому экономное использование *локальных* переменных является довольно важной задачей. Между тем, в процессе программирования блоков/модулей вполне реально появление т.н. *лишних* локальных переменных. Процедура **RedundantLocals**, которая базируется на процедуре **ProcBMQ**, в определенной степени решает данную задачу.

```
In[2171]:= RedundantLocals[x_ /; ProcQ[x] || ModuleQ[x] || BlockQ[x]] := Module[{a, b,
    c, p, g, k = 1, j, v, t = {}, z = ""}, {a, b} = {PureDefinition[x], Locals1[x]};
    If[StringQ[a], If[b == {}, True, p = Map[#[[1]] &, StringPosition[a, {"} = ", " := "}]];
    p = Select[p, ! MemberQ[{"\""}, " \"]], StringTake[a, {# - 2, #}]] &;
    c = Map[Map3[StringJoin, #, {" := ", " = "}]] &, b];
    g = Select[b, StringFreeQ[a, Map3[StringJoin, #, {" := ", " = "}]] &;
    While[k <= Length[p], v = p[[k]]; For[j = v, j >= 1, j--, z = StringTake[a, {j, j}] <> z;
        If[! SameQ[Quiet[ToExpression[z]], $Failed], t = Append[t, z]]; z = ""; k++];
        t = MinusList[g, Flatten[Map[StrToList, t]]];
        If[t == {}, t, p = Select[Map[" " <> # <> "[" &, t], ! StringFreeQ[a, #] &; g = {}];
        For[k = 1, k <= Length[p], k++, v = StringPosition[a, p[[k]]];
            v = Map[#[[2]] &, v]; z = StringTake[p[[k]], {2, -2}]; c = 1;
            For[j = c, j <= Length[v], j++,
                For[b = v[[j]], b <= StringLength[a], b++,
                    z = z <> StringTake[a, {b, b}];
                If[! SameQ[Quiet[ToExpression[z]], $Failed], g = Append[g, z];
                    c = j + 1; z = StringTake[p[[k]], {2, -2}]; Break[{}]]];
        MinusList[t, Map[HeadName[#] &, Select[g, HeadingQ1[#] &]]],
        "Object <" <> ToString[x] <> "> has multiple definitions"]]
```

```
In[2172]:= Map[RedundantLocals, {ProcQ, Locals1, RedundantLocals}]
Out[2172]= {"h"}, {"a"}, {}
In[2173]:= Vsv[x_, y_] := Module[{a, b, c = 78, d, h}, b = 460; d[z_] := z^2 + z + 460;
    h[t_] := Module[{}, t]; d[c + b] + x + y + h[x*y]]
In[2174]:= RedundantLocals[Vsv]
Out[2174]= {"a"}
In[2175]:= Map[RedundantLocals, {ProcQ, Globals, RedundantLocals, Locals, Locals1}]
Out[2175]= {"h"}, {}, {}, {}, {"a"}
In[4184]:= RedundantLocalsM[x_ /; ProcQ[x] || QFunction[x]] := Module[{b = {}, d, k = 1,
    a = Flatten[{PureDefinition[x]}], c = "$Art24Kr16$"},
```

```

While[k <= Length[a], d = c <> ToString[x]; ToExpression[c <> a[[k]]];
  b = Append[b, If[QFunction[d], "Function", RedundantLocals[d]]];
  ToExpression["ClearAll[" <> d <> "]; k++]; If[Length[b] == 1, b[[1]], b]]
In[4185]:= Vsv[x_, y_] := Module[{a, b, c = 78, d, h}, b = 460; d[z_] := z^2 + z + 460; h[t_] :=
  Module[{}, t]; d[c+b]+x+y+h[x*y]]; Vsv[x_, y_, z_] := Module[{a, b, c = 78, d, h}, a = 6;
  d[p_] := z^2 + p + 460; h[t_] := Module[{}, t]; d[c + b] + x + y + h[x*y*z]]; Vsv[x_] := x
In[4186]:= RedundantLocalsM[Vsv]
Out[4186]= {"a", "Function", {"b"}}

```

Вызов процедуры **RedundantLocals**[W] возвращает список локальных переменных в строчном формате блока или модуля W, которые процедура рассматривает *лишними* в том контексте, что им не присваивалось как начальных значений, так и значений в теле объекта W, или данные переменные не являются именами внутренних функций либо модулей/блоков, т.е. определенных внутри тела объекта W. Между тем, в такой список могут попадать и локальные переменные, используемые в качестве аргумента при вызове той либо иной функции в теле объекта W, как, например, это имеет место для нашей процедуры **Locals1**, использующей вызов **Locals**[x, a], в котором через 2-й необязательный аргумент *a* (локальную переменную) – неопределенную переменную – обеспечивается возврат дополнительного результата. Отметим, что данная процедура подобно предыдущей процедуре **ProcBMQ** ориентирована на *одиночные* объекты, чьи определения уникальны, тогда как на *одноименных* объектах возвращается сообщение "*Object <x> has multiple definitions*". Между тем, в отличие от предыдущей **ProcBMQ** процедура **RedundantLocals** вполне успешно обрабатывает и объекты, содержащие в своем теле определения типичных функций (*с заголовками*), модулей и блоков. Однако данный момент в качестве недостатка процедуры **ProcBMQ** рассматривать не совсем правомочно, ибо данная процедура тестирует только допустимость модуля/блока на его процедурность в ее классическом понимании, и наличие избыточных локальных переменных здесь не учитывается; более того, модуль согласно механизма обработки его локальных переменных является процедурой, а блок, как правило, не содержит в своем теле определений других блоков и модулей, его функция состоит в локализации требуемых переменных. Завершает фрагмент процедура **RedundantLocalsM**, которая расширяет **RedundantLocals** на одноименные объекты. Вызов **RedundantLocalsM**[W] на *единичном* объекте блок/модуль аналогичен вызову **RedundantLocals**[W], тогда как на функции возвращает **"Function"**; на *одноименном* объекте W блок/модуль/ функция возвращается список результатов применения **RedundantLocals** ко всем подобъектам типа блок/модуль объекта W и **"Function"** на традиционных функциях.

Между тем, результаты вызова обеих процедур **RedundantLocals** и **RedundantLocalsM** предполагают дополнительный анализ объекта W на *лишние* локальные переменные, т.е. процедура может рассматриваться в качестве достаточно эффективного средства для предварительного анализа блоков/модулей на предмет исключения избыточных локальных переменных. Как показывает наш опыт, в целом ряде случаев результаты применения процедур **RedundantLocals** и **RedundantLocalsM** вполне можно полагать *окончательными*, не требуя каких-либо особых исследований анализируемого объекта.

При этом, в среде программирования системы *Mathematica* вычисление определений блоков и модулей, содержащих дублированные *локальные* переменные, производится вполне корректно без инициирования каких-либо ошибочных или особых ситуаций, которые возникают только в момент вызова блока и модуля, иницилируя ошибочную ситуацию **Block::dup** и **Module::dup** соответственно, с возвратом вызова блока/модуля невычисленным. Между тем, сам механизм идентификации дублируемой локальной переменной неясен, т.к. *первыми* до нее в списке *локальных* переменных в определении процедуры **Proc** выступают *a* и *d*, как достаточно наглядно иллюстрирует следующий довольно простой фрагмент. С целью установления факта дублирования локальных переменных в определениях активированных в текущем сеансе объектов типа *модуль* или *блок* была создана процедура, чей вызов **DuplicateLocalsQ[P]** возвращает **True** при существовании в определении процедуры *P* дублирования локальных переменных, в противном случае возвращается **False**. При этом, в случае возврата **True** через второй необязательный аргумент *y* – неопределенную переменную – возвращается простой список либо список *ListList*-типа, чьи элементы определяют имена дублирующихся *локальных* переменных с их кратностями вхождений в список локальных переменных. Следующий фрагмент приводит исходный код процедуры **DuplicateLocalsQ** наряду с наиболее типичными примерами ее использования.

```
In[2060]:= Proc[x_] := Module[{a, y, d = {x}, h, c, h, d, a = 78, h = 460, c = 2013, t, t, a}, a + d]
In[2061]:= Proc[78, 460]
Module::dup: Duplicate local variable h found in local variable specification
{a, y, d={78, 460}, h, c, h, d, a=78, h=460, c=2013, t, t, a}. >>
Out[2061]= Module[{a, y, d = {78, 460}, h, c, h, d, a = 78, h = 460, c = 2013, t, t, a}, a + d]
In[2062]:= Blok[x_] := Block[{a, y, d = {x}, h, c, h, d, a = 78, h = 460, c = 2013, t, t, a}, a + d]
In[2063]:= Blok[78, 460]
Block::dup: Duplicate local variable h found in local variable specification
{a, y, d={78, 460}, h, c, h, d, a=78, h=460, c=2013, t, t, a}. >>
Out[2063]= Block[{a, y, d = {78, 460}, h, c, h, d, a = 78, h = 460, c = 2013, t, t, a}, a + d]
In[2064]:= DuplicateLocalsQ[P_;/; BlockModQ[P], y_] := Module[{a, b = Locals1[P]},
If[b == {}, False, b = If[NestListQ[b], b[[1]], b];
a = Select[Gather2[b], #[[2]] > 1 &];
If[a == {}, False, If[{y} != {} && ! HowAct[y], y = a]; True]]
In[2065]:= {DuplicateLocalsQ[Proc, y], y}
Out[2065]= {True, {{a", 3}, {d", 2}, {h", 3}, {t", 2}}}
In[2066]:= {DuplicateLocalsQ[Blok, y1], y1}
Out[2066]= {True, {{a", 3}, {d", 2}, {h", 3}, {t", 2}}}
In[2067]:= B[x_, y_] := Module[{}, x + y]; {DuplicateLocalsQ[B, y2], y2}
Out[2067]= {False, y2}
```

В определении процедуры **DuplicateLocalsQ[P]** наряду со стандартными средствами применяются наши средства *BlockModQ*, *Gather2*, *HowAct*, *Locals1*, *NestListQ*, которые рассматриваются в настоящей книге и представлены в пакете *AVZ\_Package\_1* [120].

## 8.7. Глобальные переменные процедурных и блочных объектов, и средства манипулирования с ними в системе *Mathematica*

Относительно *Maple*-процедур *Mathematica*-процедуры имеют более ограниченные возможности как по механизму *глобальных* переменных, так и по возврату результатов своего выполнения. Если в *Maple*-процедуре *глобальной* полагается *любая* переменная, продекларированная в *global*-секции описания либо не получившая значений в теле процедуры по оператору присвоения ":", или по процедуре *assign* (с релиза 11 данное утверждение потеряло силу в том смысле, что присвоение по *assign* не делает переменную локальной), то в *Mathematica*-процедуре *глобальными* полагаются все те переменные, которые явно не определены в качестве *локальных*. Следующий пример достаточно наглядно иллюстрирует вышесказанное, а именно:

```
In[2128]:= Sv[x_] := Module[{}, y := 71; z = 66; {y, z}]
In[2129]:= {y, z} = {42, 47}; {Sv[2013], y, z}
Out[2129]= {{71, 66}, 71, 66}
```

Следовательно, любое переопределение в процедуре (модуле либо блоке при выполнении соответствующих условий) системы *Mathematica* *глобальной* переменной автоматически переопределяет *одноименную* с ней переменную вне области процедуры, что требует существенно большей внимательности на предмет недопущения возможных особых и нежелательных ситуаций, чем в аналогичной ситуации с *Maple*-процедурами. Итак, уровень обеспечения *робастности* программных средств в *Mathematica*, использующих процедуры, представляется нам несколько более низким относительно упомянутого уровня системы компьютерной математики *Maple*.

Следует отметить, что *Mathematica* допускает определение глобальных переменных процедур посредством довольно простого приема модифицирования механизма для тестирования фактических аргументов в момент вызова процедуры, как это довольно наглядно иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[2133]:= Art[x_ /; If[! IntegerQ[x], h = 78; True, h = 460; True], y_] :=
Module[{a = 2013}, x + y + h + a]
In[2134]:= {Art[78, 460], Art[16.24, 460]}
Out[2134]= {1058, 651.24}
In[2135]:= Kr[x_, y_] := Module[{a = If[IntegerQ[x], 78, 460]}, x + y + a]
In[2136]:= {Kr[7.5, 460], Kr[120, 250]}
Out[2136]= {927.5, 448}
In[2137]:= Sv[x_, y_] := Module[{a = If[IntegerQ[x] && PrimeQ[y], 78, 460]}, x + y + a]
In[2138]:= {Sv[78, 460], Kr[16, 128]}
Out[2138]= {998, 222}
In[2139]:= H[x_: 78, y_, z_] := Module[{}, x + y + z]
In[2140]:= {H[120, 230, 340], H[120, 230], H[120]}
Out[2140]= {690, 428, H[120]}
```

В целом ряде случаев данный механизм оказывается достаточно полезным, тогда как для пакета *Maple* подобная модификация механизма тестирования типов фактических аргументов в момент вызова процедур недопустима. Естественно, подобный механизм допустим и для *Maple*, когда определяемый в виде теста (булевой функции) аргументов алгоритм кодируется не в самом заголовке процедуры, а задается отдельным типом с активацией его в текущем сеансе. В таком случае используется стандартный формат *x::test* для тестируемого *x*-аргумента. Естественным образом возможно определять и начальные значения локальных переменных в точке вызова процедуры в зависимости от полученных значений фактических аргументов, как иллюстрируют два примера предыдущего фрагмента. Наконец, если *Maple* не допускает присвоения значений по умолчанию промежуточным аргументам процедуры, то *Mathematica* допускает, что достаточно существенно расширяет возможность определения процедур; последний пример предыдущего фрагмента дает определенную иллюстрацию сказанному.

Между тем, следует иметь в виду, что при использовании механизма возвратов через глобальные переменные требуется повышенная внимательность, чтобы не возникало конфликтных ситуаций с одноименными глобальными переменными вне процедуры. А так как процедуры предполагается использовать многократно в различных сессиях пакета, то возврат через глобальные переменные нецелесообразен. Однако в некоторой степени данная проблема разрешаема при использовании, в частности, специальных имен, вероятность появления которых в текущей сессии пакета крайне маловероятна. В случае создания ряда процедур нашего пакета *AVZ\_Package* [90] для *Mathematica* был использован именно подобный подход для возврата результатов через глобальные переменные. В целом ряде случаев этот подход оказывается довольно эффективным.

Аналогично случаю локальных представляет интерес вопрос определения наличия в процедуре глобальных переменных; в первом случае вопрос решают рассмотренные выше процедуры **Locals** и **Locals1**, во втором – две процедуры **Globals** и **Globals1**. Так, естественным дополнением к **Locals1** выступает процедура, вызов которой **Globals[P]** возвращает список глобальных переменных в строчном формате процедуры *P*. Более того, из наших средств процедура использует процедуры **Locals1**, **DefFunc**, **ExprOfStr** и **ListListQ**. Следующий фрагмент представляет исходный код процедуры **Globals** с наиболее типичными примерами ее использования.

```
In[944]:= Globals[P_/, ProcBMQ[P]] := Module[{b = ToString1[DefFunc[P]], c, d = {}, p,
    a = If[P === ExprOfStr, {}, Sort[Locals1[P]]], g = {}, p, k = 1},
  If[a == {}, Return[{}], c = StringPosition[b, {" := ", " = "}]][[2 ;; -1]];
  For[k, k <= Length[c], k++, p = c[[k]];
  d = Append[d, ExprOfStr[b, p[[1]], -1, {" ", ",", "\"", "!", "{"}]];
  For[k = 1, k <= Length[d], k++, p = d[[k]];
  If[p != "$Failed" && p != " ", g = Append[g, If[StringFreeQ[p, {"(", "}"}, p,
    StringSplit[StringReplace[p, {"(" -> "", "}" -> ""}], ", "], Null]];
  g = Flatten[g]; d = {}; For[k = 1, k <= Length[g], k++, p = g[[k]];
  d = Append[d, If[StringFreeQ[p, {"(", "}"}, p,
    StringTake[p, {1, Flatten[StringPosition[p, "("]][[1]] - 1]}]]; g = d;
```

```

d = {}; For[k = 1, k <= Length[g], k++, p = g[[k]];
d = Append[d, StringReplace[p, {" " -> "", " " -> ""}]];
d = Sort[Map[StringTrim, DeleteDuplicates[Flatten[d]]]];
Select[d, ! MemberQ[If[ListListQ[a], a[[1]], a], #] &]]

In[945]:= Sv[x_, y_] := Module[{a, b = 78, c = 460}, a = (x^2 + y^2)/(b + c); {z, h} = {a, b};
t = z + h; t]; GS[x_] := Module[{a, b = 78, c = 42}, Kr[y_] := Module[{}, y^2 + Sin[y]];
a = x^2; {z, h, p} = {a, b, 16}; t = z + h*Kr[16] - Cos[x + Kr[78]]; t];
Ar[x_] := Module[{a, b = 78, c = 42, Kr, z}, Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2;
{z, h, p} = {a, b, 16}; t = z + h*Kr[16] - Cos[x + Kr[78]]; t]

In[946]:= Map[Globals, {Locals1, Locals, Globals, ProcQ, ExprOfStr, GS, DefFunc, Sv, Ar}]
Out[946]= {"d", "j", "v"}, {"h", "R", "s", "v", "z"}, {}, {}, {"h", "Kr", "p", "t", "z"}, {},
{"h", "t", "z"}, {"h", "p", "t"}]

```

Следует отметить, что процедура **Globals[P]** под *глобальными* переменными понимает имена объектов в теле процедуры, которым производятся присвоения по операторам `":="`, `"="` и которые отличны от локальных переменных *главной* процедуры. Поэтому вполне возможна ситуация, когда *локальная* переменная подпроцедуры в процедуре **P** может быть определена процедурой **Globals[P]** в качестве *глобальной* переменной, как это наглядно иллюстрирует пример применения **Globals** к нашим процедурам **Locals** и **Locals1**, содержащим по вложенной подпроцедуре **Sg** и **Kr** соответственно. В таком случае требуется дополнительное исследование либо процедура **Globals** может быть расширена и на данный случай, интересный в качестве полезного упражнения. Для решения этой проблемы, в частности, можно воспользоваться процедурами **Locals1** и **Globals** в сочетании с процедурой **MinusList**. Один из достаточно простых вариантов обобщения процедуры **Globals** (*базирующийся на самой Globals*) на случай вложенных процедур может быть представлен довольно простой процедурой **Globals1**, чей вызов **Globals1[x]** возвращает *список глобальных* переменных в строчном формате процедуры **x**; при этом, в качестве аргумента **x** может выступать как процедура, не содержащая в своем теле подпроцедур *различного уровня вложенности*, так и процедура, содержащая такие подпроцедуры. Возможен также ряд других довольно интересных и полезных в практическом программировании подходов для решения такой задачи. Следующий фрагмент приводит исходный код самой процедуры **Globals1** наряду с примерами ее наиболее типичного использования.

```

In[786]:= Globals1[P_;/; ProcQ[P]] := Module[{a = SubProcs[P], b, c, d = {}},
{b, c} = Map[Flatten, {Map[Locals1, a[[2]]], Map[Globals, a[[2]]]}];
MinusList[DeleteDuplicates[c], b]]

In[787]:= Map[Globals1, {Locals1, Locals, Globals, ProcQ, ExprOfStr, GS, DefFunc, Sv, Ar}]
Out[787]= {}, {}, {}, {}, {}, {"h", "Kr", "p", "t", "z"}, {}, {"h", "t", "z"}, {"h", "p", "t"}]
In[788]:= P[x_, y_] := Module[{a, b, P1, P2}, P1[z_, h_] := Module[{m, n}, T = z^2 + h^2; T];
P2[z_] := Module[{P3}, P3[h_] := Module[{}, Q = h^4; Q]; P3[z]];
V = x*P2[x] + P1[x, y] + P2[y]; V]; P1[x_] := Module[{}, {c, d} = {78, 460}; c*d + x];
Map[Globals1, {P, P1}]

```

```

Out[788]= {"Q", "T", "V"}, {"c", "d"}
In[789]:= Sv[x_, y_] := Module[{a, b = 78, c = 460}, a = (x^2 + y^2)/(b + c); {z, h} = {a, b};
                                         t = z + h; gs = t^2]; Globals1[Sv]

Out[789]= {"gs", "h", "t", "z"}

In[790]:= LocalsGlobals[x_/, ProcQ[x]] := {Locals[x], Globals1[x]}; LocalsGlobals[Sv]

Out[790]= {"a", "b = 78", "c = 460"}, {"gs", "h", "t", "z"}

```

В частности, первый пример приведен для сравнения результатов вызовов процедур **Globals** и **Globals1** на одном и том же кортеже аргументов; так, если в первом случае в качестве *глобальных* были определены переменные подпроцедур, то во втором такие переменные в качестве *глобальных* уже не выступают. Вызов функции **LocalsGlobals[x]** возвращает вложенный список, чей первый элемент – список *локальных* переменных с *начальными* значениями, если таковы имеются, в строчном формате, тогда как второй элемент определяет список *глобальных* переменных процедуры (блок либо модуль) *x*.

Довольно полезной при работе с процедурами представляется процедура **ExtrNames**, чей исходный код с примерами применения представлен следующим фрагментом.

```

In[2120]:= ExtrNames[x_/, ProcQ[x]] := Module[{a = BlockToModule[x], b, c, d, f, p = {}, g,
                                         k = 1}, {f, a} = {ToString[Locals[x]], Locals1[x]};
                                         {b, c} = {HeadPF[x], PureDefinition[x]};
g = StringReplace[c, {b <> " := Module[" -> "", ToString[f] <> ", " -> ""}];
d = Map[If[ListQ[#], #[[1]], #] &, StringPosition[g, {" := ", " = "}]];
For[k, k <= Length[d], k++, p = Append[p, ExtrName[g, d[[k]], -1]]];
p = Select[p, # != "" &]; {a, Complement[a, p], Complement[p, a]}]

In[2121]:= GS[x_] := Block[{a = 78, b, c}, b = 460; c = 6; x = a + b + c; x]
In[2122]:= ExtrNames[GS]
Out[2122]= {"a", "b", "c"}, {"a"}, {"x"}
In[2123]:= ExtrNames[ProcQ]
Out[2123]= {"a", "atr", "b", "c", "d", "h"}, {"atr", "h"}, {}
In[2124]:= ExtrNames[ExtrNames]
Out[2124]= {"a", "f", "b", "c", "d", "p", "g", "k"}, {"a", "b", "c", "f", "k"}, {}

In[2127]:= Globals2[x_/, ProcQ[x] | ModuleQ[x] | BlockQ[x]] := ExtrNames[x][[3]]

In[2128]:= GS[h_] := Module[{a = 78, b, c}, b = 460; c = 6; x = a + b + c; x + h]
In[2129]:= VG[h_] := Block[{a = 78, b, c}, b = 460; c = 6; x = a + b + c; y = h^2]
In[2130]:= Map[Globals2, {GS, VG, ProcQ, Tuples1, TestArgsTypes, LoadFile}]
Out[2130]= {"x"}, {"x", "y"}, {}, {"Res"}, {"$TestArgsTypes"}, {"$Load$Files$"}

```

Вызов процедуры **ExtrNames[x]** возвращает вложенный 3-элементный список, первый элемент которого задает список всех локальных переменных процедуры *x* в строчном формате, второй – список локальных переменных процедуры *x* в строчном формате, которым в теле процедуры *x* будут производиться присваивания значений, в то время как третий определяет список глобальных переменных, которым в теле процедуры *x*



производятся присваивания значений по операторам {":=", "="}. Завершает фрагмент достаточно простая функция **Globals2**, базирующаяся на предыдущей процедуре и в определенной степени расширяющая возможности рассмотренных процедур **Globals** и **Globals1** на процедуры произвольного типа; вызов функции **Globals2[x]** возвращает список имен в строчном формате глобальных переменных процедуры *x*.

Между тем, предыдущие средства, корректно тестирующие наличие у модуля/блока глобальных переменных, определяемых присвоениями по операторам {" := ", " = "}, не эффективны в случаях, когда определения тестируемых модулей/блоков используют присвоения типа {*a, b, c, ...*} {*= | :=*} {*a1, b1, c1, ...*} или *a[[k]]* {*= | :=*} *b*, попросту игнорируя их. Этот недостаток устраняет процедура **LocalsGlobals1**, чей вызов **LocalsGlobals1[x]** возвращает вложенный 3-х элементный список, *первый* подсписок которого содержит имена в строчном формате локальных переменных, *второй* – локальные переменные с начальными значениями в строчном формате, и *третий* – глобальные переменные в строчном формате блока/модуля *x*. На аргументе *x* типа, отличного от блок/модуль, вызов возвращается невычисленным. Следующий фрагмент представляет исходный код процедуры **LocalsGlobals1** наряду с типичными примерами ее использования.

```
In[2063]:= LocalsGlobals1[x_ /; QBlockMod[x]] := Module[{c = "", d, h = {}, k = 1, j, p, G, L,
    a = Flatten[{PureDefinition[x]}][[1]], b = Flatten[{HeadPF[x]}][[1]]},
    b = StringReplace[a, {b <> " := Module[" -> "", b <> " := Block[" -> "", 1];
    While[k <= StringLength[b], d = StringTake[b, {k, k}]; c = c <> d;
    If[StringCount[c, "{"] == StringCount[c, "}"], Break[]; k++];
    b = StringReplace[b, c <> ", " -> "", 1]; L = StrToList[StringTake[c, {2, -2}]];
    d = StringPosition[b, {" := ", " = "}; d = (#1[[1]] - 1 &)/@ d;
    For[k = 1, k <= Length[d], k++, c = d[[k]]; p = "";
    For[j = c, j >= 1, j--, p = StringTake[b, {j, j}] <> p;
    If[! Quiet[ToExpression[p]] == $Failed &&
    StringTake[b, {j - 1, j - 1}] == " ", h = Append[h, p]; Break[]];
    G = Flatten[(If[StringFreeQ[#1, "{"], #1, StrToList[StringTake[#1, {2, -2}]]] &)/@
    (StringTake[#1, {1, Quiet[Check[Flatten[StringPosition[#1, "{"][[1]], 0] - 1] &)/@ h];
    b = (If[StringFreeQ[#1, "="], #1, StringTake[#1,
    {1, Flatten[StringPosition[#1, "="][[1]] - 1] &)/@ L;
    d = DeleteDuplicates[Flatten[(StringSplit[#1, " ] &)/@ MinusList[G, b]]];
    d = Select[d, ! Quiet[SystemQ[#1]] && ! MemberQ[Flatten[{"\\", "#", "\\", "", "+", "-",
    ToString/@ Range[0, 9]}], StringTake[#1, {1, 1}] &]; {Select[b, ! MemberQ[ToString/@
    Range[0, 9], StringTake[#1, {1, 1}] &], L, MinusList[d, b]}]

In[2064]:= M[x_, y_] := Module[{a = 78, b = 460, c = {v, r}}, h = x*y*a*b; m = 71; t := (a + b);
    g[[6]] = 71; t[z_] := a; {g, p} = {65, 70}; {k, j} := {42, 71}; x + y]
In[2065]:= {h, m, t, g, p, k, j} = {1, 2, 3, 4, 5, 6, 7}; {LocalsGlobals1[M], {h, m, t, g, p, k, j}}
Out[2065]= {{{"a", "b", "c"}, {"a = 78", "b = 460", "c = {v, r}"}, {"h", "m", "t", "g", "p", "k", "j"}},
    {1, 2, 3, 4, 5, 6, 7}}
```

```

In[2066]:= Sv[x_, y_] := Module[{a, b = 78, c = {n, m}}, a = (x^2 + y^2)/(b + c); {z, h} = {a, b};
          t = z + h; gs = t^2]; LocalsGlobals1[Sv]
Out[2066]= {{{"a", "b", "c"}, {"a", "b = 78", "c = {n, m}"}, {"z", "h", "t", "gs"}}}
In[2067]:= Vt[x_, y_] := Module[{a, b = 78, c = {n, m, {42, 71}}}, a = (x^2 + y^2)/(b + c);
          {z, h} = {a, b}; t = z + h; gs = t^2]; LocalsGlobals1[Vt]
Out[2067]= {{{"a", "b", "c"}, {"a", "b = 78", "c = {n, m, {42, 71}}"}, {"z", "h", "t", "gs"}}}
In[2195]:= LocalsGlobalsM[x_/, QBlockMod[x]] := Module[{b = "$$78$", c, d = {}, k = 1,
          a = Flatten[{PureDefinition[x]}]},
          While[k <= Length[a], c = b <> ToString[x]; ToExpression[b <> a[[k]]];
          d = Append[d, LocalsGlobals1[c]]; ToExpression["Clear[" <> c <> ""]; k++];
          If[Length[d] == 1, d[[1]], d]]
In[2196]:= M[x_, y_] := Module[{a = 78, b = 460, c}, h = x*y*a*b; m = 71; t := (a + b);
          g[[6]] = 71; t[z_] := a; {g, p} = {65, 70}; x + y]; M[x_] := Block[{a, b}, y = x];
          M[x_] := Block[{a, b}, {y, z} := {a, b}]; P1[x_] := Module[{a, b = {78, 460}},
          {c, d} = {p, q}; {a, b, h, g} = {42, 47, 67, 78}; c*d + x]
In[2197]:= LocalsGlobalsM[M]
Out[2197]= {{{{"a", "b", "c"}, {"a = 78", "b = 460", "c"}, {"h", "m", "t", "g", "p", "k", "j"}},
          {{{{"a", "b"}, {"a", "b"}, {"y"}}, {{{{"a", "b"}, {"a", "b"}, {"y", "z"}}}}}
In[2198]:= LocalsGlobalsM[P1]
Out[2198]= {{{{"a", "b"}, {"a", "b = {78, 460}"}, {"c", "d", "h", "g"}}}

```

Между тем, процедура **LocalsGlobals1** корректно работает лишь с блоками/модулями, имеющими уникальные определения, т.е. с объектами, отличными от одноименных. Тогда как процедура **LocalsGlobalsM** расширяет процедуру **LocalsGlobals1** на случай одноименных блоков/модулей; вызов **LocalsGlobalsM[x]** возвращает список вложенных 3-элементных списков формата, аналогичного формату результатов возврата вызовов **LocalsGlobals1[x]**, элементы которого взаимно однозначны с *подъобъектами* *x*, согласно их порядку при применении к объекту *x* процедуры **PureDefinition**. На аргументах *x* типа, отличного от блока/модуля, вызов процедуры **LocalsGlobalsM[x]** возвращается *невычисленным*. Завершает предыдущий фрагмент исходный код процедуры наряду с примерами ее применения. Между тем, следует иметь в виду, возвращаемый список глобальных переменных не содержит кратных имен, хотя идентичные имена и могут принадлежать различного типа объектам, как весьма наглядно иллюстрирует первый пример к процедуре **LocalsGlobals1** в предыдущем фрагменте, в котором в качестве глобальных переменных дважды выступает символ **"t"**. Действительно, ниже простой пример весьма наглядно иллюстрирует вышесказанное, а именно:

```

In[2253]:= t[z_] := z; t := (a + b); Definition[t]
Out[2253]= t := (a + b)
          t[z_] := z

```

Поэтому в случае необходимости требуется проведение дополнительного анализа на предмет определения типов *глобальных* переменных, используемых блоком/модулем.

Процедура **LocalsGlobals1** наряду со стандартными использует также наши средства **HeadPF**, **MinusList**, **PureDefinition**, **QBlockMod**, **StrToList1**. Процедура имеет целый ряд применений при программировании различных задач, прежде всего, системных.

Выше нами были определены т.н. *активные* глобальные переменные как глобальные переменные, которым в объекте типа *блок/модуль* производятся присвоения значений, в то время как под *пассивными* глобальными переменными мы понимаем глобальные переменные, отличные от аргументов, чьи значения только используются в объектах указанного типа. В этой связи весьма интересными представляются средства, которые позволяют вычислять *пассивные* глобальные переменные для пользовательских *блоков* и *модулей*. Одно из подобных средств – процедура **BlockFuncModVars**, которая решает даже более общую задачу. Следующий фрагмент представляет исходный код наряду с наиболее типичными примерами использования процедуры **BlockFuncModVars**.

```
In[2117]:= BlockFuncModVars[x_/, BlockFuncModQ[x]] := Module[{c = Args[x, 78], d, t,
    a = If[QFunction[x], {}, LocalsGlobals1[x]], s = {"System"},
    u = {"Users"}, b = Flatten[{PureDefinition[x]}][[1]], h = {}},
  d = ExtrVarsOfStr[b]; If[a == {}, t = Map[If[Quiet[SystemQ[#]], s = Append[s, #],
    If[BlockFuncModQ[#], u = Append[u, #], h = Append[h, #]]] &, d];
  {s, u = Select[u, # != ToString[x] &], c, MinusList[d, Join[s, u, c, {ToString[x]}]],
    Map[If[Quiet[SystemQ[#]], s = Append[s, #],
    If[BlockFuncModQ[#], u = Append[u, #], h = Append[h, #]]] &, d];
  {Select[s, ! MemberQ[{"$Failed", "True", "False"}, #] &],
  Select[u, # != ToString[x] && ! MemberQ[a[[1]], #] &], c, a[[1]], a[[3]],
  Select[h, ! MemberQ[Join[a[[1]], a[[3]], c, {"System", "Users"}], #] &]]]

In[2118]:= A[m_, n_, p_/, IntegerQ[p], h_/, PrimeQ[h]] := Module[{a = 42.78,
  h*(m + n + p)/a + StringLength[ToString1[z]]/(Cos[c] + Sin[d])]

In[2119]:= BlockFuncModVars[A]
Out[2119]= {"System", "IntegerQ", "PrimeQ", "Module", "StringLength", "Cos", "Sin",
  {"Users", "ToString1"}, {"m", "n", "p", "h"}, {"a"}, {}, {"z", "c", "d"}}

In[2120]:= BlockFuncModVars[StringReplace$]
Out[2120]= {"System", "String", "Module", "StringLength", "StringPosition", "Characters",
  "If", "While", "Length", "Quiet", "MemberQ", "StringTake", "StringReplacePart",
  "Append"}, {"Users"}, {"S", "s1", "s2"}, {"a", "b", "c", "k", "p", "L", "R"}, {}, {}

In[2121]:= BlockFuncModVars[BlockFuncModVars]
Out[2121]= {"System", "Module", "Flatten", "If", "Quiet", "Append", "Select", "ToString",
  "MemberQ", "Join"}, {"Users", "QBlockMod", "LocalsGlobals1", "SystemQ",
  "Args", "PureDefinition", "ExtrVarsOfStr", "BlockFuncModQ"}, {"x"},
  {"a", "s", "u", "b", "c", "d", "h"}, {}, {}

In[2122]:= BlockFuncModVars[LocalsGlobals1]
Out[2122]= {"System", "Module", "Flatten", "StringReplace", "StringJoin", "Block", "While",
  "StringLength", "StringTake", "If", "StringCount", "Break", "StringPosition",
  "For", "Length", "Quiet", "ToExpression", "Append", "StringFreeQ", "Check",
```

```
"DeleteDuplicates", "StringSplit", "Select", "MemberQ", "ToString", "Range",
{"Users", "QBlockMod", "PureDefinition", "HeadPF", "StrToList1", "MinusList",
"SystemQ"}, {"x"}, {"c", "d", "h", "k", "j", "p", "a", "b", "G", "L"}, {}, {}
```

Вызов процедуры **BlockFuncModVars[x]** возвращает вложенный 6-элементный список, чей первый элемент – список системных функций, используемых блоком/модулем *x*, первый элемент которого **"System"**, тогда как остальные имена системных функций в строчном формате; второй элемент – список пользовательских средств, используемых блоком/модулем *x*, первый элемент которого **"Users"**, тогда как остальные определяют имена средств в строчном формате; третий элемент – список формальных аргументов блока/модуля; четвертый элемент – список локальных переменных блока/модуля *x* в строчном формате; пятый элемент – список *активных* глобальных переменных блока/модуля *x* в строчном формате; наконец, шестой элемент определяет список *пассивных* глобальных переменных блока/модуля *x* в строчном формате. Тогда как на функции *x* вызов **BlockFuncModVars[x]** возвращает вложенный 4-элементный список, первый элемент которого – список системных функций, используемых функцией *x*, первый элемент которого **"System"**, тогда как остальные имена системных функций; второй элемент – список пользовательских средств, используемых функцией *x*, чей первый элемент **"Users"**, тогда как остальные определяют имена средств в строчном формате; третий элемент – список формальных аргументов функции *x* и четвертый элемент – список глобальных переменных функции *x* в строчном формате. Данная процедура обеспечивает структурный анализ *блоков/функций/модулей* пользователя в следующих разрезах, а именно: (1) используемые системные функции, (2) средства пользователя, активные в текущем сеансе, (3) формальные аргументы, (4) локальные переменные, (5) активные глобальные переменные, (6) пассивные локальные переменные. Данное средство имеет ряд интересных приложений, прежде всего, системного характера.

Следующая процедура относится к группе средств, обрабатывающих строки, однако представлена именно здесь, так как весьма тесно связана с предыдущей процедурой; наряду с другими нашими средствами лежит в основе алгоритма **BlockFuncModVars**. Следующий фрагмент представляет исходный код процедуры **BlockFuncModVars** с наиболее типичными примерами ее использования.

```
In[2078]:= ExtrVarsOfStr[S_String, t_ /; MemberQ[{1, 2}, t], x___] := Module[{k, j, d = {}, p,
a = StringLength[S], q = Map[ToString, Range[0, 9]], h = 1, c = "",
L = Characters["!@#%^&*(){}:\\"\\| <>?~-=+[];:'. , 1234567890"],
R = Characters["!@#%^&*(){}:\\"\\| <>?~-=+[];:'. , "],
Label[G]; For[k = h, k <= a, k++, p = StringTake[S, {k, k}];
If[! MemberQ[L, p], c = c <> p; j = k + 1; While[j <= a, p = StringTake[S, {j, j}];
If[! MemberQ[R, p], c = c <> p, d = Append[d, c]; h = j; c = ""; Goto[G]; j++]];
d = Append[d, c]; d = Select[d, ! MemberQ[q, #] &];
d = Select[Map[StringReplace[#, {"+" -> "", "-" -> "", "_" -> ""}] &, d], # != "" &];
d = Flatten[Select[d, ! StringFreeQ[S, #] &]]; d = Flatten[Map[StringSplit[#, " " &], d]];
If[t == 1, Flatten, Sort][If[{x} != {}, Flatten, DeleteDuplicates]
```

```

[Select[d, ! MemberQ[{"\\", "#", ""}, StringTake[#, {1, 1}]] &]]]
In[2079]:= A[m_, n_, p_;/ IntegerQ[p], h_;/ PrimeQ[h]] := Module[{a = 42.78,
                                                                    h*(m + n + p)/a}
In[2080]:= ExtrVarsOfStr[Flatten[{PureDefinition[A]}][[1]], 2]
Out[2080]= {"a", "A", "h", "IntegerQ", "m", "Module", "n", "p", "PrimeQ"}
In[2081]:= G[x_, y_;/ IntegerQ[y]] := Module[{a, b = Sin[c + d], h}, z = x + y; V[m] + Z[n]]
In[2082]:= ExtrVarsOfStr[PureDefinition[G], 1, 78]
Out[2082]= {"a", "b", "c", "d", "G", "h", "IntegerQ", "m", "Module", "n", "Sin", "V", "x", "y", "z", "Z"}
In[2083]:= V[x_, y_;/ PrimeQ[y]] := Block[{a, b = 71/78, c = m*n}, If[x > t + w, x*y, S[x, y]]]
In[2084]:= ExtrVarsOfStr[PureDefinition[V], 2]
Out[2084]= {"a", "b", "Block", "c", "If", "m", "n", "PrimeQ", "S", "t", "V", "w", "x", "y"}
In[2085]:= F[x_] := a*x + Sin[b*x] + StringLength[ToString1[x + c]]; BlockFuncModVars[F]
Out[2085]= {"System", "Sin", "StringLength"}, {"Users", "ToString1"}, {"x"}, {"a", "b", "c"}
In[2086]:= ExtrVarsOfStr["G[x_] := Module[{Vg, H71}, Vg[y_] := Module[{}, y^3]", 1]
Out[2086]= {"G", "x", "Module", "Vg", "H71", "y"}
In[2087]:= ExtrVarsOfStr["(a + b)/(c + d) + Sin[c]*Cos[d + h]", 2]
Out[2087]= {"a", "b", "c", "Cos", "d", "h", "Sin"}
In[2088]:= ExtrVarsOfStr["(a + b)/(c + d) + Sin[c]*Cos[d + h]", 2, 78]
Out[2088]= {"a", "b", "c", "c", "Cos", "d", "d", "h", "Sin"}

```

Вызов процедуры **ExtrVarsOfStr**[*S*, *t*] при *t*=2 возвращает отсортированный и при *t*=1 неотсортированный список *переменных* в строчном формате, которые удалось извлечь из строки *S*; при отсутствии подобных переменных возвращается пустой список, т.е. {}. Вызов **ExtrVarsOfStr**[*S*, *t*, *x*] с 3-м необязательным аргументом *x* – *любое выражение* – возвращает список переменных, содержащихся в *S*, без приведения их кратностей к 1. Под *переменной* понимается *идентификатор*, которому можно присваивать значения:

```

In[2105]:= {a, b71, c$, $d, $125, AVZ} = {1, 2, 3, 4, 5, 6}
Out[2105]= {1, 2, 3, 4, 5, 6}

```

Опыт использования процедуры подтверждает ее достаточно высокую надежность в выборе переменных, которая довольно несложно настраивается на особые ситуации, возникающие в процессе ее выполнения. Для корректного применения **ExtrVarsOfStr** предполагается, что выражение *Expr*, определяемое строкой *S*, находится в *InputForm*-формате, т.е. *S* = **ToString**[**InputForm**[*Expr*]]. Процедура эффективно используется при манипуляциях с определениями пользовательских блоков, функций и процедур. Так, в предыдущей процедуре **BlockFuncModVars** она используется весьма существенно. В целом, процедура может использоваться и для анализа алгебраических выражений.

Наряду со стандартными механизмами *локальных* переменных система *Mathematica* в качестве *локальных* допускает механизм использования в теле процедур *глобальные* переменные текущего сеанса. Выше уже был представлен довольно полезный прием обеспечения использования таких *глобальных* переменных, не изменяющий значений одноименных переменных вне тела процедуры. Дополнительно к ранее описанному приему представим процедуру, автоматизирующую данный процесс конвертации на момент выполнения произвольной процедуры глобальных переменных в локальные

переменные процедуры. Подобная задача возникает, в частности, в том случае, когда требуется выполнить вызов процедуры, имеющей глобальные переменные, не меняя их значений во *внешней* относительно процедуры среде без изменения исходного кода процедуры в текущем сеансе. Иными словами, требуется выполнить вызов процедуры с *разделением* областей определения *глобальных* переменных текущего сеанса системы и одноименных с ними *глобальных* переменных собственно самой процедуры. Тогда как в других точках вызова таких ограничений не накладывается. Решает эту задачу процедура **GlobalToLocal**, вызов **GlobalToLocal[x]** которой обеспечивает конвертацию определения процедуры  $x$  в определение процедуры  $$$$x$ , в котором все глобальные переменные исходной процедуры  $x$  включены в состав локальных переменных; вызов процедуры возвращает имя активированной в текущем сеансе процедуры, у которой отсутствуют глобальные переменные. Тогда как вызов **GlobalToLocal[x, y]** со вторым необязательным аргументом  $y$  – неопределенной переменной – дополнительно через него возвращает вложенный список, *первый* элемент которого – подсписок локальных переменных и *второй* – подсписок глобальных переменных процедуры  $x$ . Процедура в целом ряде случаев решает задачу защиты переменных, внешних по отношению к  $x$ . Следующий фрагмент представляет исходный код процедуры **GlobalToLocal** наряду с наиболее типичными примерами ее использования.

```
In[2053]:= GlobalToLocal[x_ /; QBlockMod[x], y___] := Module[{a=LocalsGlobals1[x],b,c},
    If[Intersection[a[[1]], a[[3]]] == a[[3]] || a[[3]] == {}, x,
    b = Join[a[[2]], MinusList[a[[3]], a[[1]]];
    c = "$$$" <> StringReplace[PureDefinition[x], ToString[a[[2]]] -> ToString[b], 1];
    If[{y} != {} && ! HowAct[y], y = {a[[1]], a[[3]]};
    ToExpression[c]; Symbol["$$$" <> ToString[x]]]

In[2054]:= GS[x_] := Module[{a, b = 78, c = {m, n}}, Kr[y_] := Module[{}, y^2 + Sin[y];
    a = x^2; {z, h, p} = {a, b, 16}; t = z + h*Kr[16] - Cos[x + Kr[78]]; t]

In[2055]:= GlobalToLocal[GS]
Out[2055]= $$$GS
In[2056]:= $$$GS[x_] := Module[{a, b = 78, c = {m, n}, Kr, z, h, p, t}, Kr[y_] := Module[{}, y^2 +
    Sin[y]; a = x^2; {z, h, p} = {a, b, 16}; t = z + h*Kr[16] - Cos[x + Kr[78]]; t]
In[2057]:= {GlobalToLocal[GS, y], y}
Out[2057]= {$$$GS, {"a", "b", "c"}, {"Kr", "z", "h", "p", "t"}}
In[2058]:= LocalsGlobals1[$$$GS]
Out[2058]= {"a", "b", "c", "Kr", "z", "h", "p", "t"}, {"a", "b=6", "c={m,n}", "Kr", "z", "h", "p", "t"}, {}
```

Используемый процедурой **GlobalToLocal** алгоритм достаточно простой и состоит в следующем. В случае отсутствия у процедуры  $x$  *глобальных* переменных возвращается имя  $x$ , иначе на основе определения  $x$  формируется *определение* процедуры с именем  $$$$x$ , отличающееся от исходного лишь тем, что глобальные переменные  $x$  включены в состав локальных переменных процедуры  $$$$x$ . Затем определение активируется в текущем сеансе с возвратом имени  $$$$x$ , позволяя в текущем сеансе пакета выполнять процедуру  $$$$x$ , не изменяя значений глобальных переменных текущего сеанса. В то

же время представляет интерес задача конвертации блока и модуля  $x$  в объект того же типа, в котором *глобальные* относительно исходного объекта  $x$  переменные включены в состав локальных переменных выходного объекта  $x$  того же самого имени с теми же атрибутами и опциями. Эту задачу решает процедура **GlobalToLocalM**, исходный код которой с типичными примерами применения представляет следующий фрагмент.

```
In[2051]:= GlobalToLocalM[x_;/; QblockMod[x]] := Module[{d, h = "$$$", k = 1, n, p = {},
    a = Flatten[{PureDefinition[x]}], b = Attributes[x], c = Options[x]},
    While[k <= Length[a], d = a[[k]]; n = h <> ToString[k];
        ToExpression[h <> d]; GlobalToLocal[Symbol[n]];
        p = Append[p, PureDefinition["$$$$" <> n]];
        ToExpression["ClearAll[" <> n <> "]"]; k++];
    ClearAllAttributes[x]; ClearAll[x];
    ToExpression[Map[StringReplace[#, "$$$$$" -> "", 1] &, p]];
    SetAttributes[x, b]; If[c != {}, SetOptions[x, c]]; ]

In[2052]:= A[x_] := Block[{}, g = x; {m, n} = {6, 9}; A[x_, y_] := Module[{}, h = x + y; z = h*x];
    SetAttributes[A, {Listable, Protected}]; GlobalToLocalM[A]

In[2053]:= Definition[A]
Out[2053]= Attributes[A] = {Listable, Protected}
    A[x_] := Block[{g, m, n}, g = x; {m, n} = {6, 9}]
    A[x_, y_] := Module[{h, z}, h = x + y; z = h*x]

In[2054]:= GS[x_] := Module[{a, b = 78, c = {m, n}}, Kr[y_] := Module[{}, y^2 + Sin[y]];
    a = x^2; {z, h, p} = {a, b, 16}; t = z + h*Kr[16] - Cos[x + Kr[78]]; t];
    GS[x_, y_] := Block[{a, b = {78, 460}, c}, z = x + y; d = Art];
    SetAttributes[GS, {Protected}]; GlobalToLocalM[GS]

In[2055]:= Definition[GS]
Out[2055]= Attributes[GS] = {Protected}
    GS[x_] := Module[{a, b = 78, c = {m, n}, Kr, z, h, p, t},
        Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2; {z, h, p} = {a, b, 16};
        t = z + h*Kr[16] - Cos[x + Kr[78]]; t]
    GS[x_, y_] := Block[{a, b = {78, 460}, c, z, d}, z = x + y; d = Art]
```

Вызов процедуры **GlobalToLocalM[x]** возвращает *Null*, т.е. ничего, конвертируя блок либо модуль  $x$  в объект  $x$  того же типа и с теми же атрибутами и опциями, в котором *глобальные* переменные (если имелись) исходного объекта получают статус *локальных*. В случае одноименных объектов  $x$  процедура обеспечивает корректную конвертацию. Примеры фрагмента достаточно наглядно проясняют смысл подобной конвертации. Следует отметить, наш пакет *AVZ\_Package\_1* [120] содержит ряд и других средств для анализа процедур на предмет наличия в них локальных и глобальных переменных, а также манипулирования с аргументами, локальными и глобальными переменными объектов типов **{Block, Function, Module}**. Представленные здесь и в [120] средства для работы с *локальными* и *глобальными* переменными оказываются довольно полезными в процедурном программировании в среде системы *Mathematica*.

## 8.8. Атрибуты, опции и значения по умолчанию для аргументов пользовательских блоков, функций и модулей, дополнительные средства манипулирования с ними в среде системы *Mathematica*

Система *Mathematica* обеспечивает возможность присвоения переменным, например, именам блоков, функций и модулей специальных атрибутов, определяющих разные их свойства. В частности атрибут *Listable* для функции *F* определяет, что функция *F* будет автоматически применяться к элементам списка, который выступает в качестве ее аргумента. Текущая девятая версия системы *Mathematica* располагает списком из 19 атрибутов различного назначения, работу с которыми поддерживают 3 функции, форматы которых обеспечивают следующие операции, а именно:

**Attributes**[*x*] – возвращает список атрибутов, приписанных объекту *x*;

**Attributes**[*x*] = {*attr1*, *attr2*, ...} – приписывает объекту *x* атрибуты {*attr1*, *attr2*, ...};

**Attributes**[*x*] = {} – отменяет у объекта *x* все приписанные ему атрибуты;

**SetAttributes**[*x*, *attr*] – доопределяет атрибуты *attr1* для объекта *x*;

**ClearAttributes**[*x*, *attr*] – отменяет у объекта *x* все приписанные ему атрибуты *attr*.

Между тем, в целом ряде случаев данных средств недостаточно или они не настолько эффективны. Поэтому нами предложен ряд средств в данном направлении, которые расширяют вышеуказанные стандартные средства. Так как с течением времени новые версии *Mathematica* вполне могут как изменять стандартный набор атрибутов, так и расширять его, то вполне естественным является задача тестирования символа быть в качестве допустимого атрибута. Данную задачу и решает процедура **AttributesQ**, чей вызов **AttributesQ**[*x*] возвращает **True**, если *x* – список допустимых атрибутов текущей версии системы, и **False** в противном случае. Более того, вызов **AttributesQ**[*x*, *y*] со 2-м необязательным аргументом *y* – неопределенной переменной – возвращает через нее список элементов списка *x*, которые не являются атрибутами. Следующий фрагмент представляет исходный код процедуры с типичными примерами ее использования.

```
In[2111]:= AttributesQ[x_List, y___] := Module[{a, b = {}},
      Map[If[Quiet[Check[SetAttributes[a, #], $Failed]] === $Failed,
            b = Append[b, #] &, x];
      If[b != {}, If[{y} != {} && ! HowAct[y], y = b]; False, True]]

In[2112]:= {AttributesQ[{Listable, Agn, Protected, Kr, Art}, h], h}
Out[2112]= {False, {Agn, Kr, Art}}
In[2113]:= {AttributesQ[{Protected, Listable, HoldAll}, g], g}
Out[2113]= {True, g}
```

Данное средство в ряде системных приложений представляется довольно полезным. Выше уже было представлено расширение стандартной функции **Attributes** довольно простой функцией, чей вызов **Attributes1**[*x*] в отличие от стандартного на объектах *x*, отличных от *Symbol*, *String*, *HoldPattern*[*Symbol*] типа, возвращает **\$Failed** без вывода сообщений об ошибках, что в ряде случаев оказывается более предпочтительным для непрерывной программной обработки особых и аварийных ситуаций.



Как отмечалось выше, строгая дифференцировка блоков, функций и модулей в среде *Mathematica* выполняется не по их именам, как это принято в большинстве известных языков и систем программирования, а по их заголовкам. Именно по данной причине в целом ряде случаев продвинутого *процедурного* программирования возникает задача организации механизмов дифференцированной обработки таких объектов на основе их заголовков. Ряд таких средств представлен в настоящей книге, здесь мы определим два средства, обеспечивающие работу с атрибутами объектов на основе их заголовков. Следующий фрагмент представляет исходные коды для *двух* средств **DefAttributesH** и **AttributesH** с примерами применения. Вызов **DefAttributesH**[*x*, *y*, *z*, *p*, *h*, ...] возвращает **Null**, т.е. ничего, присваивая {*y*="Set"} или удаляя {*y*="Clear"} для объекта с заголовком *x* атрибуты, определенные аргументами {*z*, *p*, ...}. Тогда как при попытке присвоения или удаления *несуществующего* в текущей версии системы атрибута вызов процедуры возвращает список, чей *первый* элемент **\$Failed**, тогда как *второй* – список выражений, отличных от текущих атрибутов. Тогда как вызов функции **AttributesH**[*x*] возвращает список атрибутов, приписанных объекту с заголовком *x*.

```
In[2080]:= DefAttributesH[x_;/ HeadingQ[x], y_;/ MemberQ[{"Set", "Clear"}, y], z___] :=
           Module[{a}, If[AttributesQ[{z}, a = Unique[g]],
           ToExpression[y <> "Attributes[" <> HeadName[x] <>
           ", " <> ToString[{z}] <> "]", {$Failed, a}]]

In[2081]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[2082]:= DefAttributesH["M[x_, y_, z_]", "Set", Flat, Protected, Listable]
In[2083]:= Attributes[M]
Out[2083]= {Flat, Listable, Protected}
In[2084]:= DefAttributesH["M[x_, y_, z_]", "Set", AvzAgn]
Out[2084]= {$Failed, {AvzAgn}}

In[2130]:= AttributesH[x_;/ HeadingQ[x]] := Attributes1[Symbol[HeadName[x]]]
In[2131]:= AttributesH["M[x_, y_, z_]"]
Out[2131]= {Flat, Protected}

In[2473]:= ClearAllAttributes[x_] := Map[Quiet[ClearAttributes[#], Attributes[#]]; &,
                                           {x}][[1]]

In[2474]:= SetAttributes[G, {Flat, Protected}]; SetAttributes[V, {Protected}]
In[2475]:= ClearAllAttributes[G, V]; Attributes[G]
Out[2475]= {}
```

Завершает фрагмент весьма полезная функция, чей вызов **ClearAllAttributes**[*x*, *y*, *z*, ...] возвращает **Null**, т.е. ничего, отменяя все атрибуты, приписанные символам *x*, *y*, *z*, ... . Представленные средства наравне с ранее рассмотренными средствами **Attributes1** и **SetAttributes1** работы с атрибутами, которые наиболее актуальны для объектов типа {*Block*, *Function*, *Module*}, в ряде случаев достаточно полезны. При этом, эти средства можно довольно эффективно использовать при программировании и других средств различного назначения, в первую очередь, средств системного характера.

Механизм атрибутов *Mathematica* довольно эффективен как для защиты объектов от модификаций, так и для управления режимом обработки аргументов вызова блоков, функций и модулей. В частности, посредством присвоения процедуре либо функции атрибута *Listable* можно указать, что данная процедура либо функция должна быть применена автоматически ко всем фактическим аргументам как элементам списка. В следующем фрагменте представлена простая процедура, чей вызов **CallListable[x, y]** возвращает список значений **Map[x, Flatten[{y}]]**, где *x* – блок, функция или модуль от одного формального аргумента, и *y* – список либо последовательность фактических аргументов, которые могут быть и пустыми. Фрагмент представляет исходный код и наиболее типичные примеры использования процедуры **CallListable**.

```
In[2077]:= ToString[{a, b, c + d, 78, x*y, (m + n)*Sin[p - t]}]
Out[2077]= "{a, b, c + d, 78, x y, (m + n) Sin[p - t]}"

In[2078]:= CallListable[x_ /; SystemQ[x] || BlockFuncModQ[x], y___] :=
Module[{a = Attributes[x]}, If[MemberQ[a, Listable], x[Flatten[{y}]],
SetAttributes[x, Listable]; {x[Flatten[{y}]], ClearAttributes[x, Listable]}][[1]]]

In[2079]:= CallListable[ToString, {a, b, c + d, 78, x*y, (m + n)*Sin[p - t]}]
Out[2079]= {"a", "b", "c + d", "78", "x y", "(m + n) Sin[p - t]"}
In[2080]:= CallListable[ToString, a, b, c + d, 78, x*y, (m + n)*Sin[p - t]]
Out[2080]= {"a", "b", "c + d", "78", "x y", "(m + n) Sin[p - t]"}
In[2081]:= CallListable[ToString]
Out[2081]= {}
```

Подход, используемый процедурой **CallListable**, довольно эффективен и может быть использован в целом ряде программируемых в среде *Mathematica* приложений.

Определения блоков, функций и модулей пользователя в среде системы *Mathematica* позволяют в качестве условий и начальных значений для формальных аргументов, и начальных значений для локальных переменных использовать достаточно сложные конструкции, как это иллюстрирует следующий простой фрагмент, а именно:

```
In[4173]:= G[x_Integer, y_ /; {v[t_] := Module[{}, t^2], If[v[a] > 2013, True, False]}][[2]] :=
Module[{a = {g[z_] := Module[{}, z^3], If[g[b] < 2013, 71, 90]}][[2]]}, Clear[v, g]; x*y + a]

In[4174]:= {a, b} = {460, 78}; {G[42, 71], G[42, 460], G[0, 0]}
Out[4174]= {3072, 19410, 90}
In[4175]:= Map[Definition, {v, g}]
Out[4175]= {Null, Null}

In[4176]:= G[x_Integer, y_ /; {v[t_] := Module[{}, t^2], If[v[a] > 2013, True, False]}][[2]] :=
Module[{a = {g[z_] := Module[{}, z^3], If[g[b] < 2013, 71, 90]}][[2]]}, x*y + a]

In[4177]:= {a, b} = {460, 71}; {G[42, 71], G[42, 460], G[0, 0]}
Out[4177]= {3072, 19410, 90}
In[4178]:= Map[Definition2, {v, g}]
Out[4178]= {"v[t_] := Module[{}, t^2]", {}, {"g[z_] := Module[{}, z^3]", {}}}
```

Для возможности использования последовательности предложений, включая также и определения процедур, в качестве *условия* для формального аргумента *y* и *начального* значения для *локальной* переменной *a* использовался прием, базирующийся на списке. Последовательности предложений оформлялись как элементы списков со значением последнего их элемента в качестве *условия* и *начального* значения соответственно. При этом, если в теле *основной* процедуры *G* не выполнялось очистки символов *v* и *g* от их определений, процедуры *v* и *g* становились доступными в текущем сеансе, иначе нет. Этот вопрос решается в зависимости от поставленной задачи. Предыдущий фрагмент иллюстрирует сказанное. Вышепредложенный прием может достаточно эффективно применяться для процедурного программирования средств различного назначения, что иллюстрирует ряд процедур, представленных в настоящей книге несколько ниже.

Принимая во внимание важность аппарата *шаблонов* для создания пользовательских блоков, функций и модулей, рассмотрим данный вопрос несколько детальнее. Так, в качестве *альтернативного* представления шаблонов проверки системой используются конструкции вида  $\{Ш1 | Ш2 | Ш3 | \dots\}$ , где *Шk* – шаблоны проверки; так, в конструкции *x\_String | \_List* для *x*-выражения *допустимыми* являются лишь значения с заголовками  $\{String, List\}$ . Однако имеется ряд весьма существенных особенностей, которые здесь не рассматриваются. В наших книгах [100,101] показано, что в целом ряде случаев вместо применения поименованных *ША* эквивалентный им программный эквивалент более эффективен в качестве *фильтра* проверки фактических аргументов на допустимость.

Конструкция **Blank[]** (или в эквивалентной форме «\_») представляет шаблон для любого выражения, тогда как **Blank[H]** (или в эквивалентной форме «\_H») представляет шаблон для выражения с *H*-заголовком. При использовании непоименованных шаблонов они выполняют роль фиктивных аргументов функций; с другой стороны, использование непоименованного шаблона **BlankNullSequence[]** (либо в эквивалентной форме «\_\_\_»; 3 символа «\_») позволяет использовать любое число фактических аргументов при вызове пользовательской функции, включая их отсутствие, тогда как 2 символа «\_» позволяет использовать один или более фактических аргументов при вызове функции. Однако, в отсутствие необходимости в указании *отсутствия* фактических аргументов, нужно использовать 2 символа «\_» скорее, чем 3 символа «\_», как иллюстрирует пример:

In[2312]:= G[x\_, y\_] := a\*x + b + y

In[2313]:= {G[x, y], G[x, y, z, h, g, v, d, s, w, r, t]}

Out[2313]= {b + a x + y, b + d + g + h + r + s + t + v + w + a x + y + z}

Шаблон **BlankNullSequence[{ | H}]** можно определять для любой последовательности выражений, каждое из которых имеет *H*-заголовок. Относительно другого *ША* формы **BlankSequence[{ | H}]** (или в эквивалентной форме «{\_\_ | \_\_H}») имеет место сказанное выше но относительно шаблона **BlankNullSequence**. По шаблону **Optional[X\_, Y]** (или в эквивалентных формах *X:Y* и *X:\_Y*) представляется *X*-выражение, которое заменяется на *Y*-выражение в случае отсутствия первого, как иллюстрирует простой фрагмент:

In[2256]:= {G[43], G[42.68], G[14/21], G[], F[h]}

Out[2256]= {1849, G[42.68], G[2/3], 4624, F[h]}

In[2257]:= SV[x\_Integer: 43, y\_List: {68, 63, 43}] := (x + First[y])^2

```
In[2258]:= {SV[4, 6], SV[7, {3}], SV[], SV[{14, 21, 43, 63, 68}], SV[Null, {14, 21, 43, 63, 68}]}
Out[2258]= {SV[4, 6], 100, 12321, 3249, SV[Null, {14, 21, 43, 63, 68}]}
In[2261]:= Art[x_String: "rans", y_Integer: 72] := (420 + StringLength[x])*y^2
In[2262]:= {Art[42, 68], Art["IAN_2011", 14], Art["AVZ"], Art[420]}
Out[2262]= {Art[42, 68], 83888, 2192832, 74793600}
```

Такой *ША* позволяет определять значения «по умолчанию», если функция вызывается без фактических для формальных аргументов значений, что обеспечивает механизм определения фиктивных аргументов. Между тем, как следует из приведенного выше фрагмента, действие **Optional**-шаблона распространяется только на одновременное указание связанных с ним формальных аргументов фиктивными. С этим шаблоном достаточно тесно связана **Default**-функция, определяющая значения «по умолчанию» для формальных аргументов функции, получающих *фиктивные* значения. По вызову формата **Default[F, k]** задается значение «по умолчанию» для формальных аргументов *F*-функции, заданных «*\_*»-шаблоном; при кодировании второго аргумента значение «по умолчанию» присваивается *k*-у аргументу *F*-функции. Эти значения сохраняются для **Optional**-шаблона. Однако *корректное* использование *фиктивных* «*\_*»-аргументов требует предшествующего определения стандартных значений для них посредством функции **Default**, что и иллюстрирует следующий достаточно простой фрагмент:

```
In[2740]:= Default[FG] = 78; FG[x_., y_.] := x^2 + y^2
In[2741]:= {FG[], FG[42], FG[16, 24]}
Out[2741]= {12168, 7848, 832}
In[2742]:= Default[GG, 1] = 78; Default[GG, 2] = 65; Default[GG, 3] = 42;
                                                GG[x_., y_., z_.] := x + y + z
In[2743]:= {GG[16], GG[16, 24], GG[]}
Out[2743]= {123, 82, 185}
In[2744]:= DefaultValues[GG]
Out[2744]= {HoldPattern[Default[GG, 1]] :> 78, HoldPattern[Default[GG, 2]] :> 65,
            HoldPattern[Default[GG, 3]] :> 42}
```

Средства **Optional**-шаблона и **Default**-функции обеспечивают довольно эффективное определение функций пользователя с *фиктивными* аргументами и их значениями «по умолчанию». Тогда как по функции **DefaultValues[F]** возвращается *список* значений «по умолчанию» для *F*-функции, приписанных ее формальным «*\_*»-аргументам функцией **Default**, как это иллюстрирует последний пример предыдущего фрагмента.

Таким образом, наряду с механизмом типирования формальных аргументов система *Mathematica* располагает механизмами определения для них значений по умолчанию, т.е. значений, которые получают при вызове соответствующие *формальные* аргументы при их отсутствии. Между тем, механизм системы установки значений по умолчанию предполагает определение таких значений перед вычислением определений блоков, функций, модулей на базе стандартной функции **Default**, чей формат поддерживает установку различных значений по умолчанию поочередно для отдельных *формальных* аргументов или единого значения для всех аргументов. В нижеследующем фрагменте

представлена процедура **Defaults**[*F*, *y*], которая обеспечивает для любого подкортежа кортежа формальных аргументов блока, функции, модуля *F*, который определяется 2-х элементным списком *y* (*первый элемент – номер позиции аргумента, второй элемент – выражение*) установку выражения в качестве значения по умолчанию для формального аргумента. Для нескольких значений по умолчанию список *y* имеет **ListList**-тип, чьи подписки имеют указанный формат. Процедура успешно работает с *одноименными* блоком, функцией или модулем *F*, обрабатывая только первый подобъект из списка подобъектов, возвращаемых по вызову **Definition**[*F*]. Вызов возвращает **\$Failed** либо возвращается невычисленным в особых ситуациях. Фрагмент представляет исходный код процедуры **Defaults** наряду с примерами ее довольно типичного использования.

```
In[2184]:= Defaults[x_ /; BlockFuncModQ[x], y_ /; ListQ[y] && Length[y] == 2 | |
ListListQ[y] && DeleteDuplicates[Map[IntegerQ[#[[1]]] &, y]] == {True}] :=
Module[{a = Flatten[{Definition2[x]}], atr = Attributes[x], q, t, u,
b = Flatten[{HeadPF[x]}][[1]], c = Args[x], d, p, h = {}, k = 1,
g = If[ListListQ[y], y, {y}]},
If[Max[Map[#[[1]] &, y]] <= Length[c] && Min[Map[#[[1]] &, y]] >= 1,
c = Map[ToString, If[NestListQ[c], c[[1]], c]]; q = Map[#[[1]] &, y];
d = StringReplace[a[[1]], b -> "", 1];
While[k <= Length[q], p = c[[q[[k]]]]; t = StringSplit[p, "_"];
If[MemberQ[q, q[[k]]], u = If[Length[t] == 2, t[[2]] = StringReplace[t[[2]], "/" -> ""];
If[Quiet[ToExpression["{" <> t[[1]] <> "=" <> ToString[y[[k]]][[2]]] <> "," <> t[[2]] <>
"}"]][[2]] | | Quiet[Head[y[[k]]][[2]]] === Symbol[t[[2]]], True, False], True];
If[u, c[[q[[k]]]] = StringTake[p, {1, Flatten[StringPosition[p, "_"]][[2]]] <> "."]; k++];
ClearAllAttributes[x]; ClearAll[x]; k = 1;
While[k <= Length[q], ToExpression["Default[" <> ToString[x] <> ", " <>
ToString[q[[k]]] <> "]" <> " = " <> ToString1[y[[k]]][[2]]]; k++];
ToExpression[ToString[x] <> "[" <> StringTake[ToString[c], {2, -2}] <> "]" <> d];
Map[ToExpression, MinusList[a, {a[[1]]}]]; SetAttributes[x, atr, $Failed]]

In[2185]:= G[x_, y_ /; IntegerQ[y]] := x+y; G[x_, y_, z_] := x*y*z; G[x_, y_, z_, h_] := x*y*z*h
In[2186]:= SetAttributes[G, {Listable, Protected, Flat}]
In[2187]:= Defaults[G, {{2, 460}, {1, 78}}]
In[2188]:= Definition[G]
Out[2188]= Attributes[G] = {Flat, Listable, Protected}
G[x_, y_] := x + y
G[x_, y_, z_] := x y z
G[x_, y_, z_, h_] := x y z h
G /: Default[G, 1] = 78
G /: Default[G, 2] = 460
In[2189]:= {G[42, 47], G[71], G[]}
Out[2189]= {89, 531, 538}
```

```

In[2273]:= ClearAllAttributes[G]; ClearAll[G]; G[x_, y_]; IntegerQ[y] := x + y;
              G[x_, y_, z_] := x*y*z; G[x_, y_, z_, h_] := x*y*z*h
In[2274]:= SetAttributes[G, {Listable, Protected, Flat}]
In[2275]:= Defaults[G, {2, 78}]
In[2276]:= Definition[G]
Out[2276]= Attributes[G] = {Flat, Listable, Protected}
          G[x_, y_] := x + y
          G[x_, y_, z_] := x y z
          G[x_, y_, z_, h_] := x y z h
          G /: Default[G, 2] = 78
In[2277]:= Defaults[G, {1, 460}]
In[2278]:= Definition[G]
Out[2278]= Attributes[G] = {Flat, Listable, Protected}
          G[x_, y_] := x + y
          G[x_, y_, z_] := x y z
          G[x_, y_, z_, h_] := x y z h
          G /: Default[G, 1] = 460
          G /: Default[G, 2] = 78
In[2279]:= {G[], G[71, 66], G[460]}
Out[2279]= {538, 137, 538}

```

Успешный вызов процедуры **Defaults[G, y]** возвращает *Null*, т.е. ничего, выполняя все установки *y* значений по умолчанию для формальных аргументов блока, функции и модуля *G*. Следует еще раз подчеркнуть, что в случае одноименного объекта *G* вызов **Defaults[G, y]** обрабатывает только *первый* подобъект из списка подобъектов, которые возвращаются по вызову **Definition[G]**. И это достаточно существенное замечание, т.к. механизм присвоения формальным аргументам *G* значений по умолчанию для случая одноименного объекта *G*, используя функцию **Default**, отличен от приписывания для такого объекта, в частности, атрибутов. В последнем случае атрибуты присваиваются всем подобъектам объекта *G*, тогда как для значений по умолчанию механизм имеет силу только относительно первого подобъекта из списка подобъектов, возвращаемых по вызову **Definition[G]**. Данный механизм реализуется как стандартным приемом с использованием функции **Default[G, n]=default** с определением для объекта *G* его *n*-го формального аргумента шаблоном "\_.", так и процедурой **Defaults[G, {n, default}]**, как это достаточно наглядно иллюстрирует нижеследующий фрагмент, а именно:

```

In[2073]:= Clear[V]; Default[V, 2] = 78; V[x_, y_] := {x, y}; V[x_, y_, z_, h_] := {x, y, z, h}
In[2074]:= Definition[V]
Out[2074]= V[x_, y_] := {x, y}
          V[x_, y_, z_, h_] := {x, y, z, h}
          V /: Default[V, 2] = 78
In[2075]:= {V[460], V[42, 47, 67]}
Out[2075]= {{460, 78}, {42, 78, 47, 67}}

```

```

In[2076]:= Clear[V]; Default[V, 2] = 78; V[x_, y_] := {x, y}; V[x_, y_, z_, h_] := {x, y, z, h}
In[2077]:= Definition[V]
Out[2077]= V[x_, y_] := {x, y}
           V[x_, y_, z_, h_] := {x, y, z, h}
           V /: Default[V, 2] = 78
In[2078]:= {V[460], V[42, 47, 67]}
Out[2078]= {{460, 78}, V[42, 47, 67]}
In[2079]:= Clear[V]; V[x_, y_] := {x, y}; V[x_, y_, z_, h_] := {x, y, z, h}; Defaults[V, {2, 78}]
In[2080]:= Definition[V]
Out[2080]= V[x_, y_] := {x, y}
           V[x_, y_, z_, h_] := {x, y, z, h}
           V /: Default[V, 2] = 78
In[2081]:= {V[460], V[42, 47, 67]}
Out[2081]= {{460, 78}, V[42, 47, 67]}

```

В то время как процедура **DefaultsM** расширяет предыдущую процедуру **Defaults** на случай *одноименных* объектов типа {блок, функция, модуль}. Успешный вызов процедуры **DefaultsM[G, y]** возвращает *Null*, т.е. ничего, выполняя для формальных аргументов блока, функции, модуля *G* все установки значений по умолчанию *y* для формальных аргументов. Для одноименного объекта *G* указанного типа блока, функции и модуля выполняются установки значений по умолчанию *y* для формальных аргументов всех подобъектов объекта *G*. Следующий фрагмент представляет исходный код **DefaultsM** наряду с наиболее типичными примерами использования процедуры.

```

In[2056]:= DefaultsM[x_ /; BlockFuncModQ[x], y_ /; ListQ[y] && Length[y] == 2 | |
           ListListQ[y] && DeleteDuplicates[Map[IntegerQ#[[1]] &, y]] == {True}] :=
           Module[{a = Flatten[{PureDefinition[x]}], ArtKr, atr = Attributes[x],
                  g = If[ListListQ[y], y, {y}], q, k = 1,
                  ClearAllAttributes[x]; ClearAll[x]; q = Map#[[1]] &, g];
           While[k <= Length[g], ToExpression["Default[" <> ToString[x] <> ", " <>
           ToString[g[[k]][[1]]] <> "]" <> " = " <> ToString1[g[[k]][[2]]]; k++];
           ArtKr[s_String, def_List] := Module[{n = Unique[AVZ], b, c, d, t, j = 1, h},
           h = ToString[n] <> ToString[x]; ToExpression[ToString[n] <> s];
           b = HeadPF[h]; d = StringReplace[PureDefinition[h], b -> ""];
           c = Select[Map[ToString, Args[h]], # != "$Failed" &];
           While[j <= Length[c], If[MemberQ[q, j], t = c[[j]];
           c[[j]] = StringTake[t, {1, Flatten[StringPosition[t, "_"]][[2]]] <> "."; j++];
           ToExpression[ToString[x] <> "[" <> StringTake[ToString[c], {2, -2}] <> "]" <> d];
           ClearAll[h, n]; k = 1; While[k <= Length[a], ArtKr[a[[k]], g]; k++]; SetAttributes[x, atr]]
In[2057]:= G[x_, y_, z_Integer] := x + y + z; G[x_, y_] := x + y; G[x_] := Block[{}, x];
           G[x_, y_, z_, h_] := Module[{}, x*y*z*h]; SetAttributes[G, {Flat, Protected, Listable}];

```

```
In[2058]:= DefaultsM[G, {{2, 78}, {3, 460}}]
In[2059]:= Definition[G]
Out[2059]= Attributes[G] = {Flat, Listable, Protected}
          G[x_] := Block[{}, x]
          G[x_, y_] := x + y
          G[x_, y_, z_] := x + y + z
          G[x_, y_, z_, h_] := Module[{}, x y z h]
          G /: Default[G, 2] = 78
          G /: Default[G, 3] = 460
In[2050]:= {G[56], G[42, 47], G[45, 17, 24]}
Out[2050]= {56, 89, 86}
```

Процедура **DefaultsM** предоставляет достаточно полезное расширение стандартных средств данного типа, поддерживая как одинарные, так и одноименные объекты типа **{Block, Function, Module}**, как иллюстрируют примеры предыдущего фрагмента.

Еще на одном достаточно важном моменте следует еще раз акцентировать внимание. Как уже отмечалось ранее, процедуры можно определять на основе конструкций как **Module**, так и **Block**. Однако, в общем случае рекомендуется отдавать предпочтение конструкциям именно типа **Module**, ибо в целом ряде случаев (*вопрос был рассмотрен несколько выше*) конструкции типа **Block** выполняются некорректно, не выводя каких-либо диагностических сообщений. В качестве иллюстрации можно привести пример реализации процедуры **Default1**, которая относится к теме относительно значений по умолчанию, *двумя* типами конструкций – на основе **Module** и **Block**. Вызов процедуры **Default1[x, y, z]** возвращает *Null*, т.е. ничего, обеспечивая для блока/функции/модуля *x* установку значений по умолчанию, определяемых списком *z*, для аргументов *x*, чьи позиции задаются списком *y* *PosIntList*-типа. И следующий фрагмент с точки зрения оформления представляет собой, практически, идентичную реализацию определения процедуры **Default1** на основе структур и **Module**, и **Block**. И если первая реализация выполняется вполне корректно вне зависимости от имен локальных переменных, то корректность второй, вообще говоря, зависит от пересечения списка имен локальных переменных со списком значений *по умолчанию* для аргументов блока/функции либо модуля, как довольно наглядно иллюстрирует следующий фрагмент в случае, когда локальная переменная *a* имеется дополнительно и в списке значений по умолчанию для простой функции *G*. Фрагмент представляет исходный код с примерами.

```
In[1792]:= Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Module[{k = 1,
          a = Min[Map[Length, {y, z}]]}, While[k <= a, Default[x, y[[k]]] = z[[k]]; k++];]
In[1793]:= Default1[G, {1, 2}, {a, b}]; G[x_, y_] := {x, y}; Clear[Default1]; DefaultValues[G]
Out[1793]= {HoldPattern[Default[G, 1]] :> a, HoldPattern[Default[G, 2]] :> b}
In[1794]:= Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Block[{k = 1,
          a = Min[Map[Length, {y, z}]]}, While[k <= a, Default[x, y[[k]]] = z[[k]]; k++];]
In[1795]:= ClearAll[G]; Default1[G, {1, 2}, {a, b}]; G[x_, y_] := {x, y}; DefaultValues[G]
Out[1795]= {HoldPattern[Default[G, 1]] :> 2, HoldPattern[Default[G, 2]] :> b}
```



```
In[1796]:= Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Module[{k = 1,
    h = Min[Map[Length, {y, z}]]}, While[k <= h, Default[x, y[[k]]] = z[[k]]; k++];]
In[1797]:= Default1[G, {1, 2}, {a, b}]; G[x_, y_] := {x, y}; Clear[Default1]; DefaultValues[G]
Out[1797]= {HoldPattern[Default[G, 1]] :> a, HoldPattern[Default[G, 2]] :> b}
In[1798]:= Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Block[{k = 1,
    h = Min[Map[Length, {y, z}]]}, While[k <= h, Default[x, y[[k]]] = z[[k]]; k++];]
In[1799]:= ClearAll[G]; Default1[G, {1, 2}, {a, b}]; G[x_, y_] := {x, y}; DefaultValues[G]
Out[1799]= {HoldPattern[Default[G, 1]] :> a, HoldPattern[Default[G, 2]] :> b}
```

Таким образом, механизмы локальных переменных, используемые конструкциями на основе **Module** и **Block**, в общем случае идентичными не являются. Следовательно, в целом следует отдавать предпочтение определению процедур на основе конструкции **Module**, однако с учетом сказанного имеется довольно много случаев, когда оба типа организации процедур являются эквивалентными, требуя предварительного анализа на наличие такой эквивалентности. Данный вопрос детальнее рассмотрен несколько выше. В целом же, для определения процедур рекомендуется использовать структуры именно на основе **Module** во избежание необходимости проведения дополнительного анализа на процедурность и универсальность на все случаи приложений.

Определять значения по *умолчанию* формальным аргументам блока/функции/модуля можно как через **Default**, **Defaults**, **DefaultsM**, так и непосредственно в их заголовках по конструкциям формата *x\_:***Выражение** или сочетая оба указанных способа. Между тем, системная функция **DefaultValues** возвращает установки значений по *умолчанию*, выполненные только посредством стандартной функции **Default**, например:

```
In[2889]:= Default[G5, 2] = 78; G5[x_, y_ : 460, z_ : 42] := {x, y, z}; DefaultValues[G5]
Out[2889]= {HoldPattern[Default[G5, 2]] :> 78}
In[2890]:= G5[Agn]
Out[2890]= {Agn, 460, 42}
In[2891]:= Default[S4, 2] = 78; S4[x_, y_, z_ : 42] := {x, y, z}; DefaultValues[S4]
Out[2891]= {HoldPattern[Default[S4, 2]] :> 78}
In[2892]:= S4[Avz]
Out[2892]= {Avz, 78, 42}
```

При этом, если для аргумента определено значение по умолчанию и через **Default**, и непосредственно в *заголовке* по конструкции *"\_:"*, второй способ имеет максимальный приоритет, как весьма наглядно иллюстрирует предыдущий пример с функцией **G5**.

Между тем, стандартная функция **DefaultValues** обладает рядом довольно серьезных недостатков. Прежде всего, она не отражает значения по умолчанию, определенные в заголовке блока/функции/модуля, а лишь заданные через функцию **Default**. Однако в общем случае это некорректно, ибо аргументам допустимо присвоение значений по умолчанию как через функцию **Default**, так и непосредственно в заголовке; при этом, приоритет именно у второго способа, что зачастую может противоречить результату вызова функции **DefaultValues**, как проиллюстрировано предыдущими примерами. Для устранения данных недостатков запрограммирована процедура **DefaultValues1**,

вызов которой возвращает список формата  $\{\{N_1\} \rightarrow V_1, \dots, \{N_p\} \rightarrow V_p\}$ , где  $N_j$  и  $V_j$  ( $j=1..p$ ) определяют номера позиций формальных аргументов в заголовке блока/функции или модуля и присваиваемые им значения по умолчанию соответственно, вне зависимости от способа их определения с учетом приоритета (высший приоритет имеет установка значений по умолчанию в заголовках блоков/функций либо модулей). Следующий фрагмент представляет исходный код процедуры **DefaultValues1** наряду с наиболее типичными примерами ее использования.

```
In[2079]:= DefaultValues1[x_;/; BlockFuncModQ[x]] := Module[{d = {}, h, k,
    a = {SetAttributes[String, Listable]}, b = Map[ToString, Args[x]],
    c = Map[ToString, DefaultValues[x]]},
    ClearAttributes[ToString, Listable];
    If[b != {}, For[a = 1, a <= Length[b], a++, h = b[[a]]; If[StringFreeQ[h, "_:"],
        d = Append[d, ToExpression["{" <> ToString[a] <> "}"] > " <>
        StringTake[h, {Flatten[StringPosition[h, "_:"][[2]] + 1, -1}]]];
    If[c != {}, If[c == {}, c = ToExpression[Mapp[StringReplace,
        Mapp[StringReplace, c, {"HoldPattern[Default[" <> ToString[x] -> "{",
        "]" -> "}"]}, {"{", "-> "{", "}" -> "{2013}"}]]]; h = c[[1]][[1]];
        If[Op[h] == {2013}, a = {}; For[k = 1, k <= Length[b], k++,
        a = Append[a, ToExpression[ToString[k] <> " > " <> ToString[c[[1]][[2]]]]]; c = a];
        If[PosIntListQ[h] && Length[h] > 1, a = {}; b = h;
            For[k = 1, k <= Length[b], k++, a = Append[a,
            ToExpression[ToString[k] <> " > " <> ToString[c[[1]][[2]]]]]; c = a];
        If[d == {} && c == {}, Return[{}],
        c = Sort[Join[d, c], Op[#1][[1]][[1]] <= Op[#2][[1]][[1]] &]; {k, h} = {1, {}};
        While[k <= Length[c] - 1, h = Append[h, If[Op[c[[k]][[1]][[1]]] ==
            Op[c[[k + 1]][[1]][[1]], k + 1]]; k++];
    Select[ReplacePart[c, Mapp[Rule, Select[h, # != "Null" &], Null], ! SameQ[#, Null] &]]

In[2080]:= Default[G] = 460; G[x_, y_, z_] := {x, y, z}; DefaultValues1[G]
Out[2080]= {{1} -> 460, {2} -> 460, {3} -> 78}

In[2081]:= Default[S2, 2, 3] = 78; S2[x_, y_, z_] := {x, y, z}; DefaultValues1[S2]
Out[2081]= {{1} -> 78, {2} -> 78}

In[2082]:= Default1[S3, {1, 2, 3}, {42, 47, 23}]; S3[x_: 460, y_, z_] := {x, y, z};
DefaultValues1[S3]
Out[2082]= {{1} -> 460, {2} -> 47, {3} -> 23}

In[2083]:= Default[S4, 2] = 2013; S4[x_: 460, y_: 47, z_: 42] := {x, y, z}; DefaultValues1[S4]
Out[2083]= {{1} -> 460, {2} -> 47, {3} -> 42}

In[2084]:= Default[S5, 2] = 78; S5[x_, y_: 460, z_: 42] := {x, y, z}; DefaultValues1[S5]
Out[2084]= {{2} -> 460, {3} -> 42}

In[2085]:= Default1[V3, {1, 2, 3, 4}, {a, b, c, d}]; V3[x_, y_, z_, t_] := {x, y, z, t};
DefaultValues1[V3]
```

```

Out[2085]= {{1} :> a, {2} :> b, {3} :> c, {4} :> d}
In[2086]:= Default1[V4, {1, 2, 3, 4}, {a, b, c, d}]; V4[x_, y_: 70, z_, t_: 78] := {x, y, z, t};
DefaultValues1[V4]
Out[2086]= {{1} :> a, {2} :> 70, {3} :> c, {4} :> 78}

```

Таким образом, наша процедура **DefaultValues1** достаточно существенно расширяет возможности стандартной функции **DefaultValues** и вполне ее заменяет при условии наличия загруженного в текущий сеанс *Mathematica* пакета *AVZ\_Package\_1* [120].

Учитывая наличие двух допустимых механизмов присвоения значений по *умолчанию* формальным аргументам блоков, функций и модулей, вполне определенный интерес представляет задача определения такого типа значений у объектов указанного типа. Задачу решает процедура **DefaultsQ**, вызов которой **DefaultsQ[x]** возвращает **True**, если определение блока, функции либо модуля *x* содержит значения по умолчанию для формальных аргументов, и **False** в противном случае. Тогда как вызов процедуры **DefaultsQ[x, y]**, где *второй* аргумент *y* – неопределенная переменная – дополнительно через *y* возвращает список используемых типов значений по умолчанию {"\_", "\_.", "\_.:"}. В следующем фрагменте представлены как исходный код процедуры **DefaultsQ**, так и достаточно типичные примеры ее использования.

```

In[2276]:= DefaultsQ[x_/, BlockFuncModQ[x], y___] := Module[{a = Args[x], c = {}, d,
    b = {"_", "_.:"}, k = 1, a = Map[ToString, If[NestListQ[a], a[[1]], a]];
    While[k <= Length[a], d = a[[k]];
    If[! StringFreeQ[d, b[[1]]], AppendTo[c, b[[1]]],
    If[! StringFreeQ[d, b[[2]]], AppendTo[c, b[[2]]]]; k++];
    If[c == {}, False, If[{y} != {} && ! HowAct[y], y = DeleteDuplicates[Flatten[c]]; True]]
In[2277]:= PureDefinition[G]
Out[2277]= {"G[x_, y_.] := x + y", "G[x_, y_, z_] := x*y*z", "G[x_, y_, z_, h_] := x*y*z*h"}
In[2278]:= {DefaultsQ[G, t], t}
Out[2278]= {True, {"_."}}
In[2279]:= Default[S, 1] = 78; S[x_, y_: 78, z_] := x + y + z; Kr[x_, y_, z_] := Block[{}, x*y*z]
In[2280]:= {Map91[DefaultsQ, {S, Kr}, {v1, v2}], {v1, v2}}
Out[2280]= {{True, False}, {"_."}, v2}}

```

Наряду с атрибутами и значениями по *умолчанию* для формальных аргументов блока, функции либо модуля, данные объекты могут использовать механизм *опций*. Прежде всего, механизм *опций* достаточно широко используется системными средствами. Так, целому ряду функций в системе *Mathematica* (в частности, функции *Plot*), приписаны *опции*, доступные как для установки, так и для переопределения. Пакет поддерживает общие механизмы для работы с такими опциями. По вызову **Options[G]** возвращается список текущих установок в формате {*a* -> *a1*, *b* -> *b1*, ...} всех опций блока, функции, модуля *G*, в то время как по вызову **Options[G, h]** возвращается текущая установка для опции *h*. По вызову **SetOptions[G, a -> *a2*, b -> *b2*, ...]** обеспечивается переустановка значений для опций {*a*, *b*, ...} блока/функции/модуля *G*, которая остается активной до

очередной переустановки в текущем сеансе. Тогда как вызов функции **SystemOptions[]** возвращает список текущих установок для всех *предустановленных* внутренних опций и подопций пакета. Данные установки определяются как используемой платформой, так и в ряде случаев текущим сеансом *Mathematica*. Для получения количества всех системных опций и их количеств в разрезе как групп опций, так и отдельных опций используется следующая процедура **CountOptions**, исходный код которой с наиболее типичными примерами ее использования представлены ниже, а именно:

```
In[1762]:= CountOptions[h___] := Module[{a = SystemOptions[], b = {}, c = 1, d, k},
    While[c <= Length[a], d = a[[c]];
    AppendTo[b, If[ListQ[Part[d, 2]], {Part[d, 1], Length[Part[d, 2]]}, d]; c++];
    b = Flatten[Gather[b, Head[#1] == Head[#2] &], 1];
    If[{h} == {}, b, If[HowAct[h], Defer[CountOptions[h]], d = 0];
    Do[If[ListQ[b[[k]]], d = d + b[[k]][[2]], d = d + 1], {k, Length[b]}]; {h} = {d}; b]]
In[1763]:= CountOptions[]
Out[1763]= {"AlgebraicsOptions", 6}, {"AlgebraicThreadThroughHeads", 16},
{"AllocationOptions", 1}, {"ArithmeticOptions", 20}, {"BooleanComputationOptions", 9},
{"CacheOptions", 5}, {"CompileOptions", 29}, {"DifferentiationOptions", 6}, .....
"ZeroTestNumericalPrecision" -> 80.}
In[1764]:= CountOptions[g]; g
Out[1764]= 378
```

Вызов **CountOptions[]** возвращает вложенный список, элементами которого являются как списки, так и отдельные опции. Список первым элементом содержит имя группы опций, тогда как второй – *число* опций в этой группе. Тогда как вызов **CountOptions[p]** дополнительно через аргумент *p* – неопределенную переменную – возвращает общее количество предустановленных опций/подопций пакета. Установки для конкретной системной опции *p* можно переопределять по функции **SetSystemOptions[p -> value]**, однако за исключением отдельных случаев их не следует *переопределять* во избежание возможных конфликтов с системными установками. Между тем, система *Mathematica* не поддерживает операций удаления опций, поэтому в следующем фрагменте нами представлена процедура **DeleteOptsAttr**, решающую данную задачу.

Вызов процедуры **DeleteOptsAttr[x]** возвращает *Null*, т.е. ничего, отменяя для символа *x* приписанные ему опции. Тогда как вызов процедуры **DeleteOptsAttr[x,y]**, возвращая *Null*, т.е. ничего, отменяет для символа *x* приписанные ему как опции, так и атрибуты. В качестве второго необязательного аргумента *y* выступает произвольное выражение. Следующий фрагмент представляет исходный код процедуры **DeleteOptsAttr** наряду с наиболее типичными примерами ее использования.

```
In[2876]:= G[x_, y_] := x^2 + y^2
In[2877]:= Options[G] = {Art -> 24, Kr -> 17}
Out[2877]= {Art -> 24, Kr -> 17}
In[2878]:= SetOptions[G, Art -> 24, Kr -> 17]
Out[2878]= {Art -> 24, Kr -> 17}
```

```

In[2879]:= SetAttributes[G, Listable]
In[2880]:= Definition2[G]
Out[2880]= {"G[x_, y_] := x^2 + y^2", "Options[G] := {Art -> 24, Kr -> 17}", {Listable}}
In[2881]:= DeleteOptsAttr[x_ /; BlockFuncModQ[x], y___] := Module[{a = Definition2[x],
                                                                    b, c = "Options[" <> ToString[x] <> "]"}, b = a[[-1]];
                                                                    ClearAllAttributes[x]; ClearAll[x];
                                                                    ToExpression[Select[a, StringFreeQ[ToString[#], c] &]];
                                                                    If[{y} == {}, If[b != {}, SetAttributes[x, Sequences[b]]]]]
In[2882]:= DeleteOptions[G]
In[2883]:= Definition2[G]
Out[2883]= {"G[x_, y_] := x^2 + y^2", {Listable}}
In[2884]:= Vsv[x_, y_] := x^2 + y^2; Options[Vsv] = {V -> 71, G -> 66};
In[2885]:= SetOptions[Vsv, {V -> 71, G -> 66}]; SetAttributes[Vsv, Protected]
In[2886]:= Definition2[Vsv]
Out[2886]= {"Vsv[x_, y_] := x^2 + y^2", "Options[Vsv] := {V -> 71, G -> 66}", {Protected}}
In[2887]:= DeleteOptsAttr[Vsv, 78]
In[2888]:= Definition2[Vsv]
Out[2888]= {"Vsv[x_, y_] := x + y", {}}

```

При этом, следует иметь ввиду, что данная процедура не применима к стандартным функциям пакета, возвращая на них вызов *невычисленным*, как иллюстрирует пример:

```

In[2915]:= {DeleteOptsAttr[Sin, 78], DeleteOptsAttr[Sin]}
Out[2915]= {DeleteOptsAttr[Sin, 78], DeleteOptsAttr[Sin]}

```

Ряду встроенных функций *Mathematica*, например **Plot**, приписаны *опции*, значения которых можно переопределять. Более того, если при вызове функции значения для допустимых опций не определяются, для них используются значения по умолчанию. Вызов функции **Options[function, option]** позволяет получать значения по умолчанию для заданной *вторым* аргументом опции функции, определенной *первым* аргументом

```

In[2916]:= Options[Plot, {PlotLabel, FrameStyle, PlotStyle, PlotRange, ColorOutput}]
Out[2916]= {PlotLabel -> None, FrameStyle -> {}, PlotStyle -> Automatic,
            PlotRange -> {Full, Automatic}, ColorOutput -> Automatic}

```

Механизм опций, поддерживаемый *Mathematica*, может быть успешно использован в разработке различного типа как приложений, так и отдельных программных средств. Детальнее с данным механизмом можно ознакомиться как в работах [100,110,113,116], так и в достаточно хорошо развитой справочной базе системы *Mathematica*.

В заключение раздела вкратце остановимся на применении правил преобразования к процедурам. Механизм правил преобразования, поддерживаемый пакетом, сохраняет силу не только для символов, но также для алгебраических выражений. В принципе, данный механизм можно адаптировать на произвольное выражение. Более того, как существенное свойство данного механизма можно отметить то обстоятельство, что он допускает использование как символьных конструкций, так и шаблонов, например:

```
In[3837]:= Sin[x^2]^2 + Cos[y + h]^2 /. {x^2 -> x, y + h -> x}
Out[3837]= Cos[x]^2 + Sin[x]^2
In[3838]:= Sin[a + b*c]*(Sin[x^2] + Cos[y + h]) /. {Sin[_] -> x, Cos[_] -> y}
Out[3838]= x (x + y)
```

Таким образом, между чисто символьными правилами преобразований и правилами, включающими шаблоны, в частности, "\_" существует одно принципиальное отличие, а именно: если в первом случае правило преобразования производит строгую замену подвыражений преобразуемого выражения на правую часть правил, чья левая часть строго совпадает с такими подвыражениями, во втором случае замены производятся на основе соответствия *шаблонам*, т.е. производится замена *всех* входящих в выражение подвыражений, соответствующих шаблонам – левым частям правил преобразования, как весьма наглядно иллюстрирует второй пример предыдущего фрагмента. С более детальным описанием механизмов того, как для любого выражения программировать шаблоны для правил его преобразования можно ознакомиться в справке по пакету. В то же время пакет не располагает механизмом применения правил преобразования к процедурам и в качестве такового может быть предложена процедура, вызов которой **ReplaceProc**[*P*, *w*] возвращает *определение* в строчном формате процедуры – результат применения к процедуре *P* правил преобразования *w* (одно правило или их список); при этом, из правил *w* исключаются правила, чьи левые части совпадают с формальными аргументами процедуры *P*. Следующий фрагмент представляет не только исходный код процедуры с примерами ее применения, но и простую тестирующую функцию, чей вызов **RuleQ**[*x*] возвращает **True**, если *x* – правило, и **False** в противном случае. В определении процедуры **ReplaceProc** данная функция входит в ее заголовок.

```
In[2859]:= RuleQ[x_] := If[MemberQ[{Rule, RuleDelayed}, Head[x]], True, False]
In[2860]:= Map[RuleQ, {a -> b, c -> d + h, Sin, a + b, ProcQ, a -> b, c -> d + h}]
Out[2860]= {True, True, False, False, False, True, True}
In[2861]:= ReplaceProc[x_ /; ProcQ[x], r_ /; DeleteDuplicates[Map[RuleQ, Flatten[{r}]]] ==
           {True}] := Module[{a = Definition2[x], b = HeadPF[x], c, d = Flatten[{r}]},
           c = ToExpression["Hold[" <> StringTrim[a[[1]], b <> " := " <> "]" <> "]];
           d = Select[d, ! MemberQ[Args1[x], ToString[Part[#, 1]]] &];
           c = ToString1[ReplaceAll[c, d]]; b <> " := " <> StringTake[c, {6, -2}]]
In[2862]:= ArtKr[x_ /; IntegerQ[x], y_ /; StringQ[y]] := Module[{a = StringLength[y],
           b = 78, ab = 460}, (a + x)*(b + y) + ab]
In[2863]:= ReplaceProc[ArtKr, {a -> Art, b -> Kr, y -> 42, x -> 460}]
Out[2863]= "ArtKr[x_ /; IntegerQ[x], y_ /; StringQ[y]] := Module[{Art = StringLength[y],
           Kr = 78, ab = 460}, (Art + 460)*(Kr + 42) + ab]"
```

Ниже будет представлен целый ряд средств, обеспечивающий более высокий уровень *процедурного* программирования в среде системы *Mathematica*; некоторые из которых в определенной степени были навеяны подобными средствами как системы *Maple*, так и другими системами процедурного программирования.

## 8.9. Некоторые дополнительные средства для работы с блоками, функциями и модулями в программной среде *Mathematica*

Если предыдущие разделы главы представляют основные средства работы с объектом типа *{Block, Function, Module}*, то настоящий раздел представляет дополнительные, но довольно важные в целом ряде приложений средства работы с объектами этого типа. Между тем, имея свое основное назначение, данные средства функционально могут и пересекаться со средствами, представленными в предыдущих разделах главы. Этому не стоит особо удивляться, т.к. подобная ситуация имеет место сплошь и рядом среди средств, практически, любой программной среды. И все же средства данного раздела носят несколько более специфический характер и не столь востребованы как средства предыдущих разделов. Между тем, отнесение их к данному разделу в определенной степени носит условный характер и обусловлено нашим опытом их использования.

Прежде всего, вновь вернемся к вопросу синтаксической корректности *блока* и *модуля*. Ниже представлены примеры двух типов синтаксических ошибок при определении процедур типов *{Module, Block}*, которые не распознаются системой при вычислении их определений, а в ряде случаев и при вызове таких процедур. При этом, повторные вызовы процедур *Module*-типа, как очень наглядно показывают примеры фрагмента, дают формально корректные результаты. Именно для тестирования процедур обоих типов на предмет их синтаксической корректности в вышеуказанном контексте была предложена процедура *SyntCorProcQ*, чей исходный код с типичными примерами ее использования и представляет нижеследующий фрагмент, а именно:

```
In[2529]:= Art[x_, y_] := Module[{a, b}, ]; Art1[x_, y_] := Module[{a, b}]
In[2530]:= Kr[x_, y_] := Block[{a, b}, ]; Kr1[x_, y_] := Block[{a, b}]
In[2531]:= {Art[78, 460], Art1[78, 460]}
Module::argr: Module called with 1 argument; 2 arguments are expected. >>
Out[2531]= {Null, Module[{a, b}]}
In[2532]:= {Art[78, 460], Art1[78, 460]}
Out[2532]= {Null, Module[{a, b}]}
In[2533]:= {Kr[78, 460], Kr1[78, 460]}
Block::argr: Block called with 1 argument; 2 arguments are expected. >>
Out[2533]= {Null, Block[{a, b}]}
In[2534]:= SyntCorProcQ[x_ /; BlockModQ[x]] := Module[{b = PureDefinition[x], d, h,
a = HeadPF[x], c = $Kr$, ClearAll[$Kr$];
ProcFuncTypeQ[ToString[x], $Kr$];
h = Quiet[Check[Locals2[x], Locals1[x]]];
h = If[h === {}, {}, ToString[h]]; d = a <> " := " <> $Kr$ <> "[" <> h;
d = StringReplace[b, d -> "", 1]; $Kr$ = c; ! MemberQ[{"", "", Null}], d]]
In[2535]:= Map[SyntCorProcQ, {ProcQ, Kr, Kr1, Art, Art1}]
Out[2535]= {True, False, False, False, False}
```

```
In[2536]:= KrArt[x_, y_, z_] := Module[{}, 78 + x + y + z]
In[2537]:= Map[SyntCorProcQ, {Locals, Mapp, BlockToModule, KrArt}]
Out[2537]= {True, True, True, True}
In[2538]:= Map[SyntCorProcQ, {Art2, Do, If, 6}]
Out[2538]= {SyntCorProcQ[Art2], SyntCorProcQ[Do], SyntCorProcQ[If], SyntCorProcQ[6]}
```

Вызов процедуры **SyntCorProcQ[x]** возвращает **True**, если определение блока/модуля *x*, активизированное в текущем сеансе, синтаксически корректно в указанном выше контексте, в противном случае возвращается **False**. Если *x* – не блок/модуль, то вызов возвращается *невычисленным*. В определении процедуры **SyntCorProcQ** используются наши процедуры **BlockModQ**, **HeadPF**, **Locals1**, **Locals2**, **ProcFuncTypeQ**, **PureDefinition**, рассматриваемые в настоящей книге. В ряде задач процедурного программирования, прежде всего, системного характера, процедура оказывается достаточно полезной.

В ряде приложений системного характера желательно располагать информацией по пользовательским блоку, функции или модулю на предмет использования им средств в разрезе *системные* и *пользовательские*. Данную задачу решает процедура **SysUserSoft**, чей вызов **SysUserSoft[x]** в общем случае возвращает *вложенный 2-элементный* список, чей первый элемент содержит *2-элементные* подписки, первый элемент которых – имя в строчном формате системной функции, и второй элемент – ее кратность, тогда как второй элемент списка также содержит *2-элементные* подписки, первый элемент которых – имя в строчном формате пользовательского средства (*блок, функция, модуль*), и второй элемент – его кратность. При отсутствии у *x* средств указанных типов вызов процедуры **SysUserSoft[x]** возвращает *пустой* список, т.е. {}. Если же тип фактического аргумента *x* отличен от (**Block**, **Function**, **Module**), то вызов **SysUserSoft[x]** возвращается *невычисленным*. Следующий фрагмент представляет исходный код процедуры наряду с наиболее типичными примерами ее использования.

```
In[2163]:= SysUserSoft[x_;/ BlockFuncModQ[x]] := Module[{b, s = {}, u = {}, h = Args[x, 6],
  c, a = Flatten[{PureDefinition[x]}][[1]], d = If[QFunction[x], {}, LocalsGlobals1[x]]},
  b = ExtrVarsOfStr[a, 78]; c = Select[b, ! MemberQ[Flatten[{ToString[x],
    h, "True", "False", "$Failed", Quiet[d[[1]]], Quiet[d[[3]]]], #] &];
  Map[If[Quiet[SystemQ[#]], s = Append[s, #],
    If[BlockFuncModQ[#], u = Append[u, #]]] &, c]; c = Map[Gather, {s, u}];
  c = {Map[Flatten[#] &, Map[{#, Length[#]} &, c[[1]]],
    Map[Flatten[#] &, Map[{#, Length[#]} &, c[[2]]]};
  c = {Map[DeleteDuplicates[#] &, c[[1]]],
    Map[DeleteDuplicates[#] &, c[[2]]]}; If[Flatten[c] == {}, {}, c]]

In[2164]:= A[m_, n_, p_;/ IntegerQ[p], h_;/ PrimeQ[h]] := Module[{a = 42.78,
  h*(m + n + p)/a + StringLength[ToString1[z]]/(Cos[c] + Sin[d])}

In[2165]:= SysUserSoft[A]
Out[2165]= {"Cos", 1}, {"IntegerQ", 1}, {"Module", 1}, {"PrimeQ", 1}, {"Sin", 1},
  {"StringLength", 1}, {"ToString1", 1}}
```



```

In[2166]:= SysUserSoft[SysUserSoft]
Out[2166]= {{{"Append", 2}, {"DeleteDuplicates", 2}, {"Flatten", 4}, {"Gather", 1}, {"If", 3},
{"Length", 2}, {"Module", 1}, {"Quiet", 1}, {"ToString", 1}}, {"BlockFuncModQ", 2},
{"ExtrVarsOfStr", 1}, {"MinusList", 1}, {"PureDefinition", 1}, {"SystemQ", 1}}}
In[2067]:= G[x_] := x^2 + 78*x + 460; SysUserSoft[G]
Out[2067]= {}
In[2068]:= F[x_] := a*x + Sin[b*x] + StringLength[ToString1[x + c]]; SysUserSoft[F]
Out[2068]= {{{"Sin", 1}, {"StringLength", 1}}, {"ToString1", 1}}}
In[2069]:= SysUserSoft[QFunction]
Out[2069]= {{{"Block", 1}, {"CompiledFunction", 1}, {"False", 1}, {"If", 3}, {"MemberQ", 1},
{"Module", 2}, {"Quiet", 2}, {"StringFreeQ", 1}, {"StringJoin", 1}, {"StringReplace", 2},
{"ToString", 1}, {"True", 2}}, {"Definition2", 1}, {"HeadPF", 2}, {"Map3", 1},
{"SuffPref", 4}, {"ToString1", 1}}}

```

Как показал опыт, процедура **SysUserSoft** довольно полезна в программном анализе состава пользовательских средств типов *{Block, Function, Module}*.

Достаточно интересным средством в ряде случаев манипулирования одноименными блоками, функциями либо модулями представляется процедура **RenBlockFuncMod**, исходный код которой наряду с типичными примерами использования представляет следующий фрагмент, а именно:

```

In[2126]:= Pr[x_, y_String, z_ /; If[z === 78, True, False]] := {x, y, z};
Pr[x_, y_ /; StringQ[y], z_ /; If[z === 78, True, False]] := Module[{}, {x, y, z};
SetAttributes[Pr, Protected];
Pr1[x_, y_String, z_ /; If[z === 78, True, False]] := {x, y, z};
SetAttributes[Pr1, {Protected, Listable}]
In[2127]:= RenBlockFuncMod[x_ /; BlockFuncModQ[x], y_Symbol] := Module[{t = {}, h,
a = Options[x], b = Attributes[x], k = 1, n,
c = Flatten[{PureDefinition[x]}], d = Flatten[{HeadPF[x]}]},
For[k, k <= Length[c], k++, h = StringReplace[c[[k]], StringJoin[d[[k]], " := " -> ""],
h = If[SuffPref[h, "Module[{" , 1], "M", If[SuffPref[h, "Block[{" , 1], "B", "F"]];
n = ToString[Unique[y]] <> h; t = Append[t, n];
ToExpression[StringReplace[c[[k]], ToString[x] <> "[" -> n <> "[", 1];
If[a != {}, ToExpression["SetOptions[" <> n <> ", " <> ToString[a] <> ""]];
If[b != {}, ToExpression["SetAttributes[" <> n <> ", " <> ToString[b] <> ""]];
ClearAllAttributes[x]; ClearAll[x]; If[Length[t] == 1, t[[1]], t]
In[2128]:= RenBlockFuncMod[Pr1, Sv]
Out[2128]= "Sv$59655F"
In[2129]:= Definition["Sv$59655F"]
Out[2129]= Attributes[Sv$59655F] = {Listable, Protected}
Sv$59655F[x_, y_String, z_ /; If[z === 78, True, False]] := {x, y, z}

```

```
In[2130]:= RenBlockFuncMod[Pr, Sv]
Out[2130]= {"Sv$61516F", "Sv$61518M"}
In[2131]:= Definition["Sv$61516F"]
Out[2131]= Attributes[Sv$61516F] = {Protected}
          Sv$61516F[x_, y_String, z_ /; If[z === 78, True, False]] := {x, y, z}
In[2132]:= Definition[Sv$61518M]
Out[2132]= Attributes[Sv$61518M] = {Protected}
          Sv$61518M[x_, y_String, z_ /; If[z === 78, True, False]] := Module[{}, {x, y, z}]
In[2133]:= Map[Definition, {Pr, Pr1}]
Out[2133]= {Null, Null}
```

Вызов процедуры **RenBlockFuncMod**[*x*, *y*] возвращает в строчном формате новое имя блока/функции/модуля *x*, определяемое следующим форматом **Unique**[*y*] <> *H*, где *y* – произвольный символ, тогда как *H* – один из символов {"B", "F", "M"} в зависимости от типа объекта *x* или типа составляющего его подобъекта в случае *одноименного* объекта *x*. При этом, объект *x* удаляется из текущего сеанса, в то время как результат данного переименования сохраняет опции и атрибуты исходного объекта *x*. Процедура более всего удобна в случае необходимости дифференцировать одноименный объект *x* на отдельные составляющие его подобъекты.

В целом ряде случаев при вызовах процедур, находящихся в загруженном в текущий сеанс пакете пользователя (*файлы типов* {"cdf", ".m", ".mx"}), их локальные переменные в области переменных *Mathematica* ассоциируются с контекстом, приписанным этому пакету. Детальнее данный механизм здесь не рассматривается. Это же относится и к символьным результатам, возвращаемым процедурой из данного пакета через такие локальные переменные. В этом случае символьный результат принимает следующий стандартный формат, а именно:

*Контекст, связанный с пакетом`Имя процедуры`Результат*

как это наглядно иллюстрирует результат вызова **Head1**[*a* := *b*] в одном из примеров нижеследующего фрагмента. С целью устранения подобной ситуации и получения приведенного результата, существенно лучше приспособленного для последующей обработки, к локальной переменной, через которую возвращается результат, можно применять процедуру **ReductRes**, чей вызов **ReductRes**[*P*, *a*] возвращает *приведенный* результат *a*, возвращаемый процедурой *P* из пользовательского пакета, загруженного в текущий сеанс. Следующий фрагмент представляет *оба* варианта процедуры **Head1** без использования и с использованием такого механизма с иллюстрацией результатов вызова *обоих* процедур. Полученные результаты достаточно наглядно иллюстрируют весьма принципиальное различие, возникающее от механизма *приведения* результатов на основе представленной процедуры **ReductRes**.

```
In[3262]:= ReductRes[x_ /; SymbolQ[x], y_] := Module[{a = ToString[y], c, d = "", t, k,
          b = "" <> ToString[x] <> "" }, c = StringPosition[a, b]; If[c == {}, y, k = c[[1]][[2]];
          While[k >= 1, If[! MemberQ[{" ", t = StringTake[a, {k, k}]; t], d = t <> d, Break[]]; k--];
          ToExpression[StringReplace[a, d -> ""]]]
```

```

In[3264]:= ReductRes[Head1, AladjevProcedures`Head1`System]
Out[3264]= System
In[3282]:= Map[Head, {ProcQ, Sin, 78, a + b, Function[{x, y}, x + y], G[x], S[6], Head1}]
Out[3282]= {Symbol, Symbol, Integer, Plus, Function, G, S, Symbol}
In[3283]:= Map[Head1, Map[ToString, {ProcQ, Sin, 78, a + b, Function[{x, y}, x + y], G[x],
                                     S[6], Head1}]]
Out[3283]= {"Module", "System", "Integer", "Plus", "Function", "G", "S", "Module"}
In[3284]:= Head1[a := b]
Out[3284]= AladjevProcedures`Head1`System
    
```

Следующая полезная процедура **Avg** является *внутренней*, т.е. вызов процедуры **Avg[]** имеет смысл лишь в теле *другой* процедуры, возвращая список вложенности  $\{1 | 2\}$ , чьи элементы определяют 2-х элементные списки, первые элементы которых определяют *локальные* переменные в строчном формате *внешней* по отношению к **Avg** процедуры, тогда как вторые – их начальные значения в строчном формате; при этом, отсутствие начального значения кодируется символом *"None"*. В случае более одной локальной переменной возвращается *ListList*-список, чьи подписки имеют указанный формат. При отсутствии у *внешней* процедуры локальных переменных вызов процедуры **Avg[]** возвращает пустой список, т.е. {}. Вызов **Avg[]** вне другой процедуры не имеет особого смысла, возвращая список указанного выше формата для двух *локальных* переменных самой процедуры **Avg**, как наглядно иллюстрирует нижеследующий фрагмент.

```

In[96]:= Avg[] := Module[{b,
                        a = ToString[ToExpression[ToString[InputForm[Stack[_][[1]]]]]],
                        a = StringReplace[a, "$" -> ""];
                        a = StringReplace[a, If[SuffPref[a, "Block["], 1, "Block["], "Module["] -> "", 1];
                        a = SubStrSymbolParity1[a, "{", "}"][[1]];
                        b = StrToList[StringTake[a, {2, -2}]]];
                        b = Map[StringSplit[#, " = "] &, b];
                        Map[If[Length[#] == 1, {#[[1]], "None"}, #] &, b]]

In[97]:= Z[m_, n_, p_ /; IntegerQ[p]] := Module[{h, x = 78, y = {a, b}}, m+n+p; h = Avg[]; h]
In[98]:= Z[71, 78, 460]
Out[98]= {{{"h", "None"}, {"x", "78"}, {"y", "{a, b}"}}
In[99]:= G[m_, n_, p_ /; IntegerQ[p]] := Module[{a, b = 6, c, d = 78}, d = Avg[]; m + n + p; d]
In[100]:= G[t, p, 460]
Out[100]= {{{"a", "None"}, {"b", "6"}, {"c", "None"}, {"d", "78"}
In[101]:= A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] := Module[{a = 460.78, b, c, t, q,
                        d = 78, z = 45}, b = Avg[]; m + n + p + h; m*n; b]

In[102]:= A[x, y, 42, 47]
Out[102]= {{{"a", "460.78"}, {"b", "None"}, {"c", "None"}, {"t", "None"}, {"q", "None"},
            {"d", "78"}, {"z", "45"}
In[103]:= B[m_, n_, p_, h_ /; PrimeQ[h]] := Module[{a = 460.78, b, c = {h, p}, t, q, d = 78,
                        z = p*t, s}, b = Avg[]; m + n + p + h; m*n; b]
    
```

```
In[104]:= B[x, y, 42, 47]
Out[104]= {"a", "460.78"}, {"b", "None"}, {"c", "None"}, {"t", "None"}, {"q", "None"},
{"d", "78"}, {"z", "45"}
In[105]:= T[m_, n_, p_, h_ /; PrimeQ[h]] := Module[{}, m*n*p*h; Avg[]]; T[24, 17, 42, 47]
Out[105]= {}
In[106]:= Avg[]
Out[106]= {"a", "ToString[ToExpression[ToString[Stack[_][[1]]]]"], {"b", "None"}}
```

Предыдущий фрагмент представляет исходный код процедуры **Avg** с примерами ее применения для получения в теле процедуры списка ее локальных переменных. Ряд системных средств нашего пакета *AVZ\_Package\_1* [120] используют процедуру **Avg**.

Здесь еще раз вполне уместно отметить важное обстоятельство, что блоки, функции и модули различаются их заголовками, как это неоднократно иллюстрировалось выше. При этом, при вызове объекта данного типа в качестве выбранного полагается *первый* из полного списка *одноименных* подобъектов, определяемого стандартной функцией **Definition**, на котором *допустимым* является кортеж фактических аргументов. Данное обстоятельство следует учитывать при программировании и оно было учтено нами в программировании целого ряда средств из нашего пакета *AVZ\_Package\_1* [120]. Более того, в качестве одноименных могут быть как объекты типа *{Block, Function, Module}*, так и в сочетании с объектами других типов, в целом ряде случаев при вычислениях с такими объектами инициируя особые или ошибочные ситуации. Для устранения из *одноименных* объектов подобъектов типов, отличных от *{блок, функция, модуль}* служит довольно простая процедура, вызов которой **ProcCalls[g]** возвращает *Null*, т.е. ничего, удаляя из подписка *одноименного* объекта *g* (*в строчном формате*) подобъектов типов, отличных от *{блок, функция, модуль}*. Следующий фрагмент представляет исходный код процедуры **ProcCalls** наряду с наиболее типичными примерами ее использования.

```
In[2166]:= A[x_] := Module[{a = 460}, x + a]; A[x_, y_] := Module[{a = 78}, x + y + a];
A[x_, y_List] := Block[{}, {x, y}]; A[x_Integer] := Module[{a = 460}, x + a];
A := {a, b, c, d, h}; SetAttributes[A, {Flat, Listable, Protected}];
In[2167]:= Definition[A]
Out[2167]= Attributes[A] = {Flat, Listable, Protected}
A := {a, b, c, d, h}
A[x_Integer] := Module[{a = 460}, x + a]
A[x_] := Module[{a = 460}, x + a]
A[x_, y_List] := Block[{}, {x, y}]
A[x_, y_] := Module[{a = 78}, x + y + a]
In[2168]:= ProcCalls[P_String] :=
Module[{a = Select[StringSplit[ToString[InputForm[Definition[P]]], "\n"],
# != " " && # != P && ! SuffPref[#, P <> " := ", 1] &]},
If[SuffPref[a[[1]], "Attributes["], 1], a = AppendTo[a[[2 ;; -1]], a[[1]]];
ClearAttributes[P, Protected]; Clear[P]; Map[ToExpression, a]; ]
In[2169]:= ProcCalls["A"]
```

```

In[2170]:= Definition[A]
Out[2170]= Attributes[A] = {Flat, Listable, Protected}
          A[x_Integer] := Module[{a = 460}, x + a]
          A[x_] := Module[{a = 460}, x + a]
          A[x_, y_List] := Block[{x, y}]
          A[x_, y_] := Module[{a = 78}, x + y + a]

In[2180]:= ScanLikeProcs[x_: {}] := Module[{b = {}, c = {}, d, h, k = 1, a = Select[Names["*"],
                                         StringFreeQ[#, "$"] && BlockFuncModQ[#] &]},
                                         Off[Definition::ssle]; If[a == {}, Return[{}],
                                         For[k, k <= Length[a], k++, d = Definition2[a[[k]]][[1 ;; -2]];
                                         If[Length[d] > 1, b = Append[b,
                                         Map[StringTake[#, {1, Flatten[StringPosition[#, " := "][[1]] - 1]}] &, d]];
                                         c = Append[c, a[[k]]]]]; On[Definition::ssle]; If[! HowAct[x], x = b, Null]; c]

In[2181]:= G[x_] := Module[{a = 78}, x^2 + a]; G[x_ /; PrimeQ[x]] := Module[{a = 78}, x + a];
          G[x_, y_] := Module[{x+y}, G[x_, y_ /; ListQ[y], z_] := Module[{x+Length[y]+z}
In[2182]:= V[x_] := Module[{x^2}, V[x_ /; ListQ[x]] := Module[{x}, Length[x]]
In[2183]:= {ScanLikeProcs[], ScanLikeProcs[S], S}
Out[2183]= {"A", "G", "V"}, {"A", "G", "V"}, {"A[x_Integer]", "A[x_, y_List]", "A[x_, y_]",
          "A[x_]"}, {"G[x_ /; PrimeQ[x]]", "G[x_]", "G[x_, y_]"},
          "G[x_, y_ /; ListQ[y], z_]"}, {"V[x_ /; ListQ[x]]", "V[x_]"}]
    
```

В дополнение к предыдущей с целью определения одноименных блоков, функций и модулей текущего сеанса системы предназначена довольно простая процедура, вызов **ScanLikeProcs[]** которой возвращает список одноименных блоков/функций/модулей, активизированных в текущем сеансе, тогда как в результате вызова **ScanLikeProcs[W]** дополнительно через *неопределенную* переменную **W** возвращается список заголовков в строчном формате объектов указанного типа. Предыдущий фрагмент представляет исходный код процедуры **ScanLikeProcs** наряду с примерами ее применения. В ряде приложений данные средства представляются нам достаточно полезными, в первую очередь, при разработке системных средств манипулирования процедурами.

С целью устранения неоднозначности одноименных блоков, функций либо модулей рекомендуется применять стандартные средства для очистки текущего сеанса пакета от конкретных определений, в случае необходимости используя для их имен отмену **Protected**-атрибута. Для очистки символов от присвоенных им значений *Mathematica* располагает 3 функциями **Clear**, **ClearAll** и **Remove**, рассмотренными в [101]. Однако, данные функции требуют конкретного указания символов, подлежащих очистке от присвоенных им значений. Тогда как следующий фрагмент представляет исходный код процедуры **ClearCS** с примерами ее применения, вызов которой **ClearCS[ClearAll]** возвращает **Null**, т.е. ничего, очищая все символы не только от значений, полученных ими в текущем сеансе, но также от атрибутов, сообщений и значений по умолчанию, ассоциированных с такими символами; тогда как вызов процедуры **ClearCS[Remove]** возвращает **Null**, т.е. ничего, удаляя из поля имен системы все символы, получившие значения в текущем сеансе системы *Mathematica*.

```
In[2990]:= ClearCS[x_ /; MemberQ[{ClearAll, Remove}, x]] :=
Module[{a = Join[Names["Global`*"], {"a", "b", "c", "d", "h", "k", "p", "S", "x", "y"}]},
Quiet[Mapp[ClearAttributes, a, Protected]]; Quiet[Map[x, a]];]

In[2991]:= {x, y, z, g, h} = {42, 71, 47, 66, 2013}; ClearCS[Remove]; {x, y, z, g, h}
Out[2991]= {Removed[x], Removed[y], Removed[z], Removed[g], Removed[h]}
In[2992]:= {x, y, z, g, h} = {42, 71, 47, 66, 2013}; ClearCS[ClearAll]; {x, y, z, g, h}
Out[2992]= {x, y, z, g, h}
In[2200]:= G[x_] := Module[{a = 78}, x^2 + a]; G[x_ /; PrimeQ[x]] := Module[{a = 78}, x + a];
In[2201]:= V[x_] := Module[{}, x^2]; V[x_ /; ListQ[x]] := Module[{}, Length[x]];
In[2202]:= ClearCS[ClearAll]; Map[Definition, {G, V}]
Out[2202]= {Null, Null}
```

В ряде приложений процедура **ClearCS** оказывается достаточно полезным средством, во многом обеспечивая восстановление исходного состояния текущего сеанса.

Выше дополнительно к стандартной функции **Definition** нами были определены три процедуры **Definition1**, **Definition2**, **Definition3** и **PureDefinition**, которые имеют свои преимущества в программировании задач обработки одноименных объектов и/или объектов, которым были присвоены атрибуты, опции и/или значения по умолчанию для формальных аргументов. Нижеследующий фрагмент представляет исходный код процедуры **Definition4** с типичными примерами ее применения, которая дополняет указанные процедуры **Definition1**, **Definition2**, **Definition3**, **PureDefinition** в контексте облегчения программирования задач обработки *одноименных* объектов, прежде всего. Вызов процедуры **Definition4[x]** в удобном формате возвращает определение объекта *x*, имя которого кодируется в строчном формате, а именно: (1) на *системной* функции *x* возвращаются ее атрибуты, (2) на блоке, функции либо модуле пользователя вызов возвращает определение объекта *x* в *строчном* формате с приписанными атрибутами, опциями и/или значениями по умолчанию для формальных аргументов (*если такие имеются*), (3) возвращает определение объекта *x* в строчном формате для присвоений операторами `{":=", "="}` и (4) возвращает **\$Failed** в остальных случаях. Процедура имеет целый ряд интересных приложений в программировании системных приложений.

```
In[2097]:= A[x_] := Module[{a = 460}, x + a]; A[x_, y_] := Module[{a = 78}, x + y + a];
A[x_, y_List] := Block[{}, {x, y}]; A[x_Integer] := Module[{a = 460}, x + a];
A := {a, b, c, d, h}; SetAttributes[A, {Flat, Listable, Protected}];

In[2098]:= Definition4[x_String] := Module[{a},
a = Quiet[Check[Select[StringSplit[ToString[InputForm[Quiet[Definition[x]]]],
"\n"], # != " " && # != x &], $Failed]];
If[a === $Failed, $Failed, If[SuffPref[a[[1]], "Attributes["], 1],
a = AppendTo[a[[2 ;; -1]], a[[1]]]]; If[Length[a] != 1, a, a[[1]]]]

In[2099]:= Definition4["A"]
Out[2099]= {"A := {a, b, c, d, h}", "A[x_Integer] := Module[{a = 460}, x + a]",
"A[x_] := Module[{a = 460}, x + a]", "A[x_, y_List] := Block[{}, {x, y}]",
```

```
"A[x_, y_] := Module[{a = 78}, x + y + a]",
"Attributes[A] = {Flat, Listable, Protected}"
In[2100]:= W = 78; G := 460; Map[Definition4, {"W", "G", "71", "a + b", "If"}]
Out[2100]= {"W = 78", "G := 460", $Failed, $Failed, Attributes[If] = {HoldRest, Protected}}
```

В задачах формальной обработки *функциональных* выражений вполне определенный интерес представляет процедура **ExpArgs**, чей вызов **ExpArgs[G, {x, ...}]** обеспечивает расширение списка *формальных* аргументов блока, функции или модуля *G* на список аргументов {*x, y, z, ...*} вправо относительно кортежа формальных аргументов объекта *G* с возвратом значения *Null*, т.е. ничего, и активацией в текущем сеансе обновленного определения объекта *G*. Расширение кортежа формальных аргументов для объекта *G* производится только на переменные из списка {*x, y, z, ...*}, которые не являются ни его *формальными* аргументами, ни *локальными* переменными; в противном случае какого-либо расширения не производится. Элементами списка {*x, y, z, ...*} на обновление могут быть как символы в строчном формате, так и имена аргументов с приписанными им тестами на допустимость. При этом, вызов процедуры **ExpArgs[G, x]** на недопустимом объекте *G*, в частности, на системной функции, либо на пустом списке *x* возвращается невычисленным. Следующий фрагмент представляет как исходный код процедуры процедуры **ExpArgs**, так и некоторые наиболее типичные примеры ее применения.

```
In[2127]:= A[x_] := Module[{a = 460}, x + a]; A[x_, y_] := Module[{a = 78}, x + y + a];
A[x_, y_List] := Block[{}, {x, y}]; A[x_Integer] := Module[{a = 460}, x + a];
SetAttributes[A, {Flat, Listable, Protected}];
Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
In[2128]:= ExpArgs[f_ /; BlockFuncModQ[f], x_ /; ListQ[x] &&
DeleteDuplicates[Map[! StringFreeQ[ToString[#], "_"] || StringQ[#] &, x]] == {True}] :=
Module[{a, b, c, d, t, h, g = {}, k = 1},
a = Flatten[{Definition4[ToString[f]]]; b = Args[f, 78];
b = If[NestListQ[b], b[[1]], b]; d = Locals1[f];
d = If[NestListQ[d], d[[1]], d]; c = Flatten[{HeadPF[f]}][[1]];
t = Map[ToString, x]; h = Map[#[[1]] &, Map[StringSplit[#, "_"] &, t]]; b = Join[b, d];
While[k <= Length[h], If[! MemberQ[b, h[[k]]], d = t[[k]];
AppendTo[g, If[StringFreeQ[d, "_"], d <> "_", d]]; k++];
If[g == {}, Return[], g = ToString[g];
d = StringTake[c, {1, -2}] <> ", " <> StringTake[g, {2, -2}] <> ";";
ClearAllAttributes[f]; ClearAll[f];
a[[1]] = StringReplace[a[[1]], c -> d, 1]; Map[ToExpression, a]; ]
In[2129]:= ExpArgs[Art, {"x", "z_", "h", p_ /; StringQ[p], c_String, h_ /; ListQ[h] &&
Length[h] >= 78}]
In[2130]:= Definition[Art]
Out[2130]= Art[x_, y_ /; PrimeQ[y], z_, h_, p_ /; StringQ[p], h_ /; ListQ[h] &&
Length[h] >= 78] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]] a b c]
```

```

In[2131]:= ExpArgs[Art, {"x", "z_", "h", p_ /; String[p], c_Integer, h_ /; ListQ[h] &&
Length[h] >= 78}]
In[2132]:= Definition[Art]
Out[2132]= Art[x_, y_ /; PrimeQ[y], z_, h_, p_ /; String[p], h_ /; ListQ[h] &&
Length[h] >= 78] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]] a b c]
In[2133]:= ExpArgs[A, {"x", "z_", "h", p_ /; String[p], c_Integer, h_ /; ListQ[h] &&
Length[h] >= 78}]
In[2134]:= Definition[A]
Out[2134]= Attributes[A] = {Flat, Listable, Protected}
A[x_Integer, z_, h_, p_ /; String[p], c_Integer, h_ /; ListQ[h] &&
Length[h] >= 78] := Module[{a = 460}, x + a]
A[x_] := Module[{a = 460}, x + a]
A[x_, y_List] := Block[{}, {x, y}]
A[x_, y_] := Module[{a = 78}, x + y + a]
In[2135]:= ExpArgs[Art, {"x", "z_", "h", p_ /; String[p], c_Integer, h_ /; ListQ[h] &&
Length[h] >= 78}]
In[2136]:= Definition[A]
Out[2136]= Attributes[A] = {Flat, Listable, Protected}
A[x_Integer, z_, h_, p_ /; String[p], c_Integer, h_ /; ListQ[h] &&
Length[h] >= 78] := Module[{a = 460}, x + a]
A[x_] := Module[{a = 460}, x + a]
A[x_, y_List] := Block[{}, {x, y}]
A[x_, y_] := Module[{a = 78}, x + y + a]

```

Наряду с системными средствами **ExpArgs** существенно использует наши процедуры **Args**, **BlockModQ**, **ClearAllAttributes**, **Definition4**, **HeadPF**, **Locals1**, **NestListQ**, которые представлены и в настоящей книге. Процедура **ExpArgs** имеет целый ряд достаточно интересных приложений, прежде всего, приложений системного характера.

Следующий фрагмент представляет полезную процедурную переменную **\$ProcType**, реализованную простой функцией на основе системной функции **Stack** и имеющую смысл только в теле блока или модуля, возвращая тип содержащего его объекта **{Block, Module}** в строчном формате. Вне объектов указанного типа переменная принимает значение **"ToString"**, не имеющее особого значения. Ниже приводятся исходный код переменной **\$ProcType** наряду с некоторыми типичными примерами ее применения. Переменная имеет ряд полезных приложений системного и прикладного характера.

```

In[2123]:= $ProcType := ToString[Stack[{}]]
In[2124]:= Agn[x_, y_] := Block[{a = 78, b = 460, c = $ProcType}, a + b + c; {$ProcType, c}]
In[2125]:= Agn[42, 47]
Out[2125]= {"Block", "Block"}
In[2126]:= Agn[x_, y_] := Module[{a = 78, b = 460, c = $ProcType}, a + b + c; {$ProcType, c}]
In[2127]:= Agn[42, 47]
Out[2127]= {"Module", "Module"}

```



К предыдущей процедурной переменной непосредственно примыкает процедурная переменная **\$TypeProc**, также используемая только в теле блока либо модуля любого типа. Переменная **\$TypeProc** получает значение типа в строчном формате объекта **G**, который ее содержит, в разрезах {"Block", "Module", "DynamicModule"}; вне блока или модуля переменная получает значение **\$Failed**, как хорошо иллюстрирует фрагмент, представляющий исходный код процедурной переменной **\$TypeProc** с примерами ее наиболее типичного использования.

```
In[2072]:= $TypeProc := CheckAbort[If[$Art24$Kr17$ = Select[{Stack[Module],
Stack[Block], Stack[DynamicModule]}, # != {} &];
If[$Art24$Kr17$ == {}, Clear[$Art24$Kr17$];
Abort[], $Art24$Kr17$ = ToString[$Art24$Kr17$[[1]][[1]]];
SuffPref[$Art24$Kr17$, "Block[{", 1], Clear[$Art24$Kr17$]; "Block",
If[SuffPref[$Art24$Kr17$, "Module[{", 1] &&
! StringFreeQ[$Art24$Kr17$, "DynamicModule"], Clear[$Art24$Kr17$];
"DynamicModule", Clear[$Art24$Kr17$]; "Module"]], $Failed]

In[2073]:= M[x_] := Module[{a = 78, b = 460, c = $TypeProc}, c]; M[71]
Out[2073]= "Module"

In[2074]:= G[x_] := Module[{a = 78, b = 460, c}, c = a + b + x; c^2; $TypeProc]; G[71]
Out[2074]= "Module"

In[2075]:= B[x_] := Block[{a = 78, b = 460, c = $TypeProc}, c]; B[71]
Out[2075]= "Block"

In[2076]:= DM[x_] := DynamicModule[{a, c = $TypeProc}, x; c]; DM[71]
Out[2076]= "DynamicModule"

In[2077]:= $TypeProc
Out[2077]= $Failed

In[2078]:= F[x_/, ListQ[x]] := Append[Select[x, OddQ[#] &], $TypeProc]; F[{66, 71, 46, 17}]
Out[2078]= {71, 17, $Failed}
```

В целом ряде случаев процедурного программирования данная переменная наряду с ранее представленной переменной **\$ProcType** оказываются достаточно полезными. К **\$TypeProc** непосредственно примыкает и процедурная переменная **\$CallProc**, вызов которой в строчном формате возвращает содержимое тела блока или модуля, который содержит ее на момент вызова. При этом, для модуля возвращается тело с локальными переменными с приписанными им символами "\$", тогда как для блока возвращается его тело в стандартном формате. Вызов переменной вне блока либо модуля получает значение "ToString[Stack[\_] [[1]]]". Следующий фрагмент представляет исходный код процедурной переменной **\$CallProc** с типичными примерами ее использования.

```
In[2584]:= $CallProc := ToString[Stack[_] [[1]]]

In[2585]:= M[x_, y_/, StringQ[y]] := Module[{a = $CallProc, b, c}, x + StringLength[y]; a]
In[2586]:= M[78, "AvzAgnVsArtKr"]
Out[2586]= "Module[{a$ = $CallProc, b$, c$}, 78 + StringLength["AvzAgnVsArtKr"]; a$]"
```

```
In[2587]:= B[x_, y_ /; StringQ[y]] := Block[{a = $CallProc, b, c}, x + StringLength[y]; a]
In[2588]:= B[78, "AvzAgnVsvArtKr"]
Out[2588]= "Block[{a = $CallProc, b, c}, 78 + StringLength["AvzAgnVsvArtKr"]; a]"
In[2589]:= $CallProc
Out[2589]= "ToString[Stack[_][[1]]]"
```

Процедурная переменная **\$CallProc** обеспечивает возможность обработки тела блока либо модуля, его содержащего, в рамках самого объекта, представляя определенный интерес в целом ряде приложений, прежде всего, системного характера.

Использование средств сохранения определений в файлах *ASCII*-формата позволяет программировать довольно эффективные и полезные средства анализа структурной организации пользовательских блоков, функций и модулей. Следующий фрагмент представляет исходный код процедуры **CompActPF** наряду с типичными примерами ее применения, чей вызов **CompActPF[x]** возвращает *вложенный 2-элементный список*, *первый* элемент которого определяет список всех блоков, функций, модулей, которые входят в определение блока, функции, модуля *x*, включая *x*, тогда как второй элемент определяет список *заголовков* в строчном формате данных средств. При этом, в списки включаются только средства пользователя, определения которых были активированы в текущем сеансе системы *Mathematica*; более того, для входящих в объект *x* вызовов включаются соответственно и все их вызовы на полную глубину вложенности.

```
In[2088]:= G[x_] := Module[{}, a*x + b]; S[y_] := Module[{}, y^2 + 70]; S1[y_] := y^2 + G[y];
V[x_, y_] := Module[{G, S}, G[x] + S[y^2]]; G1[x_] := a*x + b + V[x, 78];
V1[x_, y_] := G1[x] + S1[y^2] + h*Sin[x*y] + v*Cos[x*y]

In[2089]:= CompActPF[x_ /; BlockFuncModQ[x]] := Module[{a = ToDefOptPF[x], b = {},
c = "", d, f = ToString[x] <> ".txt", h = ""}, Put[FullDefinition[x], f];
Quiet[While[! SameQ[h, EndOfFile], h = Read[f, String]; If[h != " ", c = c <> h;
If[HeadingQ[d = StringTake[c, {1, Flatten[StringPosition[c, " := "][[1]] - 1]}],
b = Append[b, d]; c = ""]; Continue[ ]]]; DeleteFile[Close[f]]; {Map[HeadName, b], b}]

In[2090]:= CompActPF[V1]
Out[2090]= {{ "V1", "G1", "V", "G", "S", "S1", {"V1[x_, y_]", "G1[x_]", "V[x_, y_]", "G[x_]",
"S[y_]", "S1[y_]"} }

In[2091]:= CompActPF[V]
Out[2091]= {{ "V", "G", "S", {"V[x_, y_]", "G[x_]", "S[y_]"} }

In[2099]:= CompActPF1[x_ /; BlockFuncModQ[x]] := Module[{b = Args[x, 78], d = {}, k = 1,
a = Flatten[{PureDefinition[x]}][[1]], c = Locals1[x], p},
{b, c} = {If[NestListQ[b], b[[1]], b], If[NestListQ[c], c[[1]], c]};
a = Select[ExtrVarsOfStr[a], ! MemberQ[Flatten[{ToString[x],
Join[b, c, {"Block", "Module"}]}], #] &];
While[k <= Length[a], p = a[[k]];
AppendTo[d, If[BlockFuncModQ[p], {p, HeadPF[p]},
```

```

If[SystemQ[p], {p, "System"}, {p, "Undefined"}]]]; k++;
a = Map[Flatten, Gather[d, ! StringFreeQ[#1[[2]], "_"] &&
! StringFreeQ[#2[[2]], "_"] &]];
b = Map[Flatten, Gather[a, #1[[2]] == "System" && #2[[2]] == "System" &]];
d = Map[Flatten, Gather[b, #1[[2]] == "Undefined" && #2[[2]] == "Undefined" &]];
Map[If[#1 == "System", Prepend[MinusList[#, {"System"}], "System"],
If[#1 == "Undefined", Prepend[MinusList[#, {"Undefined"}], "Undefined"], #] &, d]]
In[2100]:= CompActPF1[V1]
Out[2100]= {"System", "Cos", "Sin"}, {"G1", "G1[x_]", "S1", "S1[y_]"}, {"Undefined", "h", "v"}
In[2101]:= CompActPF1[V]
Out[2101]= {}
In[2102]:= Z[x_String, z_]; ! HowAct[x]] := Block[{a = Sin[x]}, Cos[a] + StringLength[x]]
In[2103]:= CompActPF1[Z]
Out[2103]= {"System", "Cos", "Sin", "StringLength"}, {"HowAct", "HowAct[x_]"}

```

Отметим, для эффективной обработки сохраненных полных определений функций, блоков и модулей применена процедура, вызов которой **ToDefOptPF[x]** оптимизирует в текущем сеансе *определение* блока, функции или модуля пользователя *x*. Правда, для оптимизации определений имеются и другие средства, рассмотренные в книге выше. Завершает предыдущий фрагмент одна довольно полезная модификация процедуры **CompActPF**, вызов которой **CompActPF1[x]** возвращает вложенный список, элементы которого представляют собой подписки следующего формата, а именно:

- подписок с первым элементом **"System"** определяет вызовы системных функций в определении блока, функции либо модуля *x*;
- подписок с первым элементом **"Undefined"** определяет имена объектов, которые не входят в список аргументов и локальных переменных блока, функции либо модуля *x*;
- подписок формата, отличного от вышеуказанных, содержит пользовательские пары {блок/функция/модуль, его заголовок}, чьи вызовы имеются в определении объекта *x*.

Процедура **CompActPF1** оказывается довольно полезной в целом ряде приложений, в первую очередь системного характера, обеспечивая структурный анализ средств.

Как хорошо известно [99], система *Maple* располагает рядом *процедурных* переменных (где под *процедурными* понимаются переменные, имеющие смысл только в теле блока либо модуля и получающие значения о компонентах объекта, содержащего их), обеспечивающих, например, возможность получать список формальных аргументов блока или модуля в его теле при вызове. В *Mathematica* подобные средства отсутствуют, хотя во многих случаях представляют вполне определенный интерес. Некоторые средства подобного типа для *Mathematica* представлены выше. Несложно заметить, что средства системы *Maple* в данном контексте более развиты, чем подобные средства *Mathematica*, что в ряде случаев достаточно существенно упрощает процедурное программирование.

Блок и модуль обеспечивают *четыре* основных механизма возврата результатов своего вызова: (1) через последнее предложение своего тела, (2) по функции **Return**, (3) через

глобальные переменные и (4) через формальные аргументы. Этот вопрос достаточно детально был рассмотрен в наших книгах [97-101]. Следующий фрагмент на примере достаточно простой процедуры *P* весьма наглядно иллюстрирует механизм возврата произвольного числа результатов через аргумент кортеж *z* – кортеж неопределенных переменных. При этом, для упрощения присвоения возвращаемых результатов через элементы списка *z* используется простая и одновременно полезная функция **AssignL**.

```
In[2061]:= P[x_, y_, z_] := DeleteDuplicates[Map[! HowAct[#] &, {z}]] == {True}] :=
Module[{a = 78, b = 460, c = 71},
  If[x*y > 460, AssignL[{z}[[1]], a]; AssignL[{z}[[2]], b];
  AssignL[{z}[[3]], c]; (x + y)*(a + b + c)]

In[2062]:= P[42, 47, m, n, p]
Out[2062]= 54201
In[2063]:= {m, n, p}
Out[2063]= {78, 460, 71}
In[2064]:= First[{x, y, z}] = 78
Set::write: Tag First in First[{x, y, z}] is Protected. >>
Out[2064]= 78
In[2065]:= {x, y, z}
Out[2065]= {x, y, z}
In[2066]:= {x, y, z}[[2]] = 78
Set::setps: {x, y, z} in the part assignment is not a symbol. >>
Out[2066]= 78
In[2067]:= {x, y, z}
Out[2067]= {x, y, z}
In[2068]:= AssignL[x_, y_, z_] := If[{z} != {}, x := y, x = y]
In[2069]:= AssignL[{x, y, z}[[2]], 78]
Out[2069]= 78
In[2070]:= {x, y, z}
Out[2070]= {x, 78, z}
In[2071]:= AssignL[{a1, a2, a3, a4, a5, a6}[[3 ;; 5]], {42, 47, 78}]
Out[2071]= {42, 47, 78}
In[2072]:= {a1, a2, a3, a4, a5, a6}
Out[2072]= {a1, a2, 42, 47, 78, a6}
In[2073]:= AssignL[{{a, b}, {c, d}}[[1, 2]], 78, Delayed]
In[2074]:= {{a, b}, {c, d}}
Out[2074]= {{a, 78}, {c, 78}}
```

Вызов функции **AssignL**[*x*, *y*] обеспечивает корректное присвоение элементам списка *x* (всем или заданным) выражения либо выражений из списка *y*, моделируя присвоения на основе конструкций вида  $\{x, y, z, \dots\}[[n]] = E$  и  $\{x, y, z, \dots\}[[n ;; p]] = \{En, En+1, \dots, Ep\}$  и им подобных, которые пакет не поддерживает, тогда как вызов функции **AssignL**[*x*, *y*,

`j]`, где  $j$  – произвольное выражение, обеспечивает корректные *отложенные* присвоения указанного выше вида, как наглядно иллюстрирует приведенный выше фрагмент.

Как уже отмечалось ранее и использовалось в некоторых процедурах, в *Mathematica* наряду с простыми процедурами, не содержащими в своем теле определений других процедур, допускается использование т.н. *вложенных* процедур, т.е. таких процедур, определения которых находятся в теле других процедур. Уровень вложенности таких процедур определяется лишь размером рабочей области пакета. В этой связи можно рассматривать достаточно интересную задачу по определению списка подпроцедур, *определения* которых находятся в теле произвольной процедуры типа *{Block, Module}*. Эту задачу успешно решает процедура **SubProcs**, чей вызов **SubProcs[W]** возвращает вложенный 2-элементный список *ListList*-типа, первый элемент которого определяет подсписок заголовков блоков и модулей, составляющих главную процедуру *W*, тогда как второй элемент определяет подсписок сгенерированных имен блоков и модулей, составляющих главную процедуру *W*, включая саму процедуру *W*, и активированных в текущем сеансе системы *Mathematica*. Следующий фрагмент представляет *исходный* код процедуры **SubProcs** наряду с наиболее типичными примерами ее применения.

```
In[876]:= SubProcs[P_;/; BlockModQ[P]] := Module[{a = Flatten[{PureDefinition[P]}][[1]],
      b, c = {}, d, t, h, k = 1, p = {}, g = {}},
      b = StringPosition[a, {""] := Block[{"", ""] := Module[{}];
      For[k, k <= Length[b], k++, d = b[[k]];
      p = Append[p, ExprOfStr[a, d[[1]], -1, {" ", ",", ";"}]];
      c = Append[c, h = ExprOfStr[a, d[[1]], -1, {" ", ",", ";"}] <> " := " <>
      ExprOfStr[a, d[[1]] + 5, 1, {" ", ",", ";"}]];
      t = Flatten[StringPosition[h, "["]];
      h = Quiet[StringReplacePart[h, ToString[
      Unique[ToExpression[StringTake[h, {1, t[[1]] - 1}]]], {1, t[[1]] - 1}]];
      g = Append[g, StringTake[h, {1, Flatten[StringPosition[h, "["]][[1]] - 1}]; h]];
      Map[ToExpression, c]; {p, Map[ToExpression, g]}]

In[877]:= P[x_, y_] := Module[{a, b, B, P1, P2}, P1[z_, h_] := Module[{m, n}, z + h]; B[h_] :=
Block[{}, h]; P2[z_] := Module[{P3}, P3[h_] := Module[{}, h]; P3[z]]; x*P2[x]+P1[x, y]+P2[y]]
In[878]:= P[78, 460]
Out[878]= 7082
In[879]:= SubProcs[P]
Out[879]= {"P[x_, y_]", "P1[z_, h_]", "B[h_]", "P2[z_]", "P3[h_]", {P$3115, P1$3120, B$3124,
      P2$3130, P3$3135}}

In[880]:= DefFunc[P2$3130]
Out[880]= P2$3130[z_] := Module[{P3}, P3[h_] := Module[{}, h]; P3[z]]
```

Итак, между элементами подсписков возвращаемого вложенного списка имеет место взаимно-однозначное соответствие. Наряду со стандартными процедура использует наши процедуры **BlockModQ**, **ExprOfStr**, **PureDefinition**, рассмотренные в настоящей книге. Процедура допускает целый ряд достаточно интересных расширений.

Довольно полезная процедура **SubProc1** обеспечивает тестирование блока/модуля  $x$  на предмет наличия в его определении *блоков/модулей*. Вызов процедуры **SubProcs1**[ $x$ ] в зависимости от существования *одноименных* с  $x$  процедур с различными заголовками или с одним заголовком возвращает *вложенный* или простой список; при этом, первые элементы списка либо подсписков определяют заголовки объекта  $x$ , тогда как вторые определяют количество блоков/модулей, которые входят в определение объекта  $x$  с соответствующим заголовком. Если же объект  $x$  не блок, функция или модуль, вызов процедуры **SubProcs1**[ $x$ ] возвращается *невычисленным*. Фрагмент приводит исходный код процедуры с наиболее типичными примерами ее применения. **SubProcs1** может быть довольно несложно расширена на извлечение всех подпроцедур процедуры  $x$ .

```
In[2047]:= SubProcs1[x_;/ BlockFuncModQ[x]] := Module[{b = {}, c, d, k = 1,
               a = Flatten[{PureDefinition[x]}]},
               For[k, k <= Length[a], k++, c = a[[k]];
               d = StringPosition[c, {""] := Module[{"", ""] := Block[{""]; If[d == {}, Continue[]];
               b = Append[b, {StringTake[c, {1, d[[1]][[1]]}], Length[d] - 1}];
               If[Length[b] == 1, Flatten[b], b]]

In[2048]:= G[x_, y_, z_] := x + y + z; G[x_] := Module[{V, H}, V[y_] := Module[{}, y^3];
               H[z_] := Module[{}, z^4]; x + V[x] + H[x]; G[x_, z_] := Module[{V, H, P},
               V[t_] := Module[{}, t^3 + t^2 + 460]; H[t_] := Module[{}, t^4];
               P[h_] := Module[{a = 78}, a^2 + h^2]; x + V[x] + H[z]*P[x];
               H[t_] := Module[{P}, P[h_] := Module[{a = 78}, a^2 + h^2]; x + P[x]]

In[2049]:= SetAttributes[G, {Protected, Listable}]; {G[2013], G[2013, 71]}
Out[2049]= {16428230622771, 103135191724132}
In[2050]:= SubProcs1[G]
Out[2050]= {{ "G[x_]", 2}, { "G[x_, z_]", 3}}
In[2051]:= SubProcs1[H]
Out[2051]= { "H[t_]", 1}
In[2052]:= SubProcs1[78]
Out[2052]= SubProcs1[78]
In[2096]:= P[x_;/ {w[b_] := Module[{}, b^2], If[EvenQ[x], True, False]}][[2]] :=
               Module[{a = {c[d_] := Module[{}, d]}}, {w[x], c[x]}]

In[2097]:= P[2014]
Out[2097]= {4056196, 2014}
In[2098]:= Map[Definition1, {w, c}]
Out[2098]= {"w[b_] := Module[{}, b^2]", "c[d_] := Module[{}, d]"}
```

Завершает данный фрагмент весьма простой пример, иллюстрирующий некоторые допустимые механизмы определения заголовка блока/функции/модуля, и локальных переменных блока и модуля, довольно полезных в целом ряде случаев процедурного программирования. Такие механизмы использует и ряд средств, составляющих наши пакеты *AVZ\_Package* [90] и *AVZ\_Package\_1* [120]. В то время как процедура **SubProcs2**

представляет довольно существенное расширение процедуры **SubProcs1**. Следующий фрагмент представляет исходный код процедуры **SubProcs2** наряду с примерами ее типичного применения. Более того, на всех типах аргумента *y*, отличных от **Module**, вызов процедуры **SubProcs2[y]** возвращает значение **\$Failed**.

```
In[2386]:= G[x_] := Module[{V, H}, Vg[y_] := Module[{}, y^3]; H71[z_] := Module[{}, z^4];
  x + Vg[x] + H71[x]; G[x_, z_] := Module[{Vt, H, P}, Vt[t_] := Module[{}, t^3 + t^2 + 460];
  H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 78}, a^2 + h^2]; x + Vt[x] + H[z]*P[x];
  H[t_] := Module[{P}, P[h_] := Module[{a = 78}, a^2 + h^2]; t + P[t];
  F[x_, y] := x + y; SetAttributes[G, {Protected, Listable}]; {G[2013], G[2013, 71]}
Out[2386]= {16428230622771, 103135191724132}

In[2387]:= SubProcs2[y_, z_] := Module[{v = Flatten[{PureDefinition[y]}], n = {}, m = 1, SB},
  If[BlockFuncModQ[y],
    SB[x_String] := Module[{a = Map[#[[1]] &, StringPosition[x, "Module[{"}],
      b = "Module[" , c, d, h, g = "", t, k, p, q, j, s, w],
    If[a == {}, Return[]]; If[Length[a] == 1, Return[$Failed], d = Map[# - 5 &, a];
      c = {StringTake[x, {1, d[[1]]}]}];
    For[k = Length[a], k > 1, k--, h = b; g = ""; t = "";
      For[j = a[[k]] + 7, j < Infinity, j++, h = h <> StringTake[x, {j, j}];
    If[SameQ[Quiet[Check[ToExpression[h], "Error"], "Error"], Continue[],
      For[j = d[[k]], j > 1, j--, g = StringTake[x, {j, j}] <> g;
    If[SameQ[Quiet[Check[ToExpression[g], "Error"], "Error"], Continue[], Break[]];
    While[j > 1, p = StringTake[x, {j, j}]; If[! SameQ[p, " "], t = p <> t, Break[]]; j--;
      p = StringPosition[x, " " <> t <> "["][[1]];
    s = Flatten[SubStrSymbolParity1[StringTake[x, {p[[1]], -1}], {" ", " "}]];
      w = 1; While[w <= Length[s] - 1, q = s[[w]];
    If[! StringFreeQ[q, "_"], s = t <> q <> " := Module" <> s[[w + 1]]; Break[]]; w++;
      c = Append[c, s]; Break[]]; c];
    For[m, m <= Length[v], m++, n = Append[n, SB[v[[m]]]];
      n = Select[n, ! SameQ[#, Null] &];
    If[Length[n] == 1, n[[1]], n]; If[{z} != {}, ToExpression[n]; n, $Failed]

In[2388]:= SubProcs2[G, t]
Out[2388]= {"G[x_]", "H71[z_] := Module[{}, z^4]", "Vg[y_] := Module[{}, y^3]",
  {"G[x_, z_]", "P[h_] := Module[{a = 78}, a^2 + h^2]", "H[t_] := Module[{}, t^4]",
  "Vt[t_] := Module[{}, t^3 + t^2 + 460]"}
In[2389]:= {H71[78], Vg[460], P[24], H[17], Vt[66]}
Out[2389]= {37015056, 97336000, 6660, 83521, 292312}
In[2390]:= SubProcs2[H]
Out[2390]= {"H[t_]", "P[h_] := Module[{a = 78}, a^2 + h^2]"}
In[2391]:= Map[SubProcs2, {F, 78}]
Out[2391]= {$Failed, $Failed}
```

Вызов **SubProcs2**[*y*] в зависимости от уникальной процедуры *y* или одноименных, но с *различными* заголовками, возвращает простой или вложенный список. У списка либо подсписков первый элемент – *заголовок* процедуры *y*, тогда как остальные – *определения* подпроцедур типа **Module**, входящих в определение *y*. В отсутствие у *y* подпроцедур указанного типа возвращается **\$Failed**. В случае второго необязательного аргумента *z*, в качестве которого может быть произвольное выражение, возвращается аналогичный результат с одновременной активизацией данных подпроцедур в текущем сеансе.

В качестве дальнейшего расширения процедуры **SubProcs2** предлагается процедура, чей вызов **SubProcs3**[*y*] отличен от вызова **SubProcs2**[*y*] следующими 2-я моментами, а именно: (1) в качестве аргумента *y* может выступать пользовательский блок, функция или модуль, и (2) возвращаемый список в качестве *первого* элемента содержит *заголовок* объекта *y*, тогда как остальные элементы списка представляют *определения* входящих в в определение *y* *блоков, функций и модулей* в строчном формате. В случае *одноименного* объекта *y* возвращаемый список будет вложенным, чьи подсписки имеют указанный выше формат. При этом, вызов **SubProcs3**[*y*, *z*] со вторым необязательным аргументом *z* – произвольным выражением – возвращая указанный выше список, одновременно делает активными в текущем сеансе все входящие в *y* объекты указанного выше типа. Фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2093]:= G[x_] := Module[{Vg, H71}, Vg[y_] := Module[{}, y^3]; H71[z_] := Module[{}, z^4];
  x + Vg[x] + H71[x]; G[x_, z_] := Module[{Vt, H, P}, Vt[t_] := Module[{}, t^3 + t^2 + 460];
  H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 78}, a^2 + Cos[h^2]];
  Sin[x] + Vt[x] + H[z]*P[x]; H[t_] := Module[{P}, P[h_] := Module[{a = 78}, a^2 + h^2];
  Cos[t] + P[t]; F[x_, y_] := Sin[x + y] + Cos[x - y];
  V[x_] := Block[{a, b, c}, a[m_] := m^2; b[n_] := n + Sin[n]; c[p_] := Module[{}, p];
  a[x]*b[x]*c[x]; SetAttributes[G, {Protected, Listable}]

In[2094]:= SubProcs3[y_, z_] := Module[{v = Flatten[{PureDefinition[y]}], u = {}, m = 1, Sv},
  If[BlockFuncModQ[y],
    Sv[S_String] := Module[{a = ExtrVarsOfStr[S, 1], b, c = {}, d, t = 2, k = 1, cc = {}, n, p, j,
      h = {StringTake[S, {1, Flatten[StringPosition[S, " := "][[1]] - 1]}],
      a = Select[a, ! SystemQ[Symbol[#]] && ! MemberQ[{ToString[G]}, #] &];
      b = StringPosition[S, Map[" " <> # <> "[" &, a]];
      p = Select[a, ! StringFreeQ[S, " " <> # <> "[" &];
      b = Flatten[Map[SubStrSymbolParity1[StringTake[S, {#[[1]], -1}], "[", "]" &, b]];
      For[j = 1, j <= Length[p], j++, n = p[[j]];
      For[k = 1, k <= Length[b] - 1, k++, d = b[[k]];
      If[! StringFreeQ[d, "_"] && StringTake[b[[k + 1]], {1, 1}] == "[",
        AppendTo[c, Map[n <> d <> " := " <> # <> b[[k + 1]] &, {"Block", "Module"}]]];
      c = DeleteDuplicates[Flatten[c]];
      For[k = 1, k <= Length[c], k++, d = c[[k]];
      If[! StringFreeQ[S, d], AppendTo[h, d],
```



```

AppendTo[cc, StringTake[d, {1, Flatten[StringPosition[d, " := "][[1]] - 1}]]];
{h, cc} = Map[DeleteDuplicates, {h, cc}];
p = Map[StringTake[#, {1, Flatten[StringPosition[#, "["][[1]]} &, h];
cc = Select[Select[cc, ! SuffPref[#, p, 1] &], ! StringFreeQ[S, #] &];
If[cc == {}, h, For[k = 1, k <= Length[cc], k++, p = cc[[k]];
p = StringCases[S, p <> " := " ~~ __ ~~ ";"];
AppendTo[h, StringTake[p, {1, Flatten[StringPosition[p, ";"][[1]] - 1}]]]; Flatten[h]];
For[m, m <= Length[v], m++, AppendTo[u, Sv[v[[m]]]];
u = Select[u, ! SameQ[#, Null] &];
u = If[Length[u] == 1, u[[1]], u]; If[{z} != {}, ToExpression[u]]; u, $Failed]]

In[2095]:= SubProcs3[G]
Out[2095]= {"G[x_]", "Vg[y_] := Module[{}, y^3]", "H71[z_] := Module[{}, z^4]",
{"G[x_, z_]", "Vt[t_] := Module[{}, t^3 + t^2 + 460]", "H[t_] := Module[{}, t^4]",
"P[h_] := Module[{a = 78}, a^2 + Cos[h^2]]"}

In[2096]:= SubProcs3[H]
Out[2096]= {"H[t_]", "P[h_] := Module[{a = 78}, a^2 + h^2]"}

In[2097]:= SubProcs3[F]
Out[2097]= {"F[x_, y_]"}

In[2098]:= SubProcs3[V]
Out[2098]= {"V[x_]", "c[p_] := Module[{}, p]", "a[m_] := m^2", "b[n_] := n + Sin[n]"}

In[2099]:= SubProcs3[V, 78]
Out[2099]= {"V[x_]", "c[p_] := Module[{}, p]", "a[m_] := m^2", "b[n_] := n + Sin[n]"}

In[2100]:= {V[78], a[42], b[47], c[67]}
Out[2100]= {474552 (78 + Sin[78]), 1764, 47 + Sin[47], 67}

```

Если в качестве объекта *y* выступает функция с заголовком, то возвращается лишь ее заголовок; аналогичный результат имеет место и в случае, если объект *y* не содержит подобъектов указанного типа, тогда как на объекте *y*, отличном от пользовательского блока, функции либо модуля, вызов процедуры **SubProcs3** возвращает значение **\$Failed**.

В ряде случаев возникает необходимость определения у блока и модуля *подобъектов* типа блок, функция либо модуль. Вызов процедуры **SubsProcQ[x, y]** возвращает **True**, если *y* является *глобальным* активным подобъектом объекта *x* вышеуказанного типа, и **False** в противном случае. Но так как *Math*-объекты данного типа различаются не по именам, как это принято в большинстве систем программирования, а заголовками, то через 3-й необязательный аргумент вызов процедуры возвращает вложенный список, подписки которого первым элементом содержат заголовок объекта с именем *x*, но с разными заголовками, вторым – соответствующие им *заголовки* подобъектов с именем *y*. На первых двух аргументах типов, отличных от указанных в заголовке процедуры, вызов процедуры **SubsProcQ[x, y]** возвращает **False**. Следующий фрагмент приводит исходный код процедуры **SubsProcQ** наряду с примерами применения. В принципе, на основе вышеприведенных средств можно программировать целый ряд достаточно полезных средств работы с выражениями типов *{Block, Module}*.

```

In[2100]:= SubsProcQ[x_, y_, z_] := Module[{a, b, k = 1, j = 1, Res = {}},
    If[BlockModQ[x] && BlockFuncModQ[y],
    {a, b} = Map[Flatten, {{Definition4[ToString[x]]}, {Definition4[ToString[y]]}}];
    For[k, k <= Length[b], k++, For[j, j <= Length[a], j++,
    If[! StringFreeQ[a[[j]], b[[k]], Res = Append[Res,
    {StringTake[a[[j]], {1, Flatten[StringPosition[a[[j]], " := "][[1]] - 1}},
    StringTake[b[[k]], {1, Flatten[StringPosition[b[[k]], " := "][[1]] - 1}}], Continue[]];
    If[Res != {}, If[{z} != {} && ! HowAct[z],
    z = If[Length[Res] == 1, Res[[1], Res]; True], False], False]]
In[2101]:= V[x_] := Block[{a, b, c}, a[m_] := m^2; b[n_] := n + Sin[n]; c[p_] := Module[{}, p];
    a[x]*b[x]*c[x]; c[p_] := Module[{}, p]; V[x_, y_] := Module[{a, b, c},
    a[m_] := m^2; b[n_] := n + Sin[n]; c[p_] := Module[{}, p]; a[x]*b[x]*c[x];
    c[p_] := Module[{}, p]; p[x_] := x; SetAttributes[V, Protected]
In[2102]:= {SubsProcQ[V, c, h7], h7}
Out[2102]= {True, {"V[x_]", "c[p_]"}, {"V[x_, y_]", "c[p_]"}]}
In[2103]:= SubsProcQ[V, p]
Out[2103]= False
In[2104]:= SubsProcQ[Sin, p]
Out[2104]= False
In[2105]:= SubsProcQ[p, c]
Out[2105]= False

```

В определенной степени представляет интерес процедурная переменная **\$ProcName**, которая используется лишь в теле процедуры, активизированной в текущем сеансе, и возвращает список, чей первый элемент определяет *имя*, тогда как *второй* – *заголовок* в строчном формате содержащей ее процедуры. Более того, для обеспечения данной возможности в списке *локальных* переменных процедуры, содержащей переменную **\$ProcName**, нужно закодировать выражение вида **\$\$NameProc\$\$** = «Имя\_процедуры», иначе вызов процедуры в качестве значения для переменной **\$ProcName** возвращает **"UndefinedName"**. Фрагмент представляет код процедурной переменной **\$ProcName**, реализованный на основе модульной конструкции, с примерами ее использования.

```

In[2062]:= $ProcName := Module[{d = "$$ArtKr$$", a, b, c, t = "", k}, a = ToString1[Stack[_]];
    d = Flatten[StringPosition[a, d]][[1]];
    b = Flatten[StringPosition[a, "$$NameProc$$"]][[1]];
    If[b > d || ToString[b] == "", Return["UndefinedName"], k = b];
    For[k = b, k <= d, k++, c = StringTake[a, {k, k}];
    If[MemberQ[{"", " "}, c], Break[], t = t <> c; Continue[]];
    {b = ToExpression[ToExpression[StringSplit[t, "="][[2]]], HeadPF[b]]}
In[2063]:= Avz[x_, y_, z_] := Module[{$$NameProc$$ = "Avz", b}, b = $ProcName; x+y+z; b]
In[2064]:= Agn[x_, y_, z_] := Module[{b, $$NameProc$$ = "Agn"}, x+y+z; b = $ProcName; b]

```

```
In[2065]:= Ian[x_, y_, z_] := Module[{b, c, h}, x+y+z; b = $ProcName; b]
In[2066]:= Agn[47, 66, 78]
Out[2066]= {Agn, "Agn[x_, y_, z_]"}
In[2067]:= Avz[47, 66, 78]
Out[2067]= {Avz, "Avz[x_, y_, z_]"}
In[2068]:= Ian[47, 66, 78]
Out[2068]= "UndefinedName"
```

Данная переменная в определенной степени была навеяна процедурной переменной *procname* системы *Maple*, которая играет довольно существенную роль, прежде всего, в процедурном программировании различных задач системного характера.

Вполне определенный интерес представляет достаточно несложная процедура, вызов которой **BFMSubsQ**[*x*] возвращает список формата {**True**, *Heading*}, если определение блока или модуля *x* содержит определения блоков, функций и/или модулей, и список {**False**, *Heading*} в противном случае. В случае *одноименного* объекта *x* указанного типа вызов возвращает вложенный список, подписки которого имеют указанный формат. На объекте *x* типа, отличного от {*Block*, *Module*}, вызов процедуры возвращает **False**. При этом, вызов **BFMSubsQ**[*x*, *y*] со 2-м необязательным аргументом *y* – *неопределенной переменной* – через *y* возвращает список формата {*Heading*, *N*}, где *N* определяет число блоков, функций, модулей, входящих в подобъект с заголовком *Heading* *одноименного* объекта *x*. Следующий фрагмент представляет исходный код процедуры **BFMSubsQ** наряду с рядом наиболее типичных примеров ее применения. Отметим, что данная процедура достаточно полезна в целом ряде приложений, связанных с обработкой процедур блочного и модульного типа, и, в первую очередь, системного характера.

```
In[2145]:= G[x_] := Module[{Vg, H7}, Vg[y_] := Module[{}, y^3]; H7[z_] := Module[{}, z^4];
x+Vg[x] + H7[x]; G[x_, z_] := Module[{Vt, H, P}, Vt[t_] := Module[{}, t^3 + t^2];
H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 78}, a^2 + Cos[h^2]];
Sin[x]+Vt[x]+H[z]*P[x]; H[t_] := Module[{P}, P[h_] := Module[{a = 78}, a^2+h];
T[x_] := Block[{a}, a[y_] := y^2; x + a[460]]; T[x_, y_] := Module[{a = 6}, x*y + a
Cos[t] + P[t]]; F[x_, y_] := Sin[x + y] + Cos[x - y];
SetAttributes[G, {Protected, Listable}]

In[2146]:= BFMSubsQ[x_, y_] := Module[{a, b, c, d = {}, k = 1, p, h, g = {}},
If[! BlockModQ[x], False, {a, b} = Map[Flatten, {{PureDefinition[x]}, {HeadPF[x]}}];
For[k, k <= Length[a], k++, p = a[[k]];
p = StringReplace[p, b[[k]] <> " := " -> "", 1];
t = Select[ExtrVarsOfStr[p, 1], ! SystemQ[#] &];
h = Flatten[Map[StrSymbParity[p, " " <> #, "[", "]" ] &, t]];
h = Select[h, SuffPref[#, Map[StringJoin[" " <> # <> "[" ] &, t], 1] &&
! StringFreeQ[#, "_"] &];
AppendTo[g, {b[[k]], Length[h]}]; AppendTo[d, {If[h != {}, True, False], b[[k]]}];
If[{y} != {} && ! HowAct[y], y = g]; If[Length[d] == 1, d[[1]], d]]
```

```
In[2147]:= BFMSubsQ[H]
Out[2147]= {True, "H[t_]"}
In[2148]:= BFMSubsQ[G]
Out[2148]= {{True, "G[x_]"}, {True, "G[x_, z_]"} }
In[2149]:= BFMSubsQ[T]
Out[2149]= {{True, "T[x_]"}, {False, "T[x_, y_]"} }
In[2150]:= Map[BFMSubsQ, {F, 78, Agn, Sin}]
Out[2150]= {False, False, False, False}
In[2051]:= BFMSubsQ[G, g]
Out[2051]= {{True, "G[x_]"}, {True, "G[x_, z_]"} }
In[2052]:= g
Out[2052]= {"G[x_]", 2}, {"G[x_, z_]", 3}
```

Данная процедура обобщает и расширяет наши процедуры **SubProcsQ**, **SubProcsQ1** и **SubsProcN**, представленные в [90,100,101]. Процедуры достаточно полезны в целом ряде приложений, прежде всего, приложений системного характера.

На основе процедур **BlockModQ**, **HeadPF**, **Mapp**, **PureDefinition**, **SubStrSymbolParity1**, рассматриваемых в настоящей книге, реализована и достаточно полезная процедура **ProcBody**, вызов которой **ProcBody[x]** возвращает тело блока, функции с заголовком, модуля *x* в строчном формате. Процедура успешно обрабатывает также *одноименные* объекты *x*, возвращая *список* тел подобъектов, составляющих *x*. Следующий фрагмент приводит исходный код процедуры **ProcBody** с типичными примерами применения. Процедура играет достаточно существенную роль в целом ряде задач процедурного программирования, имеющих дело с различными манипуляциями с определениями функций, процедур типа блок либо модуль и составляющими их компонентами.

```
In[2093]:= ProcBody[x_ /; BlockModQ[x] | | QFunction[x]] := Module[{c, d = {}, k = 1, p,
    a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}]},
    While[k <= Length[a], p = a[[k]];
    c = Mapp[Rule, Map[b[[k]] <> " := " <> # &, {"Block["", "Module["", ""], ""}];
    c = StringReplace[p, c, 1]; AppendTo[d, If[BlockModQ[x],
    StringTake[StringReplace[c,
    SubStrSymbolParity1[c, "{", "}"]][[1]] <> ", " -> "", 1, {1, -2}], c]]; k++;
    If[Length[d] == 1, d[[1]], d]]

In[2094]:= Art[x_, y_, z_] := Module[{a = x + y + z, c = {m, n}, b = 78}, a^2 + a + b]
In[2095]:= ProcBody[Art]
Out[2095]= "a^2 + a + b"
In[2096]:= T[x_] := Block[{a}, a[y_] := y^2; x + a[78]]; T[x_, y_] := Module[{a = 6}, x*y + a]
In[2097]:= ProcBody[T]
Out[2097]= {"a[y_] := y^2; x + a[78]", "x*y + a"}
In[2098]:= F[x_, y_] := x + y; F[x] := Sin[x] + x*Cos[x]; ProcBody[F]
Out[2098]= {"Sin[x] + x*Cos[x]", "x + y"}
```

В ряде задач, вызванных обработкой строчного представления *определений* процедур, вопрос разделения данного представления на две основных компоненты, а именно: *тела* процедуры и его *обрамления* с заключительной процедурной скобкой "]" может представить определенный интерес. В данном контексте и процедура **PartProc** может оказаться довольно полезной. Вызов процедуры **PartProc[x]** возвращает 2-элементный список, чей первый элемент в строчном формате представляет *обрамление* процедуры с заключительной *процедурной* скобкой "]" ; место *тела* процедуры занимает подстрока "*Procedure Body*", тогда как второй элемент списка в строчном формате представляет *тело* процедуры *x*. При этом, под *обрамлением* процедуры понимается конструкция формата "**Heading := Module[{locals}, ... ]**". В ошибочных ситуациях вызов процедуры возвращается невычисленным либо вызов возвращает **\$Failed**. Следующий фрагмент представляет исходный код процедуры **PartProc** с примерами ее использования.

```
In[2049]:= PartProc[P_ /; BlockModQ[P]] := Module[{a = ProcBody[P]},
      {StringReplace[PureDefinition[P], a -> "Procedure Body", 1], a}]
In[2050]:= Kr[x_, y_, z_] := Module[{a = x + y + z, b = 78}, b*a + a^2 + b]; PartProc[Kr]
Out[2050]= {"Kr[x_, y_, z_] := Module[{a = x+y+z, b = 78}, Procedure Body]", "b*a+a^2+b"}
In[2054]:= ReplaceProcBody[x_ /; BlockModQ[x], y_ /; StringQ[y]] :=
      ToExpression[StringReplace[PureDefinition[x], ProcBody[x] -> y]]
In[2055]:= ReplaceProcBody[Kr, "b*(x + y + z)"]; Definition[Kr]
Out[2055]= Kr[x_, y_, z_] := Module[{a = x + y + z, b = 78}, b*(x + y + z)]
```

Завершает фрагмент довольно простая функция, вызов которой **ReplaceProcBody[x, y]** возвращает **Null**, т.е. ничего, обеспечивая замену тела блока либо модуля *x* на новое тело *y*, которое задается в строчном формате. Как процедура **PartProc**, так и функция **ReplaceProcBody** основываются на выше рассмотренной процедуре **ProcBody**. Именно данное обстоятельство обеспечивает довольно простой алгоритм указанных средств.

Кроме рассмотренных в книге представлен ряд средств по работе с подпроцедурами, здесь мы представим довольно полезную процедуру, анализирующую блоки/модули на предмет присутствия в их определениях подобъектов типа **{Block, Module}**. Вызов процедуры **SubsProcs[x]** в общем случае возвращает *вложенный* список определений в строчном формате *всех* подобъектов типа **{Block, Module}**, чья определения находятся в теле объекта *x* типа **{Block, Module}**. *Первый* подсписок определяет подобъекты типа **Module**, *второй* – **Block**-типа. При наличии лишь *1* подписка возвращается простой список. При отсугствие подобъектов этого типа вызов **SubsProcs[x]** возвращает *пустой* список, т.е. {}, тогда как на объекте *x*, отличном от *блока* или *модуля*, вызов **SubsProcs[x]** возвращается невычисленным. Фрагмент представляет как исходный код процедуры **SubsProcs**, так и наиболее типичные примеры ее использования.

```
In[2095]:= SubsProcs[x_ /; BlockModQ[x]] := Module[{a = PureDefinition[x], j, m = 1, n = 0,
      b = {" := Module[{", " := Block[{", c = ProcBody[x],
      d, s = {}, g, k = 1, p, h = "", v = 1, R = {}, Res = {}},
      For[v, v <= 2, v++, If[StringFreeQ[c, b[[v]]], Break[], d = StringPosition[c, b[[v]]]]];
```

```

For[k, k <= Length[d], k++, j = d[[k]][[2]]; While[m != n, p = StringTake[c, {j, j}];
  If[p == "[", m++; h = h <> p, If[p == "]", n++; h = h <> p, h = h <> p]]; j++];
  Res = Append[Res, h]; m = 1; n = 0; h = "";
  Res = Map10[StringJoin, If[v == 1, " := Module[" , " := Block["], Res];
    g = Res; {Res, m, n, h} = {{}, 1, 0, ""};
    For[k = 1, k <= Length[d], k++, j = d[[k]][[1]] - 2;
      While[m != n, p = StringTake[c, {j, j}];
        If[p == "]", m++; h = p <> h, If[p == "[", n++; h = p <> h, h = p <> h]]; j--];
        Res = Append[Res, h]; s = Append[s, j]; m = 1; n = 0; h = "";
        Res = Map9[StringJoin, Res, g]; {g, h} = {Res, ""}; Res = {};
        For[k = 1, k <= Length[s], k++, For[j = s[[k]], j >= 1, j--, p = StringTake[c, {j, j}];
          If[p == " ", Break[], h = p <> h]]; Res = Append[Res, h]; h = "";
          R = Append[R, Map9[StringJoin, Res, g]];
          {Res, m, n, k, h, s} = {{}, 1, 0, 1, "", {}]; If[Length[R] == 2, R, Flatten[R]]]
In[2096]:= P[x_, y_] := Module[{Art, Kr, Gs, Vg, a}, Art[c_, d_] := Module[{b}, c + d];
  Vg[h_] := Block[{p = 78}, h^3 + p]; Kr[n_] := Module[{}, n^2];
  Gs[z_] := Module[{}, x^3]; a = Art[x, y] + Kr[x*y]*Gs[x + y] + Vg[x*y]
In[2097]:= P[78, 460]
Out[2097]= 657117089741416
In[2098]:= SubsProcs[P]
Out[2098]= {"Art[c_, d_] := Module[{b}, c + d]", "Kr[n_] := Module[{}, n^2]",
  "Gs[z_] := Module[{}, x^3]", {"Vg[h_] := Block[{p = 78}, h^3 + p]"]}
In[2099]:= H[t_] := Module[{P}, P[h_] := Module[{a = 78}, a^2 + h^2]; Cos[t] + P[t]
In[2100]:= SubsProcs[H]
Out[2100]= {"P[h_] := Module[{a = 78}, a^2 + h^2]"}

```

Процедура **SubsProcs** может быть относительно несложно расширена, в частности, на предмет определения уровней вложенности подпроцедур, а также непоименованные подпроцедуры. Оставляем это читателю в качестве довольно полезного упражнения. Процедура **SubsProcs** достаточно существенно использует также и наши процедуры **BlockModQ**, **Map10**, **Map9**, **ProcBody**, **PureDefinition**, рассмотренные в книге. Более того, в связи с проблемой вложенности блоков и модулей выявляется существенное *различие* между определениями вложенных процедур в среде систем *Mathematica* и *Maple*. Так, в среде системы *Maple* определения *подпроцедур* допускают использование списков *формальных* аргументов, идентичных с главной процедурой, содержащей их; тогда как в среде системы *Mathematica* подобное совмещение недопустимо, вызывая в процессе вычисления определения главной процедуры ошибочные ситуации [100]. Вообще говоря, именно это обстоятельство вызывает определенные неудобства, ибо требует особой внимательности в процессе программировании *вложенных* процедур. В определенной мере аналогичная ситуация возникает и при пересечениях списков формальных аргументов главной процедуры и локальных переменных подпроцедур, тогда как это вполне допустимо в программной среде системы *Maple* [44,99]. В данном

контексте процедура **SubsProcs** может вполне успешно применяться и к процедурам, содержащим подпроцедуры типа **{Block, Module}**, при условии непустого *пересечения* списка формальных аргументов главной процедуры и списка *локальных* переменных как главной процедуры, так и ее подпроцедур.

Следующая процедура обеспечивает возврат списка всех блоков, модулей и функций из пакетов пользователя, загруженных в текущий сеанс, наряду с другими активными объектами указанных типов. Фрагмент представляет ее код и примеры применения.

```
In[2041]:= ProcsAct[] := Module[{a = Names["*"], b = Names["System`*"], c, d = {}, k = 1, j,
    h, t, g = {"Module"}, {"Block"}, {"DynamicModule"}, {"Function"}, {"Others"}]},
    c = Select[a, ! MemberQ[b, #] &];
    c = Select[c, ToString[Definition[#]] != "Null" &&
ToString[Definition[#]] != "Attributes[" <> ToString[#] <> "]" = {Temporary}" &&
! MemberQ[{ToString[#] <> " = {Temporary}", ToString[#] <> " = {Temporary}"},
ToString[Definition[#]]] &];
    For[k, k <= Length[c], k++, h = c[[k]]; ClearAll[t]; Quiet[ProcQ1[Symbol[h], t]];
    If[t === "Module", AppendTo[g[[1]], h], If[t === "Block", AppendTo[g[[2]], h],
        If[t === "DynamicModule", AppendTo[g[[3]], h],
            If[QFunction[h], AppendTo[g[[4]], h], AppendTo[g[[5]], h]]];]; g

In[2042]:= ProcsAct[]
Out[2042]= {{{"Module", "ActBFMuserQ", "ActCsProcFunc", "ActiveProcess", "ActRemObj",
    "Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes", ...},
    {"Block"}, {"DynamicModule"},
    {"Function", "AssignL", "Attributes1", "AttributesH", "BinaryListQ", "BlockQ1",
    "ComplexQ", "ContextActQ", "ContextFromFile", "ContextQ", "CopyDir", ...},
    {"Others", "AcNb", "ActUcontexts", "ClearOut", "CloseAll", "CsProcsFuncs", ...}}}
```

Вызов процедуры **ProcsAct[]** возвращает вложенный 5-элементный список, подписки которого своим первым элементом определяют *типы* объектов в разрезах **"Module"**, **"Block"**, **"DynamicModule"**, **"Function"** и **"Others"**, активированных в текущем сеансе, тогда как остальные элементы определяют *имена* объектов соответствующего *первому* элементу типа. Между тем, следует отметить, что реактивность процедуры довольно существенно зависит от количества средств пользователя, активированных в текущем сеансе системы. Еще раз следует отметить, что в *Mathematica*-процедурах локальные переменные *изначально* не рассматриваются как неопределенные; между тем, придать им статус неопределенных в теле процедуры возможно, что наглядно иллюстрирует следующий достаточно прозрачный пример, а именно:

```
In[2047]:= A[x___] := Module[{a, b, c}, b = {Attributes[a], Definition[a]}; ClearAll[a];
    c = {Attributes[a], Definition[a]}; {b, c}

In[2048]:= A[]
Out[2048]= {{{Temporary}, Null}, {{}, Null}}}
```

Такой прием использован и в процедуре **ProcsAct**, обеспечивая возврат типа объекта *h* через второй аргумент *t* – *неопределенную переменную* – в вызове **ProcQ1[h, t]**. В целом,

процедура **ProcsAct** представляет вполне определенный интерес в ряде приложений и, прежде всего, в процедурном программировании задач системного характера.

Следующий фрагмент представляет достаточно полезную функцию, вызов которой **NamesProc[]** возвращает отсортированный список имен пользовательских функций, блоков и модулей, активированных в текущем сеансе. В целом ряде случаев функция **NamesProc** может оказаться достаточно полезным средством. Фрагмент представляет исходный код функции **NamesProc** наряду с типичными примерами ее применения.

```
In[4104]:= NamesProc[] := Select[Map[ToExpression, Sort[Names["*"]]],
      Quiet[BlockFuncModQ[#]] && ToString[Definition[#]] != "Null" &&
      ToString[Definition[#]] != "Attributes[" <> ToString[#] <> "]" = {Temporary} &&
      ! MemberQ[{ToString[#] <> " = {Temporary}",
      ToString[#] <> " = {Temporary}"}], ToString[Definition[#]] &]

In[4105]:= NamesProc[]
Out[4105]= {A, Art, Df, F, G, H, Kr, NamesProc, ProcQ, SortNL, Spos, Subs, T, Uprocs}
```

В качестве еще одного примера приведем процедуру **Uprocs**, как довольно полезную в практическом отношении, так и иллюстрирующую подход к некоему расширению стандартных средств *Mathematica*. Вызов **Uprocs[]** возвращает *простой* или *вложенный* список. В первом случае в текущем сеансе не активировалось процедур пользователя любого из двух типов (*модульный* или *блочный*), тогда как во втором случае элементами списка, возвращаемого процедурой **Uprocs**, являются 3-элементные *подписки*, первые элементы которых определяют *имена* блоков/модулей пользователя, активированных в текущем сеансе, вторые определяют их *заголовки* в строчном формате, третьи – *тип* процедур *{Block | Module}*. Следующий фрагмент представляет исходный код **Uprocs** наряду с наиболее типичными примерами ее использования, а именно:

```
In[2447]:= Gs[x_] := Block[{a, b, c}, Evaluate[(a*x + x^b)/c]]
In[2448]:= S[x_] := Block[{y = a, h = b}, G[Pi/2, y*x]]
In[2449]:= S[x_] := Module[{y = a, h = b}, G[Pi/2, y*x]]
In[2450]:= S[x_, y_] := Block[{z = a, h = b}, G[Pi/2, (y*x)/z]]
In[2451]:= Bl[y_] := Block[{h = z}, G[Pi/2, y]]
In[2452]:= MM[x_, y_] := Module[{}, x + y]
In[2453]:= Uprocs[] := Module[{a, b, c, d, h, g, k, t1, t2},
      a := "_$Art24_Kr17$.txt"; Save[a, "*"];
      b := Map[ToString, Flatten[DeleteDuplicates[ReadList[a, String]]]; {c, g} = {}, {};
      For[k = 1, k <= Length[b], If[StringCount[First[b][[k]]], " := Module[" != 0 &&
          StringTake[First[b][[k]]], {1}] != " " | |
          StringCount[First[b][[k]]], " := Block[" != 0 &&
          StringTake[First[b][[k]]], {1}] != " ", c = Append[c, First[b][[k]]], Null]; k = k + 1];
      For[k = 1, k <= Length[c], d = Quiet[First[c][[k]]];
          h = Quiet[Symbol[StringTake[d, First[First[StringPosition[d, "["]]] - 1]]];
```



```

t1 = If[StringCount[d, " := Module["] != 0, Module, Block];
t2 = Quiet[StringTake[d, Last[First[StringPosition[d, "]]]]];
If[BlockModQ[h], g = Append[g, {h, t2, t1}], Null]; k = k + 1]; DeleteFile[a]; g]
In[2454]:= Uprocs[]
Out[2454]= {{AK, "AK[x_]", Block}, {Art, "Art[x_]", Module}, {ProcQ, "ProcQ[x_]", Module},
{S, "S[x_]", Module}, {S, "S[x_, y_]", Block},
{StringEnd, "StringEnd[x_String, y_String]", Module}, {Sv, "Sv[x_]", Module},
{Uprocs, "Uprocs[]", Module}, {ProcName, "ProcName[]", Module},
{Bl, "Bl[y_]", Block}, {BlockQ, "BlockQ[Definition[x_]]", Module},
{Gs, "Gs[x_]", Block}, {GS, "GS[x_, y_]", Module}, {Kr, "Kr[x_]", Block},
{MM, "MM[x_, y_]", Module}, {P, "P[]", Module}}

```

Наряду со стандартными функциями системы процедура **Uprocs** весьма существенно использует и нашу процедуру **BlockModQ**, рассмотренную в книге, которая служит для тестирования объектов системы на предмет их быть объектами типа модуля либо блока. Тогда как в общем понимании модули и блоки системы вполне ассоциируются с функциями от двух аргументов – список локальных переменных и собственно тело модуля/блока. При этом оба аргумента обязательны, т.е. основой объектов типа блок и модуль является функциональная структура, а не процедурная в ее чистом виде.

Вызов процедуры **ExtrCall[x, y]** возвращает **True**, если пользовательский блок, функция либо модуль **y** содержит вызовы блока/функции/модуля **x**, и **False** в противном случае. Если вызов в качестве аргумента **x** определяет список блоков/функций/модулей, то возвращается подсписок из входящих в объект **y** вызовов блоков/функций/модулей из списка **x**. В случае, если же первый необязательный аргумент **x** отсутствует, вызов **ExtrCall[y]** возвращает список системных средств, входящих в определение функции, блока или модуля пользователя **y**. Следующий фрагмент представляет исходный код процедуры **ExtrCall** с типичными примерами ее применения. Процедура имеет ряд достаточно полезных приложений, прежде всего, в задачах системного характера.

```

In[2047]:= ExtrCall[___, P_ /; BlockFuncModQ[P]] := Module[{b, p, g, x,
a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]],
If[{N} == {}, p = PureDefinition[P]; If[ListQ[p], Return[$Failed]]; g = ExtrVarsOfStr[p, 2];
g = Select[g = Map[" " <> # <> "[" &, g], ! StringFreeQ[p, #] &]
g = Select[Map[If[SystemQ[p = StringTake[#, {2, -2}]], p] &, g], ! SameQ[#, Null] &];
If[Length[g] == 1, g[[1]], g],
b[x_] := Block[{c = DefFunc1[ToString[P]], d, h, k = 1, t = {}},
h = StringPosition[c, ToString[x] <> "["; If[h == {}, Return[False], d = Map[First, h];
For[k, k <= Length[d], k++, t = Append[t,
If[! MemberQ[a, StringTake[c, {d[[k]] - 1, d[[k]] - 1}]], True, False]]]; t[[1]]];
If[! ListQ[N], b[N], Select[N, b[#] &]]]
In[2048]:= Map3[ExtrCall, Run, {ActiveProcess, Attrib, SearchDir, SearchFile, Df, Uprocs}]
Out[2048]= {True, True, True, True, False, False}

```

```
In[2049]:= ExtrCall[{Run, Write, Read, If, Return}, Attrib]
Out[2049]= {Run, Read, If, Return}
In[2050]:= ExtrCall[BlockFuncModQ]
Out[2050]= {"Flatten", "FromCharacterCode", "If", "Module", "StringReplace", "StringTake"}
```

Наряду с системными функциями процедура **ExtrCall** весьма существенно использует также и наши процедуры **BlockFuncModQ**, **ExtrVarsOfStr**, **PureDefinition** и функцию **SystemQ**, рассматриваемые в настоящей книге. Между тем, следует иметь в виду, что процедура **ExtrCall** корректно обрабатывает только уникальные, но не одноименные объекты, возвращая на последних значение **\$Failed**.

Дополнительно к ранее представленной процедуре **TestArgsTypes**, обеспечивающей вызов указанных блока, функции, модуля таким образом, что возвращает результат данного вызова процедуры в отсутствие недопустимых фактических аргументов или список, состоящий из значений **{True, False}**, порядок которых соответствует порядку фактических аргументов при вызове тестируемого объекта указанного типа, интерес представляет процедура, чей вызов **TestProcCalls[x, y]** возвращает вложенный список, чьи элементы имеют формат **{j, "n", True | False}**, где **j** – порядковый номер аргумента, **"n"** – сам формальный аргумент в строчном формате, **{True | False}** – значение, которое определяет допустимость (**True**) либо недопустимость (**False**) фактического значения, определенного списком **y** и полученного формальным аргументом **{j, n}** в точке вызова объекта **x**. Предполагается, что объект **x** определяет фиксированное число формальных аргументов и длины списков, определяющие формальные аргументы и **y**, одинаковы, в противном случае вызов процедуры **TestProcCalls** возвращает **\$Failed**. В следующем фрагменте представлен исходный код процедуры с примерами ее использования.

```
In[2102]:= TestProcCalls[x_ /; BlockFuncModQ[x], y_List] := Module[{a = Args[x], b = {}, r,
    c = "_/"; d, p, k = 1, v}, a = Map[ToString1, If[NestListQ[a], a[[1]], a]];
    If[Length[a] != Length[y] || MemberQ[Map[! StringFreeQ[#, "_"] &, a], True], $Failed,
    v = If[NestListQ[v = Args[x, 78]], v[[1]], v]; For[k, k <= Length[a], k++, p = a[[k]];
        AppendTo[b, If[StringTake[p, {-1, -1}] == "_", True,
            If[! StringFreeQ[p, c], d = StringSplit[p, c]; r = ToExpression[d[[1]]];
                {ToExpression[d[[1]] <> "=" <> ToString1[y[[k]]], d[[2]]}][[2]],
                ToExpression[d[[1]] <> "=" <> ToString[r]]][[1]],
            d = StringSplit[p, "_"]; ToString[Head[y[[k]]]] == d[[2]]]]];
        {k, d} = {1, Partition[Riffle[v, b], 2]};
        While[k <= Length[d], PrependTo[d[[k]], k]; k++]; d]]

In[2103]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_78_460", "78_460", 2}]
Out[2103]= {{1, "S", True}, {2, "s", True}, {3, "n", True}}
In[2104]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_78_460", 460.78, 7.5}]
Out[2104]= {{1, "S", True}, {2, "s", False}, {3, "n", False}}
In[2105]:= F[x_String, y_ /; IntegerQ[y]] := {x, y}; TestProcCalls[F, {78, "avz"}]
Out[2105]= {{1, "x", False}, {2, "y", False}}
```

Процедура успешно обрабатывает как уникальные, так и одноименные объекты, во втором случае обрабатывая *первый* подобъект из списка, возвращаемого **Definition[x]**. При проверке значений на допустимость в случае формального аргумента формата "*arg\_*; *Test(arg)*" предварительно требуется вычислить *arg* и только затем проверить логическое значение *Test(arg)*. Но данная операция в теле процедуры обновляет *arg* вне процедуры, что в целом недопустимо. Поэтому для устранения данной ситуации был использован довольно простой прием (*легко усматриваемый из представленного кода процедуры*), не переопределяя глобальные переменные текущего сеанса, одноименные с формальными аргументами тестируемого объекта *x* указанного выше типа. Данный подход к организации алгоритма процедуры вполне отвечает концепции робастного программирования. При этом, процедура **TestProcCalls** допускает ряд модификаций, достаточно полезных для процедурного программирования в среде системы.

В отличие от предыдущих процедур следующая процедура **ProcActCallsQ** тестирует наличие в пользовательских блоке, функции или модуле *x* наличие вызовов *активных* в текущем сеансе средств пользователя, которые обеспечены справками (*usage*). Вызов процедуры **ProcActCallsQ[x]** возвращает **True**, если определение блока, функции или модуля *x* содержит вызовы средств подобного типа, и **False** в противном случае. Более того, через второй необязательный аргумент *y* – неопределенная переменная – вызов **ProcActCallsQ[x, y]** возвращает список *пользовательских* программных средств, вызовы которых находятся в определении блока, функции или модуля *x*. При этом, следующий фрагмент представляет *исходный* код процедуры **ProcActCallsQ** наряду с некоторыми наиболее типичными примерами ее использования.

```
In[2054]:= ProcActCallsQ[x_;/; BlockFuncModQ[x], y___] := Module[{a, b, c = {}, d, k = 1,
    h = "::usage = ", Save[b = "Art24$Kr17", x];
    For[k, k < Infinity, k++, d = Read[b, String];
    If[SameQ[d, EndOfFile], Break[], If[! StringFreeQ[d, h],
        c = Append[c, StringSplit[
    StringTake[d, {1, Flatten[StringPosition[d, h]][[1]] - 1}], "/: "]]],
    DeleteFile[Close[b]]; c = Select[c, SymbolQ[#] &];
    b = If[MemberQ[c, ToString[x]], Drop[c, 1], c];
    If[{y} != {} && ! HowAct[{y}][[1]], {y} = {b}]; If[b == {}, False, True]]

In[2055]:= {ProcActCallsQ[ProcQ, h], h}
Out[2055]= {True, {"ClearAllAttributes", "SymbolQ", "SubsDel", "Sequences", "ToString1",
    "StrDelEnds", "CharacterQ", "HeadPF", "BlockFuncModQ", "PureDefinition",
    "UnevaluatedQ", "ListStrToStr", "Definition2", "HowAct", "SystemQ", "Mapp",
    "SysFuncQ1", "SysFuncQ", "Contexts1", "SuffPref", "Map3", "MinusList"}}

In[2056]:= {ProcActCallsQ[ToString1, s], s}
Out[2056]= {True, {"StrDelEnds", "CharacterQ"}}

In[2057]:= G[x_String, y_;/; ! HowAct[y]] := If[StringLength[x] == 78, y = x, y = x <> "460"]
In[2058]:= {ProcActCallsQ[StrStr, Sv], Sv}
Out[2058]= {False, {}}
```

```

In[2059]:= {ProcActCallsQ[G, Gs], Gs}
Out[2059]= {True, {"HowAct"}}
In[2060]:= {ProcActCallsQ[ProcActCallsQ, Gsv], Gsv}
Out[2060]= {True, {"BlockFuncModQ", "PureDefinition", "UnevaluatedQ", "SymbolQ",
  "ToString1", "StrDelEnds", "CharacterQ", "ListStrToStr", "Definition2",
  "HowAct", "SystemQ", "SysFuncQ1", "Mapp", "ProcQ", "ClearAllAttributes",
  "SubsDel", "Sequences", "HeadPF", "SuffPref", "SysFuncQ", "Contexts1",
  "Map3", "MinusList"}}
In[2061]:= F[x_] := If[NestListQ[x], x, ToString1[x]]
In[2062]:= {ProcActCallsQ[F, v], v}
Out[2062]= {True, {"NestListQ", "ToString1", "StrDelEnds", "CharacterQ"}}

```

Процедура представляет интерес при анализе состава блоков/функций/модулей; при этом, исчерпывающий анализ относится только к средствам пользователя, активным в текущем сеансе системы и снабженным стандартными справками (*usages*).

В ряде случаев вполне определенный интерес представляет вопрос определения всех средств, используемых блоком, функцией или модулем пользователя, активизированных в текущем сеансе, включая средства, по которым отсутствуют справки (*usage*). Данную задачу и решает процедура **ProcContent**, обеспечивающая анализ активированного в текущем сеансе объекта *x* вышеуказанного типа с корректным заголовком, на предмет наличия в его определении пользовательских средств как *внутренних*, так и *внешних*, снабженных справочной информацией или без нее. Вызов процедуры **ProcContent[x]** возвращает вложенный список из 3 элементов, первый элемент которого определяет имя блока/функции/модуля *x*, второй – список имен всех *внешних* блоков, функций или модулей, используемых объектом *x*, и третий – список имен *внутренних* блоков, функций или модулей, определенных в теле *x*. Следующий фрагмент представляет исходный код процедуры **ProcContent** с типичными примерами ее использования.

```

In[2071]:= Kr[x_, y_] := Plus[x, y]; Art[x_] := Module[{a = 78, b = 480, c = ToString1[x], d, g},
  c = Kr[a, b]; d[y_] := Module[{y}, g[z_] := Block[{z}, z + 78]; c];
  V[x_] := Module[{c = StrStr[x], d, g}, G[a, b]; d[y_] := Module[{y}, g[z_] := Block[{z}, z]; c]
In[2072]:= ProcContent[x_ /; BlockFuncModQ[x]] := Module[{a, b = SubProcs[x][[1]], f},
  f[y_] := Module[{a1 = "$Art24Kr17$", b1 = "", c = {y}, d, h = "", p},
    Save[a1, y]; While[! SameQ[b1, EndOfFile], b1 = Read[a1, String];
      If[! MemberQ[{" ", "EndOfFile"}, ToString[b1]],
        h = h <> ToString[b1]; Continue[], d = Flatten[StringPosition[h, " := ", 1]]];
      If[d == {} , h = ""; Continue[], p = StringTake[h, {1, d[[1]] - 1}];
      If[! SameQ[Quiet[ToExpression[p]], $Failed],
        c = Append[c, StringTake[p, {1, Flatten[StringPosition[p, "[", 1]][[1]] - 1}]]; h = "", Null]]];
  a1 = Map[ToExpression, {DeleteFile[Close[a1]], c}][[2]]; DeleteDuplicates[a1];
  a = f[x]; {x, If[Length[a] > 1, a[[2 ;; -1]], {}], If[Length[b] > 1,
    Map[ToExpression, Map[HeadName, b[[2 ;; -1]]], {}]}]

```

```

In[2073]:= ProcContent[V]
Out[2073]= {V, {StrStr}, {d, g}}
In[2074]:= ProcContent2[Art]
Out[2074]= {Art, {ToString1, StrDelEnds, CharacterQ, Kr}, {d, g}}
In[2075]:= ProcContent[ToString1]
Out[2075]= {ToString1, {StrDelEnds, CharacterQ}, {}}
In[2076]:= V[x_] := Module[{a = 5, b = 6, c, d, g, Gt}, c = Gt[a, b]; d[y_] := Module[{}, y];
              g[z_] := Block[{a = 78, b = 460}, z + 78]; ToString1[x] <> StrStr[x]]
In[2077]:= ProcContent[V]
Out[2077]= {V, {ToString1, StrDelEnds, CharacterQ, StrStr}, {d, g}}
    
```

При этом, процедура **ProcContent** наряду о стандартными функциями существенно использует процедуру **BlockFuncModQ** для обеспечения при вызове тестирования на допустимость фактического аргумента и простую функцию, чей вызов **HeadName[x]** возвращает имя заголовка *x* в строчном формате. Средства рассмотрены в книге.

Вызов функции **ProcFuncCS[]** возвращает *вложенный 3-элементный список*, подписки которого определяют *имена* в строчном формате соответственно блоков, функций и модулей пользователя, чьи определения были вычислены в текущем сеансе. Фрагмент представляет исходный код функции наряду с типичным примером ее применения.

```

In[994]:= ProcFuncCS[] := Quiet[Map3[Select, Names["*"], {BlockQ[#] &, FunctionQ[#] &,
                                         ModuleQ[Symbol[#]] &}]]

In[995]:= G[x_String, y_ /; ! HowAct[y]] := If[StringLength[x] == 78, y = x, y = x <> "460"];
          GS[x_] := Block[{a = 78, b = 460}, x]; F[x_] := If[NestListQ[x], x, ToString1[x]];
          GG[y_] := Module[{a = 78, b = 460, c = 2013, d = {42, 47, 67}}, y]; ProcFuncCS[]
Out[995]= {{{"GS"}, {"F", "G"}, {"GG"}}}
    
```

Бывший в *первых* релизах пакета оператор **HeadCompose[A, B, C, ...]** (*в настоящее время не документированный*) возвращает композицию идентификаторов в форме  $A[B][C] \dots$ , которая, например, может быть весьма полезной в различного рода функциональных преобразованиях как иллюстрирует очень простой пример  $\text{HeadCompose}[F, x] \Rightarrow F[x]$ . Полезным данный оператор может быть и при организации функций пользователя, позволяя передавать в качестве фактических *значений* для ее формальных аргументов заголовки функций и/или их формальные аргументы. Между тем, данное средство в целом не представляет особого интереса, что и побудило вывести его *за рамки* пакета.

```

In[2111]:= FunCompose[L_ /; ListQ[L], x_] := Module[{a, k = 2}, a = L[[1]]@x;
              For[k, k <= Length[L], k++, a = L[[k]]@a]; a]

In[2112]:= FunCompose[{F, G, H, T, W, Q, V, U}, Sin[z]]
Out[2112]= F[G[H[T[W[Q[V[U[Sin[z]]]]]]]]]
In[2113]:= {FunCompose[{Sin, Cos, Log}, 9.42], FunCompose[{Sin, Cos, Tan, Sqrt}, 460.78]}
Out[2113]= {-0.0000114144, 0.874507}
    
```

С другой стороны, можно представить некоторый *аналог* данного средства, имеющий существенно больший прикладной интерес, а именно процедуру **FunCompose[L, x]**, с

помощью которой можно создавать вложенные функции из списка блоков, функций или модулей  $L$  от выражения, заданного ее вторым аргументом  $x$ . Итак, предыдущий довольно простой фрагмент достаточно наглядно иллюстрирует вышесказанное. Для организации передачи идентификаторов функций в качестве фактических значений можно использовать конструкции, например, следующих весьма простых форматов:

```
In[2118]:= F[x_] := x^3; SV[z_] := F@z + z^3; VSV[Id_, z_] := Module[{}, Id@(z^2 + 6);
                                                    {VSV[F, h], SV[45]}
```

```
Out[2118]= {(6 + h^2)^3, 182250}
```

а также целый ряд подобных им достаточно полезных конструкций.

Для временного удаления из текущего сеанса системы *Mathematica* блоков, функций или модулей служит довольно полезная процедура **DelRestPF**, исходный код которой с типичными примерами использования представляет следующий фрагмент.

```
In[2244]:= F[x_] := x^3; SV[z_] := F@z + z^3; VSV[Id_, z_] := Module[{}, Id@(z^2 + 6);
    F[x_, y_] := x + y; SetAttributes[F, {Protected, Listable}]; SetAttributes[SV, Listable]
In[2246]:= DelRestPF[r_ /; MemberQ[{"d", "r"}, r], x___] := Module[{f = "$Art24Kr17$.mx",
    a = Quiet[Select[{x}, BlockFuncModQ[#] &]], b, c, p, k = 1,
    If[r == "d", b = Map[Definition2, a]; Save[f, b];
    Map[ClearAllAttributes, a]; Map[Remove, a]; c = Get[f]; DeleteFile[f];
    For[k, k <= Length[c], k++, p = c[[k]]; ToExpression[p[[1 ;; -2]]];
    ToExpression["SetAttributes[" <> StringTake[p[[1]],
    {1, Flatten[StringPosition[p[[1]], "["]][[1] - 1]} <> ", " <> ToString[p[[-1]]] <> "]" ]]]]
In[2247]:= DelRestPF["d", F, SV, VSV]
In[2248]:= Map[Definition2, {F, SV, VSV}]
Out[2248]= {Definition2[F], Definition2[SV], Definition2[VSV]}
In[2249]:= DelRestPF["r"]
In[2250]:= Map[Definition2, {F, SV, VSV}]
Out[2250]= {"F[x_] := x^3", "F[x_, y_] := x + y", {Listable, Protected}},
    {"SV[z_] := F[z] + z^3", {Listable}}, {"VSV[Id_, z_] := Module[{}, Id[z^2 + 6]]", {}}
In[2284]:= DelRestPF1[r_ /; MemberQ[{"d", "r"}, r], f_String, x___] := Module[{b, c, p, k = 1,
    a = Quiet[Select[{x}, BlockFuncModQ[#] &]],
    If[r == "d", b = Map[Definition2, a]; Save[f, b];
    Map[ClearAllAttributes, a]; Map[Remove, a]; c = Get[f]; DeleteFile[f];
    For[k, k <= Length[c], k++, p = c[[k]]; ToExpression[p[[1 ;; -2]]];
    ToExpression["SetAttributes[" <> StringTake[p[[1]],
    {1, Flatten[StringPosition[p[[1]], "["]][[1] - 1]} <> ", " <> ToString[p[[-1]]] <> "]" ]]]]
In[2285]:= DelRestPF1["d", "C:/Temp/Herson", F, SV, VSV]
In[2286]:= Map[Definition2, {F, SV, VSV}]
Out[2286]= {Definition2[F], Definition2[SV], Definition2[VSV]}
```

```
In[2287]:= DelRestPF1["r", "C:/Temp/Herson"]
In[2288]:= Map[Definition2, {F, SV, VSV}]
Out[2268]= {"F[x_] := x^3", "F[x_, y_] := x + y", {Listable, Protected}},
           {"SV[z_] := F[z] + z^3", {Listable}}, {"VSV[Id_, z_] := Module[{}, Id[z^2 + 6]]", {}}
```

Вызов **DelRestPF**["d", x, y, ...] возвращает **Null**, т.е. ничего, удаляя из текущего сеанса системы блоки, функции, модули {x, y, ...}, тогда как последующий вызов процедуры **DelRestPF**["r"] восстанавливает их доступность в текущем сеансе или в другом сеансе с сохранением приписанных им опций и атрибутов, возвращая **Null**. Отметим, в ряде приложений, прежде всего, системного характера процедура довольно полезна. Более того, процедура ориентирована на работу только с одним списком объектов, создавая единственный фиксированный файл с сохраненными объектами. Между тем, весьма несложная модификация процедуры обеспечивает ее расширение на произвольное число списков блоков, функций и модулей пользователя, позволяя временно удалять их в любые моменты из текущего сеанса с последующим восстановлением в текущем либо ином сеансе системы. Предыдущий фрагмент завершается одной из подобных полезных модификаций, имеющей ряд полезных приложений системного характера. Вызов **DelRestPF**["d", f, x, y, ...] возвращает **Null**, т.е. ничего, с удалением из текущего сеанса системы блоков, функций, модулей {x, y, ...} с сохранением их в файле f, тогда как последующий вызов процедуры **DelRestPF**["r", f] восстанавливает их доступность в текущем сеансе или в ином сеансе из файла данных f с сохранением приписанных им опций и атрибутов, также возвращая значение **Null**.

Встроенный язык *Mathematica* для программирования *ветвящихся* алгоритмов наряду с предложением **If** допускает использование безусловных переходов на базе функции **Goto**, которая кодируется в форме **Goto**[h], безусловно передавая управление в точку, поименованную конструкцией **Label**[h]. Как правило, **Goto**-функция используется в процедурных конструкциях, однако в отличие от встроенной *goto*-функции системы *Maple* она может использоваться также во входной конструкции *Mathematica*. Более того, в качестве метки допускается произвольное корректное выражение, включая и *последовательность* выражений, последнее выражение которой определяет собственно саму метку; при этом, метка относительно модуля может быть как глобальной, так и локальной переменной. Между тем, во избежание возможных недоразумений *метку* рекомендуется определять локальной, ибо вычисленная вне процедуры глобальная метка всегда приемлема для модуля, однако вычисленная в теле модуля вполне может исказить вычисления вне модуля. Кратность вхождения идентичных **Goto**-функций в процедуру вполне естественна и определяется реализуемым ею алгоритмом задачи, тогда как с соответствующими им метками **Label** подобная ситуация, вообще говоря, недопустима, не распознаваясь при вычислении определения процедуры и даже на стадии ее выполнения, зачастую в значительной степени искажая запланированный алгоритм задачи. В данном случае управление получает точка *тела* модуля, которая помечена первой такой меткой. Более того, следует иметь в виду, отсутствие **Label**[a] для соответствующей функции **Goto**[a] в блоке или модуле на стадии вычисления их определений не распознается, а лишь в момент выполнения с реальным обращением к такой **Goto**[a]. Примеры, иллюстрирующие сказанное, можно найти в [100,101].

В данной связи определенный интерес может представить процедура, вызов которой **GotoLabel[P]** позволяет проанализировать процедуру *P* на *формальную* корректность использования ею **Goto**-функций и соответствующих им меток **Label**. Вызов данной процедуры возвращает вложенный 3-элементный список, первый элемент которого представляет список всех используемых процедурой **Goto**-функций, второй – список *всех* меток (без учета их кратности), третий – список, чьи подписки определяют **Goto**-функции с соответствующими им метками (при этом, первыми элементами подписков выступают вызовы **Goto**-функций; тогда как, кратности функций и меток сохраняются). Фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2056]:= GotoLabel[x_ /; BlockModQ[x]] := Module[{a = Flatten[{PureDefinition[x]}][[1]],
    b, c = {}, {}, {}, d, p, k = 1, j, h, v = {}, b = ExtrVarsOfStr[a, 1];
    b = DeleteDuplicates[Select[b, MemberQ[{"Label", "Goto"}, #] &]];
    If[b == {}, c, d = StringPosition[a, Map[" " <> # <> "[" &, {"Label", "Goto"}]];
    t = StringLength[a]; For[k, k <= Length[d], k++, p = d[[k]]; h = ""; j = p[[2]];
        While[j <= t, h = h <> StringTake[a, {j, j}];
            If[StringCount[h, "["] == StringCount[h, "]"],
                AppendTo[v, StringTake[a, {p[[1]] + 1, p[[2]] - 1}] <> h]; Break[]; j++];
            h = DeleteDuplicates[v]; {Select[h, SuffPref[#, "Goto", 1] &],
                Select[h, SuffPref[#, "Label", 1] &],
                Gather[Sort[v], #1 == StringReplace[#2, "Label[" -> "Goto[" , 1] &]]]}]]

In[2057]:= ArtKr[x_ /; IntegerQ[x]] := Module[{prime, agn}, If[PrimeQ[x], Goto[78; prime],
    If[OddQ[x], Goto[agn], Goto[Sin]]]; Label[78; prime]; Print[x^2]; Goto[Sin];
    Print[NextPrime[x]]; Goto[Sin]; Label[78; prime]; Null]

In[2058]:= Kr[x_ /; IntegerQ[x]] := Module[{prime, agn, y}, If[PrimeQ[x], Goto[prime],
    If[OddQ[x], Goto[agn], Goto[agn]]]; Label[78; prime]; y = x^2; Goto[agn];
    Label[agn]; y = NextPrime[x]; Label[agn]; y]

In[2059]:= GotoLabel[ArtKr]
Out[2059]= {{{"Goto[78; prime]", "Goto[agn]", "Goto[Sin]"}, {"Label[78; prime]"},
    {"Goto[78; prime]", "Label[78; prime]", "Label[78; prime]"}, {"Goto[agn]"},
    {"Goto[Sin]", "Goto[Sin]", "Goto[Sin]"}]]

In[2060]:= GotoLabel[Kr]
Out[2060]= {{{"Goto[prime]", "Goto[agn]"}, {"Label[78; prime]", "Label[agn]"},
    {"Goto[agn]", "Goto[agn]", "Goto[agn]", "Label[agn]", "Label[agn]"},
    {"Goto[prime]"}, {"Label[78; prime]"}]]

In[2061]:= Map[GotoLabel, {GotoLabel, TestArgsTypes}]
Out[2061]= {{}, {}, {}, {}, {}, {}]}

In[2062]:= Map[GotoLabel, {SearchDir, StrDelEnds, OP}]
Out[2062]= {{{"Goto[b]"}, {"Label[b]"}, {"Goto[b]", "Goto[b]", "Label[b]"}},
    {"Goto[p]", "Goto[Fin]"}, {"Label[1]", "Label[3]", "Label[2]", "Label[Fin]"},
    {"Goto[Fin]", "Label[Fin]"}, {"Goto[p]"}, {"Label[1]"}, {"Label[2]"}, {"Label[3]"}]},
    {"Goto[420]"}, {"Label[420]"}, {"Goto[420]", "Label[420]"}]]
```



Отметим, что наличие в возвращаемом вызовом процедуры **GotoLabel[P]** вложенном списке с третьим подписком, содержащем **Goto**-функций без соответствующих им меток, не обязательно говорит о наличии функции **Goto[x]**, для которой отсутствует метка **Label[x]**. Это может быть, например, при генерации значения *V* в зависимости от некоторого условия. Например, вызов **GotoLabel[StrDelEnds]** в третьем подписке содержит {"Goto[p]"}, что, на первый взгляд, можно было бы рассматривать в качестве некорректности соответствующего вызова **Goto**-функции. Однако, все дело в том, что значение фактического *p*-аргумента в вызове **StrDelEnds[S, h, p]** и определяет метку, реально существующую в определении данной процедуры. Таким образом, процедура **GotoLabel[P]** лишь на формальном уровне анализирует наличие «некорректных» с ее точки зрения **Goto**-функций и «лишних» меток. Тогда как уточнение полученных на основе вызова **GotoLabel[P]** результатов лежит на самом пользователе, предполагая, в первую очередь, анализ соответствия исходного кода *P* корректности алгоритма.

Структурированная парадигма программирования не предполагает использования в программах *goto*-конструкций, позволяющих передавать управление снизу-вверх. В тоже время, в целом ряде случаев использование **Goto**-функции очень эффективно, в частности, при необходимости погружения в *Math*-среду программы, использующей безусловные переходы на основе *goto*-предложения. Например, довольно типичным примером являются *Fortran*-программы, очень широко распространенные в научных приложениях. Из нашего опыта следует отметить, что использование **Goto**-функции позволило достаточно существенно упростить погружение в среду *Mathematica* ряда больших *Fortran*-программ, относящихся к инженерно-физической тематике, весьма широко использующих *goto*-конструкции. Тут же необходимо отметить, что с нашей точки зрения **Goto**-функция системы *Mathematica* более предпочтительна, чем *goto*-функция системы *Maple* в плане эффективности в свете применения в процедурном программировании приложений, включая приложения системного характера.

Как уже отмечалось, система допускает наличие *одноименных* объектов с различными *заголовками*, именно которые идентифицируют объекты, а не их имена. Стандартная функция **Definition**, а также наши процедуры **DefFunc**, **Deffunc3**, **PureDefinition** и др. позволяют по имени объекта получать определения всех активных в текущем сеансе объектов с идентичными именами, но различными *заголовками*. С учетом сказанного, возникает вполне конкретная задача удаления из текущего сеанса не всех объектов с конкретным именем, а только объектов с конкретными *заголовками*. Именно данную задачу и решает процедура **RemovePF**, вызов которой **RemovePF[x]** возвращает **Null**, т.е. ничего, обеспечивая *удаление* из текущего сеанса объектов с *заголовками* *x*, которые определены фактическим аргументом *x* (*заголовок в строчном формате либо их список*). Вызов процедуры **RemovePF[x]** возвращает **Null**, т.е. ничего, обеспечивая удаление из текущего сеанса объектов с *заголовками*, определенными фактическим аргументом *x* (*заголовок в объекта строчном формате или их список*). В случае некорректных *заголовков*, определяемых аргументом *x*, вызов **RemovePF[x]** возвращается *невычисленным*. Данная процедура оказывается довольно полезной в процедурном программировании. Ниже представлены исходный код процедуры **RemovePF** наряду с примерами ее наиболее типичного использования для удаления подобъектов у *одноименных* объектов.

```

In[2084]:= RemovePF[x_ /; HeadingQ1[x] || ListQ[x] &&
           DeleteDuplicates[Map[HeadingQ1, x]] == {True}] :=
Module[{a = DeleteDuplicates[Map[HeadName, Flatten[{x}]]], b, c = {}, d, p, k = 1, j},
  b = Map[If[UnevaluatedQ[Definition2, #], {"78", {}}, Definition2[#]] &, a];
  For[k, k <= Length[a], k++, p = b[[k]];
  AppendTo[c, Select[Flatten[{p[[1 ;; -2]], "SetAttributes[" <>
  a[[k]] <> ", " <> ToString[p[[-1]]] <> "}], ! SuffPref[#, x, 1] &]]];
  Map[ClearAllAttributes, a]; Map[Remove, a]; Map[ToExpression, c]; ]

In[2085]:= M[x_ /; SameQ[x, "avz"], y_] := Module[{a, b, c}, y]; F[x_, y_Integer] := x + y
M[x_ /; x == "avz"] := Module[{a, b, c}, x]; F[x_, y_, z_] := x + y + z;
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_, y_, z_] := x + y + z;
M[x_, y_] := Module[{a, b, c}, "agn"; x + y]; M[x_String] := x;
M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y]; F[x_, y_] := x + y;
SetAttributes[M, Protected]; SetAttributes[F, Listable]

In[2086]:= Definition[M]
Out[2086]= Attributes[M] = {Protected}
"M[x_ /; x === \"avz\", y_] := Module[{a, b, c}, y]"
"M[x_ /; x == \"avz\" ] := Module[{a, b, c}, x]"
"M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]"
"M[x_ /; ListQ[x], y_] := Block[{a, b, c}, \"agn\"; Length[x] + y]"
"M[x_, y_] := Module[{a, b, c}, \"agn\"; x + y]"
"M[x_String] := x"

In[2087]:= Definition[F]
Out[2087]= Attributes[F] = {Listable}
"F[x_, y_Integer] := x + y"
"F[x_, y_] := x + y"
"F[x_, y_, z_] := x + y + z"

In[2088]:= RemovePF[{"M[x_, y_]", "F[x_, y_, z_]", "M[x_String]", "F[x_, y_Integer]", "v[t_]"}]
In[2089]:= Definition[M]
Out[2089]= Attributes[M] = {Protected}
"M[x_ /; x === \"avz\", y_] := Module[{a, b, c}, y]"
"M[x_ /; x == \"avz\" ] := Module[{a, b, c}, x]"
"M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]"
"M[x_, y_, z_] := x + y + z"
"M[x_ /; ListQ[x], y_] := Block[{a, b, c}, \"agn\"; Length[x] + y]"

In[2090]:= Definition[F]
Out[2090]= Attributes[F] = {Listable}
"F[x_, y_] := x + y"

```

Для обеспечения корректной загрузки блока/функции/модуля в текущий сеанс при условии возможной необходимости дозагрузки в текущий *сеанс* также нестандартных

блоков/функций/модулей, вызовы которых используются в таком объекте, достаточно полезной может быть процедура **CallsInProc**[*x*], возвращающая список всех системных функций, внешних и внутренних блоков/функций/модулей, чьи вызовы использует объект *x* указанного типа. Фрагмент представляет исходный код процедуры наряду с наиболее типичными примерами ее использования.

```
In[2060]:= CallsInProc[P_;/ BlockFuncModQ[P]] := Module[{b, c = {}, k = 1, TN,
                                a = ToString[FullDefinition[P]]},
    TN[S_String, L_;/ ListQ[L] && Length[Select[L, IntegerQ[#] &]] ==
    Length[L] && L != {} := Module[{a1 = "", c1, b1 = {}, k1, p = 1},
        For[p, p <= Length[L], p++,
        For[k1 = L[[p]] - 1, k1 != 0, k1--, c1 = StringTake[S, {k1, k1}]; a1 = c1 <> a1;
            If[c1 === " ", a1 = StringTake[a1, {2, -1}];
            If[Quiet[Check[Symbol[a1], False]] === False, a1 = ""; Break[],
                b1 = Append[b1, a1]; a1 = ""; Break[[]]]; b1];
    b = TN[a, b = DeleteDuplicates[Flatten[StringPosition[a, {"["}]][[2 ;; -1]]];
    b = Sort[DeleteDuplicates[Select[b, StringFreeQ[#, "`"] &&
        ! MemberQ[{"Block", ToString[P], "Module"}, #] &&
        ToString[Definition[#]] != "Null" &]]; k = Select[b, SystemQ[#] &];
    c = MinusList[b, Flatten[{k, ToString[P]}]];
    {k, c, DeleteDuplicates[Map[Context, c]]}]

In[2061]:= CallsInProc[StringDependQ]
Out[2061]= {"Append", "Attributes", "Break", "ClearAttributes", "Close", "Continue",
    "DeleteDuplicates", "Flatten", "For", "Goto", "HoldPattern", "If", "Label",
    "Length", "ListQ", "MemberQ", "Min", "On", "Quiet", "Read", "Return",
    "Select", "SetAttributes", "StringLength", "StringQ", "StringTake", "ToString",
    "While", "Write", {"CharacterQ", "ClearAllAttributes", "Definition1",
    "ListAssignP", "ListStrQ", "Map3", "PosIntListQ", "PosIntQ", "StrDelEnds",
    "SymbolQ", "SysFuncQ1", "SystemQ", "ToString1"}, {"AladjevProcedures`"}}

In[2062]:= G[x_] := ToString1[x]; CallsInProc[G]
Out[2062]= {"Break", "Close", "Continue", "For", "Goto", "If", "Label", "MemberQ", "Read",
    "Return", "StringLength", "StringQ", "StringTake", "Write",
    {"CharacterQ", "StrDelEnds", "ToString1"}, {"AladjevProcedures`"}}
```

Вызов **CallsInProc**[*x*] возвращает вложенный 3-элементный список, первый подсписок которого содержит стандартные функции, второй – внешние и внутренние функции, блоки и модули пользователя, вызовы которых использует объект *x*, тогда как третий определяет список *контекстов*, которые соответствуют пользовательским средствам и которые используются объектом *x*. Процедура **CallsInProc** представляет существенный интерес при анализе пользовательских средств на предмет наличия в них вызовов как системных, так и пользовательских программных средств.

Для вычисления процедур и функций, чьи определения были вычислены в текущем сеансе, достаточно полезной оказывается простая функция, чей вызов **CsProcsFuncs**[

возвращает список блоков, функций и модулей, чьи определения были вычислены в текущем сеансе. Фрагмент приводит исходный код и пример применения функции.

```
In[2019]:= CsProcsFuncs[] := Select[CNames["Global`"], ProcQ[#] || FunctionQ[#] &]
In[2020]:= A[x_] := Block[{}, x]; B[x_] := Module[{}, x]; V[x_] := x + x^2; CsProcsFuncs[]
Out[2020]= {"A", "B", "V"}
```

Естественно, в данный список не включаются процедуры и функции из загруженных в текущий сеанс пакетов, как системных, так и пользовательских по той причине, что такие средства ассоциируются с контекстами соответствующих пакетов. Более того, в виду необходимости анализа довольно большого количества средств текущего сеанса выполнение функции может потребовать заметных временных издержек.

В качестве полезного расширения предыдущей функции выступает ее модификация в виде процедуры, чей вызов **CsProcsFuncs1[]** возвращает *вложенный* список, элементы которого – списки, чьи первые элементы определяют средства аналогично функции **CsProcsFuncs[]**, тогда как вторые элементы – кратность их определений. Следующий фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2057]:= CsProcsFuncs1[] := Module[{a = CsProcsFuncs[], b, c},
                                b = Map[Definition2, ToExpression[a]];
                                c = Quiet[Map[Select, b, StringFreeQ[#1, ToString[#1] <>
                                "Options[" <> ToString[#1] <> "]" := "]" &]];
                                Select[Map9[List, a, Map[Length, c]], ! MemberQ[#, "CsProcsFuncs1"] &]]
In[2058]:= CsProcsFuncs1[]
Out[2058]= {"LocalVars", 1}, {"V", 4}, {"W", 2}, {"Z", 2}
```

При этом, аналогично функции **CsProcsFuncs** вызов процедуры **CsProcsFuncs1** в виду необходимости анализа довольно большого количества средств, активизированных в текущем сеансе, может потребовать достаточно заметных временных издержек.

Следующая процедура **ActCsProcFunc** является достаточно полезной в практическом отношении, ее вызов **ActCsProcFunc[]** возвращает вложенный 2-х элементный список, в качестве элементов которого выступают два подсписка переменной длины. Первый подсписок первым элементом содержит **"Procedure"**, тогда как другие определяют 2-х элементные списки, содержащие *имена* активизированных в текущем сеансе процедур с их заголовками. Тогда как второй подсписок первым элементом содержит **"Function"**, тогда как другие определяют двух-элементные списки с именами активизированных в текущем сеансе функций с их заголовками. При этом, процедуры содержат в своем составе как *блоки*, так и *модули*. Следующий фрагмент представляет как исходный код процедуры **ActCsProcFunc**, так и наиболее типичные примеры ее использования.

```
In[3938]:= ActCsProcFunc[] := Module[{a = Names["Global`*"], c = {"Function"}, d, k = 1,
                                b = {"Procedure"}}, While[k <= Length[a], d = a[[k]];
                                If[ProcQ[d], AppendTo[b, {d, HeadPF[d]}],
                                If[FunctionQ[d], AppendTo[c, {d, HeadPF[d]}]]]; k++; {b, c}]
```

```

In[3939]:= ActCsProcFunc[]
Out[3939]= {"Procedure", {"ActCsModFunc", "ActCsModFunc[]"}, {"Function"}}
In[3940]:= g[x_] := Module[{x}; s[x_, y_] := Block[{x + y}; v[x_] := x; n[x_] := x
In[3941]:= vs[x_, y_] := x + y; gs[x_] := x^2; hg[x_] := Length[{x}]; hh[x_, y_] := x^2 + y^2
nm[x_, y_] := Module[{x*y}; ts[x_Integer] := Block[{a = 70}, x + a]; w[x_] := x
In[3942]:= ActCsProcFunc[]
Out[3942]= {"Procedure", {"ActCsModFunc", "ActCsModFunc[]"}, {"g", "g[x_]"},
{"nm", "nm[x_, y_]"}, {"s", "s[x_, y_]"}, {"ts", "ts[x_Integer]"},
{"Function", {"gs", "gs[x_]"}, {"hg", "hg[x_]"}, {"n", "n[x_]"}, {"v", "v[x_]"},
{"vs", "vs[x_, y_]"}, {"hh", "hh[x_, y_]"}, {"w", "w[x_]"}]}
In[3943]:= LoadMyPackage["AVZ_Package.mx", "AladjevProcedures`"]
In[3944]:= ActCsProcFunc[]
Out[3944]= {"Procedure", {"ActCsModFunc", "ActCsModFunc[]"}, {"g", "g[x_]"},
{"nm", "nm[x_, y_]"}, {"s", "s[x_, y_]"}, {"ts", "ts[x_Integer]"},
{"Function", {"gs", "gs[x_]"}, {"hg", "hg[x_]"}, {"n", "n[x_]"}, {"v", "v[x_]"},
{"vs", "vs[x_, y_]"}, {"hh", "hh[x_, y_]"}, {"w", "w[x_]"}]}
In[3945]:= Context["ProcQ"]
Out[3945]= "AladjevProcedures`"

```

При этом, аналогично **CsProcsFuncs** и **CsProcsFuncs1** вызов процедуры **ActCsProcFunc** ввиду необходимости анализа достаточно большого числа средств, активированных в текущем сеансе, может потребовать достаточно заметных временных издержек.

Относительно процедуры **ActCsProcFunc** следует отметить, она обеспечивает возврат только блоков, функций и модулей, чьи определения вычислены в **Input**-параграфе, не позволяя получать объекты данного типа, загруженные в текущий сеанс системы в режиме **Input**-параграфа, в частности, в результате загрузки пакета пользователя по процедуре **LoadMyPackage**, как весьма наглядно иллюстрируют последние примеры предыдущего фрагмента. Причина этого состоит в том, что данные объекты остаются ассоциированными с контекстом содержащего их пакета, а не с контекстом **"Global"**.

Как отмечалось выше, строгая дифференцировка процедур и функций в среде языка **Mathematica** выполняется не по их *именам*, как это принято в большинстве известных языков и систем программирования, а по их *заголовкам*. Именно по данной причине в целом ряде случаев продвинутого *процедурного* программирования возникает задача организации механизмов дифференцированной обработки таких объектов на основе их заголовков. Ряд таких средств представлен в настоящей книге, здесь мы определим две процедуры, обеспечивающие дифференцированную работу с атрибутами таких объектов. Тут же отметим, ряд представленных в книге средств рассчитан на работу с объектами без атрибутов, прежде всего **Protected**-атрибута, однако заинтересованный читатель вполне сможет в качестве весьма полезного упражнения адаптировать их и на работу также с объектами с приписанными им атрибутами, примеры приведены.

В отличие от процедуры **Rename**, процедура **RenameH** обеспечивает в определенной степени выборочное переименование одноименных блоков, функций и модулей на

основе их заголовков. Успешный вызов **RenameH**[*x*, *y*] возвращает **Null**, т.е. ничего, переименовывая объект с заголовком *x* на имя *y* с сохранением атрибутов; при этом, исходный объект с заголовком *x* удаляется из текущего сеанса системы *Mathematica*.

```
In[2135]:= RenameH[x_;/ HeadingQ1[x], y_;/! HowAct[y], z_] := Module[{c,
    b = ToExpression["Attributes[" <> HeadName[x] <> "]",
    a = HeadName[x], d = StandHead[x]}, c = PureDefinition[a];
    If[c == {}, Return[$Failed], ToExpression["ClearAllAttributes[" <> a <> ""]];
    ToExpression[ToString[y] <> DelSuffPref[Select[c, SuffPref[#, d <>
        " := ", 1] &][[1]], a, 1]]; If[{z} == {}, RemProcOnHead[d]];
    If[! SameQ[PureDefinition[a], $Failed],
    ToExpression["SetAttributes[" <> ToString[a] <> ", " <> ToString[b] <> ""]];
    ToExpression["SetAttributes[" <> ToString[y] <> ", " <> ToString[b] <> ""]];]

In[2136]:= RenameH["M[x_,y_,z_]", G]
In[2137]:= Definition[G]
Out[2137]= Attributes[G] = {Listable, Protected}
    G[x_, y_, z_] := x + y + z
In[2138]:= Definition[M]
Out[2138]= Attributes[M] = {Listable, Protected}
    M[x_, y_] := Module[{}, {x}]
    M[x_] := Module[{}, {x}]
In[2139]:= ClearAllAttributes[G, M]; Clear[G, M]; M[x_, y_] := Module[{}, {x};
    M[x_, y_, z_] := x+y+z; M[x_] := Module[{}, {x}]; SetAttributes[M, Protected]
In[2140]:= RenameH["M[x_,y_,z_]", G, 78]
In[2141]:= Definition[G]
Out[2141]= Attributes[G] = {Protected}
    G[x_, y_, z_] := x + y + z
In[2142]:= Definition[M]
Out[2142]= Attributes[M] = {Protected}
    M[x_, y_, z_] := x + y + z
    M[x_, y_] := Module[{}, {x}]
    M[x_] := Module[{}, {x}]
```

Более того, вызов **RenameH**[*x*, *y*, *z*] с третьим необязательным аргументом *z*, в качестве которого допускается произвольное выражение, переименовывает объект с заголовком *x* на имя *y* с сохранением атрибутов; между тем, объект с заголовком *x* сохраняется в текущем сеансе системы. Предыдущий фрагмент приводит исходный код **RenameH** с наиболее типичными примерами ее использования.

В ряде процедур, предназначенных для обработки определений либо вызовов других процедур/функций, довольно актуален вопрос идентификации выражений формата вызова, т.е. формата типа **W**[*args*], где **W** – имя процедуры/функции и *args* – кортеж *формальных* или *фактических* аргументов. Следующий фрагмент приводит исходный код процедуры **CallQ** наряду с достаточно типичными примерами ее использования.

```

In[9]:= CallQ[x_] := Module[{a = ToString[If[Quiet[Part[x, 1]] === -1, Part[x, 1]*x, x]], b, c},
                                b = Flatten[StringPosition[a, "["]];
                                If[b == {}, False, c = b[[1]]; If[SymbolQ[StringTake[a, {1, c - 1}]] &&
                                StringTake[a, {c + 1, c + 1}] != "[" && StringTake[a, -1] == "]", True, False]]

In[10]:= CallQ[A[x, y, z]]
Out[10]= True
In[2324]:= Map[CallQ, {Sin[-78], Sin[7.8]}]
Out[2324]= {True, False}

In[2109]:= FormalArgs[x_] := Module[{a, b = Quiet[Part[x, 1]]}, If[CallQ[x],
                                a = ToString[If[b === -1, Part[x, 1]*x, x]]; ToExpression["{" <> StringTake[a,
                                {Flatten[StringPosition[a, "["]][[1]] + 1, -2]} <> "}", $Failed]]

In[2110]:= Map[FormalArgs, {Agn[x, y, x], Sin[-a + b], Agn[x_ /; StringQ[x], y_Integer, z_]}]
Out[2110]= {{x, y, x}, {a - b}, {x_ /; StringQ[x], y_Integer, z_}}
In[2111]:= Map[FormalArgs, {Agn[], a + b, 78, {a, b, c}}]
Out[2111]= {{}, $Failed, $Failed, $Failed}

```

Вызов **CallQ[x]** с точностью до знака возвращает **True**, если *x* есть выражение формата **W[args]**, где *W* – имя процедуры/функции и *args* – кортеж фактических аргументов, и **False** в противном случае. Функция представляет интерес в качестве тестирующего средства для проверки фактических аргументов объектов на их допустимость. Вызов весьма полезной процедуры **FormalArgs[x]** возвращает список *формальных* аргументов заголовка *x* безотносительно от приписанного ему определения; на недопустимых *x* возвращается **\$Failed**. Фрагмент представляет исходный код процедуры с примером.

В программной среде *Maple* в задачах процедурного программирования достаточно полезной оказывается процедурная переменная *procname*, чье использование в теле процедуры позволяет получать заголовок процедуры в точке ее вызова. Переменная полезна при реализации специальных механизмов обработки внутри процедур, что достаточно широко использовалось нами при программировании системных средств, расширяющих программную среду системы *Maple* [10-13, 28-45]. Подобное средство в системе *Mathematica* отсутствует, между тем, средства подобного характера довольно полезны для процедурной парадигмы системы. В качестве одного полезного средства данного типа вполне можно рассматривать переменную, вызов которой **\$InBlockMod** в теле процедуры типа *Block* или *Module* в строчном формате возвращает исходный код процедуры без заголовка в точке ее вызова. Следующий фрагмент представляет исходный код переменной **\$InBlockMod** с примерами ее типичного использования.

```

In[2257]:= StringReplace3[S_String, x_] := Module[{a = Map[ToString, {x}], b = S, c, j = 1},
                                c = Length[a]; If[OddQ[c], S, While[j <= c/2,
                                b = StringReplace2[b, a[[2*j - 1]], a[[2*j]]]; j++]; b]]

In[2258]:= StringReplace3["Module[{a$ = 78, b$ = 460, c$ = 71}, xb$; a$*b$*47;
                                (a$+b$+c$)*(x+y); aa$]", "a$", "a", "b$", "b", "c$", "c"]
Out[2258]= "Module[{a = 78, b = 460, c = 71}, xb$; a*b*47; (a+b+c)*(x+y); aa$]"

```

```

In[2271]:= $InBlockMod := If[Stack[Block] != {}, ToString[Stack[Block]][[1]],
      If[Stack[Module] != {}, StringReplace3[ToString[Stack[Module]][[1]],
        Sequences[Riffle[Select[StringReplace[StringSplit[StringTake[
          SubStrSymbolParity1[ToString[Stack[Module]][[1]], "{", "}"][[1]], {2, -2}], " ", " " -> ""],
          StringTake[#, -1] == "$" &], Mapp[StringTake, Select[StringReplace[
            StringSplit[StringTake[SubStrSymbolParity1[ToString[
              Stack[Module]][[1]], "{", "}"][[1]], {2, -2}], " ", " " -> ""],
              StringTake[#, -1] == "$" &], {1, -2}]]]], $Failed]
In[2272]:= Avz[x_] := Block[{a = 78, b = 460, c = $InBlockMod}, Print[c]; a*b*x]
In[2273]:= Avz[42]
      Block[{a = 78, b = 460, c = $InBlockMod}, Print[c]; a*b*42]
Out[2273]= 1506960
In[2274]:= Agn[x_] := Module[{a = 78, b = 460, c = $InBlockMod}, Print[c]; a*b*x]
In[2275]:= Agn[47]
      Module[{a = 78, b = 460, c = $InBlockMod}, Print[c]; a*b*47]
Out[2275]= 1686360
In[2276]:= $InBlockMod
In[2277]:= G[x_, y_] := {x + y, $InBlockMod}
In[2278]:= G[42, 47]
Out[2278]= {89, Null}

```

Между тем, для реализации алгоритма переменной весьма существенно используется и процедура **StringReplace3**, являющаяся расширением процедуры **StringReplace2**. Ее исходный код с примерами применения приведен в начале предыдущего фрагмента. Вызов **StringReplace3[S, x, x1, y, y1, z, z1, ...]** возвращает результат постановки в строку *S* подстрок {*x1, y1, z1, ...*} вместо всех вхождений подстрок {*x, y, z, ...*} соответственно; при отсутствии таких вхождений вызов возвращает *исходную* строку *S*. Данная процедура оказывается весьма полезным средством обработки строчных конструкций, которые содержат выражения, расширяя возможности стандартных системных средств.

При использовании процедурной переменной **\$InBlockMod** следует иметь ввиду, что она имеет смысл лишь в теле процедур типа **Block** или **Module**, не возвращая ничего, т.е. **Null**, в других выражениях либо в **Input**-параграфе, как наглядно иллюстрируют примеры применения переменной **\$InBlockMod** предыдущего фрагмента. При этом, следует иметь в виду, во избежание недоразумений вызов переменной **\$InBlockMod** рекомендуется делать в начале процедур, например, в области *локальных* переменных.

На этом представление средств, служащих для обработки блоков, функций, модулей, завершается; некоторые средства, сопутствующие им, рассматриваются ниже или уже были рассмотрены выше. Классификация наших средств носит в определенной мере субъективный характер, что обусловлено их базовым функциональным назначением или частотой применения при разработке средств, представленных в настоящей книге и в целом ряде важных приложений как прикладного, так и системного характера.



## Глава 9. Средства ввода/вывода программной среды *Mathematica*

Будучи встроенным языком программирования *Mathematica*, который прежде всего ориентирован на символьные вычисления (*компьютерная алгебра*) и обработку, он располагает относительно ограниченными возможностями по обработке данных, и, прежде всего, находящихся во внешней памяти компьютера. В данном отношении он существенно уступает традиционным языкам программирования *C++*, *Basic*, *Fortran*, *Cobol*, *PL/1*, *ADA*, *Pascal* и др. Вместе с тем, ориентируясь, прежде всего, на решение задач в символьном виде, язык *Mathematica* предоставляет набор средств для доступа к файлам данных, который вполне может удовлетворить достаточно широкий охват пользователей математических приложений системы. В данной главе средства доступа к файлам данных рассматриваются достаточно поверхностно ввиду ограниченного объема, обширности этой проблематики и цели настоящей книги. Однако читатель, заинтересованный средствами доступа к файлам системы *Mathematica*, вполне может обратиться к документации, поставляемой с системой. Между тем, с целью развития способов доступа к файловой системе компьютера нами создан целый ряд достаточно эффективных средств, которые представлены в пакете *AVZ\_Package\_1* [120]. Тогда как в настоящей главе акцентируется внимание на средствах, расширяющих стандартные средства системы *Mathematica* для обеспечения работы с файлами компьютера. Ряд из них достаточно полезен для практического применения в среде *Mathematica*.

### 9.1. Средства *Mathematica* для работы с внутренними файлами

Средства *Math*-языка обеспечивают доступ пользователя к файлам нескольких типов, которые можно условно разделить на две большие группы: *внутренние* и *внешние*. При рутинной работе система имеет дело с 3-мя различными типами *внутренних* файлов, из которых отметим файлы, имеющие расширения `{"nd", ".m", ".mx"}`, структура которых распознается системой ее стандартными средствами и которые достаточно важны уже на первых этапах работы с системой. Перед дальнейшим рассмотрением отметим, что понятие *спецификатора файла (СФ)*, определяющего полный путь к искомому файлу в файловой системе ЭВМ либо к ее подкаталогу, практически, полностью совпадает с аналогичным понятием для уже упоминавшейся системы *Maple*, за исключением, что если в *Maple* для *СФ* допускается формат `{string | symbol}`-типа, в то время как для *СФ* в системе *Mathematica* является допустимым исключительно формат *string*-типа.

По вызову `Directory[]` возвращается активный подкаталог текущего сеанса системы, а по функции `SetDirectory[w]` активным определяется заданный каталог *w* с возвратом полного пути к нему; при этом, под *активным (текущим)* понимается каталог, файлы которого обрабатываются средствами доступа, если указываются лишь их имена, а не полные пути к ним. В частности, определяя при вызове `SetDirectory[w]` фактическим *w*-аргументом переменную `$UserDocumentsDirectory` системы, можно переопределять текущим подкаталог пользователя по умолчанию. Между тем, функция `SetDirectory` в качестве своего аргумента допускает лишь реально существующие подкаталоги, на несуществующих иницируя ошибочную ситуацию с возвратом значения `$Failed`. С

другой стороны, простая процедура **SetDir[x]** обеспечивает возможность определять в качестве текущих также и *несуществующие* подкаталоги *w*. Вызов процедуры **SetDir[x]** на *существующем* подкаталоге делает именно его текущим, тогда как *несуществующий* подкаталог предварительно создается, а затем определяется текущим. При этом, если в качестве фактического *w*-аргумента при вызове **SetDir[w]** определяется цепочка без имени устройства ввода/вывода (*YBB*), например, "aa\\...\\bb", то создается цепочка подкаталогов **Directory[]** <> "aa\\ ... \\bb", определяющая полный путь к созданному текущему подкаталогу. Следующий фрагмент представляет *исходный* код процедуры **SetDir** наряду с наиболее типичными примерами ее использования.

```
In[2081]:= Directory[]
Out[2081]= "C:\\Documents and Settings\\Viktor\\My Documents"
In[2082]:= SetDirectory["D:\\Math_myLib"]
Out[2082]= "D:\\Math_myLib"
In[2083]:= SetDirectory[$UserDocumentsDirectory]
Out[2083]= "C:\\Documents and Settings\\Viktor\\My Documents"
In[2084]:= SetDirectory[]
Out[2084]= "C:\\Documents and Settings\\Viktor "
In[2085]:= SetDirectory["D:\\Math_myLib\\rans"]
SetDirectory::cdir: Cannot set current directory to D:/Math_myLib/rans. >>
Out[2085]= $Failed

In[2147]:= SetDir[x_;/ StringQ[x]] := Module[{a}, If[StringLength[x] == 1 ||
StringLength[x] >= 2 && StringTake[x, {2, 2}] != ":",
Return[Quiet[SetDirectory[Quiet[CreateDirectory[
StringReplace[Directory[] <> "\\ " <> x, "\\\\" -> "\\"]]]], Null];
a = Quiet[CreateDirectory[StringTake[x, 1] <> ":\"];
If[a === $Failed, Return[$Failed], Null];
Quiet[Check[If[DirectoryQ[x], SetDirectory[x],
SetDirectory[CreateDirectory[x]]], Null]]; Directory[]]

In[2148]:= SetDir["C:\\Temp\\111\\222\\333\\444\\555\\666\\777\\888\\999"]
Out[2148]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999"
In[2149]:= SetDir["H:\\111\\222\\333\\444\\555\\666\\777\\888\\999"]
Out[2149]= $Failed
In[2150]:= Directory[]
Out[2150]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999"
In[2151]:= SetDir["rans\\ian\\avz"]
Out[2151]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999\\rans\\ian\\avz"
In[2152]:= Adrive[] := Module[{a, b, c, d, k}, {a, b} = {CharacterRange["A", "Z"], {}];
For[k = 1, k <= 26, k++, c = a[[k]] <> ":\"; d = Quiet[CreateDirectory[c]];
If[d === $Failed, Null, b = Append[b, StringTake[d, 1]]]; Sort[b]]

In[2153]:= Adrive[]
Out[2153]= {"C", "D", "E", "F", "G"}
```

Между тем, при попытке определения в качестве текущего *несуществующего* каталога вполне реально возникновение ситуации, когда в качестве *YBB* указано устройство, в настоящий момент не существующее в системе либо недоступное. Поэтому довольно актуально располагать средством, позволяющим тестировать доступность устройств в системе. Данную задачу решает процедура *Adrive*, вызов которой *Adrive[]* возвращает список логических имен *YBB*, доступных в текущий момент. Эта процедура является аналогом одноименной процедуры для пакета *Maple* [99], правда, возвращающей вместо списка последовательность имен *YBB*, доступных в текущий момент. Последняя часть фрагмента представляет исходный код процедуры *Adrive* и пример ее применения.

Следующая процедура *Adrive1* расширяет вышерассмотренную процедуру *Adrive* и возвращает 2-элементный вложенный список, чей *первый* подсписок содержит имена *всех* активных устройств *внешней* памяти прямого доступа в строчном формате, тогда как второй содержит имена всех неактивных устройств внешней памяти компьютера. В следующем фрагменте представлены исходный код *Adrive1* и пример применения.

```
In[2187]:= Adrive1[] := Module[{a = CharacterRange["A", "Z"], b = {}, c = 1, d, p = {}, h,
    t = "$Art24$Kr17$"}, For[c, c <= 26, c++, d = a[[c]] <> ":\\";
    If[DirQ[d], b = Append[b, a[[c]]]; h = Quiet[CreateDirectory[d <> t]];
    If[h === $Failed, Continue[], DeleteDirectory[d <> t];
    p = Append[p, a[[c]]]; Continue[]]; {p, MinusList[b, p]}

In[2188]:= Adrive1[]
Out[2188]= {{ "C", "D", "G"}, { "A", "E" }}

In[2211]:= SetDir1[x_ /; StringQ[x]] := Module[{a = SetDir[x], b, c, k},
    If[! SameQ[a, $Failed], a, k = 1; b = Adrive[]; c = Map[FreeSpaceVol, b];
    While[k <= Length[b], PrependTo[c[[k]], b[[k]]]; k++];
    c = SortNL1[c, 2, Greater]; SetDir[StringJoin[c[[1]][[1]], StringTake[x, {2, -1}]]]

In[2212]:= SetDir1["G:\\Gal\\Sv\\ArtKr\\Tampere\\Kherson"]
Out[2212]= "C:\\Gal\\Sv\\ArtKr\\Tampere\\Kherson"
```

В свою очередь, представленная в конце предыдущего фрагмента процедура *SetDir1*, расширяет процедуру *SetDir* на тот случай, когда попытка создать цепочку каталогов *x* встречает особую ситуацию, обусловленную отсутствием требуемого устройства, на котором планировалось создание данной цепочки подкаталогов. В отсутствие такого устройства вызов процедуры *SetDir1[x]* возвращает созданную цепочку подкаталогов на устройстве, имеющем *максимально* возможный объем доступной памяти среди всех активных в текущем сеансе системы устройств прямого доступа.

К внутренним файлам относятся файлы с документами (*notebooks*), которые в одном из 10 форматов сохраняются по цепочке команд "*File -> {Save As | Save}*" *GUI* (наиболее используемые форматы ".nb", ".m"), файлы с *Mathematica*-объектами, сохраняемыми по функции *Save* (входной формат), и файлы с пакетами *Mathematica* (форматы ".m" либо ".mx"). Эти файлы представляют вполне определенный интерес при решении многих задач, требующих как стандартных, так и продвинутых приемов программирования.

Для стандартной поддержки работы с ними *Mathematica* располагает рядом средств, тогда как для обеспечения расширенной работы с подобными файлами может быть создан комплекс средств, некоторые из которых рассматриваются в настоящей книге, а также в полном объеме включены во вторую версию нашего пакета *AVZ\_Package\_1* [120], имеющего *FreeWare*-лицензию. При этом, файлы любого из указанных форматов с сохраненными в них по функции **Save** определениями объектов в результате загрузки данных файлов по функции **Get** (<<) в последующие сеансы системы обеспечивают доступность данных объектов, что наглядно иллюстрируют примеры в [100,101].

Несложно убедиться, что файлы, созданные по функции **Save**, содержат определения объектов во входном *Mathematica*-формате независимо от расширения имени файла. Это обеспечивает возможность достаточно несложной организации обработки таких файлов для различных приложений. В частности, исходя из структуры таких файлов, можно без их загрузки в текущий сеанс получать списки *имен* объектов, находящихся в них. Для этого можно использовать процедуру **Nobj**[*x*, *y*], вызов которой возвращает список имен объектов в строчном формате, ранее сохраненных в файле *x* по функции **Save**, тогда как через второй фактический аргумент *y* возвращается список *заголовков* в строчном формате данных объектов. Такое решение достаточно существенно, т.к. в файле могут находиться одноименные объекты с различными заголовками, которые идентифицируют *уникальность* объекта. При этом, может возникнуть необходимость не загружать по функции **Get** (<<) в текущий сеанс *полностью* файл, ранее созданный по **Save**, с активацией *всех* содержащихся в нем объектов, а загружать содержащиеся в файле объекты выборочно, т.е. создавать в своем роде библиотеки пользовательских средств. Относительно пакетов, созданных по цепочке функций *GUI File* → *Save As* → *Mathematica Package (\*.m)*, эту задачу может выполнять процедура **Aobj**[*w*, *y*], вызов которой активизирует в текущем сеансе все объекты с именем *y* из *m*-файла *w*, ранее созданного по цепочке функций *GUI*. Следующий фрагмент представляет исходные коды процедур **Nobj** и **Aobj** с наиболее типичными примерами их использования.

```
In[2026]:= Art1 := #^2 &; Art2 = #^3 &; Art3 := #^4 &; Art4 = #^5 &; Kr = 17; Agn[y_] := 66
In[2027]:= Avz[x_] := x + 460; SetAttributes[Avz, {Listable, Protected}]
In[2028]:= Save["C:\\Obj.m", {Adrive, SetDir, Art1, Art2, Art3, Art4, Nobj, Kr, Agn, Avz}]
In[2029]:= Nobj[x_;/ FileExistsQ[x] && StringTake[x, -2] == ".m", y_;/ ! HowAct[y]] :=
Module[{a, b, c, d, p, h, t, k = 1},
  If[FileExistsQ[x] && MemberQ[{"Table", "Package"},
    Quiet[FileFormat[x]], {a, b, d, h} = {OpenRead[x], {}, "78", {}};
  While[! SameQ[d, "EndOfFile"], d = ToString[Read[a, String]]];
  If[! SuffPref[d, " ", 1], If[! StringFreeQ[d, "::usage = \"\""], AppendTo[b,
StringSplit[StringTake[d, {1, Flatten[StringPosition[d, "::usage"]][[1]] - 1]}, "/: "][[1]],
p = Quiet[Check[StringTake[d,
{1, Flatten[StringPosition[d, {" := ", " = "}]][[1]] - 1], $Failed]];
  If[! SameQ[p, $Failed], If[SymbolQ[p] && StringFreeQ[p, {" ", "{", "\""}] | |
StringFreeQ[p, {" ", "{", "\""}] && HeadingQ1[p] === True, AppendTo[b, p]]]; k++];
```

```

Close[a]; b = Sort[DeleteDuplicates[b]]; h = Select[b, ! SymbolQ[#] &];
t = Map[If[SymbolQ[#], #, HeadName[#]] &, h]; b = MinusList[b, h];
b = Sort[DeleteDuplicates[Join[b, t]]];
y = MinusList[Sort[DeleteDuplicates[Join[h, Select[Map[If[! UnevaluatedQ[HeadPF, #],
HeadPF[#]] &, b], ! SameQ[#, Null] &]]], b]; b, $Failed]]

In[2030]:= Clear[ArtKr]; Nobj["C:\\Obj.m", ArtKr]
Out[2030]= {"Adrive", "Agn", "Art1", "Art2", "Art3", "Art4", "Avz", "BlockFuncModQ",
"ClearAllAttributes", "Contexts1", "Definition2", "HeadingQ", "HeadingQ1",
"HeadPF", "HowAct", "Kr", "ListStrToStr", "Map3", "Mapp", "MinusList",
"Nobj", "ProcQ", "PureDefinition", "RedSymbStr", "Sequences", "SetDir",
"StrDelEnds", "StringMultiple", "StringSplit1", "SubsDel", "SuffPref",
"SymbolQ", "SymbolQ1", "SysFuncQ", "SysFuncQ1", "SystemQ", "ToString1"}

In[2031]:= ArtKr
Out[2031]= {"Adrive[]", "Agn[y_]", "Avz[x_]", "BlockFuncModQ[x_, y_]"},
"ClearAllAttributes[x_]", "Contexts1[]", "Definition2[x_ /; SymbolQ[x] === HowAct[x]]",
"HeadingQ1[x_ /; StringQ[x]]", "HeadingQ[x_ /; StringQ[x]]", "HowAct[x_]",
=====
"SymbolQ[x_]", "SysFuncQ1[x_]", "SysFuncQ[F_]", "SystemQ[S_]", "ToString1[x_]"}

In[2059]:= Aobj[x_ /; FileExistsQ[x] && StringTake[x, -2] == ".m", y_ /; SymbolQ[y] ||
ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] :=
Module[{a, b = "(*", c = "*)", d = "(* ::Input:: *)", p = {Read[x, String], Close[x]}[[1]],
h = Mapp[StringJoin, Map[ToString, Flatten[{y}]], "[", k, j, g, s, t = {}, v = {}],
If[p != "(* ::Package:: *)", $Failed, a = ReadFullFile[x];
If[StringFreeQ[a, d], $Failed, a = StringSplit[a, d][[2 ;; -1]];
a = Map[StringReplace[#, {b -> "", c -> ""}] &, a]; a = Select[a, SuffPref[#, h, 1] &];
For[k = 1, k <= Length[h], k++, g = h[[k]]; For[j = 1, j <= Length[a], j++, s = a[[j]];
c = StrSymbParity[s, g, "[", "]"]; c = If[c == {}, False,
HeadingQ1[Quiet[ToString[ToExpression[c[[1]]]]]] || HeadingQ1[c[[1]]];
If[SuffPref[s, g, 1] && c, AppendTo[t, s];
AppendTo[v, StringTake[g, {1, -2}]]]; Map[ToExpression, t];
If[v != {}, Print["Software for " <> ToString[v] <> " is downloaded"],
Print["Software for " <> ToString[Flatten[{y}]] <> " was not found"]]]]]]]

In[2060]:= Aobj["C:\\AVZ_Package\\RANS.m", {Nobj, Avz, Agn, ArtKr, Sv}]
Software for {Nobj, Avz, Agn, ArtKr, Sv} is downloaded
In[2061]:= Aobj["C:\\AVZ_Package\\RANS.m", {Nobj78, Avz460, Agn66, Vsv46}]
Software for {Nobj78, Avz460, Agn66, Vsv46} was not found

```

Верхняя часть фрагмента представляет сохранение в файле *m*-формата *Mathematica*-объектов как из данного фрагмента, так и представленных несколько выше в этом же разделе. Далее представлен исходный код процедуры *Nobj* и пример ее применения.

Тут же следует отметить, что выполнение процедуры **Nobj** потребует определенных временных издержек. При этом, если главный результат вызова процедуры содержит список имен в строчном формате средств, содержащихся в файле *x*, то через второй *y* аргумент возвращаются только заголовки средств, обладающих таковыми.

Тогда как во второй части фрагмента, отражающего работу в новом сеансе, приведен исходный код процедуры **Aobj**[*x*, *y*] с примером ее использования для активизации в текущем сеансе процедур {**Nobj**, **Avz**, **Agn**, **ArtKr**, **Sv**}, находящихся в ранее созданном по цепочке функций *GUI File* → *Save As* → *Mathematica Package (\*.m)* файле. Проверка подтверждает доступность указанных процедур в текущем сеансе. Более того, вторым аргументом *y* у процедуры **Aobj** может выступать отдельный символ либо их список. К тому же предполагается, перед сохранением в *m*-файле *x* все определения объектов в текущем документе были вычислены в отдельных *Input*-параграфах.

Успешный вызов процедуры **Aobj** ничего не возвращает с выводом сообщения о тех средствах, которые были загружены из *m*-файла *x*, или которые отсутствуют в файле. Процедуры **Nobj** и **Aobj** обрабатывают основные ошибочные ситуации, возвращая на них значение **\$Failed**. При этом, обе процедуры могут быть расширены посредством нагрузки их новыми функциями. Имеется целый ряд и других интересных процедур для обеспечения работы с файлами входного формата *Mathematica*, имена которых имеют расширения {".nb", ".m", ".txt"} и др. Все такие средства основываются на основе анализа внутренней структуры файлов такого формата. Ряд из них дает возможность создавать весьма эффективные библиотеки пользователя, содержащие определения *Mathematica*-объектов. Эти и ряд других средств оформлены специальным пакетом, поддерживающим релизы 7 ÷ 9 системы *Mathematica* [120]. Часть из данных средств будет рассмотрена нами в настоящей книге несколько ниже.

Ряд замечаний следует сделать по стандартной функции **Save**, сохраняющей объекты в заданном файле в режиме *дописывания* (**Append**); при этом, неопределенные символы в файле не сохраняются без вывода каких-либо сообщений, т.е. вызов функции **Save** возвращает **Null**, т.е. ничего. Между тем, при сохранении в файле по функции **Save** процедуры либо функции с именем *A* в файле сохраняются все активные в текущем сеансе одноименные объекты с различными заголовками, как идентификаторами их оригинальности. Для устранения такой ситуации предлагается обобщение функции **Save** на возможность сохранения объектов с конкретными заголовками. Таковую задачу решает процедура **Save1**, чей исходный код представлен следующим фрагментом.

```
In[2010]:= A[x_] := x^2; A[x_, y_] := x+y; A[x_, y_, z_] := x+y+z; A[x_] := {x}; DefFunc3[A]
Out[2010]= {"A[x_] := x^2", "A[x_, y_] := x + y", "A[x_, y_, z_] := x + y + z", "A[x_] := {x}"}
```

```
In[2011]:= Save1[x_/, StringQ[x], y_/, DeleteDuplicates[Map[StringQ, Flatten[{y}]]][[1]] :=
Module[{Rs, t = Flatten[{y}], k = 1},
Rs[n_, m_] := Module[{b, c = "$Art24$$$Kr17$",
a = If[SymbolQ[m], Save[n, m], If[StringFreeQ[m, "["], $Failed,
StringTake[m, {1, Flatten[StringPosition[m, "["]][[1]] - 1}]]],
If[a === Null, Return[], If[a === $Failed, Return[$Failed],
```

```

If[SymbolQ[a], b = DefFunc3[a], Return[$Failed]]];
If[Length[b] == 1, Save[n, a], b = Select[b, SuffPref[#, m, 1] &]];
If[b != {}, b = c <> b[[1]], Return[$Failed]]; ToExpression[b]; a = c <> a;
ToExpression["Save[" <> ToString1[n] <> ", " <> ToString1[a] <> "]];
BinaryWrite[n, StringReplace[ToString[StringJoin[Map[
FromCharCode, BinaryReadList[n]]], c -> ""]]; Close[n]; ];
For[k, k <= Length[t], k++, Rs[x, t[[k]]]]

In[2012]:= Save1["D:\\AVZ_Package\\rans_ian.m", {"A[x_, y_, z_]", "A[x_]"}]
In[2013]:= Clear[A]; DefFunc3[A]
Out[2013]= DefFunc3[A]
In[2014]:= << "D:\\AVZ_Package\\rans_ian.m"
In[2015]:= DefFunc3[A]
Out[2015]= {"A[x_, y_, z_] := x + y + z", "A[x_] := {x}"}
In[2016]:= Agn = 66; Save1["C:/Temp/Herson.m", {"A[x_, y_, z_]", "B", "A[x_] ", "Agn"}]
In[2017]:= Clear[A, B, Agn]; Map[DefFunc3, {A, B, Agn}]
In[2018]:= {DefFunc3[A], DefFunc3[B], Agn}
Out[2018]= {DefFunc3[A], DefFunc3[B], Agn}
In[2019]:= << "C:\\Temp\\Herson.m"
Out[2019]= 66
In[2020]:= DefFunc3[A]
Out[2020]= {"A[x_, y_, z_] := x + y + z", "A[x_] := {x}"}
In[2021]:= {DefFunc3["B"], Agn}
Out[2021]= {"B[x_] := x^2", 66}

```

Вызов процедуры **Save1**[*x*, *y*] сохраняет в файле, определенном первым фактическим аргументом *x*, определения объектов, определяемых вторым фактическим аргументом *y*, в качестве которого могут выступать имя активного в текущем сеансе объекта либо его заголовок в строчном формате, или их сочетания в форме списка. Таким образом, процедура **Save1** может использоваться как стандартная функция **Save**, так и решая задачу сохранения в файле дифференцированно выбранных активизированных в текущем сеансе объектов на основе их заголовков. Успешный вызов процедуры возвращает *Null*; в противном случае возвращается **\$Failed** или невычисленный вызов. Предыдущий фрагмент представляет результаты применения процедуры **Save1** для выборочного сохранения в файлах объектов, активизированных в текущем сеансе *Mathematica*.

Другим средством для сохранения определений объектов в файлах служит функция **DumpSave**, создающая файлы бинарного формата, который оптимизирован для ввода в *Mathematica*. Имена файлов данного формата имеют расширение ".mx" и их можно загружать в текущий сеанс по функции **Get** (<<) аналогично предыдущему формату. В отличие от функции **Save** вызов функции **DumpSave** возвращает список имен и/или определений объектов, сохраняемых в mx-файле. Созданные по функции **DumpSave** файлы можно загружать в текущий сеанс аналогично предыдущему формату наряду с использованием загрузки их через инициализационные файлы "init.m". Между тем,

следует иметь в виду, то весьма существенное обстоятельство, что файлы, созданные по функции **DumpSave**, не только наиболее оптимальны для ввода в *Mathematica*, но и не могут быть загружены на *вычислительной* платформе, которая отлична от той, на которой они были созданы. Следующий фрагмент иллюстрирует создание *mx*-файла с последующим тестированием результата его загрузки в новом сеансе системы.

```
In[2278]:= DumpSave["D:\\AVZ_Package\\Objects.mx", {Adrive, SetDir, Art1, Art2, Art3,
    Art4, Kr, Nobj, Aobj}]
Out[2278]= {Adrive, SetDir, #1^2 &, #1^3 &, #1^4 &, #1^5 &, 17, Nobj, Aobj}
===== Новый текущий сеанс =====
In[1281]:= Get["D:\\AVZ_Package\\Objects.mx"]
In[1282]:= Map[Definition4, {"Art1", "Art2", "Art3", "Kr"}]
Out[1282]= {"Art1 := #1^2 &", "Art2 = #1^3 &", "Art3 := #1^4 &", "Kr = 17"}
```

Таким образом, с файлами бинарного формата следует работать только в том случае, когда не планируется использование разрабатываемых в среде *Mathematica* средств в достаточно широком аспекте, т.е. в определенном смысле данный формат носит явно внутренний характер, не обеспечивая переносимости создаваемых средств.

В целом ряде случаев возникает необходимость загрузки в текущий сеанс файлов *{nb, m, mx, txt}*-типов или файлов *ASCII*-формата без расширения имени, расположенных в одном из каталогов файловой системы компьютера; более того, располагая полным именем файла, мы не располагаем информацией о его местоположении в файловой системе компьютера. Данную задачу решает процедура **LoadFile**, чей исходный код и типичные примеры применения представляет нижеследующий фрагмент, а именно:

```
In[2223]:= LoadFile[F_;/; StringQ[F]] := Module[{a, b, c},
    If[! MemberQ[{"nb", "m", "mx", "txt", ""}, ToString[FileExtension[F]]],
        Return["File <" <> F <> "> has an inadmissible type"],
        a = SearchFile[F]; $Load$Files$ = a;
        If[a == {}, Return["File <" <> F <> "> has not been found"],
        Quiet[Check[Get[$Load$Files$[[1]]], c = $Failed, {Syntax::sntxc, Syntax::sntxi}]];
        If[c === $Failed, "File <" <> $Load$Files$[[1]] <> "> has inadmissible syntax",
        "File <" <> $Load$Files$[[1]] <> "> has been loaded;\n$Load$Files$ defines the list with
        full paths to the found files."], Return["File <" <> F <> "> has not been found"]]]

In[2224]:= LoadFile["AVZ_Package_1.m"]
Out[2224]= "File <D:\\Math_myLib\\AVZ_Package_1.m> has been loaded;
    $Load$Files$ defines the list with full paths to the found files."

In[2225]:= $Load$Files$
Out[2225]= {"C:\\Temp\\AVZ_Package_1.m", "D:\\AVZ_Package\\AVZ_Package_1.m"}
In[2226]:= LoadFile["Example78.m"]
Out[2226]= "File <Example78.m> has not been found"
In[2227]:= LoadFile["AVZ_Package71.m"]
Out[2227]= "File <C:\\Temp\\AVZ_Package71.m> has inadmissible syntax"
```



Вызов **LoadFile[F]** процедуры загружает в текущий сеанс файл, заданный его полным именем *F* и имеющий расширение *{m, nb, mx, txt}* или и вовсе без расширения. Более того, при нахождении списка файлов с идентичным именем *F* производится загрузка *первого* из списка с возвратом соответствующего сообщения, тогда как через *глобальную* переменную **\$Load\$Files\$** процедура возвращает список всех файлов *F*, найденных в процессе поиска. Процедура обрабатывает такие ошибочные и особые ситуации, как отсутствие искомого файла в файловой системе компьютера либо недопустимый для функции **Get (<<)** синтаксис найденного файла. Такие ситуации идентифицируются возвратом соответствующих сообщений, пригодных для последующей обработки. Тут же следует отметить (*выше этот момент относительно глобальных переменных у модулей уже акцентировался*), что для полнофункционального выполнения процедуры следует предварительно присваивать таким переменным произвольные значения, что делает пакет **AVZ\_Package\_1** [120] в момент своей загрузки в текущий сеанс *Mathematica*.

Завершает данный раздел достаточно простая и в ряде случаев полезная процедура, вызов которой **MathematicaDF[]** возвращает список *ListList*-типа, чьи 2-элементные члены первым элементом содержат тип элемента файловой системы *Mathematica*, в то время как вторым количество данного типа элементов. При этом, *"Dir"* определяет каталог, *"NoExtension"* – файлы без расширения, остальные – тип расширения файла (*под расширением понимается, в частности, также файлы с именем формата ".xxxxxx"*). В следующем фрагменте представлены исходный код процедуры с примерами.

```
In[4063]:= MathematicaDF[] := Module[{a = "Art24$Kr17$", b = {}, c = "", d},
    Run["Dir " <> "/A/B/S " <> StrStr[$InstallationDirectory] <> " " <> a];
    While[! SameQ[c, EndOfFile], c = Read[a, String];
        Quiet[If[DirectoryQ[c], AppendTo[b, "Dir"],
            If[FileExistsQ[c], d = FileExtension[c];
                AppendTo[b, If[d === "", "NoExtension", d]],
                AppendTo[b, "NoFile"]]]]; DeleteFile[Close[a]];
    Map[#[[1]], Length[#] &, Gather[b, #1 === #2 &]]

In[4064]:= MathematicaDF[]
Out[4064]= {{{"CreationID", 8}, {"PatchLevel", 1}, {"VersionID", 1}, {"Dir", 1771}, {"exe", 134},
    {"m", 984}, {"nb", 10858}, {"gen", 40}, {"NoExtension", 951}, {"cfs", 38}, {"c", 38},
    {"jar", 137}, {"cmd", 2}, {"vbs", 2}, {"sh", 6}, {"pbs", 1}, {"class", 299}, {"dll", 336},
    =====
    {"zip", 1}, {"conf", 1}, {"manifest", 1}, {"rdf", 1}, {"js", 2}, {"xul", 1}, {"access", 2},
    {"template", 4}, {"policy", 3}, {"security", 2}, {"lib", 23}, {"cu", 37}, {"script", 3},
    {"pem", 1}, {"mlp", 14}, {"a", 3}, {"config", 2}, {"DLL", 1}, {"cs", 46}, {"resx", 1},
    {"node", 1}, {"cxx", 2}, {"cmap", 1}, {"cset", 1}, {"kbd", 1}, {"ini", 1}, {"msg", 1}}

In[4065]:= Plus[Sequences[Map[#[[2]] &, %]]]
Out[4065]= 19811
```

На этом представление средств доступа к системным файлам *Mathematica* завершено и в следующем разделе будут представлены средства работы с внешними файлами.

## 9.2. Средства *Mathematica* для работы с внешними файлами

По такому довольно важному показателю, как средства доступа система *Mathematica*, на наш взгляд, обладает рядом преимуществ по сравнению с системой *Maple*. Прежде всего, *Mathematica* производит автоматическую обработку сотен форматов данных и их подформатов на основе унифицированного использования символьных выражений. Для каждого специфического формата определяется соответствие между внутренним и внешним представлением формата, используя общий механизм элементов данных *Mathematica*. На сегодня *Mathematica 9* в общей сложности поддерживает довольно много различных форматов файлов данных самого различного назначения, их список можно получать по предопределенным переменным **\$ImportFormats** (импортируемые файлы) и **\$ExportFormats** (экспортируемые файлы) в количествах соответственно 168 и 139. Тогда как в качестве базовых файловых форматов следует отметить такие как:

"Bit"	- последовательность битов
"BMP"	- Microsoft <i>bmp</i> -формат
"Byte"	- последовательность 8-битных целых чисел без знака
"C"	- формат генерации C-кода
"Character8"	- последовательность 8-битных символов
"Character16"	- последовательность 16-битных Unicode символов
"DBF"	- формат файлов базы данных <i>dBase</i>
"Directory"	- иерархия каталогов файловой системы
"EPS"	- <i>eps</i> -формат ( <i>Encapsulated PostScript format</i> )
"GIF"	- <i>gif</i> -формат
"HTML"	- { <i>htm, html</i> }-формат
"Integer8"	- последовательность 8-битных целых чисел со знаком
"Integer16"	- последовательность 16-битных целых чисел со знаком
"JPEG"	- { <i>jpeg, jpg</i> }-формат ( <i>JPEG raster image format</i> )
"LaTeX"	- <i>latex</i> -формат ( <i>LaTeX format</i> )
"Package"	- <i>m</i> -формат пакетов <i>Mathematica</i>
"PDF"	- <i>pdf</i> -формат ( <i>Adobe Acrobat PDF format</i> )
"Real32"	- действительные IEEE числа обычной точности
"Real64"	- действительные IEEE числа двойной точности
"RTF"	- <i>rtf</i> -формат ( <i>Microsoft Rich Text Format</i> )
"String"	- формат строчных данных пакета
"Table"	- формат произвольных табличных данных
"TeX"	- <i>tex</i> -формат ( <i>TEX document format</i> )
"Text"	- <i>txt</i> -формат ( <i>ASCII format</i> )
"TIFF"	- { <i>tiff, tif</i> }-формат ( <i>TIFF raster image format</i> )
"UnsignedInteger8"	- последовательность 8-битных целых чисел без знака
"UnsignedInteger32"	- последовательность 32-битных целых чисел без знака
"UnsignedInteger64"	- последовательность 64-битных целых чисел без знака

По функции **FileFormat[x]** предпринимается попытка определить *входной* формат для файла данных, заданного своим именем *x* в строчном формате. В случае наличия для файла *x* расширения имени функция **FileFormat**, практически, аналогична функции **FileExtension**, возвращая имеющееся расширение, за исключением случая пакета (*m-файлов*), когда вместо расширения имени возвращается тип файла "**Package**". Между тем, в ряде случаев идентификация формата производится некорректно, например, попытка тестировать *doc*-файл без расширения имени возвращает "**XLS**", относя его к файлам данных, созданным *Excel 95/97/2000/XP/2003*, что в общем случае неверно.

```
In[2057]:= Map[FileFormat, {"D:/AVZ/AVZ_Package_1.nb", "D:/AVZ/AVZ_Package.m"}]
Out[2057]= {"NB", "Package"}
In[2058]:= FileFormat["D:\\AVZ_Package\\Art1"]
Out[2058]= "Text"
In[2059]:= FileExtension["D:\\AVZ_Package\\Art1"]
Out[2059]= ""
In[2060]:= FileFormat["Art1"]
FileFormat::nffil: File not found during FileFormat[Art1]. >>
Out[2060]= $Failed
In[2061]:= FileFormat["D:\\AVZ_Package\\AVZ_Package_1"]
FileFormat::nffil: File not found during
FileFormat[D:\\AVZ_Package\\AVZ_Package_1]. >>
Out[2061]= $Failed
In[2062]:= Map[FileFormat, {"C:/Temp/AVZ_P", "C:/Temp/AVZ_P1", "C:/Temp/Der"}]
Out[2062]= {"NB", "Package", "XLS"}
In[2063]:= FileFormat["C:\\Temp\\Der.doc"]
Out[2063]= "DOC"

In[2087]:= FileFormat1[x_;/ StringQ[x]] := Module[{a = If[FileExistsQ[x], {x}, SearchFile[x]],
b = {}, c, k = 1}, If[a == {}, {}, For[k, k <= Length[a], k++, c = a[[k]];
b = Append[b, {c, FileFormat[c]}]]; If[Length[b] == 1, b[[1]], b]]

In[2088]:= FileFormat1["AVZ_Package"]
Out[2088]= {"C:\\Temp\\AVZ_Package_1", "NB"}, {"D:\\AVZ_Package", "Directory"},
{"D:\\AVZ_Package\\AVZ_Package_1", "Package"}
```

Более того, по функции **FileFormat[x]** предпринимается попытка определить формат файла данных *x*, расположенного только в подкаталогах, определяемых переменной **\$Path**, в противном случае возвращая **\$Failed** с выводом соответствующего сообщения, как иллюстрирует пример предыдущего фрагмента. Для устранения такой ситуации предлагается достаточно простая процедура **FileFormat1**, расширяющая возможности стандартной функции **FileFormat** и использующая процедуру **SearchFile[x]**, которая представляется несколько ниже. Вызов процедуры **FileFormat1[x]** возвращает формат файла *x*, расположенного в любом каталоге файловой системы компьютера, подобно стандартной функции **FileFormat**; при этом, отсутствие файла *x* инициирует возврат

пустого списка, в противном случае возвращается простой либо вложенный список, в качестве первого элемента которого возвращается полный путь к файлу *x*, тогда как в качестве второго – его формат, распознаваемый функцией **FileFormat**. Более того, при нахождении нескольких файлов с *одинаковым* именем возвращается *вложенный* список, подписки которого имеют указанный формат. Предыдущий фрагмент представляет исходный код процедуры **FileFormat1** с примерами ее использования. В ряде случаев процедура **FileFormat1** предпочтительнее стандартной функции **FileFormat**. Более в деталях с форматами файлов, поддерживаемыми *Mathematica*, можно ознакомиться в справке по системе или в соответствующей литературе, например, в [100,102,110,113].

*Mathematica* обеспечивает достаточно эффективный системо-независимый доступ ко всем аспектам файлов данных любого размера. Для обеспечения операций открытия и закрытия файлов используются следующие базовые функции доступа, а именно:

**OpenRead** – открытие файла данных на чтение с возвратом *InputStream*-объекта;

**OpenWrite** – открытие файла данных на запись с возвратом *OutputStream*-объекта;

**OpenAppend** – открытие файла данных на дозапись с возвратом *OutputStream*-объекта;

**Close** – закрытие файла данных, точнее, ассоциированного с ним потока (*Stream*).

Более того, в качестве единственного формального аргумента первых *трех* функций выступает имя или полный путь к файлу данных в строчном формате; при этом, для функции **OpenWrite[]** допускается вызов без аргументов, открывая на запись новый файл, располагаемый в подкаталоге, предназначенном для временных файлов. Тогда как функция **Close** *закрывает* файл данных, заданный его именем, полным путем или *Stream*-объектом. При попытке закрыть закрытый или несуществующий файл пакет инициирует ошибочную ситуацию. Для устранения такой ситуации, нежелательной в целом ряде случаев, можно применять для закрытия *закрытого* или *несуществующего* файла очень простую функцию **Closes**, обеспечивающую закрытие любого файла без вывода каких-либо ошибочных сообщений с возвратом значения **Null**, т.е. ничего, но, возможно, имени либо полного пути к закрытому файлу данных.

Под *Stream*-объектом функций доступа **OpenRead**, **OpenWrite** и **OpenAppend** объект следующего достаточно простого формата понимается, а именно:

**{OutputStream | InputStream}[<Файл>, <Логический канал в/в>]**

По вызову **Streams[]** возвращается список *Stream*-объектов файлов данных, открытых в текущем сеансе, включая системные файлы. Для получения списка *Stream*-объектов файлов, отличных от системных, можно использовать вызов процедуры **StreamsU[]**. В следующем фрагменте представлены исходные коды упомянутых процедур наряду с примерами использования их и рассмотренных функций доступа к файлам данных.

```
In[2042]:= Streams[]
Out[2042]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[2043]:= S1 = OpenRead["D:/Math_myLib/testfile"]
Out[2043]= InputStream["D:/Math_myLib/testfile", 33]
In[2044]:= S2 = OpenWrite["D:/Math_myLib/testfile.txt"]
Out[2044]= OutputStream["D:/Math_myLib/testfile.txt", 34]
```

```

In[2045]:= S3 = OpenAppend["D:/Math_myLib/testfile1.txt"]
Out[2045]= OutputStream["D:/Math_myLib/testfile1.txt", 35]
In[2046]:= Streams[]
Out[2046]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
            InputStream["D:/Math_myLib/testfile", 33],
            OutputStream["D:/Math_myLib/testfile.txt", 34],
            OutputStream["D:/Math_myLib/testfile1.txt", 35]}
In[2047]:= Close[S3]
Out[2047]= "D:/Math_myLib/testfile1.txt"
In[2048]:= Streams[]
Out[2048]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
            InputStream["D:/Math_myLib/testfile", 33],
            OutputStream["D:/Math_myLib/testfile.txt", 34]}
In[949]:= OpenWrite[]
Out[2049]= OutputStream["C:\\Documents and Settings\\Aladjev\\Local Settings\\
            Temp\\m-0a9fa8c8-b787-4e47-958b-c392c61f793c", 36]
In[2050]:= Close["D:/Math_myLib/testfile1.txt"]
            General::openx: D:/Math_myLib/testfile1.txt is not open. >>
Out[2050]= Close["D:/Math_myLib/testfile1.txt"]
In[2051]:= Close["D:/Math_myLib/test72.txt"]
            General::openx: D:/Math_myLib/test72.txt is not open. >>
Out[2051]= Close["D:/Math_myLib/test72.txt"]
In[2052]:= Closes[x_] := Quiet[Check[Close[x], Null]]
In[2053]:= Closes["D:/Math_myLib/test72.txt"]
In[2054]:= Closes["D:/Math_myLib/testfile1.txt"]
In[2055]:= SP = Streams[]
Out[2055]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
            InputStream["D:/Math_myLib/testfile", 33],
            OutputStream["D:/Math_myLib/testfile.txt", 34],
            OutputStream["C:\\Documents and Settings\\Aladjev\\Local Settings\\
            Temp\\m-0a9fa8c8-b787-4e47-958b-c392c61f793c", 36]}
In[2056]:= StreamsU[] := Select[Streams[], !MemberQ[{"[stdout", "[stderr",
                                                    StringTake[ToString[#1], {13, 19}]] &]
In[2057]:= StreamsU[]
Out[2057]= {InputStream["D:/Math_myLib/testfile", 33],
            OutputStream["C:\\Documents and Settings\\Aladjev\\Local Settings\\
            Temp\\m-0a9fa8c8-b787-4e47-958b-c392c61f793c", 36]}
In[2058]:= Close["D:/Math_myLib/testfile"]
Out[2058]= "D:/Math_myLib/testfile"
In[2059]:= StreamsU[]
Out[2059]= {OutputStream["D:/Math_myLib/testfile.txt", 34],
            OutputStream["C:\\Documents and Settings\\Aladjev\\Local Settings\\
            Temp\\m-0a9fa8c8-b787-4e47-958b-c392c61f793c", 36]}

```

Открыв по функции **OpenWrite** требуемый файл, получаем возможность записывать в него требуемые данные по функции **Write** следующего формата, а именно:

**Write[OutputStream, W1, W2, ..., Wn]**

где **Write** записывает в файл, определенный указанным потоком (*каналом*), выражения **W<sub>j</sub> (j=1..n)**, разделяемые символами *новой строки*. При этом, **Write** является функцией доступа *нижнего* уровня системы. В качестве канала вывода может выступать как файл, так и канал либо их список, каждый элемент которого задается *Stream*-объектом либо именем в строчном формате. Более того, если **Write** применяется к несуществующему либо закрытому файлу, то это эквивалентно его открытию на запись с последующей в него записью данных и возвратом имени (*полного пути*) к файлу. Необходимо иметь в виду, что после прекращения операций записи в файл, он остается *открытым* вплоть до его *явного закрытия* функцией **Close**. Для *закрытия* всех *открытых* в текущем *сеансе* системы *каналов* и *файлов*, исключая системные, можно применять довольно простую функцию **CloseAll**, чей вызов **CloseAll[]** закрывает все упомянутые открытые каналы и файлы с возвратом списка файлов. По умолчанию, **Write** формирует запись в файл согласно с установкой опции **FormatType** для используемого *выходного* канала/файла. Опция используется при открытии файла на *запись* и определяет *формат* выражений, записываемых в данный файл. Типичные установки для **FormatType** приведены в [98]. При этом, по умолчанию вывод выражений в файл выполняется в строчном формате отдельными записями (*строками*). Следующий простой фрагмент довольно наглядно иллюстрирует использование функций **OpenWrite**, **Write**, **StreamsU** и **CloseAll**.

```
In[2048]:= S = OpenWrite["D:/AVZ_Package/file75"]
Out[2048]= OutputStream["D:/AVZ_Package/file75", 24]
In[2049]:= For[k = 1, k <= 25, k++, Write[S, "RANS_" <> ToString[k]]]
In[2050]:= {Streams[], Close[S], StreamsU[]}
Out[2050]= {{OutputStream["stdout", 1], OutputStream["stderr", 2],
OutputStream["D:/AVZ_Package/file75", 24]}, "D:/AVZ_Package/file75", {}}
In[2051]:= Write["D:/AVZ_Package/file75", "International Academy of Noosphere"]
In[2052]:= StreamsU[]
Out[2052]= {OutputStream["D:/AVZ_Package/file75", 25]}
In[2053]:= Close[OutputStream["D:/AVZ_Package/file75", 25]]
Out[2053]= "D:/AVZ_Package/file75"
In[2054]:= StreamsU[]
Out[2054]= {}
In[2055]:= Write["D:/AVZ_Package/File450", "International Academy of Noosphere"]
In[2056]:= StreamsU[]
Out[2056]= {OutputStream["D:/AVZ_Package/File450", 26]}
In[2057]:= {StreamsU[], Close[OutputStream["D:/AVZ_Package/file450", 26]], StreamsU[]}
Out[2057]= {{OutputStream["D:/AVZ_Package/file450", 26]}, "D:/AVZ_Package/file450", {}}
In[2058]:= CloseAll[] := Map[Close, StreamsU[]]
In[2059]:= CloseAll[]
Out[2059]= {}
```

```
In[2060]:= For[k = 1, k <= 3, k++, OpenWrite["C:/"] <> ToString[k]]
In[2061]:= StreamsU[]
Out[2061]= {OutputStream["C:/1", 35], OutputStream["C:/2", 36], OutputStream["C:/3", 37]}
In[2062]:= CloseAll[]
Out[2062]= {"D:/3", "C:/A3", "C:/3", "C:/1", "C:/2", "C:/15"}
```

Открыв по функции **OpenRead** требуемый файл, получаем возможность считывать в текущий сеанс из него требуемые данные по функции **Read** одного из следующих 3-х форматов, а именно:

**Read[InputStream]** – чтение одной записи из файла, указанного *InputStream*-объектом;

**Read[InputStream, type]** – чтение одной записи из файла, заданного *InputStream*-объектом, согласно указанного вторым аргументом типа;

**Read[InputStream, {type1, ..., typeN}]** – чтение последовательности из *N* записей заданного *InputStream*-объектом файла, согласно указанным вторым аргументом типам.

Функция **Read** является функцией доступа нижнего уровня системы. В качестве канала ввода может выступать как файл данных, так и канал или их список, каждый элемент которого задается *Stream*-объектом или именем в строчном формате. Более того, если **Read** применяется к закрытому файлу, то это эквивалентно его открытию на чтение с последующим чтением из него данных и с возвратом считанной записи. Если функция **Read** применяется к несуществующему файлу, инициируется ошибочная ситуация с возвратом вызова функции *невычисленным*. Следует иметь в виду, после прекращения операций чтения из файла он остается открытым вплоть до *явного* его закрытия **Close** функцией. Для закрытия всех открытых в текущем сеансе каналов и файлов, исключая системные, возможно применить ранее рассмотренную функцию **CloseAll**, чей вызов **CloseAll[]** закрывает все *открытые* каналы и файлы, исключая системные. Функция **Read** по умолчанию читает логическую запись из файла либо читает записи согласно указанным типами данных в соответствии со следующими допустимыми типами:

*Byte* – считывает единственный байт, возвращаемый его целочисленным кодом;

*Character* – считывает единственный символ, возвращаемый в строчном формате;

*Expression* – считывает корректное *Mathematica*-выражение;

*Number* – считывает целое или приближенное число, заданное в *E*-формате;

*Real* – считывает приближенное число, заданное в *E*-формате;

*Record* – считывает последовательность символов, завершенных разделителем записей;

*String* – считывает строку, завершенную символом перевода строки и возврата каретки;

*Word* – считывает последовательность символов, завершенную разделителем слов.

Смысл данных типов достаточно прозрачен и может быть детализирован в системной справке. Аналогично системе *Maple* и система *Mathematica* также имеет возможность открывать один и тот же файл на разных потоках и в различных режимах, используя разные для его имени или пути к нему кодировки (используя альтернативные регистры для букв или/и замены разделителей подкаталогов "\" на "/", и наоборот) при открытиях файлов. Следующий простой фрагмент иллюстрирует применение данного подхода для открытия одного и того же файла на 2-х разных каналах на чтение с последующим попеременным чтением записей из него. Данный фрагмент наглядно иллюстрирует

применение функций **OpenRead**, **Read**, **StreamsU** и **CloseAll**, рассмотренных выше.

```
In[2224]:= F = "D:\\AVZ_Package\\file7"; {S, S1} = {OpenRead[F],
    OpenRead[If[UpperCaseQ[StringTake[F, 1]], ToLowerCase[F], ToUpperCase[F]]]}
Out[2224]= {InputStream["D:/Mat_myLib/file7", 44], InputStream["d:/mat_mylib/file7", 45]}
In[2225]:= R = {}; For[k = 1, k <= 7, k++, R = Append[R, {Read[S], Read[S1]}]]; Flatten[R]
Out[2225]= {"RANS_1", "RANS_1", "RANS_2", "RANS_2", "RANS_3", "RANS_3", "RANS_4",
    "RANS_4", "RANS_5", "RANS_5", "RANS_6", "RANS_6", "RANS_7", "RANS_7"}
In[2226]:= Read["D:\\Mat_myLib\\file7"]
Out[2226]= "RANS_1"
In[2227]:= StreamsU[]
Out[2227]= {InputStream["D:\\Math_myLib\\file7", 59]}
In[2228]:= Read["D:\\Math_myLib\\file460"]
    Read::openx: D:\\Math_myLib\\file460 is not open. >>
Out[2228]= Read["D:\\Math_myLib\\file460"]
In[2229]:= Read["D:\\Mat_myLib/file7", {Byte, Byte, Byte, Byte, Byte, Byte}]
Out[1429]= {34, 82, 65, 78, 83, 95}
In[2230]:= {h, S} = {"", {}}; While[h != EndOfFile, h = Read["D:\\Math_myLib\\file460",
    Word]; S = Append[S, h]; CloseAll[]; Select[S, #1 != EndOfFile &]
Out[2230]= {"International", "Academy", "of", "Noosphere", "RANS", "Tallinn", "May_2013"}
```

При этом, следует иметь ввиду, что как и в случае системы *Maple*, необходимо особое внимание при открытии одного и того же файла на разных каналах и, прежде всего, в различных режимах доступа во избежание возможных особых и ошибочных моментов, включая искажение данных в файле. Тогда как в целом ряде случаев данный подход при работе с большими файлами может давать вполне ощутимый временной эффект наряду с упрощением ряда алгоритмов обработки данных, находящихся в файлах.

Подобно системе *Maple*, *Mathematica* располагает весьма полезными средствами для работы с указателем, определяющим позицию сканирования файла, т.е. позицию, в(с) которую(ой) будет производиться запись (чтение) данных в следующий момент.

**StreamPosition[F]** – возвращает целое число, определяющее текущую позицию сканирования открытого файла либо его описателя **F**;

**SetStreamPosition[F, n]** – задает номер **n** текущей позиции сканирования открытого файла либо его описателя **F**;

**Skip[F, T]** – пропускает одну запись данного типа **T** открытого файла **F** во входном потоке;

**Skip[F, T, n]** – пропускает **n** записей данного типа **T** открытого файла **F** во входном потоке;

**Find[F, "h"]** – возвращает первую строку в считываемом файле **F**, содержащую подстроку **h**;

**Find[F, {"h1", ..., "hN"}]** – возвращает первую строку в считываемом файле **F**, содержащую по меньшей мере одну из подстрок **h1, ..., hN**; при этом, функция **Find** работает и с закрытыми файлами данных.

Первые две функции **StreamPosition** и **SetStreamPosition** позволяют как производить мониторинг текущей позиции указателя открытого файла **F**, так и устанавливать для него новую позицию соответственно. При этом, на закрытых либо несуществующих



файлах вызовы *обоих* этих функций инициируют ошибочные ситуации. Аналогична реакция на статус файла данных и функции **Skip**, тогда как уже вызов функции **Find** открывает поток на чтение из файла *F*. Смысл представленных функций достаточно прозрачен и особых пояснений не требует. В связи со сказанным возникает вопрос по определению статуса файла данных – *открытый, закрытый или не существует*. В этом отношении может оказаться довольно полезной процедура **FileOpenQ**, чей исходный код с примерами применения представляет следующий фрагмент вместе с примером использования рассмотренной стандартной функции **Skip** системы *Mathematica*.

```
In[2504]:= R = OpenRead["D:\\AVZ_Package\\file78"]; {h, S} = {"", {}};
          While[h != EndOfFile, h = Read[R, String]; S = Append[S, h];
          Skip[R, String, 2]]; CloseAll[]; Select[S, #1 != EndOfFile &]
Out[2504]= {"RANS_1", "RANS_4", "RANS_7", "RANS_10", "RANS_13", "RANS_16"}

In[2505]:= FileOpenQ[F_/, StringQ[F]] := Module[{A, a = FileType[F], b, d,
          x = inputstream, y = outputstream, c = Map[ToString1, StreamsU[]],
          f = ToLowerCase[StringReplace[F, "\\\" -> "/"]]},
          A[x_] := Module[{a1 = ToString1[x], b1 = StringLength[ToString[Head[x]]],
          ToExpression["{" <> StrStr[Head[x]] <> ", " <> StringTake[a1, {b1 + 2, -2}] <> ""]];
          If[MemberQ[{Directory, None}, a], Return[$Failed], Clear[inputstream, outputstream];
          d = ToExpression[ToLowerCase[StringReplace[ToString1[Map[A,
          StreamsU[]], "\\\" -> "/"]]]; a = Select[d, #[[2]] == f &];
          If[a == {}, {inputstream, outputstream} = {x, y}; False,
          a = {ReplaceAll[a, {inputstream -> "read", outputstream -> "write"}],
          {inputstream, outputstream} = {x, y}][[1]]]; If[Length[a] == 1, a[[1]], a]]

In[2506]:= Write["rans.ian"]; Write["C:\\Temp/Summ.doc"]; Write["d:\\Book/Grin.pdf"]
In[2507]:= Map[FileOpenQ, {"rans.ian", "D:\\Book\\Grin.pdf", "Tallinn.docx",
          "C:\\Temp/Summ.doc", "Rans.Ian", "C:\\HTTP.html"}]
Out[2507]= {{{"write", "rans.ian", 362}}, {"write", "d:/book/grin.pdf", 364}}, $Failed,
          {"write", "c:/temp/summ.doc", 363}}, {"write", "rans.ian", 362}}, False}
In[2508]:= Write["rans.ian"]; Read["Rans.Ian"]; Write["RANS.IAN"];
In[2509]:= FileOpenQ["rans.ian"]
Out[2509]= {"write", "rans.ian", 362}, {"read", "rans.ian", 441}, {"write", "rans.ian", 442}}
```

Вызов функции **FileOpenQ[F]** возвращает вложенный список  $\{\{R, F, Channel\}, \dots\}$ , если файл *F* открыт на чтение/запись ( $R = \{\text{"read"} | \text{"write"}\}$ ), определяет собственно файл *F* в *стилизованном* формате (*LowerCase* + *все вхождения* *"\"* *заменены на* *" / "*), в то время как *Channel* определяет логический канал, по которому был открыт файл *F* в указанном первым элементом *R* списка режиме; если файл *F* закрыт, возвращается *False*, если же файл *F* отсутствует, то возвращается *\$Failed*. При этом, *вложенный* список использован с той целью, что файл *F* может открываться согласно синтаксически различным *СФ*, например, *"d:\\Book/Grin.pdf"* и *"d:/book\\grin.pdf"*, позволяя вести его обработку в разных режимах одновременно. Об этом шла речь несколько выше. Рекомендуются в

реализации процедуры обратить внимание на простой прием, использованный для сохранения значений глобальных переменных  $\{inputstream, outputstream\}$ , полезный в программировании различного типа процедур, использующих переменные данного типа. При таком подходе внутри процедуры глобальные переменные используются с достаточно большим произволом, не влияя на их значения вне области процедуры.

Процедура **FileOpenQ1** является достаточно полезным расширением рассмотренной выше процедуры **FileOpenQ**. Вызов процедуры **FileOpenQ1[F]** возвращает *вложенный* список формата  $\{\{R, x, y, \dots, z\}, \{\{R, x1, y1, \dots, z1\}\}$ , если файл  $F$  открыт для чтения либо записи ( $R = \{\text{"in"} \mid \text{"out"}\}$ ), и  $F$  определяет файл в произвольном формате (*Регистр* + "/" и/или "\\"); если же файл данных  $F$  закрыт или отсутствует, возвращается **False**. При этом, подписки  $\{x, y, \dots, z\}$  и  $\{x1, y1, \dots, z1\}$  определяют файлы данных или *полные* пути к ним, которые открыты на чтение и запись соответственно. Файлы данных и пути к ним возвращаются в форматах, которые определены в списке, возвращаемом вызовом функции **Streams[]**, независимо от формата файла данных  $F$ . Следующий фрагмент представляет исходный код процедуры **FileOpenQ1** с примерами ее использования.

```
In[2183]:= Write["dereks1.doc", "ian"]; Write["D:/dereks.doc", "Kr"]; Write["RANS", "ian"]
In[2184]:= Write["C:/TEMP\\DEREKS.doc", 45]; Read["C:/temp/dereks.doc"];
In[2185]:= StreamsU[]
Out[2185]= {OutputStream["dereks1.doc", 24], InputStream["C:/Temp\\dereks.doc", 25],
            OutputStream["RANS", 46], OutputStream["D:/Agn", 87],
            OutputStream["C:/TEMP\\DEREKS.doc", 88],
            OutputStream["D:/dereks.doc", 89], InputStream["C:/temp/dereks.doc", 90]}
In[2186]:= FileOpenQ1[F_;/ StringQ[F]] := Module[{a = StreamFiles[], b, c, d, k = 1, j},
            If[a === "AllFilesClosed", Return[False],
            c = StringReplace[ToLowerCase[F], "/" -> "\\"];
            b = Mapp[StringReplace, Map[ToLowerCase, a], "/" -> "\\"];
            For[k, k <= 2, k++, For[j = 2, j <= Length[b[[k]]], j++,
            If[Not[SuffPref[b[[k]][[j]], c, 2] | |
            SuffPref[b[[k]][[j]], "\\\" <> c, 2], a[[k]][[j]] = Null;
            Continue[], Continue[]]; Mapp[Select, a, ! # === Null &]]
In[2187]:= FileOpenQ1["dereks.doc"]
Out[2187]= {"in", "C:/Temp\\dereks.doc", "C:/temp/dereks.doc"},
            {"out", "D:/dereks.doc", "C:/TEMP\\DEREKS.doc"}
In[2188]:= FileOpenQ1["rans"]
Out[2188]= {"in"}, {"out", "RANS"}
In[2189]:= FileOpenQ1["Tallinn_Tampere_ArtKr.pdf"]
Out[2189]= False
```

Представленные выше функции доступа *Skip, Find, StreamPosition, SetStreamPosition* обеспечивают вполне *эффективные* средства для достаточно тонкого *манипулирования* с файлами и в сочетании с целым рядом других функций доступа они обеспечивают

пользователя как стандартным набором функций для обработки файлов данных, так и предоставляют возможность на их базе создавать собственные средства, *позволяющие* как решать специфические задачи работы с файлами, так и в определенной степени расширять стандартные возможности пакета. Ряд подобных средств представлен и в настоящей книге, и в двух наших пакетах *AVZ\_Package* [90] и *AVZ\_Package\_1* [120].

В дополнение к представленным стандартным операциям обработки файлов, целый ряд других средств пакета достаточно существенно облегчает довольно эффективное программирование более высокого уровня при решении многих задач как обработки файлов, так и управления системой. Естественно, достаточно детально рассматривать как уже представленные средства доступа к файлам, так и нижеследующие не входит в задачи настоящей книги, поэтому представим по ним лишь краткий экскурс в виде краткой справки с некоторыми комментариями по представляемым средствам.

**FileNames** – в зависимости от формата кодирования возвращает список *полных* путей к файлам и/или каталогам, содержащимся в заданном каталоге, на любую глубину в файловой системе компьютера;

```
In[49]:= FileNames["*", "d:/Book"]
```

```
Out[49]= {"d:/Book\\AVZ_Package.nb", "d:/Book\\Fonts", "d:/Book\\MapleMathem.doc"}
```

Функции **CopyFile**, **RenameFile**, **DeleteFile** служат для копирования, переименования и удаления заданных файлов. При этом, первые две функции имеют два аргумента, а именно, первый определяет исходный файл, тогда как второй – принимающий файл или новое имя файла; третья функция в качестве аргумента допускает список *файлов* или отдельный файл, подлежащие удалению из файловой системы компьютера. При успешном вызове функции возвращается значение **Null**, т.е. ничего.

```
In[2157]:= DeleteFile["D:\\Math_myLib\\Help.nb"]
```

Ряд функций предназначен для получения свойств файлов данных, в частности:

**FileDate[F]** – по вызову простейшего формата возвращается дата и время самого последнего обновления содержимого файла *F*; тогда как последующее применение функции **FileDate** дает возможность получить данную информацию в более привычном формате, например:

```
In[2237]:= d = FileDate["C:\\Book\\MapleMathem.doc"]; DateString[d]
```

```
Out[2237]= "Wed 9 Feb 2011 18:05:37"
```

**SetFileDate[F]** – изменяет даты обновления и доступа к файлу *F* на текущие, например:

```
In[2240]:= d = FileDate["C:\\Http.htm"]; DateString[d]
```

```
Out[2240]= "Tue 25 Jan 2013 18:12:42"
```

```
In[2241]:= d = SetFileDate["C:\\Http.htm"]; p = FileDate["C:\\Http.htm"]; DateString[p]
```

```
Out[2241]= "Wed 9 Feb 2011 19:14:16"
```

**FileByteCount[F]** – возвращает размер файла данных *F* в байтах, например:

```
In[2242]:= FileByteCount["C:\\Book\\MapleMathem.doc"]
```

```
Out[2242]= 6136832
```

**FileType[F]** – возвращает тип файла *F* в разрезах: *File*, *Directory*, *None* (отсутствует):

```
In[2243]:= Map[FileType, {"C:\\Book/Addition.doc", "D:/Math_myLib", "RANS_IAN"}]
```

Out[2243]= {File, Directory, None}

**FileFormat[F]** – вызов функции пытается определить, какой **Import**-формат мог бы быть использован для импортирования файла или **URL**, соответствующего аргументу **F**; между тем, на главных каталогах внешней памяти (диск, флэш-память и др.) функция вызывает ошибочную ситуацию; при этом, функция распознает только файлы данных, находящиеся в каталогах, определяемых переменной **\$Path**; для устранения последней ситуации предложена процедура **FileFormat1**, представленная выше, тогда как она довольно несложно может быть расширена с целью устранения и первой ситуации. Процедура **FileFormat2** была разработана на основе процедуры **FileFormat1**, корректно обрабатывает главные каталоги, недоступные либо несуществующие устройства внешней памяти, а также файлы из каталогов файловой системы компьютера. Фрагмент представляет исходный код процедуры с примерами:

```
In[2218]:= Map[FileFormat, {"C:/", "C:\\"}]
General::cdir: Cannot set current directory to System Volume Information. >>
General::cdir: Cannot set current directory to System Volume Information. >>
Out[2218]= {"KML", "KML"}

In[2243]:= FileFormat2[x_ /; StringQ[x]] := Module[{a, b = {}, c, k = 1},
    If[StringLength[x] == 3, If[MemberQ[{":/", ":\\"}, StringTake[x, -2]] &&
        MemberQ[Adrive[], ToUpperCase[StringTake[x, 1]]], Return["Directory"], Null],
    If[DirectoryQ[x], Return["Directory"], a = SearchFile[x]];
    If[a == {}, Return[{}], For[k, k <= Length[a], k++, c = a[[k]];
        b = Append[b, {c, FileFormat[c]}]]]; If[Length[b] == 1, b[[1]], b]]

In[2244]:= Map[FileFormat2, {"C:/", "C:\\", "C:/Temp", "C:\\Temp"}]
Out[2144]= {"Directory", "Directory", "Directory", "Directory"}
In[2145]:= FileFormat2["Obj.m"]
Out[2145]= {"C:\\Temp\\Obj.m", "Package"}, {"D:\\AVZ_Package\\Obj.m", "Package"}]
```

Итак, ранее представленная процедура **FileFormat1** обеспечивает проверку формата файлов, находящихся в каталогах файловой системы компьютера безотносительно их привязки к переменной **\$Path**. Тогда как ее *расширение* в лице процедуры **FileFormat2** дополнительно корректно обрабатывает и главные каталоги внешней памяти, имея в виду то важное обстоятельство, что они являются ключевыми элементами в файловой системе компьютера. Действительно, *первый* пример предыдущего фрагмента весьма наглядно иллюстрирует, что вызов функции **FileFormat[x]** на главном каталоге тома **x** возвращает формат **"KML"**, являющийся стандартным **GIS**-форматом, служащим для хранения картографической информации, вместо того, чтобы возвращать на данном объекте **"Directory"**. Вызов **FileFormat2[x]** устраняет данный недостаток, возвращая на подобных объектах значение **"Directory"**, в остальных ситуациях вызов **FileFormat2[x]** эквивалентен вызову **FileFormat1[x]** рассмотренной выше процедуры.

Следующая версия стандартной функции **FileFormat** попытается идентифицировать тип файла без расширения, базируясь на информации разработчика файла данных, содержащейся в самом содержимом файла. Процедура **FileFormat3** достаточно четко идентифицирует файлы следующих часто используемых типов **DOC, PDF, ODT, TXT**

и **HTML**. При этом, на тип **TXT** проверка файла производится в последнюю очередь, полагая, что файл данного типа должен состоять только из символов со следующими десятичными кодами, а именно:

- 0 ÷ 127**     - символы в *ASCII*-кодировке
- 1 ÷ 31**     -управляющие символы в *ASCII*-кодировке
- 32 ÷ 126**   - печатаемые символы в *ASCII*-кодировке
- 97 ÷ 122**   - буквы латинского алфавита в нижнем регистре
- 129 ÷ 255**  - символы *Latin-1* стандарта *ISO*
- 192 ÷ 255**  - буквы европейских языков

Вызов **FileFormat3[x]** возвращает тип файла, заданного именем или спецификатором *x*. При этом, если файл имеет расширение, то именно оно полагается типом файла *x*. Тогда как вызов **FileFormat3[x, y]** со вторым *необязательным* аргументом – произвольным выражением *y* – в случае файла без расширения имени возвращает полное имя файла с определенным для него расширением, одновременно переименовывая файл *x* под вычисленный формат. Следующий фрагмент представляет исходный код процедуры **FileFormat3** наряду с наиболее типичными примерами ее использования.

```
In[2031]:= FileFormat3[x_;/; FileExistsQ[x], t___] := Module[{a = FileExtension[x], b, c},
    If[a != "", ToUpperCase[a],
    c = If[StringTake[Read[x, String], {1, 5}] == "%PDF-",
        {Close[x, "PDF"]}[-1], b = ReadFullFile[x];
    If[! StringFreeQ[b, "MSWordDoc"], "DOC",
    If[! StringFreeQ[b, ".opendocument.textPK"], "ODT",
    If[! StringFreeQ[b, {"!DOCTYPE HTML ", "text/html"}], "HTML",
    If[MemberQ3[Range[0, 255], DeleteDuplicates[Fatten[Map[
    ToCharacterCode[#] &, DeleteDuplicates[Characters[b]]]]], "TXT", Undefined]]]]];
    If[{t} != {}, RenameFile[x, x <> "." <> c], c]]

In[2032]:= Map[FileFormat3, {"C:\\Temp.Burthday", "C:\\Temp.Protokol", "C:/Books",
    "C:\\A_Word", "C:\\Cinema", "C:\\Temp/Diverticula", "C:\\Temp/2013"}]
Out[2032]= {DOC, PDF, HTML, ODT, TXT, HTML, Undefined}
In[2033]:= FileFormat3["C:\\Temp\\Kherson\\Protokol"]
Out[2033]= "PDF"
In[2034]:= FileFormat3["C:\\Temp\\Kherson\\Protokol", 78]
Out[2034]= "C:\\Temp\\Kherson\\Protokol.PDF"
```

Используя алгоритм, реализуемый процедурой **FileFormat3**, достаточно несложно ее модифицировать под тестирование других типов файлов данных, имена которых не имеют расширения. В задачах обработки файлов данных это может быть полезным. В определенном отношении процедура дополняет процедуры **FileFormat1**, **FileFormat2**.

Кроме перечисленных для работы с *файлами* система *Mathematica* располагает целым рядом довольно полезных функций, здесь нами не рассматриваемых, но с которыми заинтересованный читатель сможет ознакомиться в справочной базе системы либо в

соответствующей литературе [100,102,110]. Между тем, здесь вполне уместно сделать одно достаточно существенное замечание. Подавляющее большинство стандартных функций доступа системы *Mathematica* при указании имени файла без полного пути к нему осуществляют поиск искомого файла лишь в пределах каталогов, отмеченных в переменной **\$Path**, при обнаружении идентифицируя ошибочную ситуацию – «*File not found*» с возвратом **\$Failed**; **\$Failed** завершается и вызов **FindFile[x]**, если файл *x* не найден в каталогах, определяемых переменной **\$Path**. Для снятия этого ограничения со стандартных средств доступа был предложен ряд их расширений, обеспечивающих поиск искомых файлов в пределах всей системы доступных каталогов компьютера. С этой целью использовался выход на уровень команд *DOS*, требующих в ряде случаев внешней реакции пользователя, а именно на запрос **<Cancel>**, **<Try Again>**, **<Continue>** следует отвечать щелчком мыши по полю **<Continue>**. Между тем, при отсутствии на компьютере устройства внешней памяти на дискетах данная ситуация не возникает.

Выше были кратко рассмотрены функции низшего уровня доступа к файлам данных: **OpenRead**, **OpenWrite**, **Read**, **Write**, **Skip** и **Streams**; между тем, не менее важными для доступа к файлам представляются также функции **Get**, **Put**, **Export**, **Import**, **ReadList** и **BinaryReadList**, которые поддерживают операции чтения и записи данных требуемого формата. Вкратце представим здесь указанные функции доступа к файлам данных.

**Export["F.ex", D]** – экспортирует данные *D* в файл "F. ex" в формате, который определяется расширением имени «.ex» файла данных;

**Export[F, D, "ft"]** – экспортирует данные *D* в файл данных *F* в заданном формате *ft*;

**Export[F, D, E]** – экспортирует данные *D* в файл данных *F*, трактуя *D* как *E*.

К сожалению, в релизе 8.0.0 системы *Mathematica* данные функции доступа в полной мере не поддерживаются, о чем свидетельствуют простые примеры [98-101], не взирая на справку по системе, где все выглядит вполне прилично. Не все примеры в справке выполняются успешно в среде упомянутой версии системы *Mathematica*. В то время как функция **Import** – обратная к **Export**, поддерживает форматы кодирования:

**Import[F]** – импортирует данные из файла *F* в формате корректного выражения пакета;

**Import[F, D]** – импортирует заданные элементы *D* из файла данных *F*;

**Import["http://url", ...]**, **Import["ftp://url", ...]** – импортирует данные с указанного сайта.

К сожалению, и для этой функции имеет место ситуация, подобная отмеченной для функции доступа **Export**, т.е. функция **Import** имеет ряд достаточно существенных ограничений, как наглядно иллюстрируют достаточно простые примеры [98-101].

В частности, по глобальным переменным **\$ImportFormats** и **\$ExportFormats** возможно получать списки форматов текущего релиза пакета, поддерживаемых соответственно функциями **Import** и **Export**. Если система поддерживает 157 форматов для функции **Import**, то для функции **Export** поддерживается 139 форматов; при этом, в частности, форматы, такие как "LaTeX", "MDB", "MKV", "MMCIF", "MPS", "MTP", "NDK", "ODS", "RSS", "SDTS", "SHP", "SP3", "SXC", "TLE", "VCF", "VCS", "XPORT" поддерживаются для **Import** и не поддерживаются для **Export**, как достаточно наглядно иллюстрируют приведенные в [98-101] весьма простые примеры.

Функция доступа **Get[F]** (<< *F*) читает данные из файла *F*, вычисляя все содержащиеся

в них определения объектов и возвращая результат вычисления последнего из таких определений, т.е. определения данных выражений активизируются в текущем сеансе. Эта функция в настоящей книге рассматривается неоднократно, поэтому в большей детализации не требуется. Обратная для нее функция доступа **Put** (**>>**) допускает три следующие формата кодирования, а именно:

**Put**[*W*, *F*] – записывает указанное выражение *W* в файл данных *F*;

**Put**[*W1*, *W2*, ..., *Wn*, *F*] – записывает последовательность выражений *Wj* (*j*=1..*n*) в файл *F*;

**Put**[*F*] – создает пустой файл с указанным именем *F*; правда, это же обеспечивает и простая связка {**OpenWrite**[*F*], **Close**[*F*]}, представленная вторым примером следующего фрагмента:

```
In[2260]:= F = "D:\\Math_myLib\\IAN"; {Put[F], StreamsU[], FileByteCount[F]}
Out[2260]= {Null, {}, 0}
In[2261]:= DeleteFile[F]
In[2262]:= {OpenWrite[F], Close[F], StreamsU[], FileByteCount[F]}
Out[2262]= {OutputStream["D:\\Math_myLib\\IAN", 8], "D:\\Math_myLib\\IAN", {}, 0}
In[2263]:= R[x_] := Sin[x]*Cos[x]; Put[R, "D:/Math_myLib/R"]; StreamsU[]
Out[2263]= {}
In[2264]:= Get["D:/Math_myLib/R"]; {StreamsU[], Definition[R]}
Out[2264]= {{}, Definition[R]}
In[2265]:= R[19.42]
Out[2265]= 0.454502
```

Из представленного фрагмента, в частности, следует, что вызов как функции **Put**, так и функции **Get** по завершению операции записи/чтения сразу *закрывает* файл, что не позволяет производить дозапись в файл, а лишь в режиме обновления. Итак, за один вызов функции **Put**[*F*] можно записать несколько выражений в файл *F*, тогда как один вызов функции **Get**[*F*] активизирует в текущем сеансе все выражения, находящиеся в *F*.

Достаточно полезными оказываются функции доступа **ReadList** и **BinaryReadList**, чьи краткие характеристики с примерами применения представлены ниже. Так, **ReadList** имеет следующие четыре формата кодирования, обеспечивающие чтение данных из заданного файла в формате списка либо вложенного списка, а именно:

**ReadList**[*F*] – читает все оставшиеся выражения файла *F*, возвращая их в формате списка;

**ReadList**[*F*, *T*] – читает выражения типа *T* из файла *F*, возвращая их в формате списка;

**ReadList**[*F*, {*T1*, *T2*, ..., *Tn*}] – читает выражения типов *Tj* (*j*=1..*n*) из файла *F*, возвращая их в формате вложенного списка;

**ReadList**[*F*, {*T1*, *T2*, ..., *Tn*}, *m*] – читает только первые *m* групп выражений типов *Tj* (*j*=1..*n*) из файла *F*, возвращая их в формате вложенного списка.

Следующий фрагмент иллюстрирует использование функции доступа **ReadList**:

```
In[647]:= ReadList["D:\\Math_myLib\\exp.txt", {Word, Number}, 7]
Out[647]= {"rans", 2013, {"ian", 460, {"tru", 78, {"avz", 71, {"agn", 66, {"art", 24, {"kr", 17}}}
In[648]:= ReadList["D:\\Math_myLib\\exp.txt", {Word, Number}]
Out[648]= {"rans", 2013, {"ian", 460, {"tru", 78, {"avz", 71, {"agn", 66, {"art", 24, {"kr", 17}}}
```

```
In[649]:= ReadList["D:\\Math_myLib\\exp.txt", String]
Out[649]= {"rans 2013 ian 460 10 2 rags 78 avz 71 agn 66 art 24 kr 17"}
In[650]:= StreamsU[]
Out[650]= {}
```

Функция **BinaryReadList** как и функция **ReadList** обладает аналогичными четырьмя форматами кодирования с той лишь разницей, что считанные данные представлены в виде списка или вложенного списка целых чисел **0..255**, которые представляют коды десятичные символов, составляющих возвращаемые данные, например:

```
In[3345]:= BinaryReadList["D:\\Math_myLib\\expert.txt"]
Out[3345]= {114, 97, 110, 115, 32, 50, 48, 49, 49, 32, 105, 97, 110, 32, 52, 50, 48, 32, 49, 48, 32,
            50, 32, 114, 97, 103, 115, 32, 55, 50, 32, 97, 118, 122, 32, 54, 56, 32, 97, 103, 110, 32,
            54, 51, 32, 97, 114, 116, 32, 50, 49, 32, 107, 114, 32, 49, 52}
In[3346]:= StreamsU[]
Out[3346]= {}
```

Наряду с представленными средствами доступа пакет обеспечивает довольно быстрые чтение и запись бинарных данных по отношению как к файлам, так и к каналам *в/в*. Данные средства включают 2 функции доступа **BinaryWrite** и **BinaryRead**, имеющие следующие достаточно простые форматы кодирования, а именно:

**BinaryRead[F]** – считывает один байт бинарных данных из входного потока либо файла **F** и возвращает результат в виде целого числа из диапазона **0..255**;

**BinaryRead[F, T]** – считывает объект заданного типа **T** из входного потока либо файла **F**;

**BinaryRead[F, {T1, T2, ..., Tn}]** – считывает последовательность объектов заданных типов **{T1, T2, ..., Tn}** из входного потока либо файла **F**;

**BinaryWrite[F, B]** – записывает в файл либо канал **F** один байт **B**, заданный целым числом в диапазоне **0..255**, определяющим его десятичный код;

**BinaryWrite[F, {b1, b2, ..., bn}]** – записывает в файл или канал **F** последовательность байтов **{b1, b2, ..., bn}**, заданных целыми числами в диапазоне **0..255**, определяющими их коды;

**BinaryWrite[F, S]** – пишет в файл или канал **F** последовательность символов, составляющих строку **S**, максимальная длина которой определяется лишь используемой вычислительной платформой.

При этом, для функции **BinaryWrite** представлены лишь 3 формата кодирования из 6 допустимых; смысл форматов достаточно прозрачен и особых пояснений не требует. Следующий фрагмент иллюстрирует примеры применения обеих функций доступа.

```
In[2004]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\ex.txt"]; If[h ==
            EndOfFile, S, S = Append[S, h]; Goto[A]]
Out[2004]= {114, 97, 110, 115, 32, 50, 48, 49, 49, 32, 105, 97, 110, 32, 52, 50, 48, 32, 49, 48, 32, 50,
            32, 114, 97, 103, 115, 32, 55, 50, 32, 97, 118, 122, 32, 54, 56, 32, 97, 103, 110, 32, 54,
            51, 32, 97, 114, 116, 32, 50, 49, 32, 107, 114, 32, 49, 52}
In[2005]:= {StreamsU[], CloseAll[]}
Out[2005]= {{InputStream["D:\\Math_myLib\\ex.txt", 8]}, {"D:\\Math_myLib\\ex.txt"}}
```



```

In[2006]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\exp.txt", "Integer16"];
           If[h === EndOfFile, S, S = Append[S, h]; Goto[A]]
Out[2006]= {24946, 29550, 12832, 12592, 8241, 24937, 8302, 12852, 8240, 12337, 12832, 29216,
           26465, 8307, 12855, 24864, 31350, 13856, 8248, 26465, 8302, 13110, 24864, 29810,
           12832, 8241, 29291, 12576}
In[2007]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\exp.txt", "Integer16"];
           If[h === EndOfFile, S, S = Append[S, h]; Goto[A]]
In[2008]:= For[k = 42, k <= 65, k++, BinaryWrite["D:\\Math_myLib\\w.txt", k]];
           {StreamsU[], CloseAll[]}
Out[2008]= {{OutputStream["D:\\Math_myLib\\w.txt", 46]}, {"D:\\Math_myLib\\w.txt"}}
In[2009]:= S = {}; Label[A]; h = BinaryRead["D:/Math_myLib/w.txt"]; If[h === EndOfFile,
           S, S = Append[S, h]; Goto[A]]
Out[2009]= {42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65}
In[2010]:= {StreamsU[], CloseAll[]}
Out[2010]= {{InputStream["D:\\Math_myLib\\w.txt", 47]}, {"D:\\Math_myLib\\w.txt"}}
In[2011]:= BinaryWrite["D:\\Math_myLib\\w.txt", Sin[x]*Cos[x], "Character8"];
           {StreamsU[], CloseAll[]}
Out[2011]= {{OutputStream["D:\\Math_myLib\\w.txt", 44]}, {"D:\\Math_myLib\\w.txt"}}
In[2012]:= Read["D:\\Math_myLib\\w.txt"]
Out[2012]= Cos[x] Sin[x]
In[2013]:= BinaryWrite["D:\\Math_myLib\\Z.txt", "426847636743624889148922"]
Out[2013]= "D:\\Math_myLib\\Z.txt"
In[2014]:= {StreamsU[], CloseAll[]}
Out[2014]= {{OutputStream["D:\\Math_myLib\\Z.txt", 53]}, {"D:\\Math_myLib\\Z.txt"}}
In[2015]:= Read["D:\\Math_myLib\\Z.txt", Record]
Out[2015]= "426847636743624889148922"
In[2016]:= {StreamsU[], CloseAll[]}
Out[2016]= {{InputStream["D:\\Math_myLib\\Z.txt", 54]}, {"D:\\Math_myLib\\Z.txt"}}

```

Из приведенного фрагмента следует, вызов функций **BinaryWrite[F]** и **BinaryRead[F]** открывает файл/поток/канал соответственно на запись и чтение данных в заданных форматах, позволяя использовать функцию **BinaryWrite** в режиме *дописывания*. Итак, данные функции доступа в совокупности с уже рассмотренными и рядом оставшихся без нашего внимания составляют достаточно развитую систему обработки файлов. С другой стороны, наряду с собственно обработкой внутреннего содержимого файлов, пакет располагает целым рядом средств для поиска файлов, их тестирования, работы с их именами и др. Перечислим только некоторые из них, а именно:

**FindFile[F]** – *отыскивает файл, заданный его именем или путем к нему F, возвращая полный путь к искомому файлу; поиск файла F, заданного только именем, производится в каталогах, нашедших отражение в глобальной переменной \$Path пакета;*

**FileExistsQ[F]** – *возвращает True, если файл F существует, и False в противном случае;*

**FileNameSplit[F]** – *возвращает список элементов, составляющих имя файла F или его путь;*

**FileNameJoin**["S1", "S2", ..., "Sk"] – возвращает конкатенацию строк {"S1", "S2", ..., "Sk"} в формате, пригодном для описания имени файла или пути к нему на текущей платформе;

**FileBaseName**[F] – возвращает главное имя файла F без его расширения или пути к нему;

**ExpandFileName**[F] – возвращает полный путь к файлу F со стандартным разделителем подкаталогов в соответствии с соглашениями текущей платформы; полный путь к файлу F рассматривается относительно текущего подкаталога пользователя; при этом, реального наличия файла F в текущем подкаталоге пользователя не требуется;

**FileNameDepth**[F] – возвращает количество элементов пути к файлу F; при этом, наличия реального файла данных F по указанному пути не требуется;

**FileNameTake**[F {, ...}] – возвращает элементы полного пути к файлу F на данную глубину; функция располагает четырьмя простыми форматами кодирования, чей смысл весьма легко усматривается из приведенных ниже примеров; при этом, реального существования файла F по указанному пути не требуется.

Следующий фрагмент иллюстрирует использование перечисленных функций.

```
In[2205]:= FindFile["init.m"]
Out[2205]= "C:/Documents and Settings/Aladjev/Application Data/Mathematica/Kernel/F"
In[2206]:= FileExistsQ["D:\\Math_myLib\\expert.txt"]
Out[2206]= True
In[2207]:= FileNameSplit["D:\\Math_myLib\\AVZ_Package.nb"]
Out[2207]= {"D:", "Math_myLib", "AVZ_Package.nb"}
In[2208]:= FileNameJoin[%]
Out[2208]= "D:\\Math_myLib\\AVZ_Package.nb"
In[2209]:= FileBaseName["D:\\Math_myLib\\AVZ_Package.nb"]
Out[2209]= "AVZ_Package"
In[2210]:= ExpandFileName["AVZ_Package.nb"]
Out[2210]= "C:\\Documents and Settings/Aladjev\\My Documents\\AVZ_Package.nb"
In[2211]:= FileExistsQ[%]
Out[2211]= False
In[2212]:= FileNameDepth["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb"]
Out[2212]= 5
In[2213]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb"]
Out[2213]= "AVZ_Package.nb"
In[2214]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", 3]
Out[2214]= "D:\\Book\\Aladjev"
In[2215]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", -2]
Out[2215]= "Math_myLib\\AVZ_Package.nb"
In[2216]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", {2, 4}]
Out[2216]= "Book\\Aladjev\\Math_myLib"
```

Как ранее отмечалось, функция **FileExistsQ** подобно ряду других функций доступа в процессе поиска ограничивается лишь каталогами, определенными в **\$Path**. С целью устранения этого недостатка предлагается довольно простая процедура **FileExistsQ1**.

```

In[2072]:= FileExistsQ1[x_ /; StringQ[{x}][[1]]] := Module[{a = SearchFile[{x}][[1]], b = {x}},
               If[a == {}, False, If[Length[b] == 2 && ! HowAct[b][[2]],
               ToExpression[ToString[b][[2]] <> " = " <> ToString[a], Null]; True]]
In[2073]:= {FileExistsQ1["Rans.ian", t], t}
Out[2073]= {True, {"C:\\AVZ_Package\\RANS.ian", "F:\\AVZ_Package\\RANS.ian"}}
In[2074]:= FileExistsQ["NewBook.doc"]
Out[2074]= True
In[2075]:= {FileExistsQ1["NewBook.doc", h], h}
Out[2075]= {True, {"D:\\NewBook\\NewBook.doc", "G:\\NewBook\\NewBook.doc"}}
    
```

Вызов процедуры **FileExistsQ1**[*x*] с одним фактическим аргументом возвращает **True**, если *x* определяет файл, реально существующий в системе каталогов компьютера, и **False** в противном случае; тогда как вызов **FileExistsQ1**[*x*, *y*] дополнительно через 2-й фактический аргумент возвращает список полных путей к найденному файлу *x*, если основным результатом вызова является **True**. Предыдущий фрагмент представляет и исходный код процедуры **FileExistsQ1**, и типичные примеры ее использования.

Наряду со средствами обработки *внешних* файлов данных система располагает также набором достаточно полезных средств для манипулирования с каталогами как самой *Mathematica*, так и файловой системы **ПК** в целом. Перечислим только некоторые из данных важных функций, а именно:

**DirectoryQ**[*D*] – возвращает **True**, если строка *D* определяет существующий подкаталог, и **False** в противном случае; к сожалению, стандартная процедура при кодировании "/" в конце строки *D* возвращает **False** независимо от существования тестируемого каталога; довольно простая процедура **DirQ** устраняет данный недостаток стандартного средства, а именно:

```

In[2677]:= DirectoryQ["C:/Documents and Settings/Aladjev/My Documents/MyVideos/"]
Out[2677]= False
In[2678]:= DirQ[d_ /; StringQ[d]] := DirectoryQ[StringReplace[d, "/" -> "\\"]]
In[2679]:= DirQ["C:/Documents and Settings/Aladjev/My Documents/MyVideos/"]
Out[2679]= True
In[2680]:= DirQ["C:/Documents and Settings/Aladjev/My Documents/MyVideos\\"]
Out[2680]= True
    
```

**DirectoryName**[*F*] – возвращает путь к каталогу, содержащему файл *F*; более того, в случае, если *F* является реальным подкаталогом, то возвращается цепочка подкаталогов до него; при этом, с учетом файловой концепции, которая отождествляет файлы и подкаталоги, и того обстоятельства, что функция **DirectoryName**[*F*] не учитывает реального существования *F*, такой подход в определенной мере можно было бы считать оправданным, однако при условии учитывания реальности тестируемого пути *F* данный подход вызывает вопросы. Поэтому с этой точки зрения предлагается довольно простая процедура **DirName**, которая возвращает **None**, если *F* – подкаталог, путь к подкаталогу, содержащему файл *F*, и **\$Failed** иначе. Более того, поиск производится в рамках всей файловой системы компьютера, а не в рамках лишь системы подкаталогов, определяемой системной переменной **\$Path**, а именно:

```
In[2267]:= Map[DirectoryName, {"D:/MathMyLib/AVZ_Package.nb", "D:/MathMyLib"}]
Out[2267]= {"D:/MathMyLib", "D:/MathMyLib"}

In[2268]:= DirName[F_]; StringQ[F] := If[DirQ[F], None, If[! FileExistsQ1[F], $Failed,
    Quiet[Check[FileNameJoin[FileNameSplit[F][[1; -2]], None]]]]

In[2269]:= Map[DirName, {"D:/MathMyLib/AVZ_Package.nb", "D:/Math MyLib", "H:/"}]
Out[2269]= {"MathMyLib", None, $Failed}
```

**CreateDirectory[D]** – создает заданный каталог **D** с возвратом пути к нему; это средство в другом контексте уже рассматривалось, однако стандартное средство не работает в случае указания несуществующего устройства внешней памяти (диск, флэшка и др.), поэтому мы и создали достаточно простую процедуру **CDir**, разрешающую данную проблему, а именно: процедура **CDir[D]** создает заданный каталог **D** с возвратом пути к нему; при отсутствии или неактивности устройства внешней памяти каталог создается на устройстве из списка активных устройств внешней памяти текущего сеанса системы, имеющем максимальный объем доступной памяти, с возвратом полного пути к нему:

```
In[2109]:= CDir[d_]; StringQ[d] := Module[{a}, Quiet[If[StringTake[d, {2, 2}] == ":",
    If[MemberQ[a, StringTake[d, 1]], CreateDirectory[d], a = Adrive[];
    CreateDirectory[Sort[a, FreeSpaceVol[#1] >= FreeSpaceVol[#2] &][[1]] <>
    StringTake[d, {2, -1}]]], CreateDirectory[d]]]

In[2110]:= CreateDirectory["H:\\Temp\\GSV\\ArtKr"]
CreateDirectory::nfil: File not found during CreateDirectory .... >>
Out[2110]= $Failed
In[2111]:= CDir["H:\\Temp\\GSV\\ArtKr"]
Out[2111]= "G:\\Temp\\GSV\\ArtKr"
In[2112]:= CDir["G:/Temp/AVZ/Tallinn/IAN/Grodno/Kherson"]
Out[2112]= "D:/Temp/AVZ/Tallinn/IAN/Grodno/Kherson"
```

**CopyDirectory[d1, d2]** – полностью копирует каталог **d1** со всем содержимым в каталог **d2**, однако при наличии принимающего каталога **d2** вызов функции **CopyDirectory** инициирует ошибочную ситуацию с возвратом значения **\$Failed**, что в целом ряде случаев нежелательно. С целью устранения этой ситуации может быть предложена достаточно простая функция **CopyDir**, в целом аналогичная стандартной функции **CopyDirectory**, но с тем отличием, что при наличии принимающего каталога **d2** каталог **d1** копируется в качестве подкаталога **d2** с возвратом полного пути к нему, например:

```
In[2579]:= CopyDirectory["D:/AVZ_Package", "C:/Temp/Temp"]
CopyDirectory::filex: Cannot overwrite existing file C:/Temp/Temp. >>
Out[2579]= $Failed

In[2580]:= CopyDir[d_]; StringQ[d], p_ := CopyDirectory[d, If[DirQ[p],
    p <> "\\ " <> FileNameSplit[d][[1]], p]]

In[2581]:= CopyDir["D:/AVZ_Package", "C:/Temp/Temp"]
Out[2581]= "C:/Temp/Temp/Temp/AVZ_Package"
```

**DeleteDirectory[D]** – удаляет из файловой системы заданный каталог **D** с возвратом **Null**, т.е. ничего, безотносительно к сочетанию атрибутов каталога (*Archive*, *Read-only*, *Hidden*, *System*). Между тем, такой подход, на наш взгляд, не вполне оправдан, полагаясь лишь на то обстоятельство, что пользователь точно уверен, что он удаляет то, что нужно. Тогда как в общем случае должна быть страховка от удаления, например, файлов и каталогов, имеющих такие атрибуты, как *Read-only* (**R**), *Hidden* (**H**) и *System* (**S**). С этой целью, например, можно перед удалением элемента файловой системы предварительно проверять его атрибуты, что обеспечивает достаточно полезная процедура **Attrib**, чей исходный код наряду с примерами применения приводится в нижеследующем фрагменте, а именно:

```
In[2070]:= Attrib[F_;/; StringQ[F], x_;/; ListQ[x] && DeleteDuplicates[Map3[MemberQ,
  {"-A", "-H", "-S", "-R", "+A", "+H", "+S", "+R"}, x]] == {True} | | x == {} | | x == "Attr"] :=
  Module[{a = "$Art24_Kr17$", b = "attrib ", c, d = " > ", h = "attrib.exe", p, f, g},
    If[LoadExtProg["attrib.exe"] === $Failed, Return[$Failed], Null];
    If[StringLength[F] == 3 && DirQ[F] && StringTake[F, {2, 2}] == ":",
      Return["Drive " <> F], If[StringLength[F] == 3 && DirQ[F], f = StandPath[F],
      If[FileExistsQ1[StrDelEnds[F, "\\ ", 2], $Art24$Kr17$], g = $Art24$Kr17$;
        f = StandPath[g[[1]]]; Clear[$Art24$Kr17$];
        Return["<" <> F <> "> is not a directory or a datafile"]]]];
    If[x === "Attr", Run[b <> f <> d <> a],
    If[x === {}, Run[b <> "-A -H -S -R " <> f <> d <> a],
    Run[b <> StringReplace[StringJoin[x], {"+" -> "+", "-" -> "-"}] <> " " <> f <> d <> a]];
    If[FileByteCount[a] == 0, Return[DeleteFile[a]],
      d = Read[a, String]; DeleteFile[Close[a]];
    h = StringSplit[StringTrim[StringTake[d, {1, StringLength[d] - StringLength[f]}]]];
    DeleteFile[$InstallationDirectory <> "\\ " <> "attrib.exe"];
    h = Flatten[h /. {"HR" -> {"H", "R"}, "SH" -> {"S", "H"},
      "SHR" -> {"S", "H", "R"}, "SRH" -> {"S", "R", "H"},
      "HSR" -> {"H", "S", "R"}, "HRS" -> {"H", "R", "S"},
      "RSH" -> {"R", "S", "H"}, "RHS" -> {"R", "H", "S"}}];
    If[h === {"File", "not", "found", "-"} | |
      MemberQ[h, "C:\\Documents", "Drive " <> f, Append[h, g]]]

In[2071]:= Attrib["Cinema Online.txt", "Attr"]
Out[2071]= {"A", "S", "H", "R", {"C:\\Temp\\Cinema Online.txt"}}
In[2072]:= FileExistsQ[$InstallationDirectory <> "\\ " <> "attrib.exe"]
Out[2072]= False
In[2073]:= Attrib["Dereks.doc", "Attr"]
Out[2073]= {"A", {"C:\\Temp\\Dereks.doc"}}
In[2074]:= Attrib["Dereks.doc", {"+A", "+S", "+R"}]
In[2075]:= Attrib["Dereks.doc", "Attr"]
Out[2075]= {"A", "S", "R", {"C:\\Temp\\Dereks.doc"}}
```

```

In[2076]:= Attrb["Dereks.doc", {}]
In[2077]:= Attrb["Dereks.doc", "Attr"]
Out[2077]= {"C:\\Temp\\Dereks.doc"}
In[2078]:= Attrb["Dereks.doc", {"+A"}]
In[2079]:= Attrb["Dereks.doc", "Attr"]
Out[2079]= {"A", {"C:\\Temp\\Dereks.doc"}}
In[2080]:= Attrb["D:\\", "Attr"]
Out[2080]= "Drive D:\\\"
In[2081]:= Attrb["H:\\", "Attr"]
Out[2081]= "<H:\\> is not a directory or a datafile"
In[2082]:= Attrb["RANS.IAN", "Attr"]
Out[2082]= "<RANS.IAN> is not a directory or a datafile"
In[2083]:= Attrb["Dereks.doc", "Attr"]
Out[2083]= {"A", "S", "H", "R", {"C:\\Temp\\Dereks.doc"}}
In[2084]:= Attrb["Dereks.doc", {"-S", "-R", "-H"}]
In[2085]:= Attrb["Dereks.doc", "Attr"]
Out[2085]= {"A", {"C:\\Temp\\Dereks.doc"}}
In[2086]:= Attrb["c:/temp\\", "Attr"]
Out[2086]= {"A", {"C:\\Temp"}}
In[2087]:= Attrb["Dereks.doc", "Attr"]
Out[2087]= {"A", "R", {"C:\\Temp\\Dereks.doc", "F:\\Dereks.doc"}}

```

Успешный вызов процедуры **Attrb**[*F*, "Attr"] возвращает список атрибутов заданного *F*-файла или каталога в разрезе *Archive* ("A"), *Read-only* ("R"), *Hidden* ("H") и *System* ("S"). При этом, возможны и другие атрибуты, присущие системным файлам и каталогам; так, в частности, на *главных* каталогах устройств внешней памяти возвращается "*Drive F*", а на несуществующем каталоге или файле – "*F is not a directory or a datafile*". При этом, вызов возвращается в форме вложенного списка формата {*x*, *y*, ..., *z*, {*a*, *b*, *c*, ..., *f*}}, где элемент-список определяет список *всех* полных путей к файлу или каталогу *F*, т.к. одноименные файлы и подкаталоги могут находиться в различных каталогах, однако обработка атрибутов производится только относительно первого файла/каталога из упомянутого *элемента-списка*. Тогда как в качестве элементов, которые предшествуют данному подписку, выступают атрибуты обрабатываемого файла/каталога. Вызов со вторым аргументом **Attrb**[*F*, {}] возвращает **Null**, т.е. *ничего*, отменяя все атрибуты для обрабатываемого файла/каталога *F*, в то время как вызов **Attrb**[*F*, {"*x*", "*y*", ..., "*z*"}], где *x, y, z* ∈ {"-A", "-H", "-S", "-R", "+A", "+H", "+S", "+R"}, также возвращает **Null**, т.е. *ничего*, устанавливая/отменяя атрибуты обрабатываемого файла/каталога *F*, определяемые вторым аргументом. При невозможности выполнить обработку атрибутов процедура **Attrb**[*F*, *x*] возвращает соответствующие *сообщения*. Данная процедура представляется нам достаточно полезной при работе с файловой системой компьютера. Процедура в отличие от стандартных средств доступа системы позволяет производить обработку и файла, и каталога, расположенных в любом месте файловой системы компьютера. В свою очередь, следующая процедура **Attrb1** во многом подобна процедуре **Attrb**[*F*, *x*]

как в функциональном, так и в описательном отношении, однако **Attrib1** имеет целый ряд отличий. Прежде всего, вызов **Attrib1[F, x]** возвращает *список* атрибутов файла или каталога в строчном формате, которые приписаны каталогу либо файлу *F*, тогда как вызов **Attrib1[F, x, y]** с третьим необязательным аргументом *y* (*произвольное выражение*) также дополнительно удаляет из каталога **\$InstallationDirectory** программный файл *"Attrib.exe"*. Следующий фрагмент представляет исходный код процедуры наряду с наиболее типичными примерами ее использования.

```
In[2228]:= Attrib1[F_ /; StringQ[F], x_ /; ListQ[x] && DeleteDuplicates[Map3[MemberQ,
  {"-A", "-H", "-S", "-R", "+A", "+H", "+S", "+R"}, x]] == {True} | | x == {} | | x == "Attr",
  y___] := Module[{a = "$ArtKr$", b = "attrib ", c, d = "> ", h = "attrib.exe", p, f, g},
  If[LoadExtProg["attrib.exe"] === $Failed, Return[$Failed], Null];
  If[StringLength[F] == 3 && DirQ[F] && StringTake[F, {2, 2}] == ":",
  Return["Drive " <> F], If[StringLength[F] == 3 && DirQ[F], f = StandPath[F],
  If[FileExistsQ1[StrDelEnds[StringReplace[F, "/" -> "\\"], "\\ ", 2], $ArtKr$],
  g = $ArtKr$; f = StandPath[g[[1]]]; Clear[$ArtKr$],
  Return["<" <> F <> "> is not a directory or a datafile"]]]];
  If[x === "Attr", Run[b <> f <> d <> a],
  If[x === {}, Run[b <> "-A -H -S -R " <> f <> d <> a],
  Run[b <> StringReplace[StringJoin[x], {"+" -> "+", "-" -> "-"}] <>
  " " <> f <> d <> a]]; If[FileByteCount[a] == 0, Return[DeleteFile[a],
  d = Read[a, String]; DeleteFile[Close[a]]];
  h = StringSplit[StringTrim[StringTake[d, {1, StringLength[d] - StringLength[f]}]]];
  If[{y} != {}, DeleteFile[$InstallationDirectory <> "\\ " <> "attrib.exe"], Null];
  h = Flatten[h /. {"HR" -> {"H", "R"}, "SH" -> {"S", "H"},
  "SHR" -> {"S", "H", "R"}, "SRH" -> {"S", "R", "H"},
  "HSR" -> {"H", "S", "R"}, "HRS" -> {"H", "R", "S"},
  "RSH" -> {"R", "S", "H"}, "RHS" -> {"R", "H", "S"}}];
  If[h === {"File", "not", "found", "-"} | | MemberQ[h, "C:\\Documents"], "Drive " <> f, h]]
In[2229]:= Mapp[Attrib1, {"C:\\temp\\aaa bbb ccc", "C:\\temp\\V g t u.doc"}, {"A", "R"}]
Out[2229]= {Null, Null}
In[2230]:= Attrib1["C:\\temp\\aaa bbb ccc ", "Attr"]
Out[2230]= {"A", "R"}
```

Обе процедуры существенно использует наши процедуры **LoadExtProg**, **StrDelEnds**, **StandPath**, **FileExistsQ1** и **DirQ** [120], наряду с использованием стандартной функции **Run** и функции *"Attrib.exe"* операционной системы *MS DOS*. При этом, возможность удаления после вызова процедуры **Attrib1** файла *«attrib.exe»* из **\$InstallationDirectory** оставляет файловую систему *Mathematica* неизменной. В реализации обеих процедур весьма существенно использовалась системная функция **Run**, а именно:

*Run[S1, S2, ..., Sn]* – выполняет в базовой операционной среде (например, *MS DOS*) команду,

сформированную из выражений  $S_j$  ( $j=1..n$ ), разделенных символами пробела, с возвратом кода успешности завершения команды в виде целого числа. Как правило, функция **Run** не требует интерактивного ввода, однако на большинстве платформ она генерирует текстовый вывод. В определенной степени функция **Run** аналогична функциям {**system**, **ssystem**} **Maple**. Ниже приводятся примеры применения **Run** для выполнения в среде **Mathematica** команд **DOS**. В затененной области представлено содержимое соответствующего текстового файла. Более того, из второго примера следует, что вызов **Run** с командой **Attrib** завершился с кодом 1.

```
In[2151]:= Run["Dir ", "F:\\", " > ", "C:\\Dir"]
Out[2151]= 0
Volume in drive F is AVZ_ARCHIVE
Volume Serial Number is 3C16-DCF1
Directory of F:\
05/14/2013  08:40 PM  <DIR>      Aladjev_sites
05/14/2013  08:40 PM  <DIR>      ARCHIVE
05/14/2013  08:50 PM  <DIR>      AVZ_Package
05/14/2013  08:51 PM  <DIR>      Kherson
05/04/2013  09:55 AM           44,544 FORTEKS.doc
06/20/2013  10:36 AM           66,202 Application.pdf
06/24/2013  10:37 AM          107,014 Raadiku_17_05.pdf
06/25/2013  02:35 PM           50,826 http.mht
06/30/2013  04:05 PM          115,664 Protocol.pdf
06/06/2013  03:15 PM           31,232 Dereks.doc
05/21/2013  10:56 AM          19,968 Kherson_Stamp.doc
06/30/2013  04:15 PM           51,200 Protocol.doc
07/03/2013  03:18 PM            378 Cinema.txt
          9 File(s)          487,028 bytes
          4 Dir(s) 11,684,929,536 bytes free
In[2152]:= Run["Attrib ", "D:/AVZ_Package\\AVZ_Package.m", " > ", "C:\\Attrib"]
Out[2152]= 1
```

Данный код говорит о неуспешном завершении вызова **Run** по причине отсутствия в каталогах, определенных переменной **\$Path**, утилиты «**attrib.exe**» – внешней команды **MS DOS**. Именно по этой причине в реализации процедуры **Attrib**, представленной выше, использован механизм поиска подкаталогов, содержащих в файловой системе компьютера данную утилиту, с последующим включение их в **\$Path**. По логике вещей это должно было бы решить проблему, однако такой естественный путь ее не решает. Представленные же примеры ее применения вполне корректны при том условии, что утилита находится, в частности, в подкаталоге, определенном **\$InstallationDirectory**, как иллюстрируют *первый* пример фрагмента, тогда как *второй* пример иллюстрирует применение процедуры **Attrib** при том условии, что каталоги с «**attrib.exe**» включены в список каталогов, определяемых переменной **\$Path**, т.е. этот прием решения не дает. Отметим, что использование процедурой функции **Run** иллюстрирует один из очень полезных методов обеспечения интерфейса с базовой операционной платформой, но



здесь имеют место два весьма существенных момента. Прежде всего, данная функция требует внешней реакции пользователя при выходе из среды пакета в операционную среду, о чем говорилось несколько выше при представлении процедуры **FileFormat2**, и во-вторых, вызов посредством функции **Run** функций или системных команд **DOS** предполагает их наличие в системе каталогов, определяемых переменной **\$Path**, ибо в противном случае пакет их не распознает. В частности, такая ситуация имеет место в случае использования внешних команд **DOS**, именно поэтому в реализации **Attrib**, которая через функцию **Run** использует *внешнюю* команду **attrib** системы **DOS**, было обеспечено подключение к системе каталогов **\$Path** каталогов, содержащих утилиту «**attrib.exe**», тогда как для внутренних команд системы **DOS** этого не требуется. Так, при использовании внутренней команды **dir** системы **DOS** не требуется расширения списка каталогов **\$Path**, как иллюстрирует простой пример, представленный выше. В то же время на основе стандартного приема на основе *расширения* списка переменной **\$Path** пакет не распознает внешние команды **DOS**. В данной связи создана достаточно простая процедура, чей успешный вызов **LoadExtProg[x]** возвращает **Null**, т.е. ничего, обеспечивая поиск в файловой системе компьютера программы **x**, заданной полным именем, с последующим ее копированием в подкаталог, определенный переменной **\$InstallationDirectory**. Следующий фрагмент представляет исходный код процедуры **LoadExtProg** с примером ее применения для загрузки в каталог **\$InstallationDirectory** копии внешней команды «**attrib.exe**» системы **MS DOS** с проверкой результата.

```
In[2131]:= LoadExtProg[x_ /; StringQ[x]] := Module[{a = $InstallationDirectory, c, d,
                                                    b = $Art24$Kr17$, d = a <> "\\\" <> x;
                                                    If[! FileExistsQ[d], Clear[$Art24$Kr17$];
                                                    a = FileExistsQ1[x, $Art24$Kr17$]; c = $Art24$Kr17$; $Art24$Kr17$ = b;
                                                    If[a, CopyFile[c[[1]], d]; d, $Failed], d]}

In[2132]:= LoadExtProg["attrib.exe"]
Out[2132]= "C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\attrib.exe"
In[2133]:= FileExistsQ[$InstallationDirectory <> "\\\" <> "attrib.exe"]
Out[2133]= True
In[2134]:= Attrib1["Kherson.doc", "Attr"]
Out[2134]= {"A", "S", "H", "R", {"C:\\Temp\\Book\\Kherson.doc"}}
```

Успешный вызов процедуры **LoadExtProg[x]** отыскивает в файловой системе файл **x** и копирует его в каталог, определяемый переменной **\$InstallationDirectory**, возвращая **\$InstallationDirectory<>"\\"<>x**, если файл уже находился в данном подкаталоге или скопирован в данный каталог. При отсутствии возможности найти искомым файл **x** с программой возвращается **\$Failed**. Следовательно, заранее по вызову **LoadExtProg[x]** можно обеспечить доступ к нужному файлу **x**. Таким образом, используя процедуру **LoadExtProg** в сочетании с системной функцией **Run**, возможно в среде *Mathematica* выполнять целый ряд весьма полезных **{exe|com}**-программ различного назначения, отсутствующих в файловой системе *Mathematica*, тем самым существенно расширяя функциональные возможности программной среды системы *Mathematica*, что вполне может быть востребовано достаточно широким кругом различных приложений.

Итак, декларируемая системой возможность расширения системы каталогов, которая определяется переменной **\$Path**, в общем случае не работает уже для *внешних* команд **DOS**, что иллюстрирует как рассмотрение процедур **Attrib**, **Attrib1** и **LoadExtProg**, так и простой пример с внешней командой **"tlist.exe"**, обеспечивающей отображение всех активных процессов текущего сеанса работы с системой **Windows XP**, а именно:

```
In[2265]:= Run["tlist", ">", "C:\\tlist.txt"]
Out[2265]= 1
In[2266]:= LoadExtProg["tlist.exe"]; Run["tlist", ">", "C:\\tlist.txt"]
Out[2266]= 0
```

```
0 System Process
4 System
488 smss.exe
520 avgchsvx.exe
676 csrss.exe
716 winlogon.exe
760 services.exe
772 lsass.exe
940 ati2evxx.exe
960 svchost.exe
1016 svchost.exe
1092 svchost.exe
1240 svchost.exe
1300 vsmon.exe
1368 ati2evxx.exe
1656 explorer.exe      Program Manager
1680 ctfmon.exe
212 spoolsv.exe
348 svchost.exe
392 avgwdsvc.exe
660 jqs.exe
1168 svchost.exe
1448 MsPMSPSv.exe
1548 AVGIDSAgent.exe
2204 avgnsx.exe
2284 avgemcx.exe
2852 alg.exe
3600 zlclient.exe
3764 avgtray.exe
3824 soundman.exe
3884 vprot.exe
3936 Skype.exe
4056 AVGIDSMonitor.exe
336 LvAgent.exe
3316 AmplusnetPrivacyTools.exe
```

2256 FreeCommander.exe	- FreeCommander
2248 WINWORD.EXE	Kherson_Book - Microsoft Word
4348 avgrsx.exe	
4380 avgcsrvx.exe	
5248 Mathematica.exe	Wolfram Mathematica 9.0 - [Running.AVZ_Package_1.nb]
4760 MathKernel.exe	
4080 javaw.exe	
4780 cmd.exe	C:\WINDOWS\system32\cmd.exe
4808 tlist.exe	

Первый пример предыдущего фрагмента иллюстрирует, попытка посредством **Run** выполнить внешнюю команду **tlist** завершается неудачно (*код возврата 1*), тогда как в результате вызова процедуры **LoadExtProg["tlist.exe"]** производится поиск и загрузка в каталог системы, определяемый переменной **\$InstallationDirectory**, файла **"tlist.exe"**, позволяя успешно выполнить посредством **Run** внешнюю команду **tlist** с сохранением результата ее выполнения в файле, который выделен в тексте затененной областью.

В качестве еще одного довольно полезного примера с использованием **Run** приведем процедуру **SearchFile[F]**, чей вызов возвращает список путей к файлу **F**, найденному в рамках файловой системы компьютера; в случае отсутствия искомого файла **F** вызов процедуры **SearchFile[F]** возвращает пустой список, т.е. **{}**. При этом, **SearchFile** весьма существенно использует рассмотренную выше функцию **Run** системы **Mathematica**.

```

In[2093]:= SearchFile[F_ /; StringQ[F]] := Module[{a, b, f, dir,
                                         h = StringReplace[ToUpperCase[F], "/" -> "\\"],
                                         {a, b, f} = {Map[ToUpperCase[#] <> ":\" &, ADrive[]], {}, "$$$Dir$$$txt"};
                                         dir[y_ /; StringQ[y]] := Module[{a, b, c, v}, Run["Dir " <> "/A/B/S " <> y <> " " <> f];
                                         c = {}; Label[b];
                                         a = StringReplace[ToUpperCase[ToString[v = Read[f, String]]], "/" -> "\\"];
                                         If[a == "ENDOFFILE", Close[f]; DeleteFile[f]; Return[c], If[SuffPref[a, h, 2],
                                         If[FileExistsQ[v], AppendTo[c, v]; Goto[b], Goto[b]]];
                                         For[k = 1, k <= Length[a], k++, b = Append[b, dir[a[[k]]]]; Flatten[b]]
In[2094]:= SearchFile["AVZ_Package.nb"]
Out[2094]= {"C:\\Book\\AVZ_Package.nb", "D:\\Book\\AVZ_Package.nb",
            "D:\\Math_myLib\\AVZ_Package.nb", "G:\\Book\\AVZ_Package.nb"}
In[2095]:= SearchFile["init.m"]
Out[2095]= {"C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica
            \\FrontEnd\\init.m", "C:\\Documents and Settings\\Aladjev\\Application Data
            \\Mathematica\\Kernel\\init.m", ..., "C:\\Program Files\\Wolfram Research\\
            Mathematica\\ 9.0\\SystemFiles\\Links\\XMLSchema\\Kernel\\init.m"}
In[2096]:= Length[%]
Out[2096]= 83
In[2097]:= SearchFile["MapleMathem.doc"]
Out[2097]= {"C:\\Book\\MapleMathem.doc", "D:\\Book\\MapleMathem.doc",

```

```
"G:\\Book\\MapleMathem.doc"}
In[2098]:= SearchFile["new_book.doc"]
Out[2098]= {"D:\\New_Book\\New_Book.doc", "G:\\New_Book\\New_Book.doc"}
In[2099]:= SearchFile["AVZ_AGN_VSV_ART_KR.460"]
Out[2099]= {}
In[2100]:= SearchFile["AVZ_Package_1.nb"]
Out[2100]= {"C:\\AVZ_Package\\AVZ_Package_1.nb", "C:\\RECYCLER\\S-1-5-21-
329068152-1645522239-1644491937-1003\\Dc10\\AVZ_Package_1.nb",
"C:\\RECYCLER\\S-1-5-21-329068152-1645522239-1644491937-1003\\Dc9\\
AVZ_Package_1.nb", "F:\\AVZ_Package\\AVZ_Package_1.nb"}
```

Значения глобальных переменных **\$System**, **\$SystemID** и **\$OperatingSystem** определяют строки, описывающие текущую операционную платформу. Между тем, в целом ряде случаев детализация текущей операционной платформы, представляемая ими, может оказаться недостаточной, в таком случае можно воспользоваться процедурой **Ver**, чей вызов **Ver[]** возвращает строку с уточнением типа операционной платформы. Весьма простой фрагмент представляет исходный код процедуры и пример использования.

```
In[2013]:= {$System, $SystemID, $OperatingSystem}
Out[2013]= {"Microsoft Windows (32-bit)", "Windows", "Windows"}
In[2014]:= Ver[] := Module[{a = "$Art24$Kr17$.txt", b}, Run["Ver > " <> a];
b = Read[a, String]; DeleteFile[Close[a]]; b]
In[2015]:= Ver[]
Out[2015]= "Microsoft Windows XP [Version 5.1.2600]"
```

Перед представлением следующих средств приведем довольно полезную процедуру, вызов которой **Attribs[x]** возвращает список атрибутов в строчном формате, которые приписаны файлу или каталогу **x**. На главных каталогах томов прямого доступа вызов процедуры **Attribs** возвращает **\$Failed**. Следующий фрагмент представляет исходный код процедуры наряду с наиболее типичными примерами ее использования.

```
In[2092]:= Attribs[x_ /; FileExistsQ[x] || DirectoryQ[x]] := Module[{a = StandPath[x], b,
c = "Attrib.exe", d = "$Art$Kr$"},
If[DirQ[x] && StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", $Failed,
If[! FileExistsQ[InstallationDirectory <> "\\ " <> c], LoadExtProg[c];
Run[c <> " " <> a <> " > ", d];
b = Characters[StringReplace[StringTake[Read[d, String],
{1, -StringLength[a] - 1}], " " -> ""]; DeleteFile[Close[d]]; b]]
In[2093]:= Attribs["C:\\Temp\\Avz"]
Out[2093]= {"A", "S", "H", "R"}
In[2094]:= Map[Attribs, {"C:/", "F:\\"}]
Out[2094]= {$Failed, $Failed}
```

```
In[2095]:= Attribs["C:/Temp/Agn/aaa bbb ccc"]
Out[2095]= {"A", "R"}
In[2096]:= Attribs["C:/Temp/Agn/Elisa.pdf"]
Out[2096]= {"R"}
In[2097]:= Attribs["C:/Temp/Agn/Vsv\\V g t u.doc"]
Out[2097]= {"A", "R"}
```

Процедура **Attribs** достаточно реактивна, дополняя процедуры **Attrib** и **Attrib1**. При этом, противоположный по смыслу успешный вызов процедуры **DelAllAttribs[x]** для файла *x* обеспечивает удаление *всех* приписанных ему атрибутов, возвращая **0**. Эти 2 процедуры достаточно эффективно применяются в программировании ряда средств доступа к *элементам* файловой системы компьютера, обрабатывая их атрибуты. Ниже представлен исходный код процедуры **DelAllAttribs** с примерами ее использования.

```
In[2039]:= DelAllAttribs[x_;/; FileExistsQ[x] || DirectoryQ[x]] := Module[{c = "Attrib.exe",
                                                                    a = StandPath[x], b},
    If[DirQ[x] && StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", $Failed,
    If[! FileExistsQ[$InstallationDirectory <> "\\\" <> c], LoadExtProg[c]];
    Run[c <> " -A -H -R -S " <> a]]]

In[2040]:= Attribs["C:\\Temp/Agn/Vsv/V g t u.doc"]
Out[2040]= {"A", "S", "H", "R"}
In[2041]:= DelAllAttribs["C:\\Temp/Agn/Vsv/V g t u.doc"]
Out[2041]= 0
In[2042]:= Attribs["C:\\Temp/Agn/Vsv/V g t u.doc"]
Out[2042]= {}
In[2043]:= Attribs["C:\\Temp"]
Out[2043]= {"A", "R"}
In[2044]:= DelAllAttribs["C:\\Temp"]
Out[2044]= 0
In[2045]:= Attribs["C:\\Temp"]
Out[2045]= {}
```

Следует отметить, что система *Mathematica* не располагает стандартными средствами обработки атрибутов файлов и каталогов и предложенные нами процедуры **Attrib1** и **Attrib** вместе с **Attribs** и **DelAllAttribs** в определенной мере заполняют данную нишу.

В отличие от системных функций **DeleteFile** и **DeleteDirectory** следующая процедура **DelDirFile** удаляет из файловой системы компьютера файл или каталог *x*, возвращая **Null**, т.е. ничего. Более того, вызов процедуры **DelDirFile[x]** с одним аргументом есть не что иное, как вызов **DeleteFile[x]** или **DeleteDirectory[x]** в зависимости от того, что определяет аргумент *x* – файл данных или каталог. Тогда как вызов **DelDirFile[x, y]** со вторым необязательным аргументом *y* – произвольным выражением – удаляет файл или каталог даже, если файл данных *x* либо элементы каталога *x* файловой системы компьютера имеют атрибут *Read-only*; в таком случае перед его удалением атрибуты

элемента  $x$  отменяются, обеспечивая корректное удаление элемента  $x$ , что в отличие от системных средств расширяет возможности по удалению элементов файловой системы компьютера. Следующий фрагмент приводит исходный код процедуры **DelDirFile** с примерами ее наиболее типичного использования.

```
In[2049]:= DelDirFile[x_;/StringQ[x] && DirQ[x] || FileExistsQ[x], y___] := Module[{c, f,
    a = {}, b = "", k = 1},
  If[DirQ[x] && If[StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", False, True],
    If[{y} === {}, Quiet[DeleteDirectory[x, DeleteContents -> True]], a = {}; b = "";
      c = StandPath[x]; f = "$Art78Kr$";
      Run["Dir " <> c <> " /A/B/OG/S > " <> f]; DelAllAttribs[c];
      For[k, k < Infinity, k++, b = Read[f, String];
        If[SameQ[b, EndOfFile], DeleteFile[Close[f]]; Break[],
          DelAllAttribs[b]]]; DeleteDirectory[x, DeleteContents -> True]],
    If[FileExistsQ[x], If[{y} != {}, DelAllAttribs[x]]; Quiet[DeleteFile[x], $Failed]]]

In[2050]:= DelDirFile["F:\\"]
Out[2050]= $Failed
In[2052]:= DelDirFile["C:\\Temp\\System.txt"]
Out[2052]= $Failed
In[2053]:= DelDirFile["C:\\Temp\\System.txt", 78]
In[2054]:= DelDirFile["C:\\Temp\\Agn"]
Out[2054]= $Failed
In[2055]:= DelDirFile["C:\\Temp\\Agn", 78]
```

Между тем, перед представлением следующего средства необходимо определить одну достаточно полезную процедуру, суть назначения которой состоит в следующем. Как уже отмечалось выше, спецификатор файла зависим как от регистра символов, так и от используемых разделителей каталогов. В частности, один и тот же файл данных с разными спецификаторами "C:\\Temp\\agn\\cinema.txt" и "C:/temp/agn/cinema.txt" открывается по двум *различным* потокам (*Streams*). По этой причине последующее его закрытие посредством стандартной функции **Close** не закрывает файл "cinema.txt", требуя закрытия *всех* потоков, по которым он был ранее открыт. Для решения данной проблемы определена процедура **Close1**, представленная следующим фрагментом.

```
In[2090]:= Streams[]
Out[2090]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[2091]:= Read["C:/Temp\\agn\\cinema.txt"]; Read["C:/Temp/Agn\\Cinema.txt"];
  Read["C:/Temp\\agn/cinema.txt"]; Read["C:/temp/agn\\Burthday.doc"];
  Read["C:/temp\\agn\\BurthDay.doc"];
In[2092]:= Streams[]
Out[2092]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
  InputStream["C:/Temp\\agn\\cinema.txt", 328],
  InputStream["C:/Temp/Agn\\Cinema.txt", 331],
```

```

        InputStream["C:/Temp\\agn/cinema.txt", 334],
        InputStream["C:/temp/agn\\Burthday.doc", 339],
        InputStream["C:/temp\\agn\\BurthDay.doc", 340]]
In[2093]:= Close["C:/Temp\\agn/cinema.txt"]
Out[2093]= "C:/Temp\\agn/cinema.txt"
In[2094]:= Streams[]
Out[2094]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
        InputStream["C:/Temp\\agn\\cinema.txt", 328],
        InputStream["C:/Temp/Ag\\Cinema.txt", 331],
        InputStream["C:/temp/agn\\Burthday.doc", 339],
        InputStream["C:/temp\\agn\\BurthDay.doc", 340]]
In[2095]:= Close1[x___String] := Module[{a = Streams[[[3 ;; -1]], b = {x}, c = {}, k = 1, j],
        If[a == {} || b == {}, {}, b = Select[{x}, FileExistsQ[#] &];
        While[k <= Length[a], j = 1; While[j <= Length[b],
        If[ToUpperCase[StringReplace[a[[k]][[1]], {"\\" -> "", "/" -> ""}]] ==
        ToUpperCase[StringReplace[b[[j]], {"\\" -> "", "/" -> ""}]]],
        AppendTo[c, a[[k]]]; j++; k++]; Map[Close, c]; If[Length[b] == 1, b[[1]], b]]
In[2096]:= Close1["C:/Temp\\agn/cinema.txt", "C:/temp\\agn\\BurthDay.doc"]
Out[2096]= {"C:/Temp\\agn/cinema.txt", "C:/temp\\agn\\BurthDay.doc"}
In[2097]:= Streams[]
Out[2097]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[2098]:= Close1[]
Out[2098]= {}
In[2099]:= Close1["C:/Temp\\agn/cinema.txt", "C:/temp\\agn\\BurthDay.doc"]
Out[2099]= {}
In[2100]:= Close1["C:/Temp\\Agn/Cinema.txt", AvzAgnVsvArtKr]
Out[2100]= Close1["C:/Temp\\Agn/Cinema.txt", AvzAgnVsvArtKr]
In[2101]:= Closes[x_] := Quiet[Check[Close[x], Null]]
In[2102]:= Closes["C:\\Temp\\Svetlana\\Kherson\\Cinema.txt"]

```

Вызов процедуры **Close1**[*x*, *y*, *z*, ...] закрывает все файлы из реально существующих в списке {*x*, *y*, *z*, ...} независимо от количества потоков, по которым они были открыты различными спецификаторами, возвращая их список. В остальных случаях вызов на допустимых фактических аргументах возвращает пустой список, т.е. {}, тогда как на недопустимых фактических аргументах возвращается невычисленным. Предыдущий фрагмент представляет исходный код процедуры **Close1** с примерами применения. В завершении фрагмента представлена простая функция, чей вызов **Closes**[*x*] ничего не возвращает, закрывая файл данных *x*, включая закрытые и пустые файлы, без вывода каких-либо ошибочных сообщений. В ряде приложений функция довольно полезна.

Следующая процедура **DelDirFile1** представляет собой весьма полезное расширение процедуры **DelDirFile** на случай открытых файлов дополнительно к атрибуту *Read-*

*only* как отдельных, так и находящихся в удаляемом каталоге. Вызов **DelDirFile1**[x] в полной мере эквивалентен вызову **DelDirFile1**[x, y], обеспечивая удаление файла или каталога независимо от открытости отдельного файла и приписанного ему атрибута *Read-only*, или наличия подобных файлов данных в каталоге. Следующий фрагмент представляет исходный код процедуры с типичными примерами ее использования.

```
In[2025]:= DelDirFile1[x_/: StringQ[x] && FileExistsQ[x] || DirQ[x] &&
    If[StringLength[x] == 3 && StringTake[x, {2, 2}] == ":", False, True]] :=
    Module[{a = {}, b = "", c = StandPath[x], d, f = "$Art460Kr$", k = 1},
    If[DirQ[x], Run["Dir " <> c <> "/A/B/OG/S > " <> f]; DelAllAttribs[c];
    For[k, k < Infinity, k++, b = Read[f, String]; If[SameQ[b, EndOfFile],
        DeleteFile[Close[f]]; Break[], DelAllAttribs[b]; Close1[b]];
    DeleteDirectory[x, DeleteContents -> True], Close1[x];
    DelAllAttribs[x]; DeleteFile[x]]]

In[2026]:= Map[Attribs, {"C:/Temp\\Agn/Cinema.txt", "C:/Temp\\Agn/BurthDay.doc",
    "C:/Temp\\Agn/page_2.doc"}]

Out[2026]= {{{"A", "S", "H", "R"}, {"A", "R"}, {"A", "R"}}}

In[2027]:= Read["C:/Temp\\Agn/Cinema.txt"]; Read["C:/Temp\\Agn/BurthDay.doc"];
    Read["C:/Temp\\Agn/page_2.doc"];

In[2028]:= Streams[]
Out[2028]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
    InputStream["C:/Temp\\Agn/Cinema.txt", 97],
    InputStream["C:/Temp\\Agn/BurthDay.doc", 101],
    InputStream["C:/Temp\\Agn/page_2.doc", 102]}

In[2029]:= DelDirFile1["C:/Temp\\Agn"]
In[2030]:= Streams[]
Out[2030]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[2031]:= DirQ["C:/Temp\\Agn"]
Out[2031]= False
In[2032]:= Read["C:/cinema.TXT"];
In[2033]:= Streams[]
Out[2033]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
    InputStream["C:/cinema.TXT", 104]}
In[2034]:= DelDirFile1["C:\\Cinema.txt"]
In[2035]:= FileExistsQ["C:\\Cinema.txt"]
Out[2035]= False
```

Следующие 2 процедуры носят общий характер при работе с устройствами прямого доступа и довольно полезны в ряде приложений, прежде всего, системного характера. Следующая процедура в значительной степени является аналогом *Maple*-процедуры *Vol\_Free\_Space*, которая возвращает объем свободной памяти на устройствах прямого доступа. Вызов процедуры **FreeSpaceVol**[x] возвращает 2-элементный список, первый



элемент которого определяет *объем* и второй единицу измерения свободного места на томе прямого доступа, определенного его *именем* в строчном формате "x". Отсутствие или неактивность устройства x идентифицируется возвратом сообщения "Drive <x> is not ready". Фрагмент представляет исходный код процедуры и примеры применения.

```
In[2015]:= FreeSpaceVol[x_ /; MemberQ[Join[CharacterRange["a", "z"],
      CharacterRange["A", "Z"]], x]] := Module[{a = "$Art$Kr$", b},
      b = Run["Dir " <> x <> ":\\ " <> " " <> " <> a];
      If[b != 0, DeleteFile[a]; Return["Drive <" <> x <> "> is not ready"],
      b = ReadList[a, String][[-1]]; DeleteFile[a];
      b = StringSplit[b][[-3 ;; -1]];
      ToExpression[{StringReplace[b[[1]], {" " -> "", "." -> ""}], b[[2]]}]

In[2016]:= Map[FreeSpaceVol, {"C", "F", "d", "A", "E"}]
Out[2016]= {{138950545408, bytes}, {11698929664, bytes}, "Drive <d> is not ready",
      "Drive <A> is not ready", "Drive <E> is not ready"}
```

Следующий фрагмент представляет процедуру, чей вызов **BootDrive[]** возвращает 2-элементный список, первый элемент которого определяет имя устройства начальной загрузки операционной системы, тогда как второй элемент определяет тип системы. Фрагмент представляет исходный код процедуры с примерами ее использования.

```
In[2030]:= BootDrive[] := Module[{b, c, d = "", h = {}, t},
      a = Select[ADrive[], FileExistsQ[# <> ":\\ " <> "Boot.ini"] &]],
      If[a == {}, Return[$Failed], b = a[[1]] <> ":\\ " <> "Boot.ini"];
      Label[avz]; c = Read[b, String];
      If[c === EndOfFile, Close[b]; Return[a[[1]], If[Length[h] == 1, h[[1]], h]],
      If[SuffPref[c, "default=", 1], d = "\\ " <>
      StrDelEnds[StringTake[c, {StringPosition[c, "\\ "][[1]][[1]] + 1, -1}], " ", 2] <> "=";
      Goto[avz], If[d == "" | | StringFreeQ[c, {d, "\\WINDOWS="}], Goto[avz],
      t = Flatten[StringPosition[c, DeleteDuplicates[{d, "\\WINDOWS=", "/" }]]];
      t = ToExpression[StringTake[c, {t[[2]] + 1, If[Length[t] == 2, 0, t[[3]] - 1}]];
      h = Append[h, t]; Goto[avz]]]]

In[2031]:= BootDrive[]
Out[2031]= {"C", "Microsoft Windows XP Professional"}
In[1942]:= BootDrive[]
Out[1942]= {"C", {"Microsoft Windows 98 rus", "Microsoft Windows 98 eng",
      "Microsoft Windows XP Professional", "Windows 2000 Professional"}}
```

В отсутствие (повреждение системы либо система старше **Windows XP**) файла «boot.ini» инициализации загрузки операционной среды вызов процедуры возвращает **\$Failed**. Процедура **BootDrive** успешно функционирует в среде **Mathematica** релизов 7 ÷ 9 на платформе **Windows {2000 | 2003 | NT | XP}** с несколькими операционными средами. В предыдущем примере она применяется в одной операционной среде и в нескольких.

Завершают данную главу средства, представляющие вполне определенный интерес в работе с *файловой* системой компьютера, как самостоятельных, так и в составе средств обработки файлов и каталогов. Используются они и средствами *AVZ\_Package\_1* [120]. В ряде случаев при работе с файлами данных может оказаться достаточно полезной процедура **OpenFiles[]**, чей вызов возвращает вложенный список из двух подсписков, первый из которых, идентифицируемый первым элементом *"read"*, содержит полные пути к файлам, открытым на чтение, тогда как второй, идентифицируемый первым элементом *"write"*, содержит полные пути к файлам, открытым на запись в текущем сеансе. В отсутствие таких файлов вызов процедуры возвращает пустой список, т.е. {}. Следующий фрагмент представляет исходный код процедуры **OpenFiles[]** наряду с достаточно типичными примерами ее использования.

```
In[2738]:= OpenRead["C:/Temp/Attention.doc"]; Read["C:/Temp/Attention!.doc", Word];
Write["D:/Art.24", S]; Write["C:/Temp/Agn.66", "TS"]; OpenRead["C:/Temp/Fultus.doc"];
In[2739]:= OpenFiles[] := Module[{a = Map[ToString1, StreamsU[]], d, h, k = 1, b = {"read"},
    c = {"write"}}, If[a == {}, Return[{}], For[k, k <= Length[a], k++, d = a[[k]];
    h = DeleteDuplicates[Flatten[StringPosition[d, {"[", ", "}]]];
    If[StringFreeQ[d, "InputStream["],
    c = Append[c, StringTake[d, {h[[1]] + 2, h[[-1]] - 2}]],
    b = Append[b, StringTake[d, {h[[1]] + 2, h[[-1]] - 2}]]]; {b, c}]
In[2740]:= OpenFiles[]
Out[2740]= {"read", "C:/Temp/Attention.doc", "C:/Temp/Fultus.doc", {"write", "D:/Art.24",
    "C:/Temp/Agn.66", "C:/Academy/RANS/IAN/Tallinn.txt"}}
In[2741]:= {CloseAll[], OpenFiles[]}
Out[2741]= {"C:/Temp/Attention.doc", "C:/Temp/Fultus.doc", "D:/Art.24",
    "C:/Temp/Agn.66", "C:/Academy/RANS/IAN/Tallinn.txt"}, {}
```

При этом, под *полным* путем понимается либо реально полный путь к файлу данных в файловой системе компьютера, либо его *полное* имя, если он находится в текущем каталоге файловой системы, определяемом вызовом системной функции **Directory[]**.

```
In[2241]:= StreamFiles[] := Module[{a = Map[ToString1, StreamsU[]], b = {}, w = {"out"},
    r = {"in"}, c, k = 1}, If[a == {}, Return["AllFilesClosed"],
    For[k, k <= Length[a], k++, c = a[[k]];
    If[SuffPref[c, "Out", 1], w = Append[w, StrFromStr[c]],
    r = Append[r, StrFromStr[c]]]; Map[Flatten, {r, w}]]
In[2242]:= Read["C:/Temp\\dereks.doc"]; Mapp[Write, {"RANS", "D:/Agn"}, "Academy"];
In[2243]:= StreamFiles[]
Out[2243]= {"in", "C:/Temp\\dereks.doc", {"out", "RANS", "D:/Agn"}}
In[2244]:= CloseAll[]; StreamFiles[]
Out[2244]= "AllFilesClosed"
```

В качестве процедуры, подобной **OpenFiles[]**, можно использовать процедуру, вызов которой **StreamFiles[]** возвращает вложенный список из двух подсписков, первый из которых, идентифицируемый *первым* элементом *"in"*, содержит полные пути/имена к файлам данных, открытым на *чтение*, тогда как второй, идентифицируемый *первым* элементом *"out"*, содержит полные пути/имена к файлам данных, открытым на *запись* в данный момент. При отсутствии открытых файлов вызов **StreamFiles[]** возвращает значение *"AllFilesClosed"*. В предыдущем фрагменте представлены как исходный код процедуры **OpenFiles[]**, так и некоторые типичные примеры ее использования.

В целом ряде случаев при работе с файлами данных может оказаться весьма полезной процедура, вызов которой **IsFileOpen[F]** возвращает *True*, если файл *F*, определенный своим именем или полным путем, открыт, и *False* в противном случае. Если аргумент *F* не определяет существующий файл, то вызов возвращается невычисленным. Тогда как вызов **IsFileOpen[F, h]** со вторым необязательным аргументом *h* – неопределенной переменной – возвращает через *h* вложенный список, элементами которого являются подписки формата *{{"read" | "write"}, {Список потоков, по которым файл открыт на чтение/запись}}*, если основной результат – *True*. Следующий фрагмент приводит исходный код процедуры **IsFileOpen** наряду с примерами ее использования.

```
In[2038]:= IsFileOpen[F_;/ FileExistsQ[F], h_] := Module[{a = Streams[[[3 ;; -1]], b, c, d,
    x = {}, y = {}], If[a == {}, False, d = StringReplace[ToUpperCase[F], "\\\" -> "/"];
    Map[If[Head[#] === InputStream, AppendTo[x, #[[1]]], AppendTo[y, #[[1]]]] &, a];
    b = Select[x, d == ToUpperCase[StringReplace[#, "\\\" -> "/"]] &];
    c = Select[y, d == ToUpperCase[StringReplace[#, "\\\" -> "/"]] &];
    If[b == {} && c == {}, False, If[{h} != {} && ! HowAct[h],
    h = If[b != {} && c != {}, {"read", b}, {"write", c}],
    If[b != {}, {"read", b}, {"write", c}]]]; True]]]

In[2039]:= OpenWrite["C:/temp/cinema.doc"]; OpenRead["C:/temp/cinema.doc"];
In[2040]:= Streams[]
Out[2040]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
    OutputStream["C:/temp/cinema.doc", 101],
    InputStream["C:/temp/cinema.doc", 102]}
In[2041]:= IsFileOpen["C:/Temp\\Cinema.doc", t78]
Out[2041]= True
In[2042]:= t78
Out[2042]= {"read", {"C:/temp/cinema.doc"}}, {"write", {"C:/temp/cinema.doc"}}}
In[2043]:= Read["C:/temp/burthday.doc"];
In[2044]:= IsFileOpen["C:\\temp\\BurthDay.doc", h460]
Out[2044]= True
In[2045]:= h460
Out[2045]= {"read", {"C:/temp/burthday.doc"}}
In[2046]:= IsFileOpen["C:\\temp\\BurthDay.doc"]
Out[2046]= True
```

```
In[2047]:= CloseAll[]; IsFileOpen["C:\\temp\\BurthDay.doc"]
Out[2047]= False
```

При работе с файловой системой компьютера достаточно полезной представляется и довольно простая процедура, чей вызов **DirEmptyQ[d]** возвращает **True**, если каталог *d* пуст, в противном случае возвращается **False**. Вызов возвращается невычисленным, если *d* не является *реальным* каталогом. Следующий фрагмент представляет исходный код процедуры **DirEmptyQ** с достаточно типичными примерами ее использования.

```
In[2279]:= DirEmptyQ[d_ /; DirQ[d]] := Module[{a = "$DirFile$", b, c, h = " 0 File(s) ",
      d1 = StandPath[d]}, b = Run["Dir " <> d1 <> "\\*. *" <> a];
      If[b != 0, Return[$Failed], Do[c = Read[a, String], {6}]];
      DeleteFile[Close[a]]; ! StringFreeQ[c, h]]
In[2280]:= Map[DirEmptyQ, {"C:\\Academy", "C:\\temp", "D:/AVZ_Package", "Rans"}]
Out[2280]= {True, False, False, DirEmptyQ["Rans"]}
```

В дополнение к предыдущей процедуре **DirEmptyQ**, следующая процедура **DirFD[d]** обеспечивает возврат вложенного списка, *первый* элемент которого определяет список подкаталогов *первого* уровня каталога *d*, тогда как *второй* – список файлов каталога *d*.

```
In[2233]:= DirFD[d_ /; DirQ[d]] := Block[{a = "$DirFile$", b, c, h, g = " File(s) ", t, k = 1,
      d1 = StandPath[d]}, If[DirEmptyQ[d1], Return[{}], b = {}, {}];
      c = Run["Dir " <> d1 <> "\\*. *" <> a];
      If[c != 0, Return[$Failed], Do[c = Read[a, String], {5}]];
      For[k, k <= Infinity, k++, c = Read[a, String];
      If[! StringFreeQ[c, g], DeleteFile[Close[a]]; Return[b],
      h = StringSplit[c]; t = StringTake[c, {Flatten[StringPosition[c, h[[4]]][[-1]] + 1, -1}];
      If[MemberQ[h, "<DIR>"], b[[1]] = Append[b[[1]], StringTrim[t]],
      b[[2]] = Append[b[[2]], StringTrim[t]]]]]]
In[2234]:= DirFD["C:\\Academy"]
Out[2234]= {"Addition", "Ian", "Rans"}, {"CoreLanguage.pdf", "GrSu.doc", "N_Book.doc"}
In[2235]:= Map[DirFD, {"C:\\RANS", "IAN"}]
Out[2235]= {}, DirFD["IAN"]
In[2236]:= DirFD["C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\
      Documentation\\English\\Packages"]
Out[2236]= {"ANOVA", "Audio", "AuthorTools", "BarCharts", "Benchmarking",
      "BlackBodyRadiation", "Calendar", "Combinatorica", "Compatibility", "FiniteFields",
      "ComputationalGeometry", "ComputerArithmetic", "Developer", "EquationTrekker",
      "ErrorBarPlots", "Experimental", "FourierSeries", "FunctionApproximations", "Geodesy",
      "GraphUtilities", "HierarchicalClustering", "Histograms", "HypothesisTesting",
      "LinearRegression", "MultivariateStatistics", "Music", "NonlinearRegression", "Notation",
      "NumericalCalculus", "NumericalDifferentialEquationAnalysis", "PhysicalConstants",
      "PieCharts", "PlotLegends", "PolyhedronOperations", "Polytopes", "PrimalityProving",
```

```
"Quaternions", "RegressionCommon", "ResonanceAbsorptionLines", "Splines",
"StandardAtmosphere", "StatisticalPlots", "Units", "VariationalMethods",
"VectorAnalysis", "VectorFieldPlots", "WorldPlot", "XML"}, {}
In[2237]:= Length[%[[1]]]
Out[2237]= 48
```

Если каталог *d* пуст, вызов процедуры возвращает пустой список, т.е. {}. Предыдущий фрагмент представляет исходный код процедуры с примерами ее применения. Так, в заключение предыдущего фрагмента возвращается список каталогов с *документацией* по пакетам, поставляемым с системой *Mathematica 9.0*. Таким образом, с релизом *9.0.1* системы *Mathematica* поставляется 48 пакетов различного назначения, которое весьма несложно усматривается из самого названия содержащих их подкаталогов.

В дополнение к процедуре **DirFD** вызов процедуры **TypeFilesD[d]** возвращает список типов файлов, расположенных в каталоге *d*, на файлах данных без *расширения* имени, возвращая значение "undefined". Более того, рассматриваются файлы, находящиеся в самом каталоге *d* и во всех его подкаталогах любого уровня вложенности. На пустом каталоге *d* и его подкаталогах вызов **TypeFilesD[d]** возвращает пустой список, т.е. {}. Следующий фрагмент представляет исходный код процедуры **TypeFilesD** наряду с достаточно типичными примерами ее использования.

```
In[2752]:= TypeFilesD[d_;/; DirQ[d]] := Module[{a = {}, b, c, h = StandPath[d], k = 1,
      f = "$Art$$Kr$"}, Run["Dir " <> h <> "/A/B/OG/S " <> f];
      For[k, k < Infinity, k++, b = Read[f, String];
      If[SameQ[b, EndOfFile], DeleteFile[Close[f]]; Break[]],
      If[! DirQ[b], AppendTo[a, c = FileExtension[b];
      If[! SameQ[c, ""], c, "undefined"]]]]; DeleteDuplicates[a]]

In[2753]:= TypeFilesD["C:\\Program Files\\Maple 11\\BIN.WIN"]
Out[2753]= {"exe", "ico", "dll", "lib", "bat", "lic", "ini", "properties", "undefined"}
In[2754]:= TypeFilesD["C:/Tallinn\\Grodno/Kherson"]
Out[2754]= {}
In[2755]:= TypeFilesD["C:/Archive"]
Out[2755]= {"jpg", "mht", "htm", "doc", "png", "pdf", "zip", "JPG", "html", "ZIP", "rtf", "avi",
"AVI", "HTM", "txt", "exe", "cab", "hdr", "bin", "ex_", "INI", "dll", "ini", "inx",
"sys", "dat", "msi", "vxd", "dft", "enu", "ibt", "iss", "mst", "DLL", "bmp", "cat",
"inf", "dl_", "sy_", "xml", "ca_", "CNT", "GID", "HLP", "EXE", "DAT", "AFE",
"BIN", "CAT", "INF", "TXT", "SYS"}
```

Дополнительно к процедуре **DirFD** определенный интерес представляет и процедура **DirFull** с исходным кодом, представленным нижеследующим фрагментом, а именно.

```
In[2250]:= DirFull[x_;/; DirQ[x]] := Module[{a = "$Art6Kr$", b = StandPath[x], c = {}, d = ""},
      Run["Dir /S/B/A ", b, " > ", a];
      While[! SameQ[d, EndOfFile], c = Append[c, d = Read[a, String]];
      DeleteFile[Close[a]]; Flatten[{x, c[[1 ;; -2]]}]]
```

```
In[2251]:= DirFull["C:\\Temp\\avz agn vsv"]
Out[2251]= "C:\\Temp\\avz agn vsv", "c:\\temp\\avz agn vsv\\Art Kr 71", "c:\\temp\\avz agn vsv\\Art Kr 71\\GRSU.doc", "c:\\temp\\avz agn vsv\\Art Kr 71\\Rans Ian", "c:\\temp\\avz agn vsv\\Art Kr 71\\Rans Ian\\Vgtu.doc"
In[2252]:= DirFull["C:\\Temp\\Aaa Bbb Ccc"]
Out[2252]= {"C:\\Temp\\Aaa Bbb Ccc"}
```

Вызов процедуры **DirFull**[*d*] возвращает список всех полных путей к подкаталогам и файлам, содержащимся в каталоге *d*; при этом, первый элемент данного списка – сам каталог *d*. Предыдущий фрагмент представляет **DirFull** с примерами ее применения.

Процедура **FindFile1** служит полезным расширением стандартной функции **FindFile**, обеспечивая поиск файла в рамках файловой системы компьютера. Вызов процедуры **FindFile1**[*x*] возвращает *полный* путь к найденному файлу *x* или список полных путей (если файл *x* находится в разных каталогах файловой системы компьютера), в противном случае вызов возвращает пустой список, т.е. {}. Фрагмент представляет исходный код процедуры **FindFile1** наряду с примерами ее достаточно типичного использования.

```
In[2130]:= FindFile1[x_ /; StringQ[x]] := Module[{a = Adrive[], c, d = {}, k = 1, j, p,
    b = "\\\" <> ToUpperCase[x]}, For[k, k <= Length[a], k++, p = a[[k]];
    c = Quiet[FileNames["*", p <> "\\\", Infinity]]; For[j = 1, j <= Length[c], j++, p = c[[j]];
    If[SuffPref[ToUpperCase[p], b, 2], d = Append[d, p]; Continue[], Continue[]];
    If[Length[d] == 1, d[[1]], d]]
In[2131]:= FindFile1["Letter_11_02_13.doc"]
Out[2131]= {"C:\\Temp\\Letter_11_02_13.doc", "F:\\Letter_11_02_13.doc"}
In[2132]:= FindFile1["New_Book.doc"]
Out[2132]= {"C:\\New_Book\\New_Book.doc", "D:\\New_Book\\New_Book.doc",
    "G:\\New_Book\\New_Book.doc"}
In[2133]:= Map[FindFile1, {"Rans.Ian", "Cinema.txt"}]
Out[2133]= {{}, "C:\\Temp\\Cinema.txt"}
In[2134]:= t = TimeUsed[]; FindFile1["Book_3.doc"]; TimeUsed[] - t
Out[2134]= 6.453
In[2135]:= t = TimeUsed[]; FileExistsQ1["Book_3.doc"]; TimeUsed[] - t
Out[2135]= 7.312
```

Как показывают 2 последних примера фрагмента, процедура **FileExistsQ1**, во многом функционально подобная процедуре **FindFile1**, во временном отношении является в некоторой степени менее реактивной в аналогичной файловой системе компьютера.

В качестве еще одного довольно показательного примера можно привести процедуру **SearchDir**[*D*], чей вызов возвращает список всех путей в файловой системе, которые в качестве подкаталога содержат *D*; в случае отсутствия таких путей вызов процедуры **SearchDir**[*D*] возвращает пустой список. Процедура содержит две процедуры уровня вложенности 2 и существенно использует рассмотренную *Run*-функцию. В сочетании с процедурой **SearchFile** процедура **SearchDir** полезна в работе с файловой системой компьютера, что и подтверждает их применение для решения задач подобного типа.

```

In[2360]:= SearchDir[D_;/ StringQ[D]] := Module[{a, b, c, d, f, h, dir, k},
  {a, b, f} = {Map[ToUpperCase[#] <> ":\\" &, ADrive[]], {}, "$$$17Dir24$$$$.txt"};
  dir[y_;/ StringQ[y]] := Module[{a, b, c, Q}, Run["Dir " <> "/B/S " <> y <> " " <> f];
  Q[x_;/ StringQ[x]] := If[! FileExistsQ[x], x, StringTake[x, {1, DeleteDuplicates[
    Flatten[StringPosition[x, "\\"]][[-1] - 1]]]; c = {}; Label[b]; a = Read[f, String];
    If[a === EndOfFile, Close[f]; DeleteFile[f]; Return[DeleteDuplicates[c]],
      h = ToUpperCase[a]; d = ToUpperCase[D];
      If[StringPosition[h, "\\"] <> d <> "\\"] != {} ||
      StringEnd[h, "\\"] <> d, c = Append[c, Q[a]]; Goto[b], Goto[b]]];
  For[k = 1, k <= Length[a], k++, b = Append[b, dir[a[[k]]]];
  b = Flatten[b]; Select[b, ! MemberQ[a, # <> "\\"] &]]

In[2361]:= SearchDir["book"]
Out[2361]= {"C:\\Book", "C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\
  SystemFiles\\FrontEnd\\StyleSheets", "C:\\Program Files\\Wolfram Research\\
  Mathematica\\9.0\\SystemFiles\\FrontEnd\\StyleSheets\\Book", "D:\\Book",
  "G:\\Book", "G:\\Book\\Fonts", "G:\\Book\\Fonts\\Type1", "G:\\Book\\
  Fonts\\TrueType", "G:\\Book\\Fonts\\SVG"}

In[2362]:= SearchDir["Old Grodno"]
Out[2362]= {"C:\\ARCHIVE", "C:\\ARCHIVE\\OLD GRODNO", "G:\\ARCHIVE",
  "G:\\ARCHIVE\\OLD GRODNO", "G:\\RANS_IAN_2012",
  "G:\\RANS_IAN_2012\\OLD GRODNO"}

In[2363]:= SearchDir["Photo"]
Out[2363]= {"C:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS",
  "C:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS\\PHOTO",
  "G:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS",
  "G:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS\\PHOTO"}

In[2364]:= SearchDir["UserLib"]
Out[2364]= {"C:\\Program Files\\Maple 11\\lib", "C:\\Program Files\\Maple 11\\
  lib\\userlib", "D:\\USERLIB6789\\USERLIB10", "D:\\USERLIB6789\\
  =====
  "G:\\USERLIB6789\\USERLIB11\\userlib", "G:\\USERLIB6789\\USERLIB10",
  "G:\\USERLIB6789\\USERLIB10\\userlib"}

In[2365]:= SearchDir["Viktor"]
Out[2365]= {"C:\\Documents and Settings", "C:\\Documents and Settings\\Viktor", ...,
  "C:\\Documents and Settings\\Viktor\\YandexDisk\\UserLib6789.ZIP"}

```

Еще раз вполне целесообразно отметить, что механизм типизации объектов, которым располагает *Mathematica*, довольно существенно уступает аналогичному механизму системы *Maple*, однако лишь относительно встроенных типов тестирования объектов. Между тем, и средства системы *Mathematica* позволяют тестировать типы наиболее важных объектов. В частности, по функции **FileType** обеспечивается проверка быть каталогом либо файлом, как иллюстрирует следующий простой пример, а именно:

```
In[3522]:= FileType["D:\\Math_myLib"]
Out[3522]= Directory
In[3523]:= FileType["D:\\Math_myLib\\ArtKr.mx"]
Out[3523]= File
In[3524]:= FileExistsQ["D:\\Math_myLib\\ArtKr.mx"]
Out[3524]= True
In[3525]:= FileExistsQ["D:\\Math_myLib"]
Out[3525]= True
```

Между тем, это средство уступает нашим процедурам *isFile* и *isDir* для системы *Maple* [45], обеспечивающим тестирование файлов и каталогов соответственно. Так, по *isFile* не только тестируется собственно наличие файла, но и режим его открытия, что в ряде случаев весьма важно. Имеются другие интересные средства [41–45] для тестирования состояния каталогов и файлов, включая их типы. В частности, у *Mathematica* имеется функция **FileExistsQ**, которая возвращает **True**, если объектом является файл данных или каталог, что с точки зрения файловой системы вполне корректно, тогда как для пользователя, работающего с файлами – это не одно и то же, что достаточно наглядно иллюстрирует следующий весьма простой пример, а именно:

```
In[2327]:= F := "D:\\Math_myLib"; If[FileExistsQ[F], OpenRead[F]; Read[F],
                                         Message[F::file, "file is absent"]]
OpenRead::noopen: Cannot open D:/Math_myLib. >>
Read::openx: D:/Math_myLib is not open. >>
Out[2327]= Read["D:/Math_myLib"]
```

Проверив по **FileExistsQ** наличие файла *F* (вместо него указан каталог) и получив **True**-значение, затем делается попытка открыть данный файл *F* на чтение с последующим чтением его первой логической записи, но обе эти процедуры доступа завершились с возвратом ошибочной диагностики. Поэтому для данной цели следует использовать тестирующую функцию **IsFile**, сочетающую функции **FileExistsQ** и **DirectoryQ**, либо для этих же целей намного более сложно организованную процедуру, вызов которой **FileQ[W]** возвращает **True**, если строка *W* определяет реально существующий файл, и **False** в противном случае. Процедура **FileQ** служит, скорее, для иллюстрации средств разработки процедур, ориентированных на работу с файловой системой компьютера. Фрагмент представляет исходные коды обоих процедур и примеры их применения.

```
In[2824]:= IsFile[x_] := If[FileExistsQ[x], If[! DirectoryQ[x], True, False], False]
In[2825]:= IsFile["D:\\Math_myLib"]
Out[2825]= False
In[2826]:= IsFile["D:\\Math_myLib\\ArtKr.mx"]
Out[2826]= True
In[2959]:= FileQ[f_;/; StringQ[f]] := Module[{d = ADrive[], s = {}, k = 1,
                                         a = ToLowerCase[StringReplace[Flatten[OpenFiles[]], "\\\\" -> "/"]],
                                         b = ToLowerCase[StringReplace[Directory[], "\\\" -> "/"]],
```



```

c = ToLowerCase[StringReplace[f, "\\\" -> "/"]],
For[k, k <= Length[d], k++, s = Append[s, d[[k]] <> ":"]; s = ToLowerCase[s];
If[StringLength[c] < 2 || ! MemberQ[s, StringTake[c, {1, 2}]], c = b <> "/" <> c, Null];
If[DirQ[c], False, If[MemberQ[a, c], True,
If[Quiet[OpenRead[c]] === $Failed, False, Close[c]; True]]]
In[2960]:= Map[FileQ, {"c:/Temp/Lugupeetud.docx", "Rans.Ian", "G:/New_Book/Art.Kr"}]
Out[2960]= {True, True, False}

```

Для дифференцированного тестирования файлов используется и функция **FileType**:

```

In[3027]:= {FileType["D:\\Math_myLib"], FileType["D:\\Math_myLib\\ArtKr.mx"]}
Out[3027]= {Directory, File}

```

Пакет *Mathematica* располагает также рядом других подобных тестирующих средств, ориентированных на элементы файловой системы компьютера. Ряд рассматривался несколько выше. В частности, в следующем фрагменте представлена процедура, чей вызов **EmptyFileQ[F]** возвращает **True**, если файл *F* пуст, и **False** в противном случае.

```

In[2130]:= EmptyFileQ[F_/, StringQ[F], y___] := Module[{a, b, c, d = {}, k = 1},
If[FileExistsQ[F], b = {F}, c = ArtKr; ClearAll[ArtKr];
a = FileExistsQ1[F, ArtKr]; If[! a, Return[$Failed], b = ArtKr; ArtKr = c];
While[k <= Length[b], AppendTo[d, Quiet[Close[b[[k]]]]];
If[Quiet[Read[b[[k]]]] === EndOfFile, Quiet[Close[b[[k]]]; True],
Quiet[Close[b[[k]]]; False]; k++];
d = If[Length[d] == 1, d[[1], d]; If[{y} != {}, {d, If[Length[b] == 1, b[[1], b]}, d]]
In[2131]:= EmptyFileQ["cinema.txt", 460]
Out[2131]= {{True, False, False, False}, {"C:\\Cinema.txt", "C:\\RECYCLER\\S-1-5-21-
329068152-1645522239-1644491937-10\\03\\Dc2\\Cinema.txt",
"C:\\Temp\\Cinema.txt", "F:\\Cinema.txt"}}
In[2132]:= EmptyFileQ["cinema.txt"]
Out[2132]= {True, False, False, False}
In[2133]:= EmptyFileQ["C:/cinema.txt"]
Out[2133]= True
In[2134]:= EmptyFileQ["C:/cinema.txt", 78]
Out[2134]= {True, "C:/cinema.txt"}

```

Если файл данных *F* отсутствует в файловой системе компьютера, вызов возвращает значение **\$Failed**. При этом, если в процессе поиска файла данных *F* устанавливается его *множественность* в файловой системе компьютера, то тестируются все файлы из их списка найденных файлов, включая и файлы, находящиеся в каталоге *Recycle Bin*. При этом, вызов процедуры **EmptyFileQ[F, y]** с *двумя* фактическими аргументами, где *y* – необязательный аргумент (*произвольное выражение*), возвращает вложенный список, первый подсписок которого определяет пустоту/непустоту (*True/False*) файла данных *F* в списке одноименных, в то время как второй подсписок определяет полные пути к

одноименным файлам *F*. При этом, между обеими подписками имеет место взаимно-однозначное соответствие. Предыдущий фрагмент представляет как исходный код, так и наиболее типичные примеры использования процедуры **EmptyFileQ**.

Следующая процедура обеспечивает поиск полных путей, содержащих подкаталог *x*, заданный полным именем, в файловой системе компьютера или в файловой системе заданных устройств внешней памяти прямого доступа, заданных *именами* в строчном формате. При этом, вызов процедуры **FindSubDir[x]** возвращает список *полных* путей в рамках всей файловой системы компьютера, тогда как вызов **FindSubDir[x, y, z, ...]** – в рамках лишь файловой системы устройств {*y, z, ...*}. Следующий фрагмент приводит исходный код **FindSubDir** наряду с довольно типичными примерами ее применения.

```
In[2040]:= FindSubDir[x_;/ StringQ[x], y___] := Module[{b = {}, c = "", k = 1, p, t,
    a = If[{y} == {}, Adrive[], {y}], f = "ArtKr.txt", h = ToLowerCase[x]},
    While[k <= Length[a], Run["Dir ", a[[k]] <> ":\\", "/B/S/L > "<>f];
    While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]];
    t = FileNameSplit[c]; p = Flatten[Position[t, h]];
    If[p != {} && DirectoryQ[FileNameJoin[t[[1 ;; p[[1]]]]], b = Append[b, c]];
    Continue[]]; Closes[f]; c = ""; k++; {DeleteFile[f], b}[[2]]]
```

```
In[2041]:= FindSubDir["Kherson", "F"]
```

```
Out[2041]= {"f:\\kherson", "f:\\archive\\books\\kherson",
    "f:\\archive\\books\\kherson\\book_1.pdf",
    =====
    "f:\\archive\\books\\kherson\\page_end.doc",
    "f:\\kherson\\book_3.doc", "f:\\kherson\\page_2.doc",
    "f:\\kherson\\page_end.doc", "f:\\kherson\\source.doc",
    "f:\\kherson\\avz_package_1.nb"}
```

```
In[2042]:= FindSubDir["Kherson"]
```

```
Out[2042]= {"c:\\kherson", "c:\\archive\\books\\kherson",
    "c:\\archive\\books\\kherson\\book_1.pdf",
    =====
    "c:\\kherson\\page_end.doc", "c:\\kherson\\source.doc",
    "f:\\kherson", "f:\\archive\\books\\kherson",
    "f:\\archive\\books\\kherson\\book_1.pdf",
    "f:\\archive\\books\\kherson\\book_2.doc",
    "f:\\archive\\books\\kherson\\book_2.pdf",
    "f:\\archive\\books\\kherson\\book_2.zip",
    "f:\\archive\\books\\kherson\\page_2.doc",
    "f:\\archive\\books\\kherson\\page_end.doc",
    "f:\\kherson\\book_3.doc", "f:\\kherson\\page_2.doc",
    "f:\\kherson\\page_end.doc", "f:\\kherson\\source.doc",
    "f:\\kherson\\avz_package_1.nb"}
```

Нижеследующая процедура **FilesDistrDirs** в определенной мере носит структурный характер для заданного ее фактическим аргументом каталога. Вызов **FilesDistrDirs[x]** возвращает вложенный список, чьими элементами являются подписки следующего формата  $\{dir\_p, f1, f2, \dots, fn\}$ , где  $dir\_p$  – каталог  $x$  и все его подкаталоги любого уровня вложенности, тогда как  $f1, f2, \dots, fn$  – имена файлов, расположенных именно в данном каталоге. Следующий фрагмент представляет исходный код процедуры с примером.

```
In[2062]:= FilesDistrDirs[x_ /; DirQ[x]] := Module[{a = {}, b, c = FromCharacterCode[17],
                                                    d, g, h = {}, f = "$Art17Kr$", k = 1, t},
  Run["Dir " <> StandPath[x] <> "/A/B/OG/S >" <> f];
  For[k, k < Infinity, k++, b = Read[f, String];
  If[SameQ[b, EndOfFile], DeleteFile[Close[f]]; Break[], AppendTo[a, b]];
  b = Gather[PrependTo[a, StringReplace[x, "/" -> "\\"]], DirQ[#1] == DirQ[#2] &];
  d = {Sort[Map[StringJoin[#, "\\"] &, b[[1]]],
  StringCount[#1, "\\"] >= StringCount[#2, "\\"] &],
  Quiet[Check[b[[2]], {}]]; a = Map[ToLowerCase, Flatten[d]];
  For[k = 1, k <= Length[d[[1]]], k++, t = ToLowerCase[d[[1]][[k]]];
  AppendTo[h, g = Select[a, SuffPref[#, t, 1] && StringFreeQ[StrDelEnds[#, t, 1], "\\"] &];
  a = MinusList[a, g]; a = {};
  For[k = 1, k <= Length[h], k++, b = h[[k]];
  AppendTo[a, {b[[1]], Map[StrDelEnds[#, b[[1]], 1] &, b[[2 ;; -1]]]}; Map[Flatten[#] &, a]]
In[2063]:= FilesDistrDirs["C:/AAA"]
Out[2063]= {{{"c:\\aaa\\bbb\\ddd\\", {"c:\\aaa\\bbb\\ccc\\", "letter_11_02_13.doc",
"letter_13_12_12.doc", "cinema.txt"}, {"c:\\aaa\\bbb\\", "burthday.doc"},
{"c:\\aaa\\", "letter_23_04_13.doc"}}
```

В ряде случаев возникает проблема определения *m*-файла, содержащего определение некоторого объекта, активного в текущем сеансе. Данную проблему успешно решает процедура, вызов которой **FindFileObject[x]** возвращает список файлов, содержащих определение объекта  $x$ , включая стандартную справку по нему; при отсутствии таких *m*-файлов вызов процедуры возвращает пустой список. Вызов **FindFileObject[x, y, ...]** с необязательными аргументами  $\{y, \dots\}$ , в качестве которых указаны имена в строчном формате устройств внешней памяти прямого доступа, обеспечивает поиск *m*-файлов на указанных устройствах вместо поиска во всей файловой системе компьютера при вызове процедуры с 1 аргументом. Следующий фрагмент представляет исходный код процедуры **FindFileObject** с некоторыми типичными примерами ее использования.

```
In[2393]:= FindFileObject[x_ /; ! SameQ[ToString[DefOpt[ToString[x]]], "Null"], y___] :=
Module[{b = {}, c = "", s = {}, d, k = 1, a = If[{y} == {}, Adrive[], {y}], f = "ArtKr",
h = "(*Begin[\"\" <> ToString[x] <> "\\"]*)", p = "(*\" <> ToString[x] <> \":usage=", t},
While[k <= Length[a], Run["Dir ", a[[k]] <> ":\\", "/B/S/L >" <> f];
While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String];
```

```

If[StringTake[c, {-2, -1}] == ".m", AppendTo[b, c]];
Continue[]]; Quiet[Close[f]]; c = ""; k++; k = 1;
While[k <= Length[b], If[Select[ReadList[b[[k]], String],
! StringFreeQ[#, h] && StringFreeQ[#, p] &] != {},
AppendTo[s, b[[k]]]; k++]; {DeleteFile[f], s}[[2]]]

In[2394]:= FindFileObject[ProcQ, "D"]
Out[2394]= {"d:\\avz_package\\aladjevproceduresandfunctions.m",
"d:\\avz_package\\avz_package.m", "d:\\avz_package\\myfunctions.m",
"d:\\avz_package\\rans.m"}
In[2395]:= Mapp[FindFileObject, {Map11, AvzAgn}, "D"]
Out[2395]= {"d:\\avz_package\\avz_package.m", FindFileObject[AvzAgn, "D"]}

```

Достаточно полезной при работе с файлами и каталогами представляется несложная функция, чей вызов **PathToFileQ[x]** возвращает **True**, если *x* определяет потенциально допустимый полный путь к каталогу или файлу данных, и **False** в противном случае. Следующий фрагмент представляет исходный код функции и пример применения.

```

In[2035]:= PathToFileQ[x_ /; StringQ[x]] := If[StringLength[x] > 3,
If[MemberQ[Join[CharacterRange["a", "z"], CharacterRange["A", "Z"]],
StringTake[x, 1]] && StringTake[x, {2, 2}] == "." &&
And[Map3[StringFreeQ, x, {"/", "\\"}] != {True, True}, True, False], False]

In[2036]:= Map[PathToFileQ, {"C:", "C:/", "G:/AVZ_Package", "H:\\AGN", "C:/Temp"}]
Out[2036]= {False, False, True, True, True}

```

Учитывая обстоятельство, что идеология файловой организации компьютера вполне допускает в целом ряде случаев работы со средствами доступа отождествлять файлы и каталоги, данное средство представляется полезным для обоих типов элементов.

В целом ряде случаев возникает необходимость считывания файла данных целиком, исключая из его содержимого символы «\r\n» возврата каретки и перевода строки. С данной задачей вполне успешно справляется следующая процедура **ReadFullFile**.

```

In[2032]:= ReadFullFile[F_ /; StringQ[F], y___] := Module[{a, b = $SvArt$Kr$},
If[FileExistsQ[F], a = F, ClearAll[$SvArt$Kr$];
If[! FileExistsQ1[F, $SvArt$Kr$], Return[$Failed],
a = $SvArt$Kr$[[1]]]; $SvArt$Kr$ = b;
StringReplace[StringJoin[Map[FromCharacterCode, BinaryReadList[a]]],
"\r\n" -> If[{y} != {}, If[StringQ[y], y, If[! HowAct[y], y = a; "", ""], ""]]]

In[2033]:= ReadFullFile["Cinema.txt"]
Out[2033]= "http://smotri-filmu.ru/http://smotri-filmu.ru/detektiv/page/24/
http://smotri-filmu.ru/3450-bratya-2012-serial-2-seriya-rossiya-satrip.html
http://xreality.ru/632-ponyat-prostit-tv-2013.html"

In[2034]:= ReadFullFile["direks.doc"]

```

```

Out[2034]= $Failed
In[2035]:= ReadFullFile["dereks.doc"]
Out[2035]= "Below, you will find information on our recent book which can be useful for
           book section on your site. Aladjev V.Z., Boiko V.K., Rovba E.A. Programming
           in packages Maple and Mathematica: Comparative Analysis.- Grodno: Grodno
           State University, 2011, 517 p., ISBN 978-985-515-481-6."
In[2036]:= ReadFullFile["dereks.doc", h]; h
Out[2036]= "C:\\Temp\\dereks.doc"
    
```

Вызов **ReadFullFile[F]** возвращает содержимое файла данных *F* с заменой в нем "**\r\n**" символов возврата каретки и перевода строки на символы **"**"; если файл *F* отсутствует в файловой системе компьютера, вызов возвращает значение **\$Failed**. Тогда как вызов **ReadFullFile[F, y]** через 2-й необязательный аргумент *y* – *неопределенную переменную* – возвращает полное имя либо полный путь к файлу *F*; если *y* является строкой, то она замещает в возвращаемом содержимом файла данных *F* символы "**\r\n**" на строку *y*.

Следующая процедура облегчает решение задачи использования внешних программ пакета либо операционной среды. Вызов процедуры **ExtProgExe[x, y, h]** обеспечивает поиск в файловой системе компьютера {*exe | com*}–файла с программой с *последующим* ее выполнением на параметрах *y* командной строки. Оба аргумента *x* и *y* кодируются в строчном формате. Успешное выполнение этой процедуры возвращает *полный* путь к *ASCII*-файлу **"\$TempFile\$"**, содержащему результат выполнения программы *x*, и его можно обрабатывать стандартными средствами на основе его структуры. При этом, в случае отсутствия файла с требуемой программой *x* возвращается **\$Failed**, в то время как при использовании третьего необязательного аргумента *h* (*произвольное выражение*) файл с программой *x*, загруженный в системный каталог, определяемый переменной **\$InstallationDirectory**, удаляется; более того, удаляется также файл **"\$TempFile\$"**, если он пуст или выполнение программы *x* завершилось аварийно. Фрагмент представляет исходный код процедуры **ExtProgExe** с типичными примерами ее использования.

```

In[2095]:= ExtProgExe[x_;/; StringQ[x], y_;/; StringQ[y], h___] :=
           Module[{a = "$TempFile$", b = $InstallationDirectory <> "\\ " <> x, c},
           Empty::datafile = "Datafile $TempFile$ is empty; the datafile had been deleted.";
           If[FileExistsQ[b], c = Run[x, " ", y, " > ", a],
           c = LoadExtProg[x]; If[c === $Failed, Return[$Failed]];
           c = Run[x, " ", y, " > ", a]; If[{h} != {}, DeleteFile[b]];
           If[c != 0, DeleteFile[a]; $Failed, If[EmptyFileQ[a], DeleteFile[a];
           Message[Empty::datafile, Directory[] <> "\\ " <> a]]]

In[2096]:= ExtProgExe["HostName.exe", "", 1]
Out[2096]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
In[2097]:= ExtProgExe["Rans.exe", "", 1]
Out[2097]= $Failed
In[2098]:= ExtProgExe["systeminfo.exe", "/fo table", 1]
Out[2098]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
    
```

```

In[2099]:= ExtProgExe["tasklist.exe", "/svc ", 1]
Out[2099]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
In[2100]:= {G, h, a} = {{}, "", "$TempFile$"}; While[h != "EndOfFile", G = Append[G,
    PrefCond[h = Read[a, String], " "]]; Close[a]; Select[G, # != "" &][[2 ;; -2]]
Out[2100]= {"System Idle Process", "System", "smss.exe", "avgchsvx.exe", "csrss.exe",
    "winlogon.exe", "services.exe", "lsass.exe", "ati2evxx.exe", "svchost.exe",
    "svchost.exe", "svchost.exe", "svchost.exe", "vsmon.exe",
    "ati2evxx.exe", "explorer.exe", "ctfmon.exe", "ISWSVC.exe", "spoolsv.exe",
    "svchost.exe", "avgwdsvc.exe", "jqs.exe", "svchost.exe", "ForceField.exe",
    "MsPMSPSv.exe", "AVGIDSAgent.exe", "avgnsx.exe", "avgemcx.exe",
    "AmplusnetPrivacyTools.exe", "alg.exe", "avgtray.exe", "soundman.exe",
    "vprot.exe", "zatray.exe", "Skype.exe", "AVGIDSMonitor.exe", "LvAgent.exe",
    "FreeCommander.exe", "avgrsx.exe", "avgcsrvx.exe", "WINWORD.EXE",
    "Mathematica.exe", "MathKernel.exe", "javaw.exe", "notepad.exe", "cmd.exe",
    "cwmaple.exe", "mserver.exe", "wmiprvse.exe"}
In[2101]:= ExtProgExe["qprocess.exe", "*"]
Out[2101]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
In[2102]:= k = 1; h = ""; While[! SameQ[h, EndOfFile], h = Read["$TempFile$", String];
    Print[h]; k++; Close["$TempFile$"];

```

USERNAME	SESSIONNAME	ID	PID	IMAGE
>network service	console	0	1232	svchost.exe
>network service	console	0	1448	svchost.exe
>local service	console	0	1496	svchost.exe
>aladjev	console	0	2312	explorer.exe
>aladjev	console	0	2624	soundman.exe
>aladjev	console	0	2632	vprot.exe
>aladjev	console	0	2672	avgui.exe
>aladjev	console	0	2680	yandexdisk.exe
>aladjev	console	0	2728	ctfmon.exe
>aladjev	console	0	2736	skype.exe
>aladjev	console	0	2792	lvagent.exe
>aladjev	console	0	2928	cap3lak.exe
>local service	console	0	4088	alg.exe
>aladjev	console	0	4424	freecommander
>aladjev	console	0	11372	mathematica.exe
>aladjev	console	0	7788	mathkernel.exe
>aladjev	console	0	13464	mathkernel.exe
>aladjev	console	0	8644	winword.exe
>aladjev	console	0	2464	cmd.exe
>aladjev	console	0	2360	qprocess.exe

Аналогично процедуре **ExtProgExe**, процедура **Memory** также довольно существенно использует процедуру **LoadExtProg**; успешный вызов **Memory[]** обеспечивает возврат списка, чьи элементы определяют структурную организацию памяти компьютера. В

то время как вызов **Memory[x]** с необязательным аргументом *x* – любым выражением – дополнительно удаляет из каталога, определяемого **\$InstallationDirectory**, программу *"mem.exe"*. Фрагмент приводит исходный код процедуры с примером применения.

```
In[2059]:= Memory[x_] := Module[{a = "$Memory$", b, c = "mem.exe", d},
    b = $InstallationDirectory <> "\\\" <> c;
    If[FileExistsQ[b], c = Run[c, " > ", a], d = LoadExtProg[c];
    If[d === $Failed, Return[$Failed], c = Run[c, " > ", a]];
    If[{x} != {}, DeleteFile[b]]; If[c != 0, DeleteFile[a]; $Failed,
    c = Map[StringTrim, ReadList[a, String]]; DeleteFile[a]; c]]

In[2060]:= Memory[]
Out[2060]= {"655360 bytes total conventional memory", "655360 bytes available to MS-Dos",
    633488 largest executable program size", "1048576 bytes total contiguous extended
    memory", "0 bytes available contiguous extended memory", "941056 bytes
    available XMS memory", "MS-Dos resident in High Memory Area"}
```

Система *Mathematica* располагает двумя стандартными функциями **RenameFile[x,y]** и **RenameDirectory[x,y]** для обеспечения переименования файлов и каталогов файловой системы компьютера соответственно. Между тем, с точки зрения файловой концепции данные функции было бы весьма целесообразно оформлять единым средством, ибо в этой концепции каталоги и файлы во многом отождествляются и их обработка может выполняться одними и теми же средствами. Упомянутые стандартные функции и по ограничениям вполне идентичны, а именно: для переименования имени *w* элемента файловой системы на *новое* имя *y* элемент с именем *y* должен отсутствовать в системе, в противном случае возвращается **\$Failed** с выводом диагностического сообщения. В то же время, если в качестве *y* кодируется только *новое* имя без полного пути к новому элементу *y*, то производится его копирование в текущий каталог; в случае каталога *w* он со всем содержимым копируется в текущий каталог под новым именем *y*. Поэтому подобная организация достаточно неудобна во многих отношениях, что и побудило нас для переименования каталогов и файлов данных определить единую процедуру **RenDirFile**, которая обеспечивает переименование элемента *w* (каталог либо файл) на месте с сохранением его типа и всех его атрибутов; при этом, в качестве аргумента *y* используется новое имя элемента *w*. Итак, успешный вызов процедуры **RenDirFile[w, y]** возвращает полный путь к переименованному элементу *w*. В случае существования элемента *y* возвращается полный путь к *исходному* элементу *w*. Следующий фрагмент представляет исходный код процедуры **RenDirFile** с примерами ее использования.

```
In[2499]:= RenameFile["C:\\Temp\\Grsu.doc", "Vgtu.doc"]
Out[2499]= "C:\\Documents and Settings\\Aladjev\\My Documents\\Vgtu.doc"
In[2500]:= RenameDirectory["C:\\Temp\\Noosphere Academy", "Rans Ian"]
Out[2500]= "C:\\Documents and Settings\\Aladjev\\My Documents\\Rans Ian"

In[2634]:= RenDirFile[x_;/; FileExistsQ[x] || DirectoryQ[x], y_;/; StringQ[y]] :=
    Module[{a = If[FileExistsQ[x], RenameFile, RenameDirectory], c = "", k = 1,
    b = FileNameSplit[x]}, If[Length[b] == 1, Quiet[Check[a[x, y],
```

```

                                FileDirStForm[x]]]; Directory[] <> "\\\" <> y,
                                Do[c = c <> b[[k]] <> "\\\"; k++, {Length[b] - 1}];
                                Quiet[Check[a[x, c <> y], FileDirStForm[x]]]]]

In[2635]:= RenDirFile["C:\\Temp/Kino Online.txt", "Cinema Online.txt"]
Out[2635]= "C:\\Temp\\Cinema Online.txt"
In[2636]:= RenDirFile["C:\\Temp/Noosphere Academy", "Rans Ian"]
Out[2636]= "C:\\Temp\\Rans Ian"
In[2637]:= RenDirFile["C:\\Temp/Noosphere Academy", "Noosphere Academy"]
Out[2637]= "c:\\temp\\noosphere academy"
In[2638]:= RenDirFile["C:\\Temp/Cinema Online.txt", "Cinema Online.txt"]
Out[2638]= "c:\\temp\\cinema online.txt"

```

Еще раз следует напомнить, что все элементы *файловой* системы компьютера должны кодироваться с разделителями, определяемыми переменной **\$PathnameSeparator**, по умолчанию в качестве разделителя полагается обратный слэш "\\\", кодируемый как двойной. Между тем, в системе *Mathematica* в общем случае обратный "\\\" и прямой "/" слэши различаются в качестве разделителей, а именно: если *обратный* слэш играет роль стандартного разделителя элементов файловой системы компьютера, то прямой слэш также вполне может выполнять эту функцию, исключая случай, когда прямой слэш кодируется в конце цепочки каталогов. Для устранения такого неудобства нами создана простая функция **DirQ**, рассмотренная выше. При этом, достаточно простая процедура **SetPathSeparator[x]** производит установку для путей к файлам/каталогам в качестве разделителя "\\\" или "/" на период текущего сеанса с системой. Следующий фрагмент представляет исходный код процедуры с примерами ее использования.

```

In[4069]:= $PathnameSeparator
Out[4069]= "\\\"

In[4107]:= SetPathSeparator[x_/: MemberQ[{"/", "\\\"}, x]] :=
            Module[{}, Unprotect[$PathnameSeparator]; $PathnameSeparator = x;
            SetAttributes[$PathnameSeparator, Protected]]

In[4108]:= {SetPathSeparator["/"]; $PathnameSeparator,
            SetPathSeparator["\\\"]; $PathnameSeparator}

Out[4108]= {"/", "\\\"}
In[4170]:= DirectoryQ["C:/Program Files/Wolfram Research/Mathematica/9.0"]
Out[4170]= True
In[4171]:= DirectoryQ["C:\\Program Files\\Wolfram Research\\Mathematica/9.0"]
Out[4171]= True
In[4172]:= DirectoryQ["C:\\Program Files\\Maple 11\\"]
Out[4172]= True
In[4173]:= DirectoryQ["C:\\Program Files\\Maple 11/"]
Out[4173]= False
In[4174]:= Map[DirQ, {"C:\\Program Files/Maple 11/", "C:/Program Files/Maple 11\\\"}]
Out[4174]= {True, True}

```



```

In[4197]:= FileDirStForm[x_;/ StringQ[x]] :=
           Module[{a = StringReplace[ToLowerCase[x], "/" -> "\\"]},
           If[StringLength[a] == 3 && StringTake[a, {2, 2}] == ":", a, StrDelEnds[a, "\\", 2]]]

In[4198]:= FileDirStForm["D:/AVZ_Package\\AVZ_Package_1.nb/"]
Out[4198]= "d:\\avz_package\\avz_package_1.nb"
In[4199]:= Map[FileDirStForm, {"D:/", "C:"}]
Out[4199]= {"d:\\", "c:"}

In[4200]:= StandPath[x_String] := Module[{a, b = "", c, k = 1}, If[PathToFileQ[x],
           a = FileNameSplit[StringReplace[ToLowerCase[x], "/" -> "\\"]];
           For[k, k <= Length[a], k++, c = a[[k]];
           If[! StringFreeQ[c, " "], b = b <> StrStr[c] <> "\\",
           b = b <> c <> "\\"]; StringTake[b, {1, -2}], x]]

In[4202]:= StandPath["C:/Program Files\\Wolfram Research/Mathematica/9.0"]
Out[4202]= "c:\\\\\"program files\\\"\\\\\"wolfram research\\\"\\\\\"mathematica\\\"9.0"
    
```

Таким образом, в большинстве случаев подобно *Maple*, для *Mathematica* также можно использовать оба типа разделителей элементов файловой системы, однако сказанное касается только относительно системы *Windows XP*, для других платформ возможны отличия, в целом ряде случаев достаточно существенные для программирования.

Как отмечалось выше, используя разные форматы для *имен* файлов и *путей* к ним, мы получаем возможность открывать один и тот же физический файл в разных потоках, что в некоторых случаях обеспечивает упрощение обработки файлов данных. Между тем, в ряде случаев такая возможность усложняет алгоритмы, связанные с обработкой файлов данных, например, файл, созданный на основе одного формата пути к нему, в общем случае не будет распознаваться стандартными средствами, в частности:

```

In[3015]:= Write["RANS.txt"]; Close["Rans.txt"]
           General::openx: Rans.txt is not open. >>
Out[3015]= Close["Rans.txt"]
    
```

Таким образом, корректное использование *имен* файлов и *путей* к ним предполагает, в общем случае, работу с одним и тем же их форматом, как и иллюстрирует пример. Поэтому в качестве довольно простого приема, позволяющего унифицировать имена файлов/каталогов и путей к ним, можно предложить следующий стандарт – символы, составляющие имена файлов и путей к ним, кодируются на нижнем регистре, тогда как в качестве разделителей используется стандартный "\\". Данную задачу успешно решают довольно простые процедуры **FileDirStForm[x]** и **StandPath[x]**, чьи исходные коды с примерами применения представлены в предыдущем фрагменте. Более того, функция **StandPath** для тестирования допустимости аргумента *x* в качестве *реального* пути использует функцию **PathToFileQ**, представляющую самостоятельный интерес и обеспечивающую корректность обработки путей, содержащих символы пробела. В частности, использованием процедурой **DirFD** процедуры **StandPath** позволяет вполне корректно получать содержимое любого каталога файловой системы компьютера, в

которых содержатся символы *пробела* и на которых команда **Dir** системы **DOS** не дает результата, как весьма наглядно иллюстрирует следующий простой пример:

```
C:\> Dir C:\\Program Files\\*.*
```

```
The filename, directory name, or volume label syntax is incorrect.
```

```
In[2460]:= DirFD["C:/Program Files"]
```

```
Out[2460]= {"ABBY Lingvo", "activePDF", "Adobe", "AML Products", "ArcSoft", "AVG",  
"AVG Secure Search", "Canon", "Common Files", "Extras", "FreeCommander",  
"Free Media Player", "FileZilla FTP Client", "Ghostgum", "gs", "Java", "JRE",  
"Internet Explorer", "K-Lite Codec Pack", "LBP1120_WinXP", "Maple 11",  
"Messenger", "Microsoft ActiveSync", "microsoft frontpage", "Movie Maker",  
"Microsoft Office", "Mozilla Firefox", "MSBuild", "MSECache", "MSXML 4.0",  
"MSN", "NetMeeting", "OpenOffice.org 3", "Outlook Express", "Pandora  
Recovery", "Real Alternative", "Skype", "VideoLAN", "Reference Assemblies",  
"Registry Clean Expert", "ScanSoft", "ToniArts", "Windows Desktop Search",  
"Windows Media Connect 2", "Windows NT", "WinRAR", "WinZip", "xerox",  
"Windows Media Player", "Wolfram Research", "Yandex"}, {}
```

Процедура **StandPath** может достаточно эффективно использоваться при разработке различного рода средств доступа в файловой системе компьютера. Используется она целым рядом средств доступа к файлам, рассматриваемым в настоящей книге. Итак, представленные в данной главе средства, порой, достаточно существенно облегчают программирование задач, имеющих дело с файловой системой компьютера. Наряду с этим, данные средства расширяют функциональные средства доступа, иллюстрируя ряд достаточно полезных приемов программирования задач данного типа. Данные средства в целом ряде случаев весьма существенно дополняют стандартные средства доступа, поддерживаемые системой, облегчая программирование целого ряда весьма важных приложений, имеющих дело с файлами данных различного формата.

Наш опыт разработки средств доступа, расширяющих аналогичные средства пакетов *Maple* и *Mathematica*, позволяет констатировать, что базовые средства доступа пакета *Mathematica* в совокупности с его *глобальными* переменными позволяют более просто и эффективно программировать оригинальные средства доступа пользователя. Более того, создаваемые средства обладают большей, порой существенно, реактивностью по отношению к аналогичным средствам, разрабатываемым в среде пакета *Maple*. Итак, в среде пакета *Mathematica* можно решать задачи, связанные с достаточно сложными алгоритмами обработки файлов данных, тогда как в среде *Maple*, в первую очередь, в случае больших файлов эффективность таких алгоритмов оставляет желать лучшего. В целом ряде приложений средства, приведенные в настоящей главе, представляются достаточно полезными, позволяя, порой, существенно упрощать программирование. Между тем, следует иметь в виду, что ряд средств, базирующихся на функции **Run** и командах **DOS**, в общем случае могут оказаться непереносимыми на другие версии системы и платформы, требуя соответствующей адаптации на новые условия.

Удаление файла в текущем сеансе производится посредством стандартной функции **DeleteFile**, вызов которой возвращает **Null**, т.е. ничего в случае успешного удаления

файла либо их списка и **\$Failed** в противном случае. При этом, если в списке файлов удаляются только те, которые не имеют *Protected*-атрибута. Более того, эта операция не сохраняет удаленные файлы в системном каталоге *Recycle Bin*, что в ряде случаев крайне нежелательно, прежде всего, в свете возможности их восстановления. Дело в том, что системная функция **DeleteFile** основывается на *Dos*-команде *Del* системы *Ms Dos*, которая согласно спецификации данной операционной системы сразу удаляет файл из файловой системы компьютера без его сохранения, что существенно отличает ее от аналогичной операции среды *Windows*, которая по умолчанию сохраняет удаленный файл в специальном каталоге *Recycle Bin* (в частности, для оболочки *Windows XP*). Для устранения данного недостатка предложена процедура **DeleteFile1** с исходным кодом и примерами ее применения, представленными следующим фрагментом. Успешный вызов процедуры **DeleteFile1[x]** возвращает 0, удаляя заданные аргументом *x* файлы с сохранением их в каталоге *Recycle Bin* операционной оболочки *Windows XP*.

Между тем, удаленные вызовом **DeleteFile1[x]** файлы сохраняются в каталоге *Recycle Bin*, но они невидимы для просмотра системными средствами, например, *Ms Explorer*, усложняя очистку данного системного каталога. Для удаления из каталога *Recycle Bin* файлов, сохраненных по процедуре **DeleteFile1**, используется процедура, успешный вызов которой **ClearRecycler[]** возвращает 0, удаляя указанного типа файлы данных из системного каталога *Recycle Bin* с сохранением в нем файлов, удаленных средствами *Windows* или ее приложениями. Наконец, команда *Disk Cleanup* в *Windows XP* в ряде случаев полностью не очищает от файлов системный каталог *Recycler* и это успешно делает вызов **ClearRecycler["ALL"]**, возвращая 0 и обеспечивая удаление всех файлов из системного каталога *Recycler*. Нижеследующий фрагмент представляет исходный код процедуры **ClearRecycler** с примерами ее наиболее типичного использования.

```
In[2094]:= DeleteFile1[x_/: StringQ[x] || ListQ[x]] := Module[{a = Map[ToString,
    Flatten[{x}]], b = FilesDistrDirs[BootDrive[]][[1]] <> ":\\Recycler"}][[1]], k = 1},
    a = Map[If[StringFreeQ[#, ":"], Directory[] <> "\\\" <> #, #] &, a]; a = Map[StandPath, a];
    While[k <= Length[a], Run["Copy /Y " <> a[[k]] <> " " <> b[[1]] <>
        FileNameTake[a[[k]]]]; k++];
    a = Map[Run["Del " <> #] &, a]; If[MemberQ[{0, {}}, Flatten[a]], 0, 0]]

In[2095]:= DeleteFile1[{"Kherson.m", "Package.nb"}]
Out[2095]= 0
In[2096]:= DeleteFile1[{"AvzKr.m", "AgnArt.nb"}]
Out[2096]= 0
In[2097]:= DeleteFile1["C:/Documents and Settings/Cinema Online.txt"]
Out[2097]= 0

In[2198]:= ClearRecycler[x_] := Module[{a, b = {}, c = "", k = 2, d = BootDrive[]][[1]] <>
    ":\\Recycler"}, a = FilesDistrDirs[d]; If[{x} != {} && x === "ALL",
    Run["Dir " <> d <> "/A/B/S/W > Dir.txt"];
    While[! SameQ[c, EndOfFile], c = Read["Dir.txt", String];
    AppendTo[b, c]; Continue[]]; DeleteFile[Close["Dir.txt"]]; b = b[[1 ;; -2]];
```

```

For[k = 1, k <= Length[b], k++, c = b[[k]];
If[DirectoryQ[c], DeleteDirectory[c, DeleteContents -> True],
  Run["Del /F /A[:AHR] " <> c]]]; 0, a = FilesDistrDirs[d][[1]];
If[a == {"c:\\recycler\\"}, 0, While[k <= Length[a], c = FileBaseName[a[[k]]];
  If[c != "desktop" && ! StringMatchQ[c, "info" ~~ ____ ~~ NumberString] &&
    ! StringMatchQ[c, "dc" ~~ ____ ~~ NumberString], AppendTo[b, a[[k]]]; k++];
c = Map[StringJoin[a[[1]], #] &, b]; c = Map[Run["Del " <> StandPath[#] &, c];
  If[MemberQ[{0}, {}], Flatten[c], 0, 0]]]]

In[2199]:= Map[ClearRecycler, {Null, "ALL"}]
Out[2199]= {0, 0}

In[2215]:= RestoreDelPackage[f_ /; StringQ[f] && MemberQ[{"cdf", "m", "mx", "nb"},
  FileExtension[f]], s_String := Module[{b, c, d = {}, k = 1, j, p, t, h = {},
  a = FilesDistrDirs[BootDrive][[1]] <> ":\Recycler"}][[1 ;; -2]]],
  For[k, k <= Length[a], k++, b = a[[k]]; For[j = 2, j <= Length[b], j++,
    If[SameQ[FileExtension[b[[j]]], FileExtension[f]] &&
      SameQ[ContextFromFile[b[[1]] <> b[[j]], s], AppendTo[d, b[[1]] <> b[[j]]]]];
    If[d == {}, $Failed, {b = {}, k = 1}; For[k, k <= Length[d], k++,
      AppendTo[b, CopyFile[d[[k]], CreateDirectory[Directory[] <> "\\$" <>
        ToString[k]] <> "\\ " <> FileNameTake[d[[k]]]; DeleteFile[d[[k]]];
        a = DirectoryName[d[[k]]];
        If[! SameQ[a, ""], DeleteDirectory[a, DeleteContents -> True]]; b]]

In[2216]:= DeleteFile1["C:\\AVZ_Package\\Kherson.m"]
Out[2216]= 0
In[2217]:= FileExistsQ["C:\\AVZ_Package\\Kherson.m"]
Out[2217]= False
In[2218]:= RestoreDelPackage["Kherson.m", "Kherson`"]
Out[2218]= {"C:\\Documents and Settings\\Viktor\\My Documents\\$1\\kherson.m",
  "C:\\Documents and Settings\\Viktor\\My Documents\\$2\\kherson.m",
  "C:\\Documents and Settings\\Viktor\\My Documents\\$3\\kherson.m",
  "C:\\Documents and Settings\\Viktor\\My Documents\\$4\\kherson.m"}

```

Представленная в конце фрагмента процедура **RestoreDelPackage** восстанавливает из системного каталога *Recycler* пакеты, удаленные по команде *Del* системы *Ms Dos* и по функции **DeleteFile1**. Успешный вызов **RestoreDelPackage[F, "Context`"]**, где первый *F* аргумент определяет имя файла формата {"cdf", "m", "mx", "nb"}, который подлежит восстановлению, и второй – контекст, ассоциированный с пакетом, возвращает список полных путей к восстановленным файлам, одновременно удаляя из каталога *Recycler* восстановленные файлы с нужным пакетом. Данные средства довольно существенно расширяют функции *Mathematica* по программному восстановлению файлов любого типа, удаленных *Windows*, ее приложениями либо нашей процедурой **DeleteFile1**.

Глава 10. Организация работы с пакетами в среде *Mathematica*

Подобно многим развитым системам *Mathematica* является расширяемой системой, т.е. дополнительно к своим *встроенным* функциональным средствам, которые вполне покрывают потребности довольно широкого круга пользователей, система позволяет в среде своего встроенного *Math*-языка программировать недостающие конкретному пользователю средства, а также расширять и исправлять стандартные средства. Более того, недостающие средства, не являющиеся встроенными, пользователь может найти в многочисленных пакетах как поставляемых с *Mathematica*, так и существующих для различных прикладных областей. Вопрос только в нахождении нужного для данного конкретного случая пакета, содержащего определения функций, процедур и других объектов, требуемых для приложения, разрабатываемого в среде *Mathematica*. Пакет имеет стандартную организацию и содержит определения различных объектов, как-то функции, процедуры, переменные и т.д., решающих вполне определенные задачи. Со своей стороны система *Mathematica* обеспечивает *стандартный* набор пакетов, чей состав определяется версией системы. Для получения состава поставляемых пакетов с текущим релизом можно воспользоваться процедурой, вызов которой `MathPackages[]` возвращает список имен пакетов, из названия которых с определенной уверенностью можно делать выводы о их основном назначении. Тогда как вызов `MathPackages[w]` с необязательным аргументом *w* – *неопределенной* переменной – обеспечивает через нее дополнительно возврат 3-элементного списка, первый элемент которого определяет текущий релиз системы *Mathematica*, второй – тип лицензии и третий – предельный срок действия лицензии. Фрагмент представляет исходный код процедуры наряду с примерами ее наиболее типичного использования.

```
In[2077]:= MathPackages[h___] := Module[{a = "$Kr_Art$", b = {}, d, k = 1,
                                         c = $InstallationDirectory},
    d = Run["Dir " <> StandPath[c] <> "/A/B/O/S > $Kr_Art$"];
    If[d != 0, $Failed, While[! SameQ[d, EndOfFile], d = Read[a, String];
    If[! DirQ[d] && FileExtension[d] == "m", AppendTo[b, FileNameBase[d]]; k++];
    DeleteFile[Close[a]]; b = MinusList[DeleteDuplicates[b], {"init", "PacletInfo"}];
    If[{h} != {} && ! HowAct[h], h = {$Version, $LicenseType, StringJoin[StringSplit[
    StringReplace[DateString[$LicenseExpirationDate], " " -> "*"], "*" ][[1 ;; -2]]]]; Sort[b]]
In[2078]:= MathPackages[]
Out[2078]= {"accessodbc", "ActionApp", "ActionAppLocalized", "AdobeStandard",
  "Aftershock", "Airflo", "all_children", "all_descendants", "all_top_level", "AngleDialog",
  =====
  "WorldPlot", "WSDL", "Xbox", "XGRID", "XML", "XMLSchema", "ZapfDingbats", "Zip"}
In[2079]:= Length[%]
Out[2079]= 453
In[2080]:= MathPackages[Sv]; Sv
Out[2080]= {"9.0 for Microsoft Windows (32-bit) (January 24, 2013)", "Professional",
  "Mon 2 Dec 2013"}
```

Из приведенного фрагмента следует, что *Mathematica* версии 9 содержит 453 пакета, ориентированных на различные приложения. Перед использованием средств пакета его предварительно требуется загрузить в текущий сеанс по функции **Get[Пакет]**.

## 10.1. Понятие контекста и его использование в среде *Mathematica*

Понятие *контекста* введено в программную среду системы для организации работы с символами, которые представляют различные объекты (*процедуры, функции, пакеты, переменные и пр.*), во избежание возможных конфликтов с одноименными символами. Основная идея состоит в том, что полное имя любого символа состоит из двух частей: *контекст* и *краткое имя*, т.е. полное имя некоторого объекта имеет следующий формат: "*контекст`краткое имя*", где символ "`" (*обратная кавычка*) выполняет роль маркера, идентифицирующего контекст в программной среде пакета. Например, **Avzagn`Vsv** представляет символ с контекстом *Avzagn* и кратким именем *Vsv*. При этом, с такими символами можно производить операции как с обычными именами; система при этом рассматривает символы **aaa`xyz** и **bbb`xyz** различными. Наиболее распространенное использование контекста состоит в присвоении его функционально одинаковым или семантически связанным символам. Например,

**AladjevProcedures`StandPath, AladjevProcedures`MathPackages**

процедуры **StandPath** и **MathPackages** принадлежат к одной и той же группе средств, ассоциированных с контекстом *AladjevProcedures`*. Текущий контекст определен для любого момента сеанса системы, он находится в глобальной переменной **\$Context**:

```
In[3315]:= $Context
Out[3315]= "Global`"
```

В текущем сеансе *Mathematica* текущим контекстом по умолчанию определен **Global`**. Тогда как глобальная переменная **\$ContextPath** определяет список *контекстов* после **\$Context** для поиска введенного в текущий сеанс символа. Обращаться к символам из текущего контекста можно просто по их кратким именам; при этом, если этот символ пересекается с символом из списка, определяемого переменной **\$ContextPath**, именно второй будет использован вместо символа из текущего контекста, например:

```
In[3316]:= $ContextPath
Out[3316]= {"AladjevProceduresAndFunctions`", "PacletManager`", "WebServices`",
            "System`", "Global`"}
```

Тогда как вызовы **Context[x]** и **Contexts[]** возвращают контекст, приписанный символу *x*, и список всех контекстов соответственно, например:

```
In[3319]:= Context[ProcQ]
Out[3319]= "AladjevProcedures`"
In[3320]:= Contexts[]
Out[3320]= {"AladjevProcedures`", "AladjevProcedures`Arity`", "AladjevProcedures`ProcQ`"
            "XML`MathML`Symbols`", "XML`NotebookML`", "XML`Parser`", "XML`RSS`"}
=====
```

По аналогии с файловой системой компьютера, *контексты* можно вполне сравнивать с каталогами. Можно определять путь к файлу, указывая каталог, его содержащий, и само имя файла. В то же время, текущий *контекст* вполне может быть ассоциирован с *текущим* каталогом, к файлам которого можно обращаться просто по их именам. При этом, подобно файловой системе *контексты* могут иметь иерархическую структуру, в частности, "*Visualization`VectorFields`VectorFieldsDump`*". Итак, путь поиска контекста символов в среде системы *Mathematica* подобен пути поиска программных файлов. В начале сессии *Mathematica* текущий контекст по умолчанию полагается **Global`**, и все вводимые в сессию символы будут ассоциироваться именно с данным контекстом, за исключением встроенных символов, например, **Do**, приписанных контексту **System`**. Путь поиска контекстов по умолчанию включает контексты для *системно-определенных* символов. Тогда как для всех символов, удаленных по функции **Remove**, контекст не может быть определен, как иллюстрирует следующий весьма простой пример:

```
In[3334]:= Avz := 71; Context["Avz"]
Out[3334]= "Global`"
In[3335]:= Remove["Avz"]; Context["Avz"]
Context::notfound: Symbol Avz not found. >>
Out[3335]= Context["Avz"]
```

При использовании контекстов нет гарантии, что два одноименных символа имеются в различных контекстах. Поэтому пакет определяет *максимальным* приоритет выбора именно того символа с этим именем, чей контекст является первым в списке, который определяется глобальной переменной **\$ContextPath**. Поэтому, для помещения такого контекста в начало указанного списка можно использовать простую конструкцию:

```
In[3343]:= $ContextPath
Out[3343]= {"AladjevProcedures`", "PacletManager`", "WebServices`", "System`", "Global`"}
In[3344]:= PrependTo[$ContextPath, "RansIanAvz`"]
Out[3344]= {"RansIanAvz`", "AladjevProcedures`", "PacletManager`", "WebServices`",
            "System`", "Global`"}
In[3345]:= $ContextPath
Out[3345]= {"RansIanAvz`", "AladjevProcedures`", "PacletManager`", "WebServices`",
            "System`", "Global`"}
```

В результате символы с тем же самым кратким именем, контексты которых находятся в списке **\$ContextPath** далее от начала, недоступны для обращения к ним по краткому имени. Поэтому для обращения к ним следует использовать полные имена формата *ContextName*; при этом, при вводе новых символов, перекрывающих одноименные в списке **\$ContextPath**, выводится соответствующее сообщение. Для ввода в текущий сеанс символа *s*, перекрывающего уже существующие одноименные символы, следует предварительно реорганизовать **\$ContextPath** или удалить этот символ по **Remove[s]**.

### *Связь контекстов и пакетов в программной среде системы Mathematica.*

Одним из основных механизмов расширения *Mathematica* являются пакеты, которые содержат определения новых символов, предназначенных для использования как вне

пакета, так и внутри его. Данные символы могут соответствовать, в частности, новым функциям или объектам, определенным в пакете и расширяющим функциональную среду *Mathematica*. Согласно принятому соглашению все новые символы, введенные в некотором пакете, помещены в контекст, имя которого связано с названием пакета. При загрузке пакета в текущий сеанс, данный контекст добавляется в начале списка, определяемого глобальной переменной **\$ContextPath**. Как правило, для обеспечения ассоциирования пакета с контекстом используется конструкция **BeginPackage["xxx"]**, кодируемая в его начале. При загрузке пакета в *текущий* сеанс именно контекст "xxx" будет обновлять текущие значения глобальных переменных **\$Context** и **\$ContextPath**. Так, наш пакет *AVZ\_Package\_1* содержит **BeginPackage["AladjevProcedures"]** и при его загрузке значения указанных переменных принимают следующий вид, а именно:

```
In[3522]:= $ContextPath
Out[3522]= {"AladjevProcedures`", "PacletManager`", "WebServices`", "System`",
                                                    "Global`"}

In[3523]:= MemberQ[Contexts["*"], "AladjevProcedures`"]
Out[3523]= True
In[3524]:= $Packages
Out[3524]= {"AladjevProcedures`", "GetFEKernelInit`", "ResourceLocator`", "WebServices`",
                                                    "PacletManager`", "QuantityUnits`", "System`", "Global`"}

In[3525]:= CNames[x_;/ ContextQ[x], y___] := Module[{a = Names[StringJoin[x, "*"]], b},
    b = Select[a, Quiet[ToString[Definition[ToString[#1]]] != "Null" &];
    If[{y} != {} && PureDefinition[y] === $Failed,
        y = Sort[DeleteDuplicates[Select[a, PureDefinition[#] === $Failed &]]];
    Select[b, Attributes[#] != {Temporary} && ToString[Definition[#]] != "Null" &]]

In[3526]:= CNames["AladjevProcedures`"]
Out[3526]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts", ...,
            "$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}

In[3527]:= Length[%]
Out[3527]= 582
In[3528]:= CNames["AladjevProcedures`", h]; h
Out[3528]= {"a", "b", "c", "k", "x"}
In[3529]:= CNames["System`"]
Out[3529]= {"AbelianGroup", "AbortKernels", "AbortProtect", "Abs", ..., "$VersionNumber"}
```

При этом, в приведенном фрагменте вместо вывода *полного* списка, определяемого по вызову **Contexts["\*"]**, в целях экономии места проведено лишь тестирование наличия в нем указанного контекста. Из полного же списка **Contexts["\*"]** легко усматривается, что в нем наряду с данным контекстом в качестве элементов имеются элементы вида *"AladjevProcedures`Name"*, определяющие полные имена всех объектов, *определения* которых находятся в пакете *AVZ\_Package\_1*. Тогда как представленная в завершении фрагмента процедура **CNames** позволяет дифференцированно получать списки всех



кратких имен в пакете с заданным контекстом как имеющих определения в нем, так и неопределенных с точки зрения текущего сеанса. Так, вызов **CNames[x]** возвращает список всех *кратких* имен в пакете с контекстом *x*, имеющих в нем определения, тогда как вызов **CNames[x, y]** дополнительно через аргумент *y* – *неопределенную переменную* – возвращает список всех *неопределенных* кратких имен в пакете с контекстом *x*. Наряду с этим, анализ списка, возвращаемого через необязательный аргумент *y*, обеспечивает дополнительную возможность проверки содержимого пакета на определенность всех объектов, запланированных для обеспечения анонсируемых его функций. **CNames** обеспечивает простой способ дифференцированного анализа содержимого пакетов, оформленных в форме документов *Mathematica* форматов {"nb", "cdf"}.

После загрузки пакета его контекст добавляется к *списку*, определяемому переменной **\$ContextPath**, поэтому каждое обращение к короткому имени объекта в таком пакете автоматически разыскивает его через активные контексты текущего сеанса. В случае, если два символа с одинаковыми короткими именами находятся в двух загруженных различных пакетах (*контекстах*), то ввод такого короткого имени инициирует вывод соответствующего сообщения о перекрываемых символах. Поэтому доступ к данным перекрытым символам возможен лишь по их *полным именам*, т.е. с контекстом. Между тем, конфликтные ситуации могут возникать и в случае, если определяется символ в текущем контексте, то такой символ может перекрываться другим символом с тем же самым кратким именем в уже загруженных пакетах. Причина этого состоит в том, что *Mathematica* ищет символ сначала в контекстах, определяемых списком **\$ContextPath**, и только после этого в текущем контексте.

Механизм *контекстов* имеет ряд существенных особенностей, которые необходимо иметь в виду при работе в среде системы, прежде всего, использовании процедурной парадигмы. В частности, после загрузки пакета в текущий сеанс *все* его объекты будут ассоциированы с контекстом, приписанным пакету, тогда как одноименные объекты, чьи определения вычислены в текущем сеансе ассоциируются с контекстом "Global`". Для определения контекстов символов можно использовать процедуру **ContextDef[x]**, вызов которой возвращает список контекстов, ассоциированных с символом *x*. Если *x* не ассоциирован с контекстом, то возвращается пустой список. Следующий фрагмент представляет исходный код процедуры **ContextDef** с примерами ее использования.

```
In[3325]:= Get["GSV.mx"]
In[3326]:= BeginPackage["RansIan`"]
           GSV::usage = "help on GSV."
           Begin["`GSV`"]
           GSV[x_, y_, z_] := Module[{a = 6}, x*y*z + a]
           End[]
           EndPackage[]
Out[3326]= "RansIan`"
Out[3327]= "help on GSV."
Out[3328]= "RansIan`GSV`"
In[3332]:= GSV[x_Integer, y_, z_Integer] := Module[{a = 78}, (x + y + z)*a]
```

```
In[3333]:= ContextDef[x_/; SymbolQ[x]] := Module[{a = $ContextPath, b = ToString[x], c,
    d, k, j = 1}, While[j <= 2, c = {}; k = 1; Quiet[While[k <= Length[a], d = a[[k]] <> b;
    If[! SameQ[ToString[ToExpression["Definition[" <> d <> ""]], "Null"],
    AppendTo[c, d]]; k++]; j++]; c]

In[3334]:= ContextDef[GSV]
Out[3334]= {"RansIa`GSV", "avzransian78`GSV", "Global`GSV"}
In[3335]:= ProcQ[x_, y_] := x*y
In[3336]:= ContextDef[ProcQ]
Out[3336]= {"RansIa`ProcQ", "AladjevProcedures`ProcQ"}
In[3337]:= Definition[avzransian78`ProcQ]
Out[3337]= ProcQ[x_, y_] := x*y
In[3338]:= Definition["Global`GSV"]
Out[3338]= Global`GSV[x_Integer, y_, z_Integer] := Module[{a = 78}, (x + y + z)*a]
In[3339]:= Definition["RansIa`GSV"]
Out[3339]= RansIa`GSV[RansIa`GSV`x_, RansIa`GSV`y_, RansIa`GSV`z_] :=
    Module[{RansIa`GSV`a = 6}, RansIa`GSV`x*RansIa`GSV`y*RansIa`GSV`z +
    RansIa`GSV`a]

In[3340]:= $ContextPath
Out[3340]= {"avzransian78", "RansIa", "AladjevProcedures", "PacletManager",
    "QuantityUnits", "WebServices", "System", "Global"}
```

Таким образом, при использовании *одноименных* объектов, вообще говоря, необходимо во избежание недоразумений ассоциировать их с приписанными им контекстами.

## 10.2. Определение пакетов и их использование в *Mathematica*

Глобальная переменная **\$Packages** определяет список *контекстов*, соответствующих всем пакетам, загруженным в текущий сеанс, как иллюстрирует следующий пример:

```
In[2041]:= $Packages
Out[2041]= {"AladjevProceduresAndFunctions", ..., "System", "Global"}
In[2042]:= Get["C:\\Avz_Package\\Aladjev.m"]; $Packages
Out[2042]= {"Aladjev", "AladjevProceduresAndFunctions", ..., "System", "Global"}
```

Каждая загрузка в текущий сеанс нового пакета добавляет в начало списка **\$Packages** соответствующий ему *контекст*, как иллюстрирует последний пример фрагмента. В целом, при наличии загруженных пакетов их средства вполне можно рассматривать в качестве средств на уровне *встроенных* средств *Mathematica*. Фактически же, довольно существенное число функций, которые встроены в *базовую* систему *Mathematica*, было реализовано в виде пакетов. Между тем, в большинстве релизов системы *Mathematica* требуется предварительная загрузка пакетов для получения доступа к содержащимся в них средствам. Большинство *релизов* *Mathematica* обеспечено стандартным набором пакетов, которые содержат определения весьма большого количества функций. Для их использовать, как правило, такие пакеты необходимо загружать явно. *Mathematica* располагает механизмом как *предварительной* загрузки, так и *автоматической* загрузки

пакетов по мере необходимости. При использовании данного механизма все средства таких пакетов можно рассматривать на уровне *стандартных*, однако здесь имеет место одно весьма существенное обстоятельство – справки по таким пакетным средствам не отражаются в справочной базе *Mathematica*, и к справкам по ним можно обращаться, например, по конструкции *?Name*. Такая организация не совсем удобна, существенно уступая, в частности, подобному механизму справочной системы *Maple* [21-23,25-27].

Основными формами сохранения определений объектов являются *документ (notebook)* и *пакет (package)*, находящиеся в файлах форматов *{nb, cdf}* и *{m, mx}* соответственно. В то же время между ними существует определенное различие. Если загрузка первого в текущий сеанс позволяет работать с ним, как документом (*просматривать, выполнять, редактировать, сохранять*), то пакет предназначен лишь для загрузки в текущий сеанс. Между тем, многие документы частично либо полностью можно рассматривать как пакеты. В частности, для удобства работы с пакетом *AVZ\_Package\_1* он представлен в трех основных платформо-независимых форматах, а именно *{nb, cdf, m}*. Отметим, что бинарные файлы *mx*-формата, оптимизированного под быструю загрузку в текущий сеанс, являются *непереносимыми* как между версиями самой системы *Mathematica*, так и между операционными платформами. Поэтому здесь они не рассматриваются.

**Загрузка пакета в текущий сеанс.** В общем случае типичный пакете обеспечен *двумя* типами символов, определяющими как экспортируемые символы, так и символы для внутреннего использования. Для *различия* эти символы ассоциируются с различными контекстами. Стандартный прием состоит в помещении определений *экспортируемых* символов в контекст с именем *Name`*, который соответствует имени пакета. Тогда, при загрузке пакета он дополняет список *\$ContextPath* для обеспечения вызова символов, находящихся в данном контексте, по их *коротким* именам. Тогда как определения *всех* символов, предназначенных для внутреннего использования, помещаются в контекст с именем *Package`Private`*, который не добавляется в список *\$ContextPath*, не позволяя получать доступ к символам такого контекста по их коротким именам. Как правило, в целях настройки контекстов пакета и *глобальных* переменных *\$Context* и *\$ContextPath* используется стандартная последовательность функций в пакете, а именно:

```

BeginPackage["Package`"] - установка для пакета текущего контекста "Package`;
F1::usage = "Help" - справка по экспортируемому символу F1; в дальнейшем позволяет
                        получать справку по конструкциям ?F1 и Information[F1];
F2::usage = "Help" - справка по экспортируемому символу F2; в дальнейшем позволяет
                        получать справку по конструкциям ?F2 и Information[F2];
=====
Begin["`Private`"] - установка контекста "`Private`" для локальных символов;
F1[args] := Definition1; ... - определения локальных и глобальных символов пакета;
F2[args] := Definition2; ... - определения локальных и глобальных символов пакета;
=====
End[]
EndPackage[] - закрывающая скобка для пакета; одновременно добавляющая в начало
                списка $ContextPath контекст "Package`".
    
```

Предыдущий фрагмент одновременно представляет типичную схему пакета. Пакет, представленный ниже, служит иллюстрацией-наполнением данной схемы, а именно:

```
In[2040]:= BeginPackage["Kherson`"]
      G::usage = "Function G[x, y] := 71*x^2 + 66*y + 46 + S[x, y]."
```

```
      Begin["Private`"]
      S[x_, y_] := x^3 + y^3
      G[x_;/ IntegerQ[x], y_;/ IntegerQ[y]] := 71*x^2 + 66*y + 46 + S[x, y]
      End[]
      EndPackage[]
```

```
Out[2040]= "Kherson`"
Out[2041]= "Function G[x, y] := 70*x^2 + 65*y + 45 + S[x, y]."
```

```
Out[2042]= "Kherson`Private`"
Out[2045]= "Kherson`Private`"
```

```
In[2047]:= {S[78, 460], G[78, 460]}
Out[2047]= {S[75, 450], 631170}
In[2048]:= $ContextPath
Out[2048]= {"Kherson`", "AladjevProceduresAndFunctions`", "PacletManager`",
      "WebServices`", "System`", "Global`"}
```

```
In[2049]:= $Context
Out[2049]= "Kherson`"
```

```
In[2050]:= Information[S]
Out[2050]= Kherson`S
In[2051]:= Information[G]
Out[2051]= Function G[x, y] := 71*x^2 + 66*y + 46 + S[x, y].
      G[Kherson`Private`x_;/ IntegerQ[Kherson`Private`x],
      Kherson`Private`y_;/ IntegerQ[Kherson`Private`y]] :=
      71 Kherson`Private`x^2 + 66 Kherson`Private`y + 46 +
      Kherson`Private`S[Kherson`Private`x, Kherson`Private`y]
```

```
In[2052]:= Kherson`Private`S[78, 460]
Out[2052]= 97810552
```

```
In[2053]:= $Packages
Out[2053]= {"Kherson`", "AladjevProceduresAndFunctions`", "ResourceLocator`",
      "DocumentationSearch`", "GetFEKernelInit`", "JLink`", "PacletManager`",
      "WebServices`", "System`", "Global`"}
```

Отметим, определение справок (*usage*) для средств, экспортируемых пакетом, служит своего рода *индикатором* того, что именно данные средства *экспортируются* пакетом, тогда как определения средств без справок определяют *локальные* символы, которые вне пакета невидимы, однако могут использоваться как *локальными*, так и *глобальными* символами пакета. Такая организация более проста и в ряде случаев несколько более предпочтительна. Ввиду этого организационная схема пакета может быть упрощена, приняв достаточно простой вид, а именно:

<p><b>BeginPackage["Package`"]</b> – установка для пакета текущего контекста "Package`";  <b>F::usage = "Текст справки"</b> – справки по экспортируемым символам <i>F</i>; в дальнейшем они позволяют получать справки по конструкциям <b>?F</b> и <b>Information[F]</b>;  <b>Begin["`F`"]</b> – установка контекста "`F`" для глобального символа;  <b>F[Formal args] = Definition F; ...</b> – определение глобального символа <i>F</i> пакета;  <b>V[Formal args] = Definition V1; ...</b> – определение локального символа <i>V</i> пакета;  =====</p>
<p><b>End[]</b>  <b>EndPackage[]</b> – закрывающая скобка для пакета; одновременно добавляющая в начало списка <b>\$ContextPath</b> контекст "Package`".</p>

Таким образом, программирование пакета может быть упрощено путем определения *локальных* переменных без соответствующим им справок, в то время как все *экспорты* пакета определяются соответствующими им справками, как иллюстрирует фрагмент:

<pre> In[2054]:= BeginPackage["Kherson78`"]            G78::usage = "Function G78[x, y] := 71*x^2 + 66*y + 46 + S78[x, y]."</pre>
<pre>            Begin["`G78`"]            S78[x_, y_] := x^4 + y^4            G78[x_/, IntegerQ[x], y_/, IntegerQ[y]] := 71*x^2 + 66*y + 46 + S78[x, y]            End[]            EndPackage[]</pre>
<pre> Out[2054]= "Kherson78`" Out[2055]= "Function G78[x, y] := 71*x^2 + 66*y + 46 + S78[x, y]."</pre>
<pre> Out[2056]= "Kherson78`G78`" Out[2059]= "Kherson78`G78`"</pre>
<pre> In[2061]:= {S78[78, 460], G78[78, 460]} Out[2061]= {S78[78, 460], 44812037426}</pre>
<pre> In[2062]:= \$ContextPath Out[2062]= {"Kherson78`", "Kherson`", "AladjevProcedures`", "PacletManager`",            "WebServices`", "System`", "Global`"}</pre>
<pre> In[2063]:= \$Packages Out[2063]= {"Kherson78`", "Kherson`", "AladjevProcedures`", "ResourceLocator`",            "DocumentationSearch`", "GetFEKernelInit`", "JLink`", "PacletManager`",            "WebServices`", "System`", "Global`"}</pre>
<pre> In[2064]:= Information[S78]            Global`S78</pre>
<pre> In[2065]:= Information[G78]            Function G78[x, y] := 71*x^2 + 66*y + 46 + S78[x, y].            G78[Kherson78`G78`x_/, IntegerQ[Kherson78`G78`x], Kherson78`G78`y_/,            IntegerQ[Kherson78`G78`y]] := 71 Kherson78`G78`x^2 + 66 Kherson78`G78`y +            46 + Kherson78`G78`S78[Kherson78`G78`x, Kherson78`G78`y]</pre>

Вызов стандартной функции **Context[x]** возвращает контекст, который ассоциирован с символом *x*. Между тем, достаточно интересным представляется *m*-файл с пакетом, содержащим заданный контекст. Вызов **FindFileContext[x]** возвращает список файлов с пакетами, содержащими заданный контекст *x*; при отсутствии таких файлов вызов процедуры возвращает *пустой* список, т.е. {}. При этом, вызов **FindFileContext[x,y,...]** с *необязательными* аргументами {*y*,...}, в качестве которых выступают имена в строчном формате устройств внешней памяти прямого доступа, обеспечивает поиск искомых файлов именно на указанных устройствах вместо поиска по всей файловой системе компьютера в случае вызова процедуры с одним фактическим аргументом. Фрагмент представляет исходный код процедуры **FindFileContext** с примерами ее применения.

```
In[2060]:= FindFileContext[x_/, ContextQ[x], y___] := Module[{b = {}, c = "", s = {}, k = 1,
    d = "BeginPackage[" <> StrStr[x] <> "]", j = 1,
    a = If[{y} == {}, Adrive[], {y}], f = "$Kr17_Art24$.txt"},
  While[k <= Length[a], Run["Dir ", a[[k]] <> ":\*.*", "/A/B/O/S > " <> f];
    While[! SameQ[c, EndOfFile], c = Read[f, String];
      If[! DirQ[c] && FileExtension[c] == "m", AppendTo[b, c]]; j++;
      DeleteFile[Close[f]]; c = ""; j = 1; k++; k = 1;
    While[k <= Length[b], c = ToString[ReadFullFile[b[[k]]]];
      If[! StringFreeQ[c, d], AppendTo[s, b[[k]]]; k++]; s]

In[2061]:= FindFileContext["Kherson`"]
Out[2061]= {"C:\\AVZ_Package\\Kherson.m"}
In[2062]:= FindFileContext["Aladjev`"]
Out[2062]= {"C:\\AVZ_Package\\Aladjev.m", "F:\\Kherson\\Aladjev.m"}
In[2063]:= FindFileContext["AvzAgnSvetArtKr`", "F"]
Out[2063]= {}
In[2064]:= FindFileContext["AladjevProcedures`"]
Out[2064]= {"C:\\AVZ_Package\\AVZ_Package_1.m",
  "C:\\RECYCLER\\S-1-5-21-329068152-1645522239-1644491937-1003\\Dc50.m"}
```

Следует иметь в виду, что поиск в рамках всей файловой системы компьютера может потребовать достаточно существенных временных затрат. Более того, поиск искомых файлов *m*-формата производится также в каталоге *Recycle Bin* операционной среды, как весьма наглядно иллюстрирует пример предыдущего фрагмента.

Для определения статуса существования контекста (*отсутствует, текущий без файла, текущий с файлом m-формата, неактивный с файлом m-формата*) может использоваться следующая процедура **FindFileContext1**, чей исходный код с типичными примерами использования представляет нижеследующий фрагмент, а именно:

```
In[1879]:= FindFileContext1[x_/, ContextQ[x]] := Module[{a = FindFileContext[x],
    b = If[MemberQ[$Packages, x], "Current", {}]},
  If[a != {} && ! SameQ[b, {}], {b, a}, If[a != {} && SameQ[b, {}], a,
    If[a == {} && ! SameQ[b, {}], b, {}]]]
```

```
In[1880]:= FindFileContext1["Kherson`"]
Out[1880]= "Current"
In[1881]:= FindFileContext1["AladjevProcedures`"]
Out[1881]= {"Current", {"c:\avz_package\avz_package_1.m"}}
In[1882]:= FindFileContext1["Aladjev`"]
Out[1882]= {"f:\avz_package\aladjev.m"}
In[1883]:= FindFileContext1["RansIanRacRea`"]
Out[1883]= {}
In[1884]:= FindFileContext1["PacletManager`"]
Out[1884]= {"Current", {"C:\Program Files\Wolfram Research\Mathematica\9.0\SystemFiles\Autoload\PacletManager\PacletManager.m"}}
```

В зависимости от статуса контекста  $x$  вызов процедуры **FindFileContext1**[ $x$ ] возвращает:

- {"Current",  $\{m\text{-файлы}\}$ } - контекст  $x$  является текущим и находится в  $m$ -файлах;
- "Current" - контекст  $x$  является текущим, но не ассоциирован с  $m$ -файлами;
- $\{m\text{-файлы}\}$  - контекст  $x$  находится в  $m$ -файлах, но не в списке **\$Packages**;
- {} - контекст  $x$  является формально корректным, но не фактическим.

Данная процедура представляется достаточно полезной в работе с пакетами. Между тем, существенно используя предыдущую процедуру **FindFileContext**, реализующую поиск  $m$ -файлов с заданным контекстом по всей файловой системе компьютера, она, как правило, может потребовать достаточно существенных временных затрат. Ниже будет представлен ряд полезных процедур для работы с пакетами и их контекстами.

В определенном смысле *обратными* к процедуре **FindFileContext** являются процедуры **ContextMfile** и **ContextNBfile**, чьи успешные вызовы **ContextMfile**[ $x$ ], **ContextNBfile**[ $x$ ] возвращают *контекст*, ассоциированный с пакетом, находящимся в файле форматов соответственно  $m$ ,  $\{nb, cdf\}$ , заданном именем либо полным путем к нему. Следующий фрагмент представляет исходные коды процедур **ContextMfile**, **ContextNBfile** наряду с наиболее типичными примерами их использования.

```
In[2289]:= ContextMfile[x_;/ FileExistsQ[x] && FileExtension[x] == "m"] := Module[{b,
    a = ReadFullFile[x, c], b = Substring[a, {"BeginPackage[\"", "\"}"]];
    c = If[b != {}, StringTake[b, {14, -2}]];
    If[b === {}, $Failed, c = Flatten[StringSplit[c, ","]];
    c = Select[Quiet[ToExpression[c]], ContextQ[#] &]; If[Length[c] > 1, c, c[[1]]]}];
In[2290]:= ContextMfile["D:\AVZ_Package\AVZ_Package.m"]
Out[2290]= "AladjevProceduresAndFunctions`"
In[2291]:= ContextMfile["D:\AVZ_Package\RansIan.m"]
Out[2291]= $Failed
In[2092]:= ContextMfile["C:/AVZ_Package/AVZ_Package_1.m"]
Out[2092]= "AladjevProcedures`"
In[2093]:= ContextMfile["C:/temp\A A A\Aladjev.m"]
Out[2093]= "Aladjev`"
```

```

In[2210]:= ContextNBfile[x_;/; FileExistsQ[x] && MemberQ[{"cdf", "nb"},
                                         FileExtension[x]] := Module[{a = ""},
  While[! SameQ[a, EndOfFile], a = Read[x, String]; If[! StringFreeQ[a, "BeginPackage"],
    a = Quiet[ToExpression[ToExpression[StringSplit[a, ","][[3]]]];
    Break[]; Continue[]; Close[x];
  If[! ContextQ[a] || SameQ[a, EndOfFile], $Failed, a]]

In[2211]:= ContextNBfile["D:\\AVZ_PACKAGE\\AVZ_Package.nb"]
Out[2211]= "AladjevProceduresAndFunctions`"
In[2212]:= ContextNBfile["D:\\AVZ_PACKAGE\\Book_3.nb"]
Out[2212]= $Failed
In[2213]:= ContextNBfile["D:\\AVZ_PACKAGE\\AVZ_Package.cdf"]
Out[2213]= "AladjevProceduresAndFunctions`"
In[2214]:= ContextNBfile["C:/AVZ_Package/AVZ_Package_1.nb"]
Out[2214]= "AladjevProcedures`"
In[2215]:= ContextNBfile["C:/Temp/A A A\\AVZ_Package.nb"]
Out[2215]= "AladjevProceduresAndFunctions`"

```

Завершает фрагмент процедура **ContextNBfile**, подобная процедуре **ContextMfile**, но ориентированная на пакеты пользователя, оформленные файлами типа {"cdf", "nb"}, внутренняя организация которых отличается от организации *m*-файлов с пакетами. Вызов процедуры **ContextNBfile[x]** возвращает контекст, ассоциированный с данным пакетом, находящимся в файле *x* формата {"cdf", "nb"}, заданном именем или полным путем к нему. Если файл *x* не содержит контекста, вызов процедуры **ContextNBfile[x]** возвращает **\$Failed**. Обе процедуры имеют ряд достаточно важных приложений.

На основе процедуры **ContextMfile** для тестирования системных пакетов (*m*-файлов), находящихся в каталоге, определяемом переменной **\$InstallationDirectory**, довольно полезна процедура **SystemPackages**, чей вызов **SystemPackages[]** возвращает список, в котором 2-элементные подписки имеют формат {Пакет, его контекст}, в то время как вызов **SystemPackages[j]** через необязательный аргумент – неопределенную переменную – дополнительно возвращает список системных пакетов, не обладающих контекстами, т.е. используемых для внутренних нужд системы. Фрагмент приводит исходный код процедуры **SystemPackages** наряду с типичными примерами ее использования.

```

In[2078]:= SystemPackages[y___] := Module[{a, b = Directory[]},
  a = FileNames["*.m", SetDirectory[$InstallationDirectory], Infinity]; SetDirectory[b];
  b = Quiet[DeleteDuplicates[Map[{FileBaseName[#], ContextMfile[#]} &, a]]];
  b = Select[b, # != {} &];
  If[{y} != {} && ! HowAct[y], y = Select[Map[If[SameQ[#[[2]], $Failed], #[[1]]] &, b],
    ! SameQ[#, Null] &]; Select[b, ! SameQ[#[[2]], $Failed] &]]

In[2079]:= SystemPackages[]
Out[2079]= {"Common", {"AuthorTools`Common`, "AuthorTools`MakeProject`"},
  {"DiffReport", {"AuthorTools`DiffReport`, "AuthorTools`Common`"}},

```



```

{"Experimental", "AuthorTools`Experimental`"},
=====
{"DateString", "XMLSchema`DateString`"}, {"XMLSchema", "XMLSchema`"}}
In[2080]:= Length[%]
Out[2080]= 188
In[2081]:= SystemPackages[Sv]; Sv
Out[2081]= {"init", "PacletInfo", "Default", "DemonstrationsTools", "Information", "autoload",
            "Preferences", "sysload", "Interface", "SysInfo", "PredictiveInterfaceLoader",
            =====
            "JLinkCommon", "MakeNETObject", "MathKernel", "NETBlock", "NET",
            "UI", "TerraService", "WebServicesNavigator", "Implementation", "WSDL"}
In[2082]:= Length[%]
Out[2082]= 289
In[2083]:= t = TimeUsed[]; SystemPackages[Kr]; Kr; TimeUsed[] - t
Out[2083]= 36.235

```

Ввиду того, что, в частности, каталог, содержащий инсталлированную *Mathematica 9*, содержит **17095** файлов различного типа, то их тестирование требует определенных временных затрат, как иллюстрирует последний пример предыдущего фрагмента. В то же время следует иметь ввиду, что в виду сказанного доступ к внутренним пакетам системы *Mathematica* посредством механизма контекстов невозможен.

Для получения доступа к пакетным средствам необходимо, чтобы пакет, содержащий их, был загружен в текущий сеанс, а список **\$ContextPath** должен включать контекст, соответствующий данному пакету. Пакет можно загружать в любом месте документа по функции **Get["context"]**, либо по функции **Needs["context"]** определять загрузку пакета, если контекст, ассоциированный с пакетом, отсутствует в списке **\$Packages**. В случае, если пакет начинается **BeginPackage["Package"]**, то при его загрузке в списки **\$ContextPath** и **\$Packages** помещается лишь контекст **"Package"**, обеспечивая доступ к *экспортам* пакета и системным средствам. Если же пакет использует средства других пакетов, он должен начинаться **BeginPackage["Package", {"Package1", ..., "Package2"}]** с указанием списка контекстов, ассоциированных с такими пакетами. Это позволяет дополнительно включать в списки **\$ContextPath** и **\$Packages** требуемые контексты.

Пакет подобно процедурам допускают *вложенность*; при этом, в системе распознаются и регистрируются *все* входящие в него *подпакеты*. Более того, объекты, определенные как в главном пакете, так и в подпакетах, *полностью* доступны в текущем сеансе после загрузки в него вложенного пакета, как довольно наглядно иллюстрирует следующий весьма простой фрагмент. Между тем, для выполнения вышесказанного необходимо после загрузки вложенного пакета *переопределить* переменную **\$ContextPath**, добавив в определяемый ею список все контексты подпакетов главного пакета, а именно:

```

In[2050]:= BeginPackage["Kiev"]
           W::usage = "Help on W."
           Begin["`W`"]

```

```
W[x_Integer, y_Integer] := x^2 + y^2
End[]
BeginPackage["Kiev1`", {"Kiev`"}]
W1::usage = "Help on W1."
Begin["`W1`"]
W1[x_Integer, y_Integer] := x*y + W[x, y]
End[]
EndPackage[]
EndPackage[]
```

```
Out[2050]= "Kiev`"
Out[2051]= "Help on W."
Out[2052]= "Kiev`W`"
Out[2054]= "Kiev`W`"
Out[2055]= "Kiev`"
Out[2056]= "Help on W1."
Out[2057]= "Kiev`W1`"
Out[2059]= "Kiev`W1`"
In[2062]:= $ContextPath = AppendTo[$ContextPath, "Kiev1`"]
Out[2062]= {"Kiev`", "AladjevProcedures`", "System`", "Kiev1`"}
In[2063]:= $Packages
Out[2063]= {"Kiev1`", "Kiev`", "AladjevProcedures`", "GetFEKernelInit`", "ResourceLocator`",
"PacketManager`", "QuantityUnits", "WebServices`", "System`", "Global`"}
In[2064]:= CNames["Kiev`"]
Out[2064]= {"W"}
In[2065]:= CNames["Kiev1`"]
Out[2065]= {"W1"}
In[2066]:= {W[42, 71], W1[42, 71]}
Out[2066]= {6805, 9787}
In[2067]:= Definition[W]
Out[2067]= W[Kiev`W`x_Integer, Kiev`W`y_Integer] := Kiev`W`x^2 + Kiev`W`y^2
In[2068]:= Definition[W1]
Out[2068]= W1[Kiev`W1`x_Integer, Kiev`W1`y_Integer] := Kiev`W1`x Kiev`W1`y +
W[Kiev`W1`x, Kiev`W1`y]
```

После вычисления *определения* пользовательского пакета любого уровня вложенности его можно сохранять в файлах следующих трех системных форматов, а именно:

**File.nb** – файл со *стандартным* документом (*notebook*) системы *Mathematica*; более того, имеется возможность конвертации такого файла в файлы 9 форматов, включая {*cdf, m*};

**File.m** – файл с пакетом исходного формата системы *Mathematica*;

**File.mx** – файл с пакетом формата *DumpSave* системы *Mathematica*; оптимизированный под используемую операционную платформу (как правило, *MacOSX, Unix, Windows*).

Как уже отмечалось выше, объекты, определенные в *главном* пакете и в его *подпакетах*, полностью доступны в текущем сеансе после загрузки в него *главного* пакета, а также

переопределения переменной **\$ContextPath** добавлением в список, определяемый ею, *всех* контекстов, ассоциированных с *подпакетами* главного пакета. Автоматизирует эту задачу процедура, чей вызов **ToContextPath[x]** обеспечивает обновление содержимого текущего списка, определяемого переменной **\$ContextPath**, путем дописывания в его начало всех контекстов из *m*-файла *x*, содержащего простой или вложенный пакет. В следующем фрагменте приведен исходный код процедуры с примером применения.

```
In[2082]:= ToContextPath[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] := Module[{c,
    a = ReadFullFile[x], b = "BeginPackage["],
    c = StrSymbParity[a, b, "[", "]"]];
    c = Map[If[! StringFreeQ[#, {"\""}, {"\""}]], StringTake[#, {14, -2}] &, c];
    c = ToExpression[Flatten[Map[StringSplit[#, ","] &, c]]];
    c = DeleteDuplicates[Map[If[ListQ[#, #][[1]], #] &, c]];
    $ContextPath = DeleteDuplicates[c, Join[$ContextPath]]; $ContextPath]

In[2083]:= ToContextPath["C:\\AVZ_Package\\Kiev.m"]
Out[2083]= {"Kiev`", "Kiev1`", "AladjevProcedures`", "System`"}
```

Успешный вызов **ToContextPath[x]** процедуры возвращает обновленное значение для переменной **\$ContextPath**. С учетом сказанного, рекомендуется производить загрузку вложенного пакета *x* (*m*-файла) в текущий сеанс системы следующей парой вызовов, а именно **Get[x]; ToContextPath[x]**, обеспечивая доступ ко *всем* средствам пакета *x*. Выше представлен пример для ранее приведенного вложенного пакета *"Kiev.m"*.

По функции **Get["Имя"] Mathematica**, прежде всего, делает попытку автоматически загрузить версию файла *"Имя.mx"*, который оптимизирован для текущей платформы, если же такой файл не найден, делается попытка загрузить файл *"Имя.m"*, который содержит код, переносимый на другие платформы. При этом, предполагается, что *m*-файл с пакетом должен располагаться в одном из каталогов, определяемых системной переменной **Path**. Если используется имя каталога, делается попытка считать *m*-файл *"init.m"*, предназначенный для настройки пакетов каталога. Для обеспечения режима автоматической загрузки пакетов используется системная функция **DeclarePackage**. В то же время для удаления символов некоторого контекста, точнее, экспортов пакета с данным контекстом, используется вызов нашей процедуры **RemovePackage["Name"]**.

Как отмечалось ранее, для каждого экспортируемого объекта пакета для него следует определить справку (*usage*). В результате загрузки такого пакета в текущий сеанс все его *экспорты* становятся доступными, тогда как локальные объекты, расположенные в секции, скажем *Private*, в текущем сеансе *недоступны*. Для тестирования загруженного в текущий сеанс пакета или незагруженного пакета, находящегося в *m*-файле данных, на предмет наличия в нем *глобальных* и *локальных* объектов может быть использована процедура, вызов которой **DefInPackage[x]**, где *x* определяет файл или полный путь к нему, или контекст, ассоциированный с пакетом, возвращает вложенный список, чей первый элемент определяет контекст пакета, второй – список локальных переменных и третий – список глобальных переменных пакета *x*. Если *x* не определяет пакет либо контекст, вызов **DefInPackage[x]** возвращается невычисленным. В случае нереального

контекста  $x$  вызов процедуры возвращает **\$Failed**. Фрагмент представляет исходный код процедуры **DefInPackage** с наиболее типичными примерами ее использования.

```
In[2084]:= BeginPackage["Kherson`"]
Gs::usage = "Help on Gs."
Ga::usage = "Help on Ga."
Vgs::usage = "Help on Vgs."
Begin["`Private`"]
W[x_, y_] := x + y
Vt[y_] := y + Sin[y]
Sv[x_] := x^2 + 23*x + 16
End[]
Begin["`Gs`"]
Gs[x_Integer, y_Integer] := x^2 + y^2
End[]
Begin["`Ga`"]
Ga[x_Integer, y_Integer] := x*y + Gs[x, y]
End[]
Begin["`Vgs`"]
Vgs[x_Integer, y_Integer] := x*y
End[]
EndPackage[];
```

```
Out[2084]= "Kherson`"
Out[2085]= "Help on Gs."
Out[2086]= "Help on Ga."
Out[2087]= "Help on Vgs."
Out[2088]= "Kherson`Private`"
Out[2092]= "Kherson`Private`"
Out[2093]= "Kherson`Gs`"
Out[2095]= "Kherson`Gs`"
Out[2096]= "Kherson`Ga`"
Out[2099]= "Kherson`Ga`"
Out[2099]= "Kherson`Vgs`"
Out[2101]= "Kherson`Vgs`"
```

```
In[2103]:= Map[FunctionQ, {Ga, Gs, Vgs, W, Vt, Sv}]
Out[2103]= {True, True, True, False, False, False}
```

```
In[2104]:= BeginPackage["Kherson1`"]
Gs1::usage = "Help on Gs1."
Ga1::usage = "Help on Ga1."
Begin["`Gs1`"]
Gs1[x_Integer, y_Integer] := x^2 + y^2
End[]
Begin["`Ga1`"]
```

```

Ga1[x_Integer, y_Integer] := x*y + Gs1[x, y]
End[]
EndPackage[];

Out[2104]= "Kherson1`"
Out[2105]= "Help on Gs1."
Out[2106]= "Help on Ga1."
Out[2107]= "Kherson1`Gs1`"
Out[2109]= "Kherson1`Gs1`"
Out[2110]= "Kherson1`Ga1`"
Out[2112]= "Kherson1`Ga1`"

In[2222]:= StringDependAllQ[s_String, a_ /; StringQ[a] || ListQ[a] &&
! MemberQ[Map[StringQ, a], False] := DeleteDuplicates[
Map[StringFreeQ[s, #] &, If[StringQ[a], {a}, a]]] == {False}

In[2223]:= Map3[StringDependAllQ, "abcnq", {{ "a", "b", "n", "q"}, {"a", "x", "y"}, {"y", "x", "h"}}]
Out[2223]= {True, False, False}
In[2224]:= Map[! StringFreeQ["abcnq", #] &, {{ "a", "b", "n", "q"}, {"a", "x", "y"}, {"y", "x", "h"}}]
Out[2224]= {True, True, False}
In[2225]:= StringDependAllQ["abcdmnpq", "a"]
Out[2225]= True

In[2228]:= MfilePackageQ[x_] := If[FileExistsQ[x] && FileExtension[x] == ".m",
StringDependAllQ[ReadFullFile[x], {"(* ::Package:: *)", "(* ::Input:: *)",
"::usage", "BeginPackage[\"", "EndPackage[\"]"], False]

In[2229]:= MfilePackageQ["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2229]= True
In[2230]:= Map[MfilePackageQ, {"C:\\AVZ_Package\\66.nb", "Avz.agn"}]
Out[2230]= {False, False}

In[2315]:= DefInPackage[x_ /; MfilePackageQ[x] || ContextQ[x]] := Module[{a, d, p, g, t,
b = {"Begin[\"", "\"\""], c = "BeginPackage[\", k = 1, f, n = x},
Label[Avz];
If[ContextQ[n] && Contexts[n] != {}, f = "$Kr17Art24$"; Save[f, x];
g = FromCharacterCode[17]; t = n <> "Private`";
a = ReadFullFile[f, g]; DeleteFile[f]; d = CNames[n];
p = SubsString[a, {t, g}]; p = DeleteDuplicates[Map[StringCases[#,
t ~~ Shortest[___] ~~ "[" <> t ~~ Shortest[___] ~~ " := "]] &, p]];
p = Map[StringTake[#, {StringLength[t] + 1,
Flatten[StringPosition[#, "["]][[1]] - 1}] &, Flatten[p]];
{n, DeleteDuplicates[p], d},
If[FileExistsQ[n], a = ReadFullFile[n];
f = StringTake[SubsString[a, {c, "\"\""}, {15, -3}][[1]]];

```

```

If[MemberQ[$Packages, f], n = f; Goto[Avz]]; b = StringSplit[a, "*)(*");
d = Select[b, ! StringFreeQ[StringReplace[#, " " -> ""], "::usage="] &];
d = Map[StringTake[#, {1, Flatten[StringPosition[#, "::"]][[1]] - 1}] &, d];
p = DeleteDuplicates[Select[b, StringDependAllQ[#, {"Begin[\"", "\"]"}] &];
p = MinusList[Map[StringTake[#, {9, -4}] &, p], {"Private"}];
t = Flatten[StringSplit[SubsString[a, {"Begin[\"`Private`\""], "End[]"}], "*)(*");
If[t == {}, {f, MinusList[d, p], p}, g = Map[StringReplace[#, " " -> ""] &, t[[2 ;; -1]]];
g = Select[g, ! StringFreeQ[#, "!="] &];
g = Map[StringTake[#, {1, Flatten[StringPosition[#, "="][[1]] - 1}] &, g];
g = Map[Quiet[Check[StringTake[#, {1, Flatten[StringPosition[#, "["][[1]] - 1], #]] &, g];
{f, g, d}], $Failed]]]

```

```

In[2316]:= DefInPackage["Kherson1`"]
Out[2316]= {"Kherson1`", {}, {"Ga1", "Gs1"}}
In[2317]:= DefInPackage["C:\\AVZ_Package\\Kiev.m"]
Out[2317]= {"Kiev`", {}, {"W", "W1"}}
In[2318]:= DefInPackage["C:\\AVZ_Package\\Kherson1.m"]
Out[2318]= {"Kherson1`", {"W1", "Vt1", "Sv1"}, {"Gs1", "Ga1", "Vgs1"}}
In[2319]:= DefInPackage["C:\\AVZ_Package\\Kherson.m"]
Out[2319]= {"Kherson`", {"Vt", "Sv", "W"}, {"Ga", "Gs", "Vgs"}}
In[2320]:= DefInPackage["Kherson`"]
Out[2320]= {"Kherson`", {"Vt", "Sv", "W"}, {"Ga", "Gs", "Vgs"}}

```

В целях упрощения алгоритма процедуры **DefInPackage** выявилась целесообразность дополнительного определения двух довольно простых функций, а именно. Функция **StringDependAllQ** расширяет стандартную конструкцию **!StringFreeQ** на тот случай, когда требуется тестирование принадлежности строке всех подстрок из заданного их списка. Вызов **StringDependAllQ[s, a]** возвращает **True** лишь в том случае, если строка *a* является подстрокой строки *s*, или каждая строка из списка *a* содержится в строке *s*. Тогда как вызов функции **MfilePackageQ[x]** возвращает **True** лишь в том случае, если строка *x* определяет реальный файл *m*-формата, являющийся стандартным пакетом. Предыдущий фрагмент представляет исходные коды обеих функций с примерами.

Предполагается, что локальные символы пакета располагаются в его секции *Private*, что является довольно устоявшимся соглашением. Между тем, *локальными* объектами пакета выступают также и те, для которых в его определении не определены справки (*usage*). Процедура **DefInPackage** успешно обрабатывает пакеты с другими именами локальных секции либо без таких секции вовсе, т.е. определения локальных символов распагаются в пакете произвольно. Оставляем анализ алгоритма данной процедуры в качестве весьма полезного упражнения для заинтересованного читателя.

В целом ряде случаев возникает необходимость полного удаления из текущего сеанса загруженного в него пакета. Частично данную задачу решают стандартные функции **Clear** и **Remove**, однако они не очищают от пакетной информации списки **\$Packages**,

**\$ContextPath** и **Contexts[]**. Данную задачу решает процедура **RemovePackage**, вызов которой **RemovePackage[x]** возвращает **Null**, т.е. ничего, полностью удаляя из сеанса пакет, определяемый контекстом *x*, включая все экспорты пакета *x* и соответственно обновив указанные системные списки. Следующий фрагмент представляет исходный код процедуры **RemovePackage** с наиболее типичными примерами ее применения.

```
In[2288]:= RemovePackage[x_/: ContextQ[x]] := Module[{a = CNames[x],
               b = ClearAttributes[{$Packages, Contexts}, Protected]],
  Quiet[Map[Remove, a]]; $Packages = Select[$Packages, StringFreeQ[#, x] &];
  Contexts[] = Select[Contexts[], StringFreeQ[#, x] &];
  SetAttributes[{$Packages, Contexts}, Protected];
  $ContextPath = Select[$ContextPath, StringFreeQ[#, x] &]; ]

In[2289]:= $ContextPath
Out[2289]= {"Kherson1`", "Kherson`", "AladjevProcedures`", "PacletManager`",
            "QuantityUnits`", "WebServices`", "System`", "Global`"}

In[2290]:= $Packages
Out[2290]= {"Kherson1`", "Kherson`", "AladjevProcedures`", "GetFEKernelInit`",
            "ResourceLocator`", "PacletManager`", "QuantityUnits`", "WebServices`",
            "System`", "Global`"}

In[2291]:= Contexts[]
Out[2291]= {"AladjevProcedures`", "AladjevProcedures`ActBFMuserQ`",
            "AladjevProcedures`ActRemObj`", "AladjevProcedures`ActUcontexts`",
            "AladjevProcedures`Adrive1`", "AladjevProcedures`Affiliate`",
            "AladjevProcedures`Aobj`", "AladjevProcedures`Args`",
            "AladjevProcedures`ArgsBFM`", "AladjevProcedures`ArgsTypes`",
            "AladjevProcedures`Arity`", "AladjevProcedures`ArrayInd`",
            "AladjevProcedures`AssignL`", "AladjevProcedures`AssignToList`", ...}

In[2292]:= RemovePackage["AladjevProceduresAndFunctions"]
In[2293]:= $ContextPath
Out[2293]= {"Kherson`", "AladjevProcedures`", "PacletManager`", "QuantityUnits`",
            "WebServices`", "System`", "Global`"}

In[2294]:= $Packages
Out[2294]= {"Kherson`", "AladjevProcedures`", "GetFEKernelInit`", "ResourceLocator`",
            "PacletManager`", "QuantityUnits`", "WebServices`", "System`", "Global`"}

In[2295]:= Map[PureDefinition, {"Ga1", "Gs1"}]
Out[2295]= {$Failed, $Failed}
```

Между тем, следует отметить, что загруженные в текущий сеанс пакеты могут иметь одноименные объекты, о чем выводятся сообщения. При этом, активным становится тот объект, чей контекст находится ранее в списке **\$Packages**, как довольно наглядно иллюстрирует следующий фрагмент с применением процедуры **RemovePackage**. В этой связи вызов **RemovePackage[x]** удаляет пакет именно с заданным контекстом *x*.

```

In[2077]:= BeginPackage["Pac1`"]
           W::usage = "Help on W."
           Begin["`W`"]
           W[x_Integer, y_Integer] := x^2 + y^2
           End[]
           EndPackage[]

Out[2077]= "Pac1`"
Out[2078]= "Help on W."
Out[2079]= "Pac1`W`"
Out[2081]= "Pac1`W`"

In[2083]:= BeginPackage["Pac2`"]
           W::usage = "Help on W."
           Begin["`W`"]
           W[x_Integer, y_Integer] := x^3 + y^3
           End[]
           EndPackage[]

Out[2083]= "Pac2`"
           W::shdw: Symbol W appears in multiple contexts {Pac2`, Pac1`}; definitions ...
Out[2084]= "Help on W."
Out[2085]= "Pac2`W`"
Out[2087]= "Pac2`W`"
In[2089]:= $Packages
Out[2089]= {"Pac2`", "Pac1`", "AladjevProcedures`", "GetFEKernelInit`", ...}
In[2090]:= W[78, 460]
Out[2090]= 97810552
In[2091]:= Definition[W]
Out[2091]= W[Pac2`W`x_Integer, Pac2`W`y_Integer] := Pac2`W`x^3 + Pac2`W`y^3
In[2092]:= RemovePackage["Pac1`"]
In[2093]:= W[78, 460]
Out[2093]= 97810552
In[2094]:= Definition[W]
Out[2094]= W[Pac2`W`x_Integer, Pac2`W`y_Integer] := Pac2`W`x^3 + Pac2`W`y^3
In[2095]:= RemovePackage["Package2`"]
In[2096]:= Definition[W]
Out[2096]= Null
In[2097]:= $Packages
Out[2097]= "AladjevProcedures`", "GetFEKernelInit`", "ResourceLocator`", ...}

```

Довольно удобный способ сохранения пакетов представляет функция **DumpSave**, чей вызов **DumpSave[f, x]** возвращает контекст  $x$  пакета, сохраняемого в бинарном файле  $f$  в формате, оптимизированном для его последующей загрузки в *Mathematica*. Данная функция имеет четыре формата кодирования, из которых упомянутый представляет особый интерес для сохранения пакетов. Являясь бинарным, данный файл содержит в заголовочной части следующую текстовую информацию, а именно:



(\*This is a Mathematica binary dump file. It can be loaded with Get.\*) ■ Windows ■  
CONT ■ Контекст` ■ ENDCONT ...

В текстовом формате представляются и справки (*usage*) по всем содержащимся в нем средствам. Пакет, сохраненный описанным способом, загружается в текущий сеанс по функции **Get[f]** с автоматической активацией всех содержащихся в нем определений; при этом, корректно загружаются лишь те файлы, которые были сохранены на той же самой вычислительной платформе функцией **DumpSave** системы *Mathematica*.

Относительно файлов *mx*-формата с пакетами пользователя возникает интересная и достаточно полезная задача по определению контекста и объектов, чьи определения находятся в файле данного типа. Решает данную задачу процедура **ContMxFile**, чей исходный код с типичными примерами применения представлен фрагментом.

```
In[2164]:= DumpSave["Package_2013.mx", "AladjevProcedures`"]
Out[2164]= {"AladjevProcedures`"}
In[2165]:= DumpSave["Kherson1.mx", "Kherson1`"]
Out[2165]= {"Kherson1`"}
In[2166]:= DumpSave["Kiev.mx", "Kiev`"]
Out[2166]= {"Kiev`"}

In[2167]:= ContMxFile[x_;/; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d = "", h, t},
h = StringTake[SubsString[a, {b, c}], {5, -8}]; If[h == {}, {}, a = StringReplace[a, h[[1]] -> ""];
h = StringJoin[FromCharacterCode[Select[ToCharacterCode[h[[1]]], # >= 32 &]]];
If[MemberQ[$Packages, h] && {y} != {}, {h, CNames[h]},
If[! MemberQ[$Packages, h] && {y} != {}, Quiet[Get[x]];
{{h, CNames[h]}, RemovePackage[h]}][[1]],
t = SubsString[a, {h, ""}]; t = Select[t, ! MemberQ[ToCharacterCode[#], 0] &];
{h, Sort[DeleteDuplicates[Map[StringReplace[#, {h -> "", "`" -> ""}] &, t]]]}]]

In[2168]:= ContMxFile["Kiev1.mx"]
Out[2168]= {"Kiev1`", {"W", "W1", "W2", "W3"}}
In[2169]:= ContMxFile["Kiev1.mx", 78]
Out[2169]= {"Kiev1`", {"W", "W1"}}
In[2170]:= ContMxFile["Package_2013.mx"]
Out[2170]= {"AladjevProcedures78`", {"ActBFMuserQ", "ActRemObj", "ActUcontexts",
"Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes", "Arity",
"ArrayInd", "AssignL", "AssignToList", "AtomicQ", "Attrib", "Attrib1",
=====
"VizContext", "WhatObj", "WhatType", "WhichN", "$ProcName", "$TypeProc"}}

In[2171]:= Length[%[[2]]]
Out[2171]= 460
In[2172]:= ContMxFile["Kherson1.mx"]
Out[2172]= {"Kherson1`", {"Ga1", "Gs1"}}
```

Вызов процедуры **ContMxFile[x]** возвращает вложенный список, чей первый элемент определяет контекст, ассоциированный с пакетом, содержащимся в *mx*-файле *x*, в то время как второй элемент определяет список имен в строчном формате всех объектов данного пакета безотносительно наличия для них справки (*usage*), т.е. как локальных, так и глобальных объектов. Тогда как вызов процедуры **ContMxFile[x, y]**, где аргумент *y* – произвольное выражение – возвращает вложенный список аналогичной структуры, но с тем отличием, что в качестве его второго элемента выступает список имен объектов данного пакета, снабженных справками (*usage*), т.е. только глобальных объектов.

Между тем, функция **DumpSave** имеет один достаточно существенный недостаток, а именно: сохраняет контексты, которые являются только формально контекстами, т.е. соответствуют им лишь по формату. В данной связи более предпочтительна функция **DumpSaveP[f, x]**, которая обеспечивает сохранение в файле *f* пакета с контекстом *x* с условием, что данный пакет содержит *глобальные* символы; в противном случае вызов функции **DumpSaveP** возвращает **\$Failed**. Фрагмент представляет как исходный код функции **DumpSaveP**, так и наиболее типичные примеры ее использования.

```
In[3942]:= PackageQ[x_]; ContextQ[x] := If[CNames[x] != {}, True, False]
In[3947]:= DumpSaveP[f_String, x_]; ContextQ[x] := If[PackageQ[x], DumpSave[f, x],
                                                    $Failed]

In[3957]:= DumpSave["AVZ_Package.mx", "AladjevProcedures`"]
Out[3957]= {"AladjevProcedures`"}
In[3958]:= RemovePackage["AladjevProcedures`"]
In[3959]:= Map[Definition, {ProcQ, RemovePackage, Mapp, Map14, Definition2, StrStr}]
Out[3959]= {Null, Null, Null, Null, Null, Null}
In[3960]:= Get["AVZ_Package.mx"]
In[3961]:= Definition2[StrStr]
Out[3961]= {StrStr[x_] := If[StringQ[x], StringJoin["\"", x, "\""], ToString[x]], {}
In[3962]:= PackageQ["AvzAgnVsvArtKr`"]
Out[3962]= False
In[3963]:= DumpSave["AvzAgnVsvArtKr.mx", "AvzAgnVsvArtKr`"]
Out[3963]= {"AvzAgnVsvArtKr`"}
In[3964]:= DumpSaveP["AvzAgnVsvArtKr.mx", "AvzAgnVsvArtKr`"]
Out[3964]= $Failed
```

Заголовок функции **DumpSaveP** в качестве теста на допустимость второго аргумента использует логическую функцию, вызов которой **PackageQ[x]** возвращает **True**, если *x* – пакет, содержащий глобальные символы, и **False** в противном случае. Естественно, пакет без *глобальных* символов какого-либо интереса не представляет. В данной связи вызов **DumpSaveP[f, x]**, где *x* не является пакетом, возвращает **\$Failed**, позволяя весьма просто программно обрабатывать ситуации данного типа. Ниже приведен довольно простой фрагмент, иллюстрирующий тестирования формально корректных пакетов на предмет идентификации их в качестве пакетов, определяющих *глобальные символы*, т.е. символы, определяющие объекты, становящиеся *глобальными* при загрузке пакета:

```
In[2076]:= BeginPackage["Kiev`"]
           Avz::usage = "Help on Avz."
           Agn::usage = "Help on Agn."
           EndPackage[]

Out[2076]= "Kiev`"
Out[2077]= "Help on Avz."
Out[2078]= "Help on Agn."
In[2080]:= BeginPackage["Kiev1`"]
           EndPackage[]

Out[2080]= "Kiev1`"
In[2082]:= $Packages
Out[2082]= {"Kiev1`", "Kiev`", "AladjevProcedures`", "GetFEKernelInit`", "ResourceLocator`",
           "PacletManager`", "QuantityUnits`", "WebServices`", "System`", "Global`"}
In[2083]:= Map[PackageQ, {"Kiev`", "Kiev1`"}]
Out[2083]= {False, False}
```

Результат тестирования контекстов {"Kiev1`", "Kiev`"} в простейшего формата пакетах не ассоциирует их с пакетами. Действительно, в соответствии с соглашениями пакет должен определять глобальные символы, без чего он таковым считаться не может.

Довольно актуальной представляется задача определения контекста, ассоциированного с пакетом, находящимся в бинарном *mx*-файле. Задачу решает процедура, чей вызов **PackageMxCont[x]** возвращает контекст *mx*-файла *x*; вызов **PackageMxCont[x, y]** через 2-й необязательный аргумент – неопределенную переменную *y* – возвращает вложенный список, чей первый элемент – список локальных и второй – список глобальных символов пакета *mx*-файла *x*. На *mx*-файлах без контекста или локальных/глобальных символов вызов процедуры **PackageMxCont[x]** возвращает пустой список, т.е. {}.

```
In[2084]:= BeginPackage["Kherson`"]
           Gs::usage = "Help on Gs."
           Ga::usage = "Help on Ga."
           Vgs::usage = "Help on Vgs."
           Begin["`Private`"]
           W[x_, y_] := x + y
           Vt[y_] := y + Sin[y]
           End[]
           Begin["`Gs`"]
           Gs[x_Integer, y_Integer] := x^2 + y^2
           End[]
           Begin["`Ga`"]
           Ga[x_Integer, y_Integer] := x*y + Gs[x, y]
           End[]
           Begin["`Vgs`"]
           Vgs[x_Integer, y_Integer] := x*y
           End[]
           EndPackage[];
```

```

In[2102]:= DumpSave["Kherson.mx", "Kherson`"]
Out[2102]= {"Kherson`"}

In[2104]:= PackageMxCont[x_;/; FileExistsQ[x] && FileExtension[x] == "mx", y___] :=
Module[{a = $Packages, b, c}, Get[x]; If[Length[a] == Length[$Packages],
c = ContMxFile[x, 78]; If[c == {} || c[[2]] == {}, Return[{}], b = c[[1]], b = $Packages[[1]]];
If[{y} != {} && ! HowAct[y] && ! SameQ[b, {}], y = DefInPackage[b][[2 ;; -1]]];
If[! MemberQ[a, b], RemovePackage[b]]; b]

In[2105]:= {PackageMxCont["Kherson.mx", s], s}
Out[2105]= {"Kherson`", {"Vt", "W"}, {"Ga", "Gs", "Vgs"}}

In[2106]:= CurrentPackageQ[x_;/; ContextQ[x]] := MemberQ[$Packages, x]

In[2107]:= CurrentPackageQ["Kherson`"]
Out[2107]= True

In[2108]:= RemovePackage["Kherson`"]
In[2109]:= {PackageMxCont["Kherson.mx", g], g}
Out[2109]= {"Kherson`", {"Vt", "W"}, {"Ga", "Gs", "Vgs"}}
In[2110]:= CurrentPackageQ["Kherson`"]
Out[2110]= False
In[2111]:= PackageMxCont["Kherson.mx"]
Out[2111]= "Kherson`"

In[2112]:= Packages[] := Select[$Packages, Quiet[PackageQ[#]] &]

In[2113]:= Packages[]
Out[2113]= {"AladjevProcedures", "ResourceLocator", "PacletManager", "WebServices",
"System", "Global"}

In[2114]:= {DumpSave["J7.mx", "J7`"], DumpSaveP["J7.mx", "J7`"], ContMxFile["J7.mx"]}
Out[2114]= {"J7`", $Failed, {}}

```

Предыдущий фрагмент представляет код процедуры **PackageMxCont** с примерами. Вызов простой, но полезной функции **CurrentPackageQ[w]** возвращает **True**, если *w* – активный контекст в текущем сеансе *Mathematica*, и **False** в противном случае. Итак, несмотря на *формальную* корректность определения пакетов без экспортируемых ими *глобальных* символов, что *Mathematica* подтверждает помещением соответствующих им контекстов в список, определяемый *глобальной* переменной **\$Packages**, или *mx*-файлов без контекста, необходимо тестирование фактических пакетов, которые загружены в текущий сеанс, что обеспечивает простая функция, чей вызов **Packages[]** возвращает список из контекстов фактических пакетов, загруженных в текущий сеанс системы. В двух предыдущих фрагментах содержатся как примеры двух формально корректных пакетов с контекстами {"Kiev", "Kiev1"}, так и примеры их тестирования и исходный код функции **Packages** с примерами ее использования для определения фактических пакетов, активных в текущем сеансе *Mathematica*. Все вышепредставленные средства включены в наш пакет *AVZ\_Package\_1* [120], имеющий *FreeWare* лицензию. Из опыта использования выявилась его эффективность в разработке целого ряда приложений.

### 10.3. Дополнительные средства работы с пакетами в *Mathematica*

Средства *Mathematica* работы с файлами можно условно подразделить на 2 группы: средства, поддерживающие работу с файлами, автоматически распознаваемыми при обращении к ним, и средства, поддерживающие работу с произвольными файлами данных. Данная тема довольно обширна, более детально рассматривается несколько ниже, здесь же кратко коснемся средств работы с файлами собственно *Mathematica*, т.е. файлами, формат которых автоматически распознаваем системой. В частности, в таких файлах сохраняются определения, результаты вычислений, пакеты, документы и другие объекты. За редким исключением, данные файлы имеют текстовый формат, что позволяет их обрабатывать не только системой и текстовыми процессорами, но и делает их переносимыми с одной вычислительной платформы на другую. Из данных файлов наибольший интерес представляют файлы следующих форматов, а именно:

- file.m*        - платформо-независимый файл с пакетом *Mathematica* на *Math*-языке
- file.nb*       - платформо-независимый файл с документом (*notebook*) *Mathematica*
- file.cdf*      - файл с документом, формата пригодного для плеера *Mathematica*
- file.mx*       - платформо-зависимый оптимизированный бинарный файл *Mathematica*

Имеется еще ряд системных файлов и других форматов, здесь не рассматриваемых. С ними можно ознакомиться как в справке по системе, так и в изданиях, например, [100, 116]. В настоящей книге рассматриваются, в основном, средства, ориентированные на обработку файлов именно указанных форматов. Целый ряд средств такого типа был рассмотрен выше, здесь приведены некоторые полезные дополнительные средства.

Функция **Get["Файл"]** (<< *Файл*) читает *Файл* в текущий сеанс с возвратом последнего содержащегося в нем выражения. При этом, если в качестве файла указан контекст, то обеспечивается поиск файла, ассоциированного с данным контекстом. Функция **Get** может читать файлы форматов {"*.m*", "*.nb*", "*.cdf*"}, однако наиболее оптимальным для загрузки является формат "*.mx*" при условии, что файлы создавались на той же самой операционной платформе посредством функции **DumpSave** или нашей **DumpSaveP**, исключающей помещение в файл пакетов, не соответствующих стандарту. При этом, поиск файла по умолчанию **Get** производит в каталогах, определенных переменной **\$Path**. Между тем, **Get** допускает использование опции *Path*, обеспечивающей режим поиска файла в указанном списке каталогов {*d1, d2, ..., dp*}, а именно: **Get[File, Path -> {*d1, d2, d3, ..., dp*}]**. На период загрузки файла по **Get** глобальные переменные **\$Input** и **\$InputFileName** устанавливаются соответственно в *имя* загружаемого файла и *полный путь* к данному файлу. Детальнее с описанием функции **Get** можно ознакомиться по справочной системе *Mathematica*, несколько выше и в книгах [100,102,116].

Обратной к **Get** является функция **Put** (>>), имеющая три формата кодирования:

- Выражение* >> *Файл*, **Put[Выражение, Файл]** - пишет выражение в заданный *Файл*;
- Put[V1, V2, ..., Vp, Файл]** - пишет выражения {*V1, V2, ..., Vp*} в заданный *Файл*;
- Put[Файл]** - создает пустой *Файл* с указанным именем.

Функция **PutAppend** (>>>) допускает те же *три* формата кодирования, но в отличие

от **Put** она пишет информацию в файл не в режиме полного обновления, а в режиме дополнения (*Append*). По умолчанию обе эти функции пишут выражения во входном формате **InputForm**, завершая их символом перевода строки. Более того, как функция **Get**, так и **Put**, **PutAppend** после успешной обработки файла оставляют его закрытым.

Тогда как вызов функции **FilePrint[f]** на монитор выводит тестовое содержимое файла *f*. Поэтому из файлов упомянутых форматов данной функцией корректно выводятся лишь файлы форматов {"*m*", "*nb*", "*cdf*"}, тогда как *mx*-файл выводится некорректно. Между тем, для файлов достаточно большого размера форматов {"*nb*", "*cdf*"} вполне возможны проблемы вывода, в ряде случаев требующие перезагрузки пакета. В связи с тем, что средства доступа к файлам даже форматов, автоматически распознаваемых пакетом, при их большей развитости, чем, в частности, *Maple* не решают целого ряда достаточно важных задач, пользователь вынужден программировать собственные на основе стандартных и, возможно, с использованием своих средств. В качестве одного полезного примера приведем процедуру **DefFromPackage**, вызов **DefFromPackage[x]** которой возвращает 2-х элементный список, *первый* элемент которого – определение в строчном формате символа *x*, контекст которого отличен от {"Global", "System"}, в то время как *второй* элемент – справка (*usage*) по нему, тогда как *третий* определяет атрибуты символа. На символах, ассоциированных с указанными контекстами, вызов процедуры возвращает только список их атрибутов. Следующий фрагмент приводит исходный код процедуры **DefFromPackage** с типичными примерами ее применения.

```
In[2140]:= DefFromPackage[x_/: SymbolQ[x]] := Module[{a = Context[x], b = "", c = "", p,
                                                    d = ToString[x], k = 1, h},
  If[MemberQ[{"Global", "System"}, a], Return[Attributes[x]], h = a <> d;
  ToExpression["Save[" <> ToString1[d] <> "," <> ToString1[h] <> "]"];
  For[k, k < Infinity, k++, c = Read[d, String]; If[c === " ", Break[], b = b <> c]];
  p = StringReplace[RedSymbStr[b, " ", " "], h <> "" -> ""]; {c, k, b} = {"", 1, ""};
  For[k, k < Infinity, k++, c = Read[d, String];
  If[c === " " || c === EndOfFile, Break[],
  b = b <> If[StringTake[c, {-1, -1}] == "\\ ", StringTake[c, {1, -2}], c]];
  DeleteFile[Close[d]]; {p, StringReplace[b, "/" <> d -> ""], Attributes[x]]

In[2141]:= DefFromPackage[StrStr]
Out[2141]= {StrStr[x_] := If[StringQ[x], StringJoin["\"", x, "\""], ToString[x]],
  StrStr::usage = "The call StrStr[x] returns an expression x in string format if x
  is different from string; otherwise, the double string obtained from an
  expression x is returned.", {}}

In[2142]:= DefFromPackage[AvzAgn]
Out[2142]= {}

In[2143]:= SetAttributes[AgnAvzKr, {Listable, Protected}]; DefFromPackage[AgnAvzKr]
Out[2143]= {Listable, Protected}

In[2144]:= DefFromPackage[Cos]
Out[2144]= {Listable, NumericFunction, Protected}
```

Процедура **DefFromPackage** предназначена для получения полной информации по символу *x*, *определение* которого находится в некотором загруженном в текущий сеанс пакете. В отличие от стандартных функций **FilePrint** и **Definition** данная процедура, *во-первых*, не выводит, а возвращает указанную информацию, полностью доступную для последующей обработки, и, *во-вторых*, информация возвращается в оптимальном формате. В частности, в целом ряде случаев при выводе *определения* символа, которое находится в активном пакете, вывод его стандартными средствами сопровождается и ассоциированным с пакетом контекстом, что не только затрудняет его просмотр, но и последующую обработку. Возврат **DefFromPackage** устраняет и данную проблему.

Алгоритм, реализуемый данной процедурой, базируется на *анализе* структуры файла, полученного в результате сохранения контекста "*y`x*", в котором *x* – символ в вызове **DefFromPackage**[*x*], "*y`*" – контекст, ассоциированный с загруженным в текущий сеанс пакетом, содержащим определение символа *x*. Для создания временного файла "*x*" в текущем каталоге используется функция **Save**, располагающая *четырьмя* форматами, из которых для наших целей наиболее приемлем формат **Save**[*файл*, "*контекст`*"], по которому в указанном *файле* сохраняются определения всех символов, находящихся в заданном *контексте*. Используя знания о структуре созданного файла и стандартную функцию чтения **Read** наряду с другими стандартными функциями, включая и нашу процедуру **RedSymbStr**, рассмотренную в книге, получаем искомый результат. Более детально алгоритм, реализованный процедурой, усматривается из ее исходного кода.

В качестве второго примера, развивающего алгоритм предыдущей процедуры в свете применения функций доступа, можно представить достаточно полезную процедуру **FullCalls**, вызов которой **FullCalls**[*x*] возвращает список, первым элементом которого является контекст, ассоциированный с загруженным в текущем сеансе пакетом, тогда как остальные его элементы – символы из данного пакета, используемые процедурой либо функцией пользователя *x*, либо вложенный список из данного типа подсписков при использовании *x* символов (*имен процедур/функций*) из нескольких пакетов. Ниже представлен исходный код процедуры с типичными примерами ее использования.

```
In[2235]:= FullCalls[x_;/; ProcQ[x] || FunctionQ[x]] := Module[{a = {}, b, c = "::usage = ",
                                                                d, k = 1}, Save[b = ToString[x], x];
  For[k, k < Infinity, k++, d = Read[b, String]; If[d === EndOfFile, Break[],
    If[StringFreeQ[d, c], Continue[], a = Append[a,
      StringSplit[StringTake[d, {1, Flatten[StringPosition[d, c]][[1]] - 1}], "/: "][[1]]];
    a = Select[a, SymbolQ[#] &]; DeleteFile[Close[b]];
    a = Map[{#, Context[#]} &, DeleteDuplicates[a]];
    a = If[Length[a] == 1, a, Map[DeleteDuplicates,
      Map[Flatten, Gather[a, #1[[2]] === #2[[2]] &]]]; {d, k} = {{}, 1};
    While[k <= Length[a], b = Select[a[[k]], ContextQ[#] &];
    c = Select[a[[k]], ! ContextQ[#] &]; d = Append[d, Flatten[{b, Sort[c]}]]; k++;
    d = MinusList[If[Length[d] == 1, Flatten[d], d], {ToString[x]}];
    If[d == {Context[x]}, {}, d]]
```

```

In[2236]:= FullCalls[StrStr]
Out[2236]= {}
In[2237]:= G[x_] := StrStr[x] <> "RansIan78"; FullCalls[G]
Out[2237]= {"AladjevProcedures", "StrStr"}
In[2238]:= F[x_Integer, y_Integer] := x^2 + y^2; FullCalls[F]
Out[2238]= {}
In[2239]:= FullCalls[ProcQ]
Out[2239]= {"AladjevProcedures", "BlockFuncModQ", "ClearAllAttributes", "Contexts1",
  "Definition2", "HeadPF", "HowAct", "ListStrToStr", "Map3", "Mapp", "MinusList",
  "PureDefinition", "Sequences", "StrDelEnds", "SubsDel", "SuffPref", "SymbolQ",
  "SysFuncQ", "SysFuncQ1", "SystemQ", "ToString1", "UnevaluatedQ"}
In[2240]:= FullCalls[Args]
Out[2240]= {"AladjevProcedures", "BlockFuncModQ", "CharacterQ", "ClearAllAttributes",
  "CompileFuncQ", "Contexts1", "Definition2", "FunctionQ", "HeadPF", "HowAct",
  "Ind", "ListStrToStr", "Map3", "Mapp", "MinusList", "OP", "ProcQ", "PureFuncQ",
  "PureDefinition", "QFunction", "QFunction1", "Sequences", "StrDelEnds",
  "SubsDel", "SubStrSymbolParity1", "SuffPref", "SymbolQ", "SysFuncQ",
  "SysFuncQ1", "SystemQ", "ToList", "ToString1", "UnDefVars", "UnevaluatedQ"}
In[2241]:= GS[x_/, RuleQ[x], y_String] := ArtKr[StringLength[StringReplace[y, x]], 460] +
  Vgs[StringLength[y], 78]; FullCalls[GS]
Out[2241]= {"AladjevProcedures", "RuleQ"}, {"Kherson", "ArtKr", "Vgs"}
In[2242]:= GS["Avz" -> "2013", "AgnAvzVsvArtKr"]
Out[2242]= 71.1406

```

Таким образом, вызов процедуры **FullCalls[x]** обеспечивает возможность тестирования процедуры либо функции, отличной от стандартной, на предмет использования ею средств, определения которых находятся в пакетах, загруженных в текущий сеанс. В развитие данной процедуры может быть предложена процедура **FullCalls1**, исходный код которой с примерами использования представлены следующим фрагментом.

```

In[968]:= FullCalls1[x_/, ProcQ[x] | | FunctionQ[x]] := Module[{a = {}, b, c = "", d, k = 1, n, p},
  Save[b = ToString[x], {x, c}]; For[k, k < Infinity, k++, d = Read[b, String];
  If[d === EndOfFile, Break[],
    If[d != " ", c = c <> d,
      If[n = Flatten[StringPosition[c, " := "]]; n != {},
        If[Quiet[HeadingQ[p = StringTake[c, {1, n[[1]] - 1}]]],
          a = Append[a, Quiet[HeadName[StringTake[c, {1, n[[1]] - 1}]]]]; c = ""];
        DeleteFile[Close[b]]; {b = FullCalls[x],
  Select[MinusList[a, {ToString[x]}, ! MemberQ[Flatten[b], #] &]]]
In[969]:= ArtKr[x_Integer, y_Integer] := Module[{}, N[Sqrt[x^2 + y^2]];
  Vgs[x_Integer, y_Integer] := N[Sin[x] + Cos[y]]; GS[x_/, RuleQ[x], y_String] :=
  ArtKr[StringLength[StringReplace[y, x]], 42] + Vgs[StringLength[y], 78];

```



```

In[2163]:= FullCalls1[GS]
Out[2163]= {"AladjevProcedures`", "RuleQ"}, {"Kherson`", "ArtKr", "Vgs"}}
In[2164]:= FullCalls1[StrStr]
Out[2164]= {}, {}
In[2165]:= FullCalls1[ProcQ]
Out[2165]= {"AladjevProcedures`", "BlockFuncModQ", "ClearAllAttributes", "Contexts1",
            "Definition2", "HeadPF", "HowAct", "ListStrToStr", "Map3", "Mapp", "MinusList",
            "PureDefinition", "Sequences", "StrDelEnds", "SubsDel", "SuffPref", "SymbolQ",
            "SysFuncQ", "SysFuncQ1", "SystemQ", "ToString1", "UnevaluatedQ"}, {}
    
```

Процедура **FullCalls1** тестирует процедуру/функцию  $x$  на предмет использования ею как пакетных, так и иных средств, отличных от стандартных. Так, вызов **FullCalls1**[ $x$ ] возвращает вложенный список, первый элемент которой соответствует **FullCalls**[ $x$ ], в то время как второй определяет список имен средств, используемых  $x$ , исключая, при этом, средства, принадлежащие загруженным пользовательским пакетам. Между тем, следует иметь ввиду, что обе процедуры обрабатывают лишь средства, используемые  $x$ , которые определены механизмом отложенных вычислений. Распространение этих процедур на механизм немедленных вычислений каких-либо особых затруднений не вызывает, и такое расширение может представить достаточно полезное упражнение заинтересованному читателю. Мы же исходили из того, что определения процедур и функций по целому ряду довольно существенных причин целесообразнее определять именно механизмом отложенных вычислений. Обе процедуры **FullCalls** и **FullCalls1** оказываются довольно полезными при программировании целого ряда приложений.

Использованная в реализации процедур **FullCalls** и **FullCalls1** функция **Save** может оказаться довольно полезной для организации библиотек пользовательских средств. Действительно, вызов **Save**[ $f$ , { $a$ ,  $b$ ,  $c$ , ...}] сохраняет в файле  $f$  текстового формата все определения не только объектов с именами { $a$ ,  $b$ ,  $c$ , ...}, но и все определения средств, с которыми связаны указанные объекты на всех уровнях их структурных деревьев. В то же самое время, вызов функции производит запись в файл в режиме дописывания (*Append*), оставляя файл закрытым. Созданный файл легко редактируется текстовыми редакторами, позволяя относительно несложно создавать программные средства его редактирования (*удаление объектов, добавление объектов, замену объектов и др. функции*). Для загрузки такой библиотеки в текущий сеанс достаточно вызова **Get**[ $f$ ], обеспечив доступ ко всем средствам, определения которых были сохранены в файле  $f$ . Фрагмент представляет примеры ряда операций с функциями **Save** и **Get**, которые определяют основу *библиотечной* организации программных средств пользователя. Ниже данный вопрос рассматривается несколько детальнее с иллюстрацией примера библиотеки.

```

In[2079]:= ArtKr[x_Integer, y_Integer] := Module[{}, N[Sqrt[x^2 + y^2]];
           Vgs[x_Integer, y_Integer] := N[Sin[x] + Cos[y]]; GS[x_ /; RuleQ[x], y_String] :=
           ArtKr[StringLength[StringReplace[y, x]], 42] + Vgs[StringLength[y], 78];
In[2080]:= Rans[x_, y_] := Module[{}, x + y]; Save["Library.txt", {GS, Rans}]
In[2081]:= {GS["Agn" -> "2013", "AvzAgnVsvArtKr"], Rans[42, 78]}
Out[2081]= {44.731, 120}
    
```

```
In[2082]:= Remove[GS, Rans]
In[2083]:= {GS["Agn" -> "2013", "AvzAgnVsvArtKr"], Rans[42, 78]}
Out[2083]= {GS["Agn" -> "2013", "AvzAgnVsvArtKr"], Rans[42, 78]}
In[2084]:= Get["Library.txt"]
In[2085]:= {GS["Agn" -> "2013", "AvzAgnVsvArtKr"], Rans[42, 78]}
Out[2085]= {44.731, 120}
```

На этом краткое рассмотрение средств доступа в среде системы *Mathematica*, которые позволяют пользователю относительно несложно организовывать свои программные средства, завершается и будет продолжено на более детальном уровне ниже.

В целом ряде случаев возникает необходимость в загрузке в текущий сеанс не пакета, а только отдельного содержащегося в нем средства, например, процедуры/ функции, или их списка. В нижеследующем фрагменте представлена процедура, вызов которой **ExtrOfMfile**[*x*,*y*] ничего не возвращает, загружая в текущий сеанс определения только тех средств, которые определены аргументом *y* и находятся в файле *x* *m*-формата. При этом, в случае наличия в *m*-файле нескольких одноименных средств, в текущий сеанс загружается последнее. Тогда как вызов **ExtrOfMfile**[*x*, *y*, *z*] с третьим необязательным аргументом *z* – *неопределенной переменной* – дополнительно через *z* возвращает список определений средств *y*, находящихся в *m*-файле *x*. В отсутствие в *m*-файле *x* средств *y* вызов процедуры возвращает **\$Failed**. Ниже представлены исходный код процедуры **ExtrOfMfile** наряду с наиболее типичными примерами ее использования.

```
In[2250]:= ExtrOfMfile[f_ /; FileExistsQ[f] && FileExtension[f] == "m", s_ /; StringQ[s] | |
ListQ[s], z_] := Module[{Vsv, p = {}, v, m}, m = ReadFullFile[f];
If[StringFreeQ[m, Map["(*Begin[\"\" <> # <> \"\")"] &, Map[ToString, s]]], $Failed,
Vsv[x_, y_] := Module[{a = m, b = FromCharacterCode[17], c = FromCharacterCode[24],
d = "(*Begin[\"\" <> y <> \"\")", h = "(*End[\"")", g = {}, t}, a = StringReplace[a, h -> c];
If[StringFreeQ[a, d], $Failed, While[! StringFreeQ[a, d], a = StringReplace[a, d -> b, 1];
t = StringTake[SubStrSymbolParity1[a, b, c][[1]], {4, -4}];
t = StringReplace[t, {"(" -> "", ")" -> ""}];
AppendTo[g, t]; a = StringReplace[a, b -> "", 1]; Continue[]];
{g, ToExpression[g][[-1]]}];
If[StringQ[s], v = Quiet[Check[Vsv[f, s][[1]], $Failed]],
Map[{v = Quiet[Check[Vsv[f, #][[1]], $Failed]], AppendTo[p, v]} &, Map[ToString, s]];
If[{z} != {} && ! HowAct[z], z = If[StringQ[s], v, p]; ]

In[2251]:= ExtrOfMfile["C:\\AVZ_Package\\Kiev.m", "W"]
In[2252]:= ExtrOfMfile["C:\\AVZ_Package\\Kiev.m", "W", w]
In[2253]:= {W[71, 78, 460], w}
Out[2253]= {98168463, {"W[x_Integer, y_Integer] := x^2 + y^2",
"W[x_Integer, y_Integer, z_Integer] := x^2 + y^2 + z^2",
"W[x_Integer, y_Integer, z_Integer] := x^3 + y^3 + z^3"}}
In[2254]:= ExtrOfMfile["C:\\AVZ_Package\\Kiev.m", {"W", "W1"}]
```

```
In[2255]:= ExtrOfMfile["C:\\AVZ_Package\\Kiev.m", {"W", "W1"}, w1]
In[2256]:= {W[71, 78, 460], W1[71, 78], w1}
Out[2256]= {98168463, 5538 + W[71, 78], {"W[x_Integer, y_Integer] := x^2 + y^2",
      "W[x_Integer, y_Integer, z_Integer] := x^2 + y^2 + z^2",
      "W[x_Integer, y_Integer, z_Integer] := x^3 + y^3 + z^3",
      {"W1[x_Integer, y_Integer] := x*y + W[x, y]}"}
In[2257]:= ExtrOfMfile["C:\\AVZ_Package\\Kiev.m", {"W", "W1", "GSArtKr"}, w2]
In[2258]:= w2
Out[2258]= {"W[x_Integer, y_Integer] := x^2 + y^2",
      "W[x_Integer, y_Integer, z_Integer] := x^2 + y^2 + z^2",
      "W[x_Integer, y_Integer, z_Integer] := x^3 + y^3 + z^3",
      {"W1[x_Integer, y_Integer] := x*y + W[x, y]}, $Failed}
In[2259]:= ExtrOfMfile["C:\\AVZ_Package\\Kiev.m", {"AgnVsvArtKr", "Avz"]}
Out[2259]= $Failed
In[2260]:= Remove[StrStr]
In[2261]:= Definition[StrStr]
Out[2261]= Null
In[2262]:= {ExtrOfMfile["C:\\AVZ_Package\\AVZ_Package_1.m", "StrStr", G], G}
Out[2262]= {Null, {"StrStr[x_] := If[StringQ[x], "\\\\"<x>"\\\\"<x>"\\\\"<x>", ToString[x]]}}
```

Следует отметить, что данная процедура может оказаться довольно полезной в случае необходимости восстановления в текущем сеансе поврежденных средств без загрузки пользовательских пакетов, содержащих их определения.

К процедуре **ExtrOfMfile** непосредственно примыкает также и процедура **DefFromM**, вызов которой **DefFromM[x, y]** возвращает определение объекта с именем *y*, который находится в файле *x* *m*-формата с пакетом, тогда как **DefFromM[x, y, z]**, где *z* – любое выражение, дополнительно вычисляет в текущем сеансе данное определение, делая объект *y* доступным. Для упрощения алгоритма процедуры **DefFromM** использована процедура **SubListsMin**, в целом полезная для работы со списками. Вызов процедуры **SubListsMin[L, x, y, t]** возвращает подписки списка *L*, ограниченные элементами {*x, y*} и имеющими минимальную длину; при *t="r"* выборка производится слева направо, а при *t="l"* справа налево. Тогда как вызов **SubListsMin[L, x, y, t, z]** с необязательным 5-м аргументом *z* – произвольным выражением – возвращает подписки без ограничивающих элементов {*x, y*}. Фрагмент представляет исходные коды *обеих* процедур с примерами.

```

In[2033]:= SubListsMin[L_;/; ListQ[L], x_, y_, t_;/; MemberQ[{"r", "l"}, t], z___] :=
Module[{a, b, c, d = {}, k = 1, j},
{a, b} = Map[Flatten, Map3[Position, L, {x, y}]];
If[a == {} || b == {} || a == {} && b == {} || L == {}, {},
b = Select[Map[If[If[t == "r", Greater, Less][#, a[[1]]], #] &, b], ! SameQ[#, Null] &];
For[k, k <= Length[a], k++, j = 1; While[j <= Length[b],
If[If[t == "r", Greater, Less][b[[j]], a[[k]],
AppendTo[d, If[t == "r", a[[k]] ;; b[[j]], b[[j]] ;; a[[k]]]; Break[]; j++];

```

```

d = Sort[d, Part[#1, 2] - Part[#1, 1] <= Part[#2, 2] - Part[#2, 1] &];
d = Select[d, Part[#1, 2] - Part[#1, 1] == Part[d[[1]], 2] - Part[d[[1]], 1] &];
d = Map[L[[]] &, d]; d = If[{z} != {}, Map[#[[2 ;; -2]] &, d], d];
If[Length[d] == 1, Flatten[d], d]]

In[2034]:= SubListsMin[{a, b, a, c, d, q, v, w, j, k, d, h, f, d, h}, a, h, "r", 78]
Out[2034]= {c, d, q, v, w, j, k, d}
In[2035]:= SubListsMin[{h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h}, a, h, "r"]
Out[2035]= {{a, b, h}, {a, q, h}}
In[2036]:= SubListsMin[{h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h}, a, h, "r", 78]
Out[2036]= {{b}, {q}}
In[2037]:= SubListsMin[{h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h}, a, h, "l"]
Out[2037]= {h, g, a}
In[2038]:= SubListsMin[{h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h}, a, h, "l", 78]
Out[2038]= {g}
In[2039]:= DefFromM[x_/: FileExistsQ[x] && FileExtension[x] == "m", y_/: SymbolQ[y],
z_] := Module[{a = ReadList[x, String], b, c, d},
{b, c} = {"(*Begin[\"\"\" <> ToString[y] <> \"\"\"*)", "(*End[\"\"\"*)"};
d = StringJoin[Map[StringTake[#, {3, -3}] &, Flatten[SubListsMin[a, b, c, "r", 78]]]];
If[{z} != {}, ToExpression[d]; d, d]]

In[2040]:= DefFromM["C:\\AVZ_Package\\AVZ_Package_1.m", StrStr, 78]
Out[2040]= "StrStr[x_] := If[StringQ[x], \"\\\"\\\"\\\"\" <> x <> \"\\\"\\\"\\\"\", ToString[x]]"

```

Базируясь на подходе, использованном в предыдущей процедуре **ExtrOfMfile**, а также на механизме строчных шаблонов, получаем довольно полезную процедуру, которая обеспечивает получение списка средств, чьи *определения* находятся в пакете (*m-файле*). Вызов процедуры **ContentOfMfile[f]** возвращает список *имен* в строчном формате всех средств, чьи *определения* находятся в пакете (*m-файле*), определяемом аргументом *f*. В отсутствие в *m-файле* определений средств в стандартном пакетном формате вызов возвращает пустой список, т.е. {}. Следующий фрагмент представляет исходный код процедуры **ContentOfMfile** наряду с типичными примерами ее использования.

```

In[2130]:= ContentOfMfile[f_/: FileExistsQ[f] && FileExtension[f] == "m"] := Module[{b,
a = ReadFullFile[f]}, b = StringSplit[a, {"(*", "*)"}];
b = Select[b, ! StringFreeQ[#, {"Begin[\"\"\", \"\"\""}]] &&
StringFreeQ[#, "BeginPackage["] &];
b = Flatten[Map[StringCases[#, "\"\"\" ~ ~ __ ~ ~ \"\"\""] &, b]];
b = DeleteDuplicates[Map[StringTake[#, {3, -3}] &, b]];
Select[b, StringFreeQ[#, {"=", ",", "\"", "[", "]", "(", ")", "^", "^", ";", "{", "}", "\\\", "/"}}] &]]

In[2131]:= ContentOfMfile["C:\\AVZ_Package\\Kiev.m"]
Out[2131]= {"W", "W1"}
In[2132]:= ContentOfMfile["C:\\AVZ_Package\\AVZ_Package_1.m"]

```

```
Out[2132]= {"ExtrExpr", "ExtrCall", "QBlockMod", "RenDirFile", "CharacterQ", "WhichN",
            "Characters1", "$Load$Files$", "FileExistsQ1", "TypeActObj", "Gather2", "Ind",
            =====
            "$Line1", "PackNames", "PackageQ", "Packages", "DumpSaveP"}
In[2133]:= Length[%]
Out[2133]= 582
```

Предыдущая процедура **ContentOfMfile** может быть упрощена и сведена к функции, используя функцию **SubsString**, обеспечивающую выделение подстрок из строки при условии удовлетворения выделяемых подстрок заданным условиям. Вызов процедуры **SubsString[s, {a, b, c, d, ...}]** возвращает список подстрок строки *s*, которые ограничены подстроками {*a, b, c, d, ...*}, тогда как вызов процедуры **SubsString[s, {a, b, c, d, ...}, p]** с 3-м необязательным аргументом *p* – чистой функцией в кратком формате – возвращает список подстрок строки *s*, ограниченных подстроками {*a, b, c, d, ...*}, удовлетворяющих условию, определяемому чистой функцией *p*. В то время как вызов **SubsString[s, {a, b, c, d, ...}, p]** с 3-м необязательным аргументом *p* – произвольным выражением, отличным от чистой функции – возвращает список подстрок, ограниченных подстроками {*a, b, c, d, ...*}, с удаленными префиксами и суффиксами {*a, b, c, d, ...*}[[1]] и {*a, b, c, d, ...*}[[-1]] соответственно. При отсутствии в строке *s* по меньшей мере одной из подстрок {*a, b, c, ...*} вызов процедуры **SubsString** возвращает пустой список. Используя процедуру **SubsString**, уже достаточно несложно модифицировать процедуру **ContentOfMfile** в форме функции **ContentOfMfile1**, исходный код которой наряду с исходным кодом процедуры **SubsString** и примерами их типичных применений достаточно наглядно иллюстрирует нижеследующий фрагмент, а именно.

```
In[2213]:= SubsString[s_String, y_List, pf_] := Module[{a = "", b, c, k = 1},
            If[Set[c, Length[y]] < 2, s, b = Map[ToString1, y];
            While[k <= c - 1, a = a <> b[[k]] <> "~ Shortest[___] ~"; k++];
            a = a <> b[[-1]]; b = StringCases[s, ToExpression[a]];
            If[{pf} != {} && PureFuncQ[pf], Select[b, pf], If[{pf} != {},
            Map[StringTake[#, {StringLength[y][[1]] + 1, -StringLength[y][[-1]] - 1}] &, b],
            Select[b, StringQ[#] &]]]]
In[2214]:= SubsString["adfgbfffxbavzgagngbArtgbKrg", {"b", "g"}, StringFreeQ[#, "f"] &]
Out[2214]= {"bavzg", "bArtg", "bKrg"}
In[2215]:= SubsString["adfgbfffxbavzgagngbArtgbKrg", {"b", "g"}]
Out[2215]= {"bfff", "bavzg", "bArtg", "bKrg"}
In[2216]:= SubsString["abcxxxx12345abcyyyyy12345", {"ab", "12"}, 78]
Out[2216]= {"cxxxx", "cyyyyy"}
In[2227]:= ContentOfMfile1[f_ /; FileExistsQ[f] && FileExtension[f] == "m"] :=
            DeleteDuplicates[Select[Map[StringTake[#, {9, -4}] &,
            SubsString[ReadFullFile[f], {"Begin[\"", "\"]"}],
            StringFreeQ[#, {"=", ",", "~", "[", "]", "(", ")"}, {"^", ";", "{", "}", "\\\", "/"}] &]]
```

```
In[2228]:= ContentOfMfile1["C:\\AVZ_Package\\Kiev.m"]
Out[2228]= {"W", "W1"}
In[2129]:= ContentOfMfile1["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2129]= {"ExtrExpr", "ExtrCall", "QBlockMod", "RenDirFile", "CharacterQ", "WhichN",
            "Characters1", "$Load$Files$", "FileExistsQ1", "TypeActObj", "Gather2", "Ind",
            "BitGet1", "StringPosition1", "SubsPosSymb", "TwoHandQ", "GC", "ExprOfStr",
            =====
            "Locals2", "SyntCorProcQ", "Subs", "BlockQ", "BlockQ1", "SymbolToList",
            "FreeQ2", "Uprocs", "ActCsProcFunc", "ListOp", "ProcQ", "ProcFuncTypeQ",
            "$Line1", "PackNames", "PackageQ", "Packages", "DumpSaveP"}
In[2130]:= Length[%]
Out[2130]= 582
```

В целом следует отметить, что система *Mathematica* располагает достаточно развитым механизмом *строчных* шаблонов, позволяющим программировать довольно развитые средства обработки строчных конструкций. В качестве весьма полезного упражнения, где это целесообразно, рекомендуется модифицировать средства обработки строк из настоящей книги с использованием механизма строчных шаблонов. В целом, данный механизм весьма эффективен при обработке различного рода строчных структур.

Следующие *две* процедуры довольно полезны при манипуляциях с пакетом, который находится в файле *mх*-формата. Так, вызов процедуры **ContextMXfile[x]** возвращает *контекст*, ассоциированный с пакетом пользователя, находящимся в файле формата *"mх"*. Между тем, загрузки *mх*-файла не производится. Процедура **MxToTxt** допускает от 2 до 4 фактических аргументов. Вызов процедуры **MxToTxt[x, y]** возвращает **Null**, т. е. ничего, сохраняя в файле *y txt*-формата и в текущем сеансе *все* определения пакета, находящегося в файле *x mх*-формата. При этом, определения файла *x* сохраняются в оптимальном формате (*без ассоциированного с пакетом контекста*). В случае, если вызов **MxToTxt[x, y, z]**, начиная с 3-го аргумента, содержит необязательный аргумент *"Del"*, то пакет *x* не загружается в текущий сеанс, в противном случае все его определения сохраняются в текущем сеансе в оптимальном формате. Если при вызове аргументы процедуры, начиная с третьего, содержат неопределенную переменную, то через нее возвращается список всех объектов, определения которых находятся в файле данных *x* с пакетом пользователя. Следующий фрагмент представляет исходные коды всех 3 упомянутых процедур наряду с наиболее типичными примерами их использования.

```
In[2256]:= ContextMXfile[x_;/ FileExistsQ[x] && FileExtension[x] == "mx"] := Module[{a,
            b = Flatten[Map7[Range, Sequences, {{48, 57}, {65, 90}, {96, 122}}]], c},
            a = BinaryReadList[x]; a = a[[1 ;; If[Length[a] >= 500, 500, Length[a]]]];
            c = Flatten[Map3[PosSubList, a, {{67, 79, 78, 84}, {69, 78, 68, 67, 79, 78, 84}}]];
            If[Length[c] < 5, $Failed, FromCharacterCode[Select[a[[c[[2]] + 1 ;; c[[5]] - 1]],
            MemberQ[b, #1] &]]]]
In[2257]:= ContextMXfile["F:\\AVZ_Package\\AVZ_Package.mx"]
Out[2257]= "AladjevProcedures`"
```

```

In[2258]:= ContextFromFile[x_String] := If[Quiet[FileExistsQ[x]] && MemberQ[{"m", "nb",
"mx", "cdf"}, FileExtension[x]], Quiet[ToExpression[StringJoin["Context", ToUpperCase[
If[FileExtension[x] == "cdf", "nb", FileExtension[x]]], "file["], ToString1[x], "]" ]], $Failed]

In[2259]:= Map[ContextFromFile, {"Kherson.m", "AVZ_Package_1.mx", "Package.nb"}]
Out[2259]= {"Kherson", "AladjevProcedures", "AladjevProcedures"}

In[2260]:= MxToTxt[x_;/; FileExistsQ[x] && FileExtension[x] == "mx", y_;/; StringQ[y],
z_] := Module[{b, c, a = ContextMXfile[x]}, LoadMyPackage[x, a];
b = CNames[a]; Map[{Write[y, Definition[#]], Write[y]} &, b];
Close[y]; If[MemberQ[{z}, "Del"], RemovePackage[a];
c = Select[{z}, ! HowAct[#] && ! SameQ[#, "Del"] &];
If[c != {}, ToExpression[ToString[c[[1]]] <> "=" <> ToString[b]]]; ]

BeginPackage["Kherson"]
Gs::usage = "Function Gs[x, y] := 71*x^2 + 66*y + 46 + S[x, y].\"
G::usage = "Function G[x, y] := N[Sin[x] + Cos[y]] + S[x, y].\"
V::usage = "Function S[x_, y_] := x^2 + y^2.\"
Begin["Private"]
V[x_, y_] := x^2 + y^2
Gs[x_;/; IntegerQ[x], y_;/; IntegerQ[y]] := 71*x^2 + 66*y + 46 + V[x, y]
G[x_;/; IntegerQ[x], y_;/; IntegerQ[y]] := N[Sin[x] + Cos[y]] + V[x, y]
End[]
EndPackage[]

In[2261]:= $Packages
Out[2261]= {"AladjevProcedures", "GetFEKernelInit", "ResourceLocator",
"PacletManager", "QuantityUnits", "WebServices", "System", "Global"}
In[2262]:= ContextMXfile["Kherson.mx"]
Out[2262]= "Kherson"
In[2263]:= MxToTxt["Kherson.mx", "Kherson.txt"]
In[2264]:= $Packages
Out[2264]= {"Kherson", "AladjevProcedures", "GetFEKernelInit", "ResourceLocator",
"PacletManager", "QuantityUnits", "WebServices", "System", "Global"}
In[2265]:= MxToTxt["Kherson.mx", "Kherson.txt", g]; g
Out[2265]= {G, Ga, Gs, V, Vgs}
In[2266]:= $Packages
Out[2266]= {"Kherson", "AladjevProcedures", "GetFEKernelInit", "ResourceLocator",
"PacletManager", "QuantityUnits", "WebServices", "System", "Global"}
In[2267]:= MxToTxt["Kherson.mx", "Kherson.txt", "Del"]
In[2268]:= $Packages
Out[2268]= {"AladjevProcedures", "GetFEKernelInit", "ResourceLocator",
"PacletManager", "QuantityUnits", "WebServices", "System", "Global"}
In[2269]:= MxToTxt["Kherson.mx", "Kherson.txt", t, "Del"]; t
Out[2269]= {Ga, Gs, Vgs}

```

```

In[2270]:= $Packages
Out[2270]= {"AladjevProcedures`", "GetFEKernelInit`", "ResourceLocator`",
            "PacletManager`", "QuantityUnits`", "WebServices`", "System`", "Global`}
In[2271]:= MxToTxt1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y],
            z_] := Module[{a = ContextFromFile[x], c}, LoadMyPackage[x, a];
            Map[PutAppend[Definition[#], "OK!", y] &, CNames[a]];
            If[MemberQ[{z}, "Del"], RemovePackage[a]];
            c = Select[{z}, ! HowAct[#] && ! SameQ[#, "Del"] &];
            If[c != {}, ToExpression[ToString[c[[1]]] <> "=" <> ToString[b]]; ]

In[2272]:= MxToTxt1["AVZ_Package_1.mx", "RansIan"]
In[2273]:= Get["RansIan"]
Out[2273]= "OK!"

In[2293]:= MxToTxt2[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y]] :=
            Module[{a = ContextFromFile[x], c}, If[! MemberQ[$Packages, a], c = 78];
            Get[x]; Map[PutAppend[Definition[#], "OK!", y] &, CNames[a]];
            If[c === 78, RemovePackage[a]]; "OK!"

In[2294]:= MxToTxt2["Kherson.mx", "RansIanAvzKr.txt"]
Out[2294]= "OK!"
In[2295]:= Get["RansIanAvzKr.txt"]
Out[2295]= "OK!"

```

Процедура **MxToTxt1** представляет собой модификацию процедуры **MxToTxt**, также допуская от 2 до 4 фактических аргументов. Вызов **MxToTxt1[x, y]** возвращает **Null**, т. е. ничего, сохраняя как в файле *y txt*-формата, так и в текущем сеансе все *определения* пакета, находящегося в файле *x mx*-формата. Более того, определения *x* сохраняются в *оптимальном* формате (без ассоциированного с пакетом контекста). В случае, если вызов **MxToTxt1[x, y, z]**, начиная с 3-го аргумента, содержит необязательный аргумент *"Del"*, то пакет *x* не загружается в текущий сеанс, иначе его определения будут сохраняться в текущем сеансе в *оптимальном* формате. Если же при вызове аргументы процедуры, начиная с третьего, содержат неопределенную переменную, через нее возвращается список всех объектов, определения которых находятся в файле *x* с пользовательским пакетом. Фрагмент представляет исходный код процедуры с примером применения. При этом, загрузка файла *y* с определениями пакета *x* возвращает *"OK!"*, активируя в текущем сеансе содержащиеся в нем определения программных средств. Процедура **MxToTxt2** представляет собой довольно полезную модификацию процедур **MxToTxt** и **MxToTxt1**, допуская 2 фактических аргумента и сохраняя в *ASCII*-файле, который определен вторым аргументом, *все* определения, содержащиеся в *mx*-файле, который определен первым аргументом. Вызов **MxToTxt2[x, y]** возвращает *"OK!"*, сохраняя в *y*-файле определения *mx*-файла *x*, разделенные строкой *"OK!"*. При этом, если пакет из *mx*-файла *x* не был загружен в текущий сеанс, то и после вызова **MxToTxt2** в текущем сеансе он недоступен. В частности, на основе процедур **MxToTxt** ÷ **MxToTxt2** можно



создавать довольно эффективные и простые библиотеки пользовательских средств с системой их ведения. Подобная организация оказывается достаточно привычной для пользователей, имеющих опыт работы в традиционных системах программирования.

Следующая процедура **ActUcontexts** базируется на механизме **%**-оператора, ее вызов **ActUcontexts[]** возвращает список *всех* контекстов, которые ассоциированы с пакетами пользователя, загруженными в текущий сеанс из файлов форматов {"cdf", "nb", "m"}. В то время как вызов **ActUcontexts[h]** возвращает список всех контекстов, порожденных загруженными пакетами из файлов вышеуказанного формата. Следующий фрагмент представляет исходный код процедуры **ActUcontexts** с примерами ее использования.

```
In[2110]:= ActUcontexts[h___] := Module[{a = {}, b = "", c = "Out[" , k = 1},
      For[k, k < Infinity, k++, b = ToString[Out[k]];
      If[b == c <> ToString[k] <> "]", Return[DeleteDuplicates[a]]];
      If[ContextQ[b] && If[{h} != {}, True, StringCount[b, "" ] == 1],
      a = Append[a, b]]]

In[2111]:= ActUcontexts[]
Out[2111]= {"AladjevProcedures`", "Kiev`", "Kiev1`", "Kherson`"}
In[2112]:= ActUcontexts[78]
Out[2112]= {"AladjevProcedures`", "AladjevProcedures`ExtrExpr`",
      "AladjevProcedures`ExtrCall`", "AladjevProcedures`QBlockMod`", .....,
      "AladjevProcedures`DumpSaveP`", "Kiev`", "Kiev1`", "Kherson`",
      "Kherson`Private`", "Kherson`Gs`", "Kherson`Ga`", "Kherson`Vgs`"}
```

Вызов **DumpSave[x, y]** возвращает список контекстов *y*, ассоциированные с которыми объекты или объекты с определениями якобы были выгружены в файл *x* независимо от наличия *определений* для этих объектов или контекстов в переменной **\$ContextPath**, не позволяя производить *программную* обработку результата вызова **DumpSave**. Итак, результат вызова **DumpSave** невозможно программно тестировать и устранению этой ситуации способствует процедура, чей успешный вызов **DumpSave1[x, y]** возвращает вложенный список, *первый* элемент которого определяет путь к файлу *x* *mx*-формата (*при необходимости файлу присваивается расширение "mx"*), тогда как *второй* определяет список объектов *и/или* контекстов из списка, определяемого *y*, определения которых выгружены в файл *x*. При отсутствии объектов (*определенные символы и/или контексты, существующие в списке \$ContextPath*), которые были определены аргументом *y*, вызов **DumpSave1** возвращает **\$Failed**. Фрагмент представляет исходный код с примерами.

```
In[14]:= DumpSave1[x_, y_] := Module[{a, b, c}, If[StringQ[x], If[FileExtension[x] == "mx",
      c = x, c = x <> ".mx"]; a = Flatten[{y}];
      b = Select[a, (ContextQ[#] && MemberQ[$ContextPath, #]) ||
      ! MemberQ[{"", "Null"}, Quiet[ToString[Definition[#]]]] &];
      If[b != {}, {c, Flatten[DumpSave[c, b]]}, $Failed, $Failed]]

In[15]:= DumpSave1["C:\\AVZ_Package\\AVZ_Package42.mx", {"Art`", "Kr`", GS}]
Out[15]= $Failed
```

```

In[3124]:= ReplaceSubLists[x_List, y_ /; RuleQ[y] || ListRulesQ[y]] := Module[{a, f,
    d = FromCharacterCode[2013]}, f[z_List] := StringJoin[Map[ToString1[#] <> d &, z]];
    a = Map[f[Flatten[{{#1}}]] -> f[Flatten[{{#2}}]] &, Flatten[{y}]];
    ToExpression[StringSplit[StringReplace[f[x], a], d]]]

In[3125]:= ReplaceSubLists[{a, b, c, "d", m, x, b, c}, {{b, c} -> {x, y}, a -> {m, n}, "d" -> "78"}]
Out[3125]= {m, n, x, y, "78", m, x, x, y}

In[2321]:= SubsList[x_List, y_, z_] := Module[{a = FromCharacterCode[2013], b, c},
    b = StringJoin[Map[ToString1[#] <> a &, x]];
    c = Map[StringJoin[Map[ToString1[#] <> a &, Flatten[{{#1}}]] &, {y, z}];
    c = ToExpression[StringSplit[SubsString[b, {c[[1]], c[[2]]}], a]]; If[Length[c] == 1, c[[1]], c]]

In[2322]:= SubsList[{a, b, c, d, x, y, x, b, c, n, a + b, x, y, z}, {b, c}, {x, y}]
Out[2322]= {{b, c, d, x, y}, {b, c, n, a + b, x, y}}

In[2346]:= ContextToSymbol[x_Symbol, y_ /; ContextQ[y], z_ / StringQ[z]] :=
    Module[{a = If[FileExtension[z] == "mx", z, z <> ".mx"],
        b = Flatten[{PureDefinition[x]}]},
    If[b === {$Failed}, $Failed, $ContextPath = AppendTo[$ContextPath, y];
    Quiet[ToExpression[Map[y <> # &, b]]]; {a, DumpSave[a, y]]]

In[2347]:= Art[x_] := Module[{a = 6}, x + a]; ContextToSymbol[Art, "Veeroja", "AgnAvz"]
Out[2347]= {"AgnAvz.mx", {"Veeroja"}}

In[2531]:= ContextRepMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx",
    y_ /; ContextQ[y]] := Module[{a = ContextMXfile[x], b, c, d, h, n, m, f},
    a = If[SameQ[a, $Failed], "None", a]; b = ReadList[x, Byte];
    c = Map[ToCharacterCode, {"Windows", "ENDCONT"}]; f = ToString[Unique[]] <> ".mx";
    ContextToSymbol[d, y, f]; h = ReadList[f, Byte]; n = SubsList[b, c[[1]], c[[2]]];
    m = SubsList[h, c[[1]], c[[2]]]; DeleteFile[f]; h = ReplaceSubLists[b, n -> m];
    f = FileNameSplit[x]; f = FileNameJoin[AppendTo[f[[1 ;; -2]], "$" <> f[[-1]]];
    BinaryWrite[f, h]; Close[f]; {f, a, y}]

In[2532]:= ContextRepMx["C:\\AVZ_Package\\Kiev.mx", "Grodno"]
Out[2532]= {"C:\\AVZ_Package\\$Kiev.mx", "avzransian", "Grodno"}

In[2320]:= ContextSymbol[x_Symbol] := Select[Map[If[MemberQ[CNames[#],
    ToString[x]] || MemberQ[CNames[#], # <> ToString[x]], #] &,
    DeleteDuplicates[$ContextPath]], ! SameQ[#1, Null] &]

In[2321]:= ContextSymbol[GSV]
Out[2321]= {"avzransian"}

```

Вызов **ReplaceSubLists**[*x*, *y*] возвращает результат замены элементов (включая смежные) списка *x* на основе правила или списка правил *y*; более того, в качестве частей правил могут выступать *списки*. Тогда как вызов процедуры **SubsList**[*x*, *y*, *z*] возвращает *список*

подписков элементов списка  $x$ , ограниченных элементами  $\{y, z\}$ ; в качестве элементов  $\{y, z\}$  могут выступать списки. Если любой из элементов  $\{y, z\}$  не принадлежит  $x$ , вызов процедуры возвращает пустой список, т.е.  $\{\}$ . Представленные процедуры **SubsList** и **ReplaceSubLists** наряду с обработкой списков представляют интерес для присвоения  $mx$ -файлам контекста в его отсутствие. В то время как вызов **ContextToSymbol** $[x, y, z]$  возвращает список формата  $\{z, \{y\}\}$ , приписывая определенному символу  $x$  контекст  $y$  с сохранением его определения в  $mx$ -файле  $z$ . В частности, данное средство довольно полезно при необходимости сохранения объектов в  $mx$ -файлах с контекстом. На базе 3 процедур **ReplaceSubLists**, **SubsList** и **ContextToSymbol** создана процедура, которая обеспечивает замену контекстов в  $mx$ -файлах без их загрузки в текущий сеанс. Вызов **ContextRepMx** $[x, y]$  обеспечивает замену контекста  $mx$ -файла  $x$  на новый контекст  $y$ , возвращая список формата  $\{File, h, y\}$ , где *File* – файл с результатом такой замены, *h* – старый контекст или "None" – если его не было, и  $y$  – новый контекст. Тогда как, вызов функции **ContextSymbol** $[x]$  возвращает контекст, ассоциированный с символом  $x$ .

При вычислении определения символа  $x$  в текущем сеансе он будет ассоциироваться с контекстом "Global", который сохраняется при его выгрузке в  $mx$ -файл функцией **DumpSave**. Тогда как в ряде случаев возникает необходимость сохранения символов в  $mx$ -файле с иными контекстами. Данную задачу решает процедура, вызов которой **DumpSave2** $[f, x, y]$  ничего не возвращает, выгружая в  $mx$ -файл  $f$  определение символа или списка символов  $x$ , имеющих контекст "Global", с контекстом  $y$ . Исходный код с типичными примерами использования процедуры приводит следующий фрагмент.

```
In[99]:= DumpSave2[x_/: FileExtension[x] == "mx", y_/: SymbolQ[y] || ListQ[y] &&
DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}, z_/: ContextQ[z]] := Module[{b,c,
a = Flatten[Select[Map[PureDefinition[#] &, Flatten[{y}]], ! SameQ[#, $Failed] &]],
Map[ToExpression[z <> #] &, a]; $ContextPath = AppendTo[$ContextPath, z];
c = Map[z <> ToString[#] &, Flatten[{y}]]]; c = AppendTo[c, $ContextPath];
DumpSave[x, c]; ]

In[100]:= Agn[x_] := x; Agn[x_, y_] := x + y; Agn[x_Integer] := x + 460
In[101]:= Avz[x_] := x^2; Avz[x_, y_] := 78*(x + y); Avz[x_Integer] := x + 460
In[102]:= Map[ContextSymbol, {Agn, Avz}]
Out[102]= {"Global", "Global"}
In[103]:= DumpSave2["Tallinn.mx", {Agn, Avz}, "Rans`"]
In[104]:= Clear[Avz, Agn]; Map[PureDefinition, {Agn, Avz}]
Out[104]= {$Failed, $Failed}
In[105]:= Get["Tallinn.mx"]; PureDefinition[Rans`Agn]
Out[105]= {"Rans`Agn[x_Integer] := x+460", "Rans`Agn[x_] := x", "Rans`Agn[x_, y_] := x+y"}
In[3331]:= Ian`Agn[x_, y_] := x + y; $ContextPath = PrependTo[$ContextPath, "Ian`"]
Out[3331]= {"Ian`", "AladjevProcedures`", "PacletManager`", "QuantityUnits`", ...}
In[3332]:= ContextSymbol[Agn]
Out[3332]= {"Ian`"}
In[3333]:= DumpSave["Tampere.mx", "Ian`"];
In[3335]:= $ContextPath = MinusList[$ContextPath, {"Ian`"}]; PureDefinition[Agn]
```

```
Out[3335]= $Failed
In[3336]:= Get["Tampere.mx"]; CNames["Ian`"]
Out[3336]= {"Agn"}
In[3337]:= ContextMXfile["Tampere.mx"]
Out[3337]= "Ian`"
In[3338]:= PureDefinition[Agn]
Out[3338]= "Agn[x_, y_] := x + y"
```

Фрагмент завершают примеры, иллюстрирующие принцип *сохранения* объектов, чьи определения вычислены в текущем сеансе, в *mx*-файлах с заданным контекстом. Этот принцип использовался при написании процедур, наделяющих символ контекстом.

Определенный интерес представляет процедура **NamesCS[P, Pr, Pobj]**, вызов которой возвращает *Null*, т.е. ничего, тогда как через три аргумента *P*, *Pr* и *Pobj* возвращаются соответственно *список* контекстов, соответствующих пакетам, загруженным в *текущем* сеансе пакета, *список* процедур пользователя, чьи определения активированы в *Input*-параграфе текущего сеанса пакета, и *вложенный* список, чьи *подписки* в общем случае имеют различную длину и структурно форматированы следующим образом:

- *первый элемент подписки определяет контекст, соответствующий пакету, который был загружен в текущем сеансе Mathematica на момент вызова процедуры NamesCS;*
- *все последующие элементы подписки определяют объекты данного пакета, которые были в текущем сеансе системы Mathematica активизированы.*

```
In[2213]:= NamesCS[P_;/! HowAct[P], Pr_;/! HowAct[Pr], Pobj_;/! HowAct[Pobj]] :=
Module[{a = Quiet[Select[Map[ToExpression, Names["*"], ProcQ[#] &]],
b = Contexts[], c = $Packages, d, k = 1, p, n, m, h},
{P, Pr} = {c, a}; c = Map[List, c];
For[k, k <= Length[b], k++, For[p = 1, p <= Length[c], p++,
n = b[[k]]; m = c[[p]][[1]]; If[n === m, Null, If[SuffPref[n, m, 1],
d = StringReplace[n, b -> ""]; If[d == "", Null, c[[p]] = Append[c[[p]],
ToExpression[StringTake[StringReplace[n, b -> ""], {1, -2}]]]],
Continue[]]]; c = Map[DeleteDuplicates, c];
For[k = 1, k <= Length[c], k++, h = c[[k]];
If[Length[h] == 1, h = Null, h = Select[h, StringQ[#] ||
ToString[Quiet[DefFunc[#]]] != "Null" &]];
Pobj := Select[c, Length[#] > 1 && ! # === Null &];
Pobj = Mapp[Select, Pobj, If[! StringQ[#], True,
If[StringTake[#, -1] == "", True, False]] &]; ]

In[2214]:= NamesCS[P, Pr, Pobj]
In[2215]:= {P, Pr}
Out[2215]= {"Grodno`", "Kherson`", "Kiev1`", "Kiev`", "AladjevProcedures`", ....., {}}
In[2216]:= Pobj
Out[2216]= {"Kherson`", Ga, Gs, Private, Vgs},
```

```
{ "AladjevProcedures`", ActBFMuserQ, ActRemObj, ActUcontexts, Adrive1, ...
  "ResourceLocator`", Private},
{ "PacletManager`", Collection`Private, Private, Extension`Private, ...},
{ "QuantityUnits`", Private},
{ "WebServices`", Information},
{ "System`", BesselParamDerivativesDump, BinaryReadDump}}
```

Более того, список, возвращаемый через **Pobj**-аргумент, содержит только подписки, чьи соответствующие пакеты имеют в текущем сеансе активизированные объекты. Тогда как весьма простая функция **Npackage[x]** возвращает список имен в строчном формате всех объектов, определения которых содержатся в пакете *x*, активированном в текущем сеансе. В случае неактивности в текущем сеансе пакета *x* либо в случае его отсутствия вызов функции **Npackage[x]** возвращает значение **\$Failed**. Исходный код функции **Npackage** с примером ее применения представляет следующий фрагмент.

```
In[2380]:= Npackage[x_./; StringQ[x]] := If[MemberQ[Contexts1[], x],
      Sort[Select[Names[x <> "*"], StringTake[#, -1] != "$" &&
      ToString[Definition[#]] != "Null" &]], $Failed]

In[2381]:= Npackage["AladjevProcedures`"]
Out[2381]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts", ...}
```

Достаточно интересной представляется и процедура, вызов которой **ContOfContext[x]** возвращает вложенный 2-элементный список, первый элемент которого определяет подписок *всех* имен в строчном формате программных средств пакета пользователя, заданного своим *контекстом x*, определения которых в текущем сеансе возвращаются по функции **Definition** с включенным в них *контекстом x*, тогда как второй элемент определяет подписок *всех* имен в строчном формате средств пакета, заданного своим *контекстом x*, определения которых в текущем сеансе возвращаются по **Definition** без *контекста x*. Следующий фрагмент представляет как исходный код процедуры, так и примеры ее применения относительно контекста **"AladjevProcedures"**. В завершение вычисляется *длина* обоих подписков возвращаемого результата наряду с выборочной проверкой по **Definition** определений средств из обеих подписков. При этом, нужно отметить, что с ростом числа объектов анализируемого контекста *x*, вызов процедуры **ContOfContext[x]** требует все больших временных ресурсов.

```
In[2105]:= ContOfContext[x_./; StringQ[x] && StringTake[x, -1] == ""] := Module[{b = {},
      a = Select[CNames[x], # != "a" &], c = {}, h, k = 1},
      If[a == {}, $Failed, While[k <= Length[a], h = a[[k]];
      If[StringFreeQ[StringReplace[ToString[Definition1[h]], "\n\n" -> ""],
      x <> h <> ""], AppendTo[c, h], AppendTo[b, h]]; k++]; {b, c}]

In[2106]:= ContOfContext["AladjevProcedures`"]
Out[2106]= {"ActBFMuserQ", "ActRemObj", "ActUcontexts", "Adrive1", "Affiliate", .....},
      {"AcNb", "ActCsProcFunc", "Adrive", "ArityBFM", "Attribs", "Attributes1", ...,
      "$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}}
```

```

In[2107]:= Map[Length, %]
Out[2107]= {446, 136}
In[2108]:= Definition["DirName"]
Out[2108]= DirName[AladjevProcedures`DirName`F_ /;
                StringQ[AladjevProcedures`DirName`F]] :=
                If[DirQ[AladjevProcedures`DirName`F], None,
                If[! FileExistsQ1[AladjevProcedures`DirName`F], $Failed,
                Quiet[Check[FileNameJoin[FileNameSplit[
                AladjevProcedures`DirName`F][[1; -2]]], None]]]]
In[2109]:= Definition["StrStr"]
Out[2109]= StrStr[x_] := If[StringQ[x], StringJoin["\\", x, "\\"], ToString[x]]
In[2110]:= ContOfContext["AladjevProceduresAndFunctions`"]
Out[2110]= $Failed

```

На неактивных контекстах вызовы процедуры **ContOfContext[x]** возвращают **\$Failed**, в то время как на остальных случаях вызов процедуры возвращается невычисленным. Во фрагменте представлено использование процедуры **ContOfContext** для контекста вполне реального *nb*-документа (одной из последних версий пакета *AVZ\_Package\_1* [120]), загруженного и вычисленного в текущем сеансе. Как одно из возможных приложений данной процедуры можно отметить задачи, которые имеют дело с исходными кодами программных средств пакетов пользователя.

В качестве довольно полезного дополнения к процедуре **ContOfContext** вполне может быть рассмотрена и процедура, чей вызов **NamesContext[x]** возвращает список имен в строчном формате программных объектов текущего сеанса, которые ассоциированы с контекстом *x*. В случае отсутствия данного контекста возвращается пустой список, т. е. {}. В случае значения *x*, отличного от контекста, вызов возвращается *невычисленным*. Следующий фрагмент представляет исходный код процедуры **NamesContext** наряду с некоторыми наиболее типичными примерами ее использования.

```

In[1]:= LoadPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] := Module[{a},
                Quiet[ToExpression["Off[shdw::Symbol]"]; Get[x];
                a = ToExpression["Packages"][[1]];
                ToExpression["LoadMyPackage[" <> "\"" <> x <> "\"" <> ", " <> "\"" <> a <> "\"" <> ""]"];
                ToExpression["On[shdw::Symbol]"]]]
In[2]:= LoadPackage["C:\\AVZ_Package\\AVZ_Package_1.mx"]
In[3]:= G := 66; S := 46; Art := 24; Kr := 17;
In[4]:= Sv[x_] := x^2; Vt[x_] := x^3; Gal[x_] := Sv[x] + Vt[x]
In[5]:= NamesContext[x_ /; ContextQ[x]] := Module[{a = Names[x <> "*"], b, c = {}, k = 1, h},
                While[k <= Length[a], b = a[[k]];
                h = ToString[ToExpression["Definition[" <> b <> ""]"];
                If[h != "Null" && h != "Attributes[" <> b <> "]"] = {Temporary} &&
                ! SuffPref[b, "a$", 1], AppendTo[c, a[[k]]]; k++]; c]

```

```

In[6]:= NamesContext["AladjevProcedures`"]
Out[6]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
        "Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes", "Arity",
        =====
        "WhatObj", "WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1",
        "$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}
In[7]:= Length[%]
Out[7]= 582
In[8]:= NamesContext["Global`"]
Out[8]= {"Art", "G", "Gal", "Kr", "Global`LoadPackage", "NamesContext", "Sv", "Vt"}
In[9]:= Length[%]
Out[9]= 8
In[10]:= NamesContext["System`"]
Out[10]= {"\[FormalA]", "\[FormalB]", "\[FormalC]", "\[FormalD]", "\[FormalE]",
        =====
        "$UserAddOnsDirectory", "$UserBaseDirectory", "$UserDocumentsDirectory",
        "$UserName", "$Version", "$VersionNumber", "\[SystemsModelDelay]"}
In[11]:= Length[%]
Out[11]= 4280
In[12]:= NamesContext["Aladjev`"]
Out[12]= {}
    
```

Следующая простая функция обеспечивает тестирование строки на ее допустимость в качестве *контекста*, при этом, допустимость рассматривается лишь с точки зрения синтаксической корректности. Вызов функции **ContextQ[x]** возвращает **True**, если *x* – потенциально допустимый контекст, и **False** в противном случае. Функция **ContextQ** представляет вполне определенный интерес в задачах, использующих тестирование строк на предмет допустимости их в качестве формальных контекстов. Следующий достаточно простой фрагмент представляет исходный код функции **ContextQ** наряду с некоторыми наиболее типичными примерами ее использования.

```

In[2118]:= ContextQ[x_] := StringQ[x] && StringLength[x] > 1 &&
        Quiet[SymbolQ[Symbol[StringTake[x, {1, -2}]]]] && StringTake[x, -1] == ""
In[2119]:= ContextQ["AladjevProceduresAndFunctions`"]
Out[2119]= True
In[2120]:= ContextQ["71AladjevProcedures78AndFunctions`"]
Out[2120]= False
In[2121]:= ContextQ["LinearAlgebra`Internal`"]
Out[2121]= True
    
```

Вызов процедуры **Contexts1**, являющейся простой модификацией функции **Contexts**, возвращает список контекстов, соответствующих пакетам, компоненты которых были активизированы в текущем сеансе. Следующий фрагмент представляет *исходный* код процедуры **Contexts1** наряду с достаточно типичным примером ее использования.

```
In[2125]:= Contexts1[] := Module[{a = {}, b = Contexts[], c, k = 1}, For[k, k <= Length[b],
    k++, c = b[[k]]; If[Length[DeleteDuplicates[Flatten[StringPosition[c, ""]]]] == 1
    && StringTake[c, {-1, -1}] == "", a = Append[a, c], Next[]]; a]
In[2126]:= Contexts1[]
Out[2126]= {"AladjevProcedures`, "Algebra`, "AlphaIntegration`, "Assumptions`,
    "AssumptionsDump`, "AugmentedData`, "BinningUtilities`, "BoxForm`,
    "BoxFormat`, "BrowserCategoryLoad`, "Calendar`, "CCodeGenerator`,
    =====
    "Visualization`, "Wavelets`, "WebServices`, "WrappersDump`, "XML`"}
In[2127]:= Length[%]
Out[2127]= 161
```

Тогда как весьма простая функция **ContextActQ** обеспечивает проверку *контекста* на его наличие в списке контекстов текущего сеанса. Вызов **ContextActQ[w]** возвращает *True*, если контекст *w* находится в списке *всех* контекстов текущего сеанса, иначе *False* возвращается. Следующий фрагмент представляет исходный код функции наряду с примерами ее применения. Функция имеет целый ряд весьма важных приложений, прежде всего, в задачах, требующих тестирования доступности контекстов.

```
In[2111]:= ContextActQ[x_ /; ContextQ[x]] := MemberQ[DeleteDuplicates[Contexts1[]], x]
In[2112]:= Map[ContextActQ, {"AladjevProcedures`, "RansIan78460`, "Kherson`}]]
Out[2112]= {True, False, True}
```

Наконец, следующая достаточно простая процедура обеспечивает визуализацию всех определений, находящихся в пакете, ассоциированном с заданным контекстом. Вызов **VizContext[x]** возвращает *nb*-документ, содержащий все определения пакета, который загружен в текущий сеанс и ассоциируется с контекстом *x*. Документ возвращается в отдельном окне, обеспечивая просмотр определений с возможностью последующего его сохранения в файле форматов {*cdf*, *m*, *nb*, ...}. Следующий фрагмент представляет исходный код процедуры **VizContext** с типичным примером ее использования.

```
In[2246]:= VizContext[x_ /; ContextQ[x]] := Module[{a = CNames[x]}, If[a == {}, $Failed,
    CreateDocument[Map[StringReplace[#, x -> ""]] &,
    Map[ToString1, Map[Definition, a]]]]]
In[2247]:= VizContext["AladjevProcedures`"]
Out[2247]= NotebookObject[Untitled-78]
```

Между тем, определения объектов из пакета, ассоциированного с данным контекстом, отсортированы согласно именам этих объектов; при этом, справки по ним не даются. Данная процедура наиболее удобна для визуализации определений пакета из файла *m*-формата и конвертации его определений в файлы других форматов. Эффективно использование процедуры **VizContext** также с процедурами, рассмотренными выше.

В процессе работы в текущем сеансе со средствами загруженного пакета (*формат "m"*) вполне реальны такие ситуации, когда некоторые из его активированных средств по



той или иной причине удаляются из текущего сеанса или искажаются. С этой целью для их восстановления может быть использована процедура **ReloadPackage**.

```

In[2092]:= ReloadPackage[x_ /; FileExistsQ[x] && FileExtension[x]=="m", y___List, t___] :=
Module[{a = NamesMPackage[x], b = ContextMfile[x], c = "$ArtKr$.txt", p, k = 1,
d = If[{y} != {}, ToExpression[Map14[StringJoin, Map[ToString, y], {"", 78}], {}]},
Put[c]; While[k <= Length[a], p = a[[k]]; PutAppend[StringReplace[ToString1[
ToExpression["Definition" <> p <> ""]], b <> p <> "" -> ""], c]; k++];
If[d == {}, ToExpression["Clear" <> StringTake[ToString[a], {2, -2}] <> ""], Null];
While[b != "EndOfFile", b = ToString[Read[c]]; If[b === "EndOfFile", Break[]];
If[d == {}, Quiet[ToExpression[b]]; Continue[],
If[If[{t} == {}, MemberQ, ! MemberQ]
[d, StringTake[b, {1, Quiet[StringPosition[b, "{", 1][[1]][[1]]]}],
Quiet[ToExpression[b]]; Break[], Continue[]]]]; Close[c]; DeleteFile[c]]

In[2093]:= ReloadPackage["C:\\AVZ_Package\\AVZ_Package_1.m"]
In[2094]:= Definition[FindFile1] (1)
Out[2094]= FindFile1[x_ /; StringQ[x]] := Module[{a = Adrive[], c, d = {}, k = 1,
AladjevProcedures`FindFile1`, p, b = "\\ " <> ToUpperCase[x]},
For[k, k <= Length[a], k++, p = a[[k]];
c = Quiet[FileNames["*", p <> ":\\ ", \[Infinity]]];
For[AladjevProcedures`FindFile1` = 1,
AladjevProcedures`FindFile1` <= Length[c],
AladjevProcedures`FindFile1`++,
p = c[[AladjevProcedures`FindFile1`]];
If[SuffPref[ToUpperCase[p], b, 2], d = Append[d, p]; Continue[],
Continue[]]; If[Length[d] == 1, d[[1]], d]]

In[2095]:= Definition[FindFile1] (2)
Out[2095]= FindFile1[x_ /; StringQ[x]] := Module[{a = Adrive[], c, d = {}, k = 1, j, p,
b = "\\ " <> ToUpperCase[x]}, For[k, k <= Length[a], k++, p = a[[k]];
c = Quiet[FileNames["*", p <> ":\\ ", \[Infinity]]]; For[j = 1, j <= Length[c], j++,
p = c[[j]]; If[SuffPref[ToUpperCase[p], b, 2], d = Append[d, p]; Continue[],
Continue[]]; If[Length[d] == 1, d[[1]], d]]

In[2096]:= ReloadPackage["C:\\AVZ_Package\\AVZ_Package_1.m", {Locals, ProcQ}, 78]

```

Успешный вызов процедуры **ReloadPackage[x]** ничего не возвращает, обеспечивая в текущем сеансе активацию всех средств пакета, находящегося в *m*-файле *x*, как если бы их определения вычислялись во *входном* потоке. Если же вызов **ReloadPackage[x, y]** содержит второй необязательный *y*-аргумент, в качестве которого выступает список имен, то перезагрузка производится лишь для средств пакета с указанными именами. Вызов же **ReloadPackage[x, y, t]** дополнительно с 3-м необязательным аргументом, где *t* – произвольное выражение, также ничего не возвращает, обеспечивая перезагрузку в текущем сеансе всех средств пакета *x*, исключая лишь средства пакета с указанными

в списке *y* именами. Предыдущий фрагмент представляет исходный код процедуры с примерами ее применения. В частности, иллюстрируется, что перезагрузка пакета обеспечивает более компактный вывод определений средств, содержащихся в нем, т. е. вывод определений производится в т.н. *оптимальном* формате (без контекстов).

Следующий фрагмент представляет исходный код наряду с примерами применения процедуры **ReloadPackage1**, функционально эквивалентной процедуре **ReloadPackage**.

```
In[4436]:= ReloadPackage1[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ : 0, t_ : 0] :=
Module[{a = NamesMPackage[x], b = ReadFullFile[x], c,
        d = Map[ToString, Flatten[{y}]]},
c = Flatten[Map[SubsString[b, {""}(*Begin["\" <> # <> "\""]*)(", ")(*End[""]*)"], 78] &, a]];
c = Map[StringReplace[#, {""}(*" -> "" &, c];
Map[If[d == {"0"}, ToExpression[#, If[ListQ[y], If[{t} == {0},
If[MemberQ[d, StringTake[#, Flatten[StringPosition[#, {"[", " :=", "="}]]][[1]] - 1]],
ToExpression[#, If[! MemberQ[d, StringTake[#,
Flatten[StringPosition[#, {"[", " :=", "="}]]][[1]] - 1]], ToExpression[#]]]]]] &, c];]

In[4437]:= Map[Clear, {StrStr, Map1}]
Out[4437]= {Null, Null}
In[4438]:= Definition[StrStr]
Out[4438]= Null
In[4439]:= Definition[Map1]
Out[4439]= Null
In[4440]:= ReloadPackage1["C:\\AVZ_Package\\AVZ_Package_1.m", {StrStr, Map1}]
In[4441]:= Definition[StrStr]
Out[4441]= StrStr[x_] := If[StringQ[x], "\" <> x <> "\" , ToString[x]]
In[4442]:= Definition[Map1]
Out[4442]= Map1[x_ /; ListQ[x] && DeleteDuplicates[(SymbolQ[#1] &) /@ x] === {True},
y_List] := (Symbol[ToString[#1]][Sequences[y]] &) /@ x
```

Успешный вызов процедуры **ReloadPackage1[x]** ничего не возвращает, обеспечивая в текущем сеансе активацию всех средств пакета, находящегося в *m*-файле *x*, как если бы их определения вычислялись во *входном* потоке. Если же вызов **ReloadPackage1[x,y]** содержит второй необязательный *y*-аргумент, в качестве которого выступает список имен, то перезагрузка производится лишь для средств пакета с указанными именами. Вызов же **ReloadPackage1[x,y,t]** дополнительно с 3-м необязательным аргументом, где *t* – произвольное выражение, также ничего не возвращает, обеспечивая перезагрузку в текущем сеансе всех средств пакета *x*, исключая лишь средства пакета с указанными в списке *y* именами. Подобно процедуре **ReloadPackage** процедура **ReloadPackage1**, в частности, также обеспечивает вывод определений в оптимальном формате. Данная модификация представляет интерес с точки зрения используемых в ней подходов.

Такой подход позволяет избавиться от *контекстных ссылок* в определениях функций/процедур, загруженных в текущий сеанс из пользовательского пакета. Для выявления

средств пакета, определения которых в текущем сеансе содержат *контекстные* ссылки, служит процедура, чей вызов **DefWithContext[x]** возвращает 2-элементный *вложенный* список: его первый элемент определяет список имен средств пакета, загруженного из файла *x*, определения которых не содержат контекстных ссылок, тогда как второй – список имен средств пакета, определения которых содержат контекстные ссылки. Во фрагменте представлены исходный код процедуры и примеры ее применения как до вызова процедуры **ReloadPackage1**, так и после него, что достаточно иллюстративно.

```
In[2982]:= DefWithContext[x_ /; FileExistsQ[x] && FileExtension[x] == ".m"] :=
           Module[{a = ContextMfile[x], b, c = {}, d = {}}, b = CNames[a];
           Map[If[StringFreeQ[Definition4[#], a <> # <> ""], AppendTo[c, #],
           AppendTo[d, #]] &, b]; {c, d}]

In[2983]:= DefWithContext["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2983]= {"AcNb", "ActCsProcFunc", "Adrive", "ArityBFM", "Attribs", "Attributes1",
           =====
           "$CallProc", "$Line1", "$Load$Files$", "$ProcType", "$TestArgsTypes"},
           {"ActBFMuserQ", "ActRemObj", "ActUcontexts", "Adrive1", "Affiliate",
           =====
           "VizContext", "WhatObj", "WhatType", "WhichN", "$ProcName", "$TypeProc"}]

In[2984]:= Map[Length, %]
Out[2984]= {126, 456}

In[2985]:= ReloadPackage1["C:\\AVZ_Package\\AVZ_Package_1.m"]
In[2986]:= DefWithContext["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2986]= {"AcNb", "ActBFMuserQ", ..., "$ProcType", "$TestArgsTypes", "$TypeProc"}, {}]
In[2987]:= Map[Length, %]
Out[2987]= {582, 0}
```

Более того, под результатом вычисления определения некоторого средства *x* здесь мы понимаем результат вызова пакетной функции **Definition[x]**, а не наших процедур, в частности, **DefOpt[x]**, возвращающих определения в компактном каноническом виде.

Из приведенного фрагмента следует, более 78% определений средств нашего пакета *AVZ\_Package\_1*, загруженного в текущий сеанс, полученных по системной функции **Definition[x]**, будут содержать контекстные ссылки формата *"AladjevProcedures`x`"*.

Еще раз на достаточно существенном моменте следует акцентировать внимание. При загрузке пакета пользователя в текущий сеанс соответствующий ему контекст занесен в список, определяемый переменной **\$Packages**, в то время как при попытке получить определения по функции **Definition** для целого ряда его средств, такие определения будут содержать контекст, ассоциированный с данным пакетом. Прежде всего, такие определения значительно *менее* читабельны, но не это самое главное. У программных средств оптимального формата, использующих подобные определения, при работе с ними возможны ошибочные ситуации, как это уже отмечалось выше. В данной связи ниже представлены три подхода к загрузке пакета пользователя в текущий сеанс. При этом, первые 2 подхода выполняются в режиме документа и целиком ориентированы

конкретно на наш пакет *AVZ\_Package\_1*, который расположен в файле *mx*-формата в текущем каталоге *Mathematica*, например, "C:\\Documents and Settings\\Viktor\\My Documents". В качестве примера ниже приводятся определения процедуры **ContextQ**, полученные сразу после загрузки пакета по **Get**-функции, и после перевычисления определений всех средств из отмеченного пакета. Результаты достаточно разительны уже в визуальном отношении. Более того, оба первых подхода различаются только по организации циклической **While**-конструкции, из которых вторая существенно более реактивна в случае достаточно большого количества содержащихся в пакете средств.

```
In[1]:= Quiet[Get["AVZ_Package_1.mx"]]
In[2]:= {a, k} = {CNames["AladjevProcedures`"], 1};
In[3]:= Definition["ContextQ"]
Out[3]= ContextQ[AladjevProceduresAndFunctions`ContextQ`x_] :=
    StringQ[AladjevProceduresAndFunctions`ContextQ`x] &&
    StringLength[AladjevProceduresAndFunctions`ContextQ`x] > 1 &&
    Quiet[SymbolQ[Symbol[StringTake[
        AladjevProceduresAndFunctions`ContextQ`x, {1, -2}]]]] &&
    StringTake[AladjevProceduresAndFunctions`ContextQ`x, {-1, -1}] == ""
In[4]:= While[k <= Length[a], ToExpression[Definition4[a[[k]]]]; k++]
In[5]:= Definition["ContextQ"]
Out[5]= ContextQ[x_] := StringQ[x] && StringLength[x] > 1 &&
    Quiet[SymbolQ[Symbol[StringTake[x, {1, -2}]]]] && StringTake[x, {-1, -1}] == ""
=====
In[4]:= While[k <= Length[a], ToExpression[StringReplace[Definition4[a[[k]]],
    "AladjevProcedures`" -> ""]]; k++]
=====
In[1]:= LoadMyPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_] :=
    Module[{a, Cn, Ts, k = 1},
        Ts[g_] := Module[{p = "$ArtKr$.txt", b = "", c, d, v = 1}, Write[p, g]; Close[p];
        While[v < Infinity, c = Read[p, String]; If[SameQ[c, EndOfFile],
            Close[p]; DeleteFile[p]; Return[b], b = b <> c]; Continue[]];
        Cn[t_] := Module[{s = Names[StringJoin[t, "*"]], b}, b = Select[s,
            Quiet[ToString[Definition[ToString[#1]]]] != "Null" &];
            Quiet[Get[x]]; a = Cn[y]; While[k <= Length[a],
            Quiet[ToExpression[StringReplace[StringReplace[Ts[ToExpression[
                "Definition[" <> a[[k]] <> ""]], y -> ""], a[[k]] <> "" -> ""]]; k++]]
In[2]:= LoadMyPackage["AVZ_Package_1.mx", "AladjevProcedures`"]
In[3]:= Definition["ContextQ"]
Out[3]= ContextQ[x_] := StringQ[x] && StringLength[x] > 1 &&
    Quiet[SymbolQ[Symbol[StringTake[x, {1, -2}]]]] && StringTake[x, {-1, -1}] == ""
```

Тогда как третий подход базируется на процедуре и ориентирован на произвольный пакет пользователя. Вызов процедуры **LoadMyPackage[x, y]** в самом начале текущего

сеанса *Mathematica* возвращает **Null**, т.е. ничего, загружая пользовательский пакет *x* с приписанным ему контекстом *y*, с последующим перевычислением определений его средств, что обеспечивает оптимальный формат таких определений. В предыдущем фрагменте представлен исходный код процедуры с примером ее применения. Такой именно подход рекомендуется применять при загрузке пакета пользователя, включая пакет *AVZ\_Package\_1*, сохраненный в файле *mx*-формата, для устранения указанных нежелательных моментов и для упрощения программирования с использованием его средств, а также для расширения собственно самого пакета на основе его средств. При этом, вызов **LoadMyPackage**[*x*, *y*] с отмеченными целями может быть выполнен и при загруженном в текущий сеанс пользовательском пакете *x* с контекстом *y*. Более того, наиболее эффективно сохранение пакета в *mx*-файле с последующей загрузкой его в каждом новом сеансе по функции **Get**, обеспечивая доступ ко всем средствам пакета с получением их определений в оптимизированном (в вышеуказанном смысле) формате.

До сих пор рассматривалась явная загрузка пакета пользователя по функциям **Needs**, **Get**, процедуре **Need** и в автоматическом режиме по функции **DeclarePackage**. Между тем, в целом ряде случаев целесообразно загружать пакет в момент начальной загрузки *Mathematica*, как это имеет место для целого ряда стандартных пакетов *Mathematica*. Для этого можно использовать инициализационные файлы с именами «*init.m*», которые располагаются в ряде подкаталогов *Mathematica*. Эти файлы содержат коды, которые выполняются каждый раз при загрузке ядра или препроцессора *Mathematica*. Между тем, для допустимого расположения инициализационных файлов в целях обеспечения ими требуемых функций можно использовать следующие подкаталоги *Mathematica*:

**\$BaseDirectory/Kernel** – файл инициализации ядра для всех пользователей

**\$UserBaseDirectory/Kernel** – файл инициализации ядра для текущего пользователя (ТП)

**\$BaseDirectory/FrontEnd** – файл инициализации препроцессора для всех пользователей

**\$UserBaseDirectory/FrontEnd** – файл инициализации препроцессора для ТП

Для конкретной версии *Mathematica 9.0.1.0* данные подкаталоги принимают вид:

```
In[1]:= $BaseDirectory <> "\\Kernel"
```

```
Out[1]= "C:\Documents and Settings\All Users\Application Data\Mathematica\Kernel"
```

```
In[2]:= $UserBaseDirectory <> "\\Kernel"
```

```
Out[2]= "C:\Documents and Settings\Viktor\Application Data\Mathematica\Kernel"
```

```
In[3]:= $BaseDirectory <> "\\FrontEnd"
```

```
Out[3]= "C:\Documents and Settings\All Users\Application Data\Mathematica\FrontEnd"
```

```
In[4]:= $UserBaseDirectory <> "\\FrontEnd"
```

```
Out[4]= "C:\Documents and Settings\Viktor\Application Data\Mathematica\FrontEnd"
```

Для удобства автоматической загрузки пакета пользователя при запуске *Mathematica* можно воспользоваться вызовом системной функции **Get**[*W*], производящей загрузку файла *W* *mx*-формата с пользовательским пакетом, расположенном в произвольном подкаталоге файловой системы компьютера, в текущий сеанс. Данный вызов можно поместить в инициализационный файл, например, в «*init.m*» из подкаталога, который предназначен для инициализации ядра для текущего пользователя, в частности:

**"C:\\Documents and Settings\\Viktor\\Application Data\\Mathematica\\Kernel"**

Выполнить эту процедуру можно посредством, например, *Notepad*, т.к. файл «*init.m*» имеет *txt*-формат. После чего содержимое файла «*init.m*» принимает следующий вид

```
(** User Mathematica initialization file - 20.11.2013 **)
Get["C:\\AVZ_Package\\AVZ_Package_1.mx"]
```

Такой подход обеспечивает автоматическую загрузку пакета пользователя в текущий сеанс сразу же при загрузке ядра *Mathematica*, с первых же шагов пользователя давая возможность использовать *все* находящиеся в данном файле определения функций и/или процедур наряду с определениями других объектов. Между тем, следует иметь в виду, что такое решение целесообразно лишь для пакетов, содержащих весьма часто используемые средства, предназначенные для решения задач пользователя. При этом, предварительно хорошо протестированный пакет рекомендуется сохранять в файле именно *mx*-формата, обеспечивая его наиболее быструю загрузку в текущий сеанс. В целом, подготовка такого файла *mx*-формата с пакетом пользователя довольно проста и проводить ее рекомендуется в следующие три этапа, а именно:

1. Загрузить в текущий сеанс *nb*-документ с отлаженным пакетом с ассоциированным контекстом, скажем, "Context`";
2. Вычислить загруженный в текущий сеанс *nb*-документ;
3. Выполнить вызов `DumpSave["C:\\...\\Package_Name.mx", "Context`"]`, в результате которого создается искомый *mx*-файл с пакетом `Package_Name`, например:

```
In[100]:= DumpSave["C:\\AVZ_Package\\AVZ_Package_1.mx", "AladjevProcedures`"]
Out[100]= {"AladjevProcedures`"}
```

Единственным недостатком подобной организации режима автоматической загрузки пользовательского пакета при загрузке системы *Mathematica* является обстоятельство, что загружаемый пакет *непереносим* на вычислительные платформы, отличные от той, на которой он был создан. Однако, ввиду его направленности на конкретную версию это не является сколько-нибудь существенным недостатком, полагая, что его копии в текстовых файлах форматов *{m, nb}* являются вполне переносимыми. Представленные выше две процедуры `DumpSave1` и `DumpSave2` расширяют функцию `DumpSave`.

Следующий фрагмент представляет результат *тестирования* успешности загрузки в текущий сеанс (*сразу после запуска Mathematica*) определений функций и процедур из пакета *AVZ\_Package\_1*, загрузка которого определена в инициализационном файле.

```
In[1]:= ?Df
Df[x, y] returns the result of differentiation of x on y. The procedure expands the
standard function Df.
In[2]:= Map[ProcQ, {ProcQ, Spos, BlockQ, QFunction, Subs, Locals, SortNL, NamesProc,
Mapp, Df, Uprocs, FunCompose, UprocQ}]
Out[2]= {True, True, True, True, True, True, False, False, True, True, True, True, True}
In[3]:= Map[QFunction, {ProcQ, Spos, BlockQ, QFunction, Subs, Locals, SortNL,
NamesProc, Mapp, Df, Uprocs, FunCompose, UprocQ}]
Out[3]= {False, False, False, False, False, False, True, True, False, False, False, False, False}
```

```
In[4]:= Directory[]
```

```
Out[4]= "C:\\Documents and Settings\\Viktor\\My Documents"
```

Пользовательские файлы инициализации подкаталога **\$UserBaseDirectory** читаются после файлов инициализации **\$BaseDirectory**, поэтому одноименные символы будут определены именно пользовательскими файлами инициализации. При этом, *"init.m"* для ядра в подкаталогах пользователей автоматически создаются *пустыми*, если ранее они не были определены пользователем. Таким образом, данные файлы можно легко *модифицировать* под конкретные нужды пользователей, что было проиллюстрировано нами выше. Если в процессе выполнения *Mathematica* изменяются препроцессорные опции, то соответствующим образом модифицируется и препроцессорный *"init.m"* пользователя, тогда как системный препроцессорный файл *"init.m"* не изменяется. В целом, при запуске ядра *Mathematica* первыми выполняет следующие процедуры:

- выполняет процедуру проверки на лицензионную допустимость системы;
- выполняет команды, указанные в опциях *-run*, передаваемых ядру системы;
- выполняет процедуры из системного файла **\$BaseDirectory/Kernel/init.m**;
- выполняет процедуры из файла пользователя **\$UserBaseDirectory/Kernel/init.m**;
- загружает файлы *"init.m"* и *"Kernel/init.m"* из подкаталогов *"Autoload"*;
- система начинает выполнение своего основного цикла.

В деталях с вопросами использования файлов *инициализации* возможно ознакомиться по справочной системе *Mathematica* либо в соответствующей литературе по системе.

В случае явной либо автоматической загрузки пакета в текущий сеанс по умолчанию полагается, что он располагается в *одном* из подкаталогов, определяемых переменной **\$Path**. Но так как местоположение пользовательского подкаталога с пакетами ничем, вообще говоря, не ограничивается, то для обеспечения успешной загрузки в текущий сеанс таких пакетов требуется *переопределять* текущий каталог сеанса. Именно так мы и поступали выше. Между тем, в ряде случаев имеется более простой путь, состоящий в переопределении системной переменной **\$Path** на весь период текущего сеанса; он сводится к расширению списка подкаталогов, определяемых **\$Path**, на один или более подкаталогов. Данную функцию можно осуществлять процедурой **UpdatePath[x]**, чей аргумент определяет путь к одному или списку подкаталогов файловой системы *ПК*. Исходный код процедуры с примерами обновления переменной **\$Path** представлены выше. Обновление переменной **\$Path** тестируется получением ее нового значения и возможностью *загрузки* в текущий сеанс пакета, находящегося в подкаталоге, который до обновления переменной **\$Path** не входил в список ее подкаталогов. Эта процедура может быть использована и в инициализационных файлах *"init.m"* для обновления системной переменной **\$Path**, упрощая загрузку пакетов в текущий сеанс сразу после запуска *Mathematica*; при этом, переопределение системной переменной **\$Path** может преследовать и ряд других целей, облегчающих программирование в *Mathematica*. В следующем фрагменте первая часть представляет содержимое файла *"init.m"*, тогда как вторая представляет результат вызова процедуры **ProcQ** из пакета *AVZ\_Package\_1* для тестирования ряда средств, чьи определения находятся также в этом же пакете, а также проверяет наличие дополнительных подкаталогов в переменной **\$Path**.

```
(** User Mathematica initialization file - 20.11.2013 **)

ListStringQ[x_] := ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True};
UpdatePath[x_/: StringQ[x] || ListStringQ[x]] := Module[{}, ClearAttributes[$Path,
Protected]; $Path = DeleteDuplicates[Flatten[AppendTo[$Path, x]]];
SetAttributes[$Path, Protected];

UpdatePath[{"C:\\AVZ_Package", "D:\\UserLib6789", "F:\\New_Book_2013"}];
Get["AVZ_Package_1.mx"]

=====
In[1]:= Map[ProcQ, {ProcQ, Spos, BlockQ, FunctionQ, Subs, SortNL, Locals, NamesProc,
Mapp, Df, Uprocs, FunCompose, UprocQ}]
Out[1]= {True, True, True, False, True, False, True, False, True, True, True, True}
In[2]:= $Path[[-3 ;; -1]]
Out[2]= {"D:\\AVZ_Package", "D:\\UserLib6789", "F:\\New_Book_2013"}
In[3]:= Map[UprocQ, {ProcQ, Uprocs, FunCompose, UprocQ}]
Out[3]= {{True, Module}, {True, Module}, {True, Module}, {True, Module}}
In[4]:= $Packages
Out[4]= {"GetFEKernelInit", "AladjevProcedures", "ResourceLocator", "PacletManager",
"QuantityUnits", "WebServices", "System", "Global"}
```

При этом, следует отметить, что автоматически загруженный в процессе запуска ядра контекст пакета *AVZ\_Package\_1* отражается и в системной переменной **\$Packages**. Для упрощения реализации процедуры **UpdatePath** применена весьма простая функция, исходный код которой представлен началом фрагмента, вызов которой **ListStringQ[x]** возвращает **True**, если *x* – строка либо список строк, и **False** в противном случае. Более того, в процессе загрузки ядра системы в случае наличия инициализационного файла *"init.m"*, определяющего автоматическую загрузку пакета пользователя, вполне могут появляться сообщения, обусловленные наличием *одноименных* средств, определяемых как в файла *"init.m"*, так и в загружаемом пакете пользователя, которые, как правило, носят информационный характер, не связанный с последующими вычислениями.

Тогда как для обновления переменных **\$Packages**, **\$ContextPath** новыми контекстами используются процедуры **UpdatePackages** и **UpdateContextPaths** соответственно, чьи исходные коды с примерами использования представляет следующий фрагмент.

```
In[2337]:= UpdatePackages[x_/: StringQ[x] || ListStringQ[x]] := Module[{},
ClearAttributes[$Packages, Protected];
$Packages = DeleteDuplicates[Flatten[ReleaseHold[Map[AppendTo[$Packages, #] &,
Flatten[{x}]]]]]; SetAttributes[$Packages, Protected]; ]

In[2338]:= UpdatePackages[{"Kherson", "Kiev"}]
In[2339]:= $Packages
Out[2339]= {"JLink", "DocumentationSearch", "HTTPClient`OAuth",
"HTTPClient`CURLInfo", "HTTPClient`CURLLink", "HTTPClient",
```



```

"GetFEKernelInit`, "AladjevProcedures`, "ResourceLocator`,
"PacletManager`, "QuantityUnits`, "WebServices`, "System`, "Global`,
"Kherson`, "Kiev`"
In[2340]:= Attributes[$Packages]
Out[2340]= {Protected}
In[2350]:= UpdateContextPaths[x_/: StringQ[x] || ListStringQ[x]] := Module[{},
    $ContextPath = DeleteDuplicates[Flatten[ReleaseHold[
        Map[AppendTo[$ContextPath, #] &, Flatten[{x}]]]]; ]
In[2351]:= $ContextPath
Out[2351]= {"AladjevProcedures`, "PacletManager`, "QuantityUnits`, "WebServices`,
    "System`, "Global`"}
In[2352]:= UpdateContextPaths[{"Kherson`, "Kiev`"}]; $ContextPath
Out[2352]= {"AladjevProcedures`, "PacletManager`, "QuantityUnits`, "WebServices`,
    "System`, "Global`, "Kherson`, "Kiev`"}

```

В качестве аргумента  $x$  обеих процедур выступает контекст, который ассоциирован с пакетом пользователя, или их список; при этом, вызовы процедур **UpdatePackages** $[x]$ , **UpdateContextPaths** $[x]$  возвращают **Null**, т.е. ничего, обновляя переменные **\$Packages** и **\$ContextPath** соответственно. Однако, следует иметь в виду, что обновление данных предопределенных переменных производится формально, не учитывая фактическое существование контекстов, обновляющих указанные переменные. Данные процедуры вполне могут сочетаться с ранее представленными средствами по работе с пакетами пользователя, в ряде случаев существенно упрощая программирование приложений.

Как отмечалось ранее, в результате загрузки в текущий сеанс пакета пользователя из файла формата  $\{m, nb\}$  с его последующей активизацией большая часть определений его средств, получаемых по вызову стандартной функции **Definition**, будет включать контекстные ссылки формата **"Context`x`"**, где  $x$  – имя средства и **"Context"** – контекст, приписанный данному пакету. Выше представлены средства выявления тех объектов пакета, определения которых имеют контекстные ссылки. Но эти средства полагают, что анализируемый пакет активирован в текущем сеансе. Тогда как нижеследующая процедура обеспечивает такого типа анализ незагруженного пакета, расположенного в файле  $m$ x-формата, ранее полученном из загруженного и активированного файла, например,  $m$ x-формата с пакетом. Фрагмент представляет исходный код процедуры **MxPackNames** с примером ее применения. Вызов **MxPackNames** $[x]$  возвращает список имен объектов в строчном формате  $nb$ -файла  $y$  – аналога файла  $x$ , чьи определения в случае загрузки данного файла  $y$  в текущий сеанс и активизации системная функция **Definition** будет возвращать с контекстными ссылками вышеуказанного формата.

```

In[3235]:= MxPackNames[x_/: FileExistsQ[x] && FileExtension[x] == "mx"] := Module[{b,
    a = FromCharacterCode[Select[ToCharacterCode[ReadFullFile[x]], # > 31 &]],
    b = Substring[a, {"CONT", "END"}, 78][[1]];
    Sort[Select[Substring[a, {b, ""}, 78], StringFreeQ[#, {"ENDCONT", "ÿ"}] &]]]

```

```
In[3236]:= MxPackNames["C:\\AVZ_Package\\AVZ_Package_1.mx"]
Out[3236]= {"ActBFMuserQ", "ActRemObj", "ActUcontexts", "Adrive1", "Affiliate", "Aobj",
  "Args", "ArgsBFM", "ArgsTypes", "Arity", "ArrayInd", "AssignL", "AtomicQ",
  =====
  "Unique1", "Uprocs", "UserLib", "VarExch", "VarExch1", "Ver", "VizContext",
  "WhatObj", "WhatType", "WhichN", "$ProcName", "$TypeProc"}
In[3237]:= Length[%]
Out[3237]= 458
In[3238]:= N[458*100/Length[CNames["AladjevProcedures`"]], 3]
Out[3238]= 78.7
```

Примеры предыдущего фрагмента еще раз подтверждают, что число средств нашего пакета *AVZ\_Package\_1*, загруженного в текущий сеанс из *nb*-файла, чьи определения по системной функции **Definition** содержат контекстные ссылки, более 78%.

Довольно интересным представляется получение списка *имен* объектов, определения которых находятся в пакете, находящимся в файле данных формата *{m, nb}*, в пакете по которым имеются справки (*usage*). При этом, предполагается, что загрузка пакета в текущий сеанс не обязательна. Такую задачу решает довольно полезная процедура, чей вызов **PackNames[x]** возвращает список имен вышеуказанных объектов в пакете, находящимся в файле данных *x* формата *{m, nb}*. Следующий фрагмент представляет исходный код процедуры с примером применения к пакету *AVZ\_Package\_1*, который находится в файлах данных *"AVZ\_Package\_1.m"* и *"AVZ\_Package\_1.nb"* [120].

```
In[2100]:= PackNames[x_ /; FileExistsQ[x] && MemberQ[{"m", "nb"}, FileExtension[x]] :=
  Module[{a = ReadFullFile[x], b, c = {}, d = "", k = 1, j, h},
    If[FileExtension[x] == "m",
      a = StringReplace[a, {ContextMfile[x] <> "\""} -> "#", "(**)(*" -> "#"}];
      b = Substring1[a, {"#", "::usage="}, StringQ[#] &, 0];
      c = DeleteDuplicates[Sort[Select[b, StringFreeQ[#,
        Characters["~!@#%^&*()_+{}<>?:|`-=[];'\\/,."]] || SymbolQ[#] &]]]; Sort[c],
      b = Quiet[Substring1[a, {"RowBox[{ RowBox[{ RowBox[{", "\"::\"", "\"usage\""}]}],
        StringQ[#] &, 0]];
      b = Map[StringTake[#, {2, -8}] &, b]; Sort[Map[If[SymbolQ[#, #] &, b]]]]
In[2101]:= PackNames["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2101]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
  "Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes",
  =====
  "WhatObj", "WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1",
  "$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}
In[2102]:= Length[%]
Out[2102]= 567
In[2103]:= PackNames["C:\\AVZ_Package\\AVZ_Package_1.nb"]
```

```
Out[2103]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
            "Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes",
            =====
            "WhatObj", "WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1",
            "$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}
In[2104]:= Length[%]
Out[2104]= 577
```

Отметим, в реализации процедуры **PackNames** существенно использована процедура **SubsString1**, являющаяся расширением процедуры **SubsString**, представляя интерес в программировании задач, связанных с обработкой строк. Вызов **SubsString1[s, y, pf, t]** возвращает список подстрок строки *s*, ограниченных подстроками списка *y*; при этом, если в качестве аргумента *pf* выступает тестирующая чистая функция, то выходной список будет содержать лишь подстроки, удовлетворяющие данной функции. Более того при *t = 1* возвращаемые подстроки ограничены крайними подстроками списка *y*, тогда как при *t = 0* подстроки возвращаются без ограничивающих крайних подстрок списка *y*. Наконец, при наличии пятого необязательного аргумента *r* – произвольного выражения – поиск подстрок в строке *s* производится справа налево, что в целом ряде упрощает алгоритмы поиска искомых подстрок. В частности, **SubsString1** позволила существенно упростить алгоритм процедуры **PackNames**. Нижеследующий фрагмент представляет исходный код процедуры **SubsString1** с примерами ее использования.

```
In[2159]:= SubsString1[s_String, y_List, pf_, t_ /; MemberQ[{0, 1}, t], r_] := Module[{c, h,
            a = "", b = Map[ToString1, y], d = s, k = 1},
            If[Set[c, Length[y]] < 2, s,
            If[{r} != {}, b = Map[StringReverse, Reverse[b]]; d = StringReverse[s]];
            While[k <= c - 1, a = a <> b[[k]] <> "~~ Shortest[___] ~~ "; k++];
            a = a <> b[[-1]]; h = StringCases[d, ToExpression[a]];
            If[t == 0, h = Map[StringTake[#, {StringLength[b[[1]]] - 1, -StringLength[b[[-1]]] + 1}] &, h]];
            If[PureFuncQ[pf], h = Select[h, pf]];
            If[{r} != {}, Reverse[Map[StringReverse, h]], h]]

In[2160]:= SubsString1["12345#xyztttttabc::usage=45678", {"#", "::usage=4"}, 0, 0]
Out[2160]= {"xyztttttabc"}
In[2161]:= SubsString1["12345#xyz::usage=45612345#x78yz::usage=456#Avz::usage=460",
            {"#", "::usage="}, 0, 0]
Out[2161]= {"xyz", "x78yz", "Avz"}
In[2162]:= SubsString1["12345#xyz::usage=45612345#x78yz::usage=456#Avz::usage=460",
            {"#", "::usage="}, 0, 1]
Out[2162]= {"#xyz::usage=", "#x78yz::usage=", "#Avz::usage="}
In[2163]:= SubsString1["12345#xyz::usage=45612345#x78yz::usage=456#Avz::usage=460",
            {"#", "::usage="}, LetterQ[#] &, 0]
Out[2163]= {"xyz", "Avz"}
```

```
In[2164]:= Substring1["12345#xyz::usage=45612345#x78yz::usage=456#Avz::usage=460",
                    {"#", "::usage="}, LetterQ[#] &, 0, 78]
Out[2164]= {"xyz", "Avz"}
```

Здесь в связи с вышесказанным вполне уместно затронуть еще *один* довольно важный вопрос, касающийся глобальных переменных, определяемых процедурой, а именно. Согласно соглашениям процедурного программирования переменная, определяемая в процедуре *глобальной*, видима извне процедуры, т.е. может изменять свое значение как внутри самой процедуры, так и вне ее, точнее, областью ее определения является текущий сеанс соответствующей программной среды. В принципе, данное *соглашение* справедливо и для текущего сеанса пакета *Mathematica*, но с весьма существенными оговорками. Если процедура, определяющая *глобальные* переменные, активизирована в *Input*-режиме пакета, то упомянутое соглашение имеет силу. Между тем, если такая процедура предварительно была *сохранена* в файле *{m|nb}*-формата, то последующая загрузка такого файла в новый сеанс пакета активизирует все содержащиеся в файле средства, делая их доступными, однако механизм *глобальных* переменных не работает. Следующий фрагмент наглядно иллюстрирует указанное важное обстоятельство на примере достаточно простой процедуры *VSG*, а именно:

```
In[2108]:= VSG[x_, N_;/ IntegerQ[N], y_, z_;/ StringQ[z], L_;/ ListQ[L] &&
           MemberQ[{{0}, {1}, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]] :=
           Block[{}, L[[StringLength[y <> z] + N]]]

In[2109]:= LoadNameFromM[F_;/ FileExistsQ[F] && FileExtension[F] == ".m" &&
           StringTake[ToString[ContextFromFile[F]], -1] == "",
           p_;/ StringQ[p] || ListStringQ[p]] :=
           Module[{a = ReadFullFile[F], b, c = ""}(*End[*]), h = Flatten[{p}],
           b = Map[Substring[a, {"(*Begin[\"\" <> # <> \"\"]*)(\"\", c), 78] &, h];
           b = If[Length[b] == 1, Flatten[b], Map[#[[1]] &, b]];
           Map[ToExpression, Map[StringReplace[#, ""(*)->"] &, b]]; ]

In[2111]:= LoadNameFromM["F:\\Math_myLib\\AVZ_Package_1.m", {"StrStr", "ProcQ",
           "TestArgsTypes", "Substring1", "PackNames", "Map3"}]

Out[2111]= Null
In[2112]:= TestArgsTypes[VSG, VSG[9, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[2112]= $Failed
In[2113]:= $TestArgsTypes
Out[2113]= {2, 7.2}
In[2114]:= TestArgsTypes[VSG, VSG[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[2114]= $Failed
In[2115]:= $TestArgsTypes
Out[2115]= {{2, 7.2}, {4, vz}, {5, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}}}
```

Рассмотренная процедура *TestArgsTypes*, определяющая глобальную *\$TestArgsTypes*-переменную, сохраняется в файле *"F:/Math\_MyLib/avz\_package.nb"*, который в новом

сеансе загружается по цепочке *GUI*: «File → Open → Evaluate Notebook», обеспечивая доступность всех содержащихся в файле процедур и функций, включая и процедуру **TestArgsTypes**. Для устранения этой ситуации в пакет с определением **TestArgsTypes** помещается и произвольное определение для **\$TestArgsTypes**-переменной, например, **\$TestArgsTypes:=78**. В результате этого указанная ситуация устраняется и переменная **\$TestArgsTypes** выступает глобальной для всех объектов, в определениях которых она определена *глобальной*. В противном случае данная переменная системой *Mathematica* глобальной может и не рассматриваться. В качестве второго приема можно повторно загружать в текущий сеанс *объект*, чье определение содержит *глобальную* переменную.

Для обеспечения механизма *глобальных* переменных (*в том числе*) создана достаточно полезная процедура **LoadNameFromM[F, N]**, обеспечивающая загрузку и активацию в текущем сеансе системы процедуры *N* либо их списка, сохраненных в файле данных *F* *m*-формата с пакетом. Предыдущий фрагмент представляет исходный код данной процедуры с примерами ее использования для загрузки процедуры **TestArgsTypes** с сопутствующими ей еще пятью процедурами наряду с последующей иллюстрацией значений, возвращаемых глобальной **\$TestArgsTypes**-переменной. К этой процедуре в определенном отношении примыкает и нижеследующая процедура **ExtrPackName**.

Основываясь на внутренней структуре файла *m*-формата с пакетом, представляется достаточно полезной процедура **ExtrPackName** для обработки файлов данного типа. Успешный вызов **ExtrPackName[F, N]** процедуры возвращает **Null** с одновременным возвратом вычисленного определения объекта *N*, которое содержалось в *m*-файле *F* с пакетом, делая это определение доступным в текущем сеансе. Если же формат файла *F* отличен от *m*-формата, то вызов процедуры возвращает значение **\$Failed**, тогда как в отсутствие в файле *F* запрошенного объекта *N* вызов **ExtrPackName[F, N]** процедуры возвращает соответствующее сообщение. Следующий фрагмент приводит исходный код процедуры **ExtrPackName** наряду с типичными примерами ее использования.

```
In[2132]:= ExtrPackName[F_;/; StringQ[F], N_;/; StringQ[N]] := Module[{a, b, c, d, Art, Kr},
    If[FileExistsQ[F] && FileExtension[F] == "m" &&
    StringTake[ToString[ContextFromFile[F]], -1] == "", a = OpenRead[F], Return[$Failed]];
    If[Read[a, String] != "(* ::Package:: *)", Close[a]; $Failed,
    {c, d} = {"", StringReplace["(*Begin[\"`Z`\"]*)", "Z" -> N]};
    Label[Art]; b = Read[a, String]; If[b == EndOfFile, Close[a];
    Return["Definition of " <> N <> " is absent in file <" <> F <> ">"], Null];
    If[b != d, Goto[Art], Label[Kr]; b = StringTake[Read[a, String], {3, -3}];
    c = c <> b <> " "; If[b == "End[]", Close[a];
    Return[ToExpression[StringTake[c, {1, -8}]]], Goto[Kr]]];

In[2133]:= ExtrPackName["F:\\AVZ_Package\\AVZ_Package_1.m", "Df"]
In[2134]:= ExtrPackName["F:\\AVZ_Package\\AVZ_Package_1.m", "Subs"]
In[2135]:= ExtrPackName["F:\\AVZ_Package\\AVZ_Package_1.m", "HowAct"]
In[2136]:= ExtrPackName["F:\\AVZ_Package\\AVZ_Package_1.m", "ArtKr"]
Out[2136]= "Definition of ArtKr is absent in file <F:\\AVZ_Package\\AVZ_Package_1.m>"
```

```
In[2137]:= ExtrPackName["C:\\AVZ_Package\\AVZ_Package_6.m", "ProcQ"]
Out[2137]= $Failed
In[2138]:= Df[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2]
Out[2138]= x^2 (- (-1 + x^2) Cos[1/x^2] - (1 + x^2) Sin[1/x^2])
In[2139]:= Subs[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2, h]
Out[2139]= (Cos[h] + Sin[h])/h
In[2140]:= ExtrPackName["C:\\AVZ_Package\\Kherson.m", "Gs"]; Definition[Gs]
Out[2140]= Gs[x_Integer, y_Integer] := x^2 + y^2
```

Данная процедура обеспечивает активацию в текущем сеансе конкретной функции или процедуры, находящейся в *m*-файле данных, без загрузки файла полностью. По функциональности данная процедура пересекается с процедурой **LoadNameFromM**, рассмотренной выше, однако обладает рядом полезных возможностей.

В качестве дополнения к вышеприведенным средствам определенный интерес может представить процедура **NamesNbPackage[W]**, возвращающая список *имен* в строчном формате всех средств, находящихся в файле *W* *nb*-формата с пакетом и снабженных краткими справками "*usages*" по их использованию в среде системы *Mathematica*. В следующем фрагменте представляется исходный код процедуры **NamesNbPackage** с примером ее применения к *nb*-файлу "*D:\\Math\_myLib\\AVZ\_Package.nb*". Тогда как вызов процедуры **NamesNbPackage1[W]** – довольно эффективной модификации предыдущей процедуры – возвращает аналогичный список *имен* в строчном формате всех средств, находящихся в файле данных *W* *nb*-формата с пакетом; предполагается, что все средства снабжены краткими справками (*usages*) по их использованию в среде системы *Mathematica*, в отсутствие таких средств возвращается пустой список, т.е. {}.

```
In[2208]:= NamesNbPackage[F_;/; IsFile[F] && FileExtension[F] == "nb" &&
! SameQ[ContextFromFile[F], $Failed]] := Module[{Res = {}, Tr},
Tr[x_;/; StringQ[x]] := Module[{c, d, h, g = "\"::\"", v = "\"="\",
p = "\"usage\"", a = OpenRead[x], s = " RowBox[{",
Label[c]; d = Read[a, String]; If[d == EndOfFile, Close[a]; Return[Res], Null];
If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False} &&
SuffPref[d, s, 1], h = Flatten[StringPosition[d, g]];
Res = Append[Res, StringTake[d, {12, h[[1]] - 3}]]; Goto[c], Goto[c]];
Map[ToExpression, Sort[Tr[F]]]]

In[2209]:= NamesNbPackage["C:\\AVZ_Package\\AVZ_Package_1.nb"]
Out[2209]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
"Arity", "ArityBFM", "ArrayInd", "AssignL", "AssignToList", "AtomicQ",
"Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes",
====
"WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1", "$Load$Files$",
"$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}
```

```

In[2210]:= NamesNbPackage1[F_;/ IsFile[F] && FileExtension[F] == ".nb" &&
! SameQ[ContextFromFile[F], $Failed]] := Module[{c, d, g = ":",
a = OpenRead[F], p = "usage", v = "=", Res = {}, s = "  RowBox[{}],
Label[c]; d = Read[a, String]; If[d === EndOfFile, Close[a];
Return[Sort[Map[ToExpression, Res]]],
If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False} && SuffPref[d, s, 1],
Res = Append[Res, StringReplace[StringSplit[d, ", "][[1], s -> ""]]; Goto[c]; Goto[c]]]

In[2211]= NamesNbPackage1["C:\\AVZ_Package\\AVZ_Package_1.nb"]
Out[2211]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
"Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes", ....
"WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1", "$Load$Files$",
"$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}

In[2212]:= Length[%]
Out[2212]= 576

```

Следующая процедура **NamesMPackage** представляет собой аналог *двух* предыдущих процедур **NamesNbPackage**, **NamesNbPackage1**, ориентированном на случай пакетов пользователя, которые находятся в файлах *m*-формата. Успешный вызов процедуры **NamesMPackage[W]** возвращает список имен объектов в строчном формате, которые были сохранены в пакетном файле *W* *m*-формата; при этом, полагается, что объекты снабжены краткими справками «*usages*» по использованию в среде *Mathematica*. При отсутствии таких объектов возвращается *пустой* список, т.е. {}. Фрагмент представляет исходный код процедуры **NamesMPackage** наряду с примером применения. Данная процедура хорошо дополняет процедуры **NamesNbPackage** и **NamesNbPackage1**.

```

In[2233]:= NamesMPackage[F_;/ IsFile[F] && FileExtension[F] == ".m" &&
! SameQ[ContextFromFile[F], $Failed]] := Module[{c, d, Res = {},
s = ":", usage = "\", a = OpenRead[F]},
Label[c]; d = Read[a, String]; If[SuffPref[d, "(*Begin[\"", 1] ||
d === EndOfFile, Close[a]; Return[Sort[DeleteDuplicates[Res]]],
If[SuffPref[d, "(*", 1] && ! StringFreeQ[d, s], Res = Append[Res,
StringTake[d, {3, Flatten[StringPosition[d, s]][[1]] - 1}]]; Goto[c], Goto[c]]]

In[2234]:= NamesMPackage["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2234]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
=====
"WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1", "$Load$Files$",
"$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}

In[2235]:= Length[%]
Out[2235]= 581

```

Представленная во фрагменте функция **ContextFromFile[x]** обобщает три процедуры **ContextMfile**, **ContextMXfile** и **ContextNBfile**, возвращая контекст, ассоциированный с пакетами, сохраненными в файлах формата {*cdf*, *m*, *mx*, *nb*}, и значение **\$Failed** иначе.

Рассмотрев вопрос извлечения определений функций и процедур из пакета, который расположен в файле *m*-формата, представим процедуру, решающую данную задачу для пакета, расположенного в файле формата {"cdf", "nb"}. При этом, принципиальная организация файла данных форматов с пакетом представляется в начале следующего фрагмента, используемого и в качестве одного из примеров для данной процедуры.

```
BeginPackage["Kherson`"]
Gs::usage = "Help on Gs."
Ga::usage = "Help on Ga."
Vgs::usage = "Help on Vgs."
GSV::usage = "Help on GSV."
Begin["`Private`"]
Sv[x_] := x^2 + 23*x + 16
End[]
Begin["`Gs`"]
Gs[x_Integer, y_Integer] := x^2 + y^2
End[]
Begin["`Ga`"]
Ga[x_Integer, y_Integer] := x*y + Gs[x, y]
End[]
Begin["`Vgs`"]
Vgs[x_Integer, y_Integer] := x*y
End[]
Begin["`GSV`"]
GSV[x_Integer, y_Integer] := Module[{a = 78, b = 460, c = 2013}, x*y + Gs[x, y]*(a + b + c)] +
a*Sin[x]/(b+c)*Cos[y]
End[]
EndPackage[]
In[995]:= ExtrFromNBfile[x_;/ FileExistsQ[x] && MemberQ[{"cdf", "nb"}, FileExtension[x],
n_String] := Module[{a = ToString[InputForm[Get[x]]],
b = "" <> n <> "", c = "RowBox[List[RowBox[List[" k],
a = StringReplace[a, {"\`\"\\[IndentingNewLine]\"`" -> "", "\"\\n\"`" -> ""}]];
If[StringFreeQ[a, b], $Failed,
a = StringTake[a, {Flatten[StringPosition[a, b]][[1]] + 1, -1}];
a = StringTake[a, {Flatten[StringPosition[a, c]][[1]], -1}];
c = StringTake[a, {1, Flatten[StringPosition[a, "RowBox[List[\"End\\","]"][[1]] - 1}];
For[k = StringLength[c], k >= 1, k--, c = StringTake[c, {1, k}];
If[! SameQ[Quiet[ToExpression[c]], $Failed], Break[]];
c = Quiet[ToString[InputForm[ToExpression[c]]]];
c = StringReplace[c, {"\\(\" -> "", "\\)" -> ""}];
If[SuffPref[c, "RowBox[{", 1] && SuffPref[c, ", Null]"] 2],
StringTake[c, {9, -9}]; If[SuffPref[c, "RowBox[{", 1], $Failed,
Quiet[Check[ToExpression[c], Return[$Failed]]]; c]]]
```



```

In[996]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson.nb", "Vgs"]
Out[996]= "Vgs[x_Integer, y_Integer] := x * y"
In[997]:= Definition[Vgs]
Out[997]= Vgs[x_Integer, y_Integer] := x y
In[998]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson.nb", "Ga"]
Out[998]= "Ga[x_Integer, y_Integer] := x * y + Gs[x, y]"
In[999]:= Definition[Ga]
Out[999]= Ga[x_Integer, y_Integer] := x y + Gs[x, y]
In[1000]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson.nb", "Gs"]
Out[1000]= "Gs[x_Integer, y_Integer] := x^2 + y^2"
In[1001]:= Definition[Gs]
Out[1001]= Gs[x_Integer, y_Integer] := x^2 + y^2
In[1002]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson.nb", "GSV"]
Out[1002]= "GSV[x_Integer, y_Integer] := Module[{a = 78, b = 460, c = 2013}, x * y + Gs[x, y] *
(a + b + c) + a * Sin[x]/(b + c) * Cos[y]]
In[1003]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson1.nb", "Ga1"]
Out[1003]= "Ga1[x_Integer, y_Integer] := x * y + Gs1[x, y]"
In[1004]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson1.nb", "Gs1"]
Out[1004]= "Gs1[x_Integer, y_Integer] := x^2 + y^2"
In[1005]:= Gs1[78, 460]
Out[1005]= 217684
In[1006]:= ExtrFromNBfile["C:\\AVZ_Package\\Kherson.nb", "Avz"]
Out[1006]= $Failed
In[1007]:= ExtrFromNBfile["C:\\AVZ_Package\\AVZ_Package_1.nb", "StrStr"]
Out[1007]= "StrStr[x_] := If[StringQ[x], "\\\"\\\"\\\"\" <> x <> "\\\"\\\"\\\"\"", ToString[x]]
In[1008]:= ExtrFromNBfile["C:\\AVZ_Package\\AVZ_Package_1.nb", "Df"]
Out[1008]= "Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y], Simplify[Subs[D[Subs[x, y, a],
a], a, y]]]"
In[1009]:= ExtrFromNBfile["C:\\AVZ_Package\\AVZ_Package_1.nb", "ListStringQ"]
Out[1009]= "ListStringQ[x_] := ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}"

```

Успешный вызов **ExtrFromNBfile**[*x*, *y*] процедуры возвращает определение объекта в строчном формате из файла *x* формата {"cdf", "nb"} с именем *y*, заданном в строчном формате, одновременно активируя это определение в текущем сеансе. В противном случае вызов процедуры возвращает **\$Failed**. Полезным свойством данной процедуры является то обстоятельство, что загрузки в текущий сеанс файла *x* не требуется. В то же время данная процедура не носит *универсального* характера, не позволяя получать все определения, находящиеся в файле *x* формата {"cdf", "nb"} с пакетом. Между тем, на основе использованного процедурой подхода ее можно обобщить, что оставляем в качестве весьма полезного упражнения заинтересованному пользователю системы.

Достаточно интересной представляется задача редактирования пакета, находящегося в *m*-файле, которую решает следующая процедура **RedMfile**, исходный код которой

с типичными примерами использования представляет нижеследующий фрагмент.

```
In[2564]:= PosListTest[L_List, p_/: PureFuncQ[p]] := Module[{a = {}, k = 1},
    While[k <= Length[L], If[Select[{L[[k]]}, p] != {}, AppendTo[a, k]]; k++]; a]

In[2565]:= PosListTest[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, EvenQ[#] &]
Out[2565]= {2, 4, 6, 8, 10}

In[2566]:= RedMfile[x_/: FileExistsQ[x] && FileExtension[x] == "m", p_/: SymbolQ[p],
    r_/: MemberQ[{"add", "delete", "replace"}, r]] := Module[{a = ReadList[x, String],
    b = "("Begin["`" <> ToString[p] <> "`\""])", c = "("End[]*)",
    d = ToString[p], h, save},
    save[q_] := Module[{f, k = 1}, f = DirectoryName[x] <> FileBaseName[x] <> ".$m";
    While[k <= Length[q], WriteString[f, q[[k]], "\n"]; k++]; Close[f];
    If[! MemberQ[a, "("Package:: *)"), $Failed,
    If[r === "delete", h = Select[a, SuffPref[#, "(" <> d <> "::usage", 1] &];
    If[h == {}, x, a = Select[a, ! SuffPref[#, "(" <> d <> "::usage", 1] &];
    d = SubListsMin[a, b, c, "r"]; d = MinusList[a, d]; save[d]],
    If[r === "add" && Select[a, SuffPref[#, "(" <> d <> "::usage = ", 1] &] == {}
    && Head[p::usage] == String && (ProcQ[p] || FunctionQ[p]),
    h = PosListTest[a, SuffPref[#, "("BeginPackage["`", "("EndPackage[]"], 1] &];
    a = Insert[a, "(" <> d <> "::usage = " <> ToString1[p::usage] <> ")", h[[1]] + 1];
    a = Flatten[Insert[a, "("Begin["`" <> d <> "`\""])",
    "(" <> PureDefinition[p] <> ")", "("End[]*)"), h[[2]] + 1]; save[a],
    If[r === "replace" && Head[p::usage] == String && (ProcQ[p] || FunctionQ[p]),
    h = PosListTest[a, SuffPref[#, "("Begin["`" <> d <> "`\""])", 1] &];
    a[[h[[1]] ;; h[[1]] + 2]] = "("Begin["`" <> d <> "`\""])",
    "(" <> PureDefinition[p] <> ")", "("End[]*)");
    h = PosListTest[a, SuffPref[#, "(" <> d <> "::usage = ", 1] &];
    a[[h[[1]]]] = "(" <> d <> "::usage = " <> ToString1[p::usage] <> ")",
    save[Flatten[a]]], x]]]]

(* ::Package:: *)
(* ::Input:: *)
(*BeginPackage["Kherson`"]*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2+y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_Integer, y_Integer] := x*y*)
(*End[]*)
```

Содержимое исходного т-файла

```
(*EndPackage[]*)
In[2567]:= Avz[x_] := Module[{}, x^2 + 78]; Vgs[x_, y_] := x^2 + y^2
In[2568]:= Avz::usage = "Help on Avz."; Vgs::usage = "Help on Vgs_1.";
In[2569]:= RedMfile["C:/AVZ_Package/Kherson.m", Vgs, "delete"]
Out[2569]= "C:\\AVZ_Package\\Kherson$.m"
(* ::Package:: *)                               Содержимое т-файла после операции "delete"
(* ::Input:: *)
(*BeginPackage["Kherson`"]*)
(*Gs::usage = "Help on Gs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2+y^2*)
(*EndPackage[]*)
In[2570]:= RedMfile["C:\\AVZ_Package\\Kherson.m", Avz, "add"]
Out[2570]= "C:\\AVZ_Package\\Kherson$.m"
(* ::Package:: *)                               Содержимое т-файла после операции "add"
(* ::Input:: *)
(*BeginPackage["Kherson`"]*)
(*Avz::usage = "Help on Avz."*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2+y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_Integer, y_Integer] := x*y*)
(*End[]*)
(*Begin["`Avz`"]*)
(*Avz[x_] := Module[{}, x^2 + 78]*)
(*End[]*)
(*EndPackage[]*)
In[2571]:= RedMfile["C:\\AVZ_Package\\Kherson.m", Vgs, "replace"]
Out[2571]= "C:\\AVZ_Package\\Kherson$$$$.m"
(* ::Package:: *)                               Содержимое т-файла после операции "replace"
(* ::Input:: *)
(*BeginPackage["Kherson`"]*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs_1."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2+y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_, y_] := x^2 + y^2*)
(*End[]*)
(*EndPackage[]*)
```

```
In[2572]:= RedMfile["C:\\AVZ_Package\\Kherson.m", Gs, "add"]
Out[2572]= "C:\\AVZ_Package\\Kherson.m"
In[2573]:= RedMfile["C:\\AVZ_Package\\Rans78.m", Gs, "add"]
Out[2573]= $Failed
```

Прежде всего, предыдущий фрагмент предваряет достаточно простая процедура, чей вызов **PosListTest**[*L*, *p*] возвращает список позиций списка *L*, которые удовлетворяют тесту, определяемому чистой функцией *p*. В дальнейшем предполагается, что файл *x* *m*-формата структурно соответствует стандартному файлу с пакетом; пример такого файла *m*-формата приведен в 1-й затененной области предыдущего фрагмента. Вызов **RedMfile**[*x*, *n*, *y*] возвращает полный путь к *m*-файлу, чье базовое имя (*FileName*) имеет вид **FileName**[*x*] <> "\$", являющемуся результатом применения к исходному *m*-файлу операции *y* относительно объекта, определенного именем *n*, а именно:

**"delete"** – из файла *x* удаляется справка (*usage*) и определение объекта с именем *n*, сам исходный файл не изменяется; если такой объект в файле отсутствует, возвращается полный путь к исходному файлу данных *x*;

**"add"** – в файл *x* добавляются справка (*usage*) и определение объекта с именем *n*, тогда как сам исходный файл не изменяется; если такой объект в файле уже существует, то возвращается полный путь к исходному файлу данных *x*;

**"replace"** – в файле *x* заменяются справка (*usage*) и определение объекта с именем *n*, но сам исходный файл не меняется; если такой объект в файле отсутствует, возвращается полный путь к исходному файлу данных *x*.

Если исходный файл *x* имеет структуру, отличную от указанной, то вызов процедуры возвращает **\$Failed**; при этом, для успешного выполнения операций **"add"** и **"replace"** требуется предварительное вычисление в текущем сеансе *n::usage* и определения для *n*, как иллюстрируют примеры предыдущего фрагмента. В целом, данная процедура допускает целый ряд расширений, которые оставляем заинтересованному читателю.

Совершенно иная ситуация при необходимости обновить объект из пакета, который находится в файле данных *mx*-формата. Ради наглядности алгоритма модификацию пакета представим в виде достаточно прозрачной *схемы*, описанной нижеследующим фрагментом на весьма простом примере замены функции в файле *mx*-формата.

```
In[2201]:= Get["C:\\AVZ_Package\\Kherson.mx"]
In[2202]:= ?GSV
Help on GSV
In[2203]:= Definition[GSV]
Out[2203]= GSV[Book`GSV`x_Integer, Book`GSV`y_] := Module[{Book`GSV`a = 78,
Book`GSV`b = 460}, Book`GSV`x*Book`GSV`y + Book`GSV`a*Book`GSV`b]
In[2204]:= GSV[78, 460]
Out[2204]= 71760
In[2205]:= Clear[GSV]; GSV::usage = "A new help on GSV.";
In[2206]:= GSV[Book`GSV`h_, Book`GSV`t_;/ IntegerQ[Book`GSV`h_]] :=
Module[{Book`GSV`a = 78}, Book`GSV`h* Book`GSV`t + Book`GSV`a]
```

```

In[2207]:= DumpSave["C:\\AVZ_Package\\Kherson_1.mx", "Book`"]
Out[2207]= {"Book`"}
===== Новый сеанс системы Mathematica =====
In[1001]:= Get["C:\\AVZ_Package\\Kherson_1.mx"]
In[1002]:= ?GSV
      A new help on GSV
In[1003]:= Definition[GSV]
Out[1003]= GSV[Book`GSV`h_, Book`GSV`t_ /; IntegerQ[Book`GSV`t]] :=
      Module[{Book`GSV`a = 78}, Book`GSV`h*Book`GSV`t + Book`GSV`a]
In[1004]:= GSV[78, 460]
Out[1004]= 35958
    
```

На *первом* шаге вызов функции **Get[x]** загружает в текущий сеанс *x* файл *mx*-формата с пакетом, что обеспечивает доступность всех средств, содержащихся в нем. Тогда как на *втором* шаге проверяется справка и определение функции **GSV**, подвергающейся обновлению, и результат конкретного вызова данной функции. На *следующем* шаге из текущего сеанса вызовом **Clear[GSV]** удаляется функция **GSV** и для нее определяется новая справка (*usage*). Затем, вычисляется новое определение для функции **GSV**, все параметры которого, включая *формальные* аргументы и *локальные* переменные, будут сцеплены с контекстом, принимая следующий формат, а именно:

*Context\_from\_File`Object\_Name`Variable\_of\_New\_Definition*

В нашем конкретном случае данный контекст принимает следующий вид **Book`GSV`**. Пример фрагмента довольно наглядно иллюстрирует сказанное. После чего вызовом функции **DumpSave[y, "Book`"]** определения *всех* объектов текущего сеанса, которые снабжены контекстом **"Book`"**, вместе с их справками (*usages*) сохраняются в новом *y* файле *mx*-формата. Наконец, *заключительный* этап в новом текущем сеансе тестирует корректность полученного *y* файла *mx*-формата с пакетом – результата *модификации* исходного *x* файла *mx*-формата с пакетом. Отметим лишь, с достаточно очевидными изменениями описанный алгоритм вполне успешно работает и в случае дополнения (*удаления*) объектов в (*из*) *x* файл(*а*) *mx*-формата с пакетом. Представленный алгоритм достаточно прост, однако имеет недостаток при необходимости модификации файла *mx*-формата с пакетом довольно большими исходными кодами объектов, в подобном случае может быть использован нижеследующий прием, а именно. В текущем сеансе оформляется простой пакет, содержащий справку по объекту, предназначенному для модификации файла *mx*-формата с пакетом, и его определение. Затем данный пакет вычисляется с последующим вычислением конструкции, представленной следующим фрагментом, из которого достаточно несложно усматривается сам механизм замены у функции **Name** контекста **"Context`"** на **"Kiev`"**. После чего полученное определение функции **Name** можно использовать в вышеописанной схеме обновления *mx*-файлов.

```

In[3209]:= BeginPackage["Context`"]
      Name::usage = "Help on Name."
      Begin["`Name`"]
    
```

```

Name[x_Integer, y_, z_Integer] := Module[{}, x*y*z]
End[]
EndPackage[]

Out[3209]= "Context`"
Out[3210]= ""
Out[3211]= "Context`Name`"
In[3215]:= $ContextPath = MinusList[PrependTo[$ContextPath, "Kiev`"], {"Context`"}];
StringReplace[ToString1[Definition[Name]], "Context`" -> "Kiev`"]
Out[3215]= "Name[Kiev`Name`x_Integer, Kiev`Name`y_, Kiev`Name`z_Integer] :=
Module[{}, Kiev`Name`x*Kiev`Name`y*Kiev`Name`z]"
In[3216]:= ToExpression[%]
In[3217]:= Definition[Name]
Out[3217]= Name[Kiev`Name`x_Integer, Kiev`Name`y_, Kiev`Name`z_Integer] :=
Module[{}, Kiev`Name`x*Kiev`Name`y*Kiev`Name`z]
In[3218]:= $ContextPath
Out[3218]= {"Kiev`", "AladjevProcedures`", "PacletManager`", "QuantityUnits`", "System`",
"WebServices`", "Global`"}

```

Представленный во фрагменте пример пакета (выделен затенением) для определения функции **Name** может содержать произвольное число объектов (определения + справки), которыми можно обновлять файл *mx*-формата с пакетом путем замены либо удаления его объектов, как это позволяет выполнять вышеописанный механизм редактирования *mx*-файлов с пакетами. Между тем, следует иметь в виду, представленный алгоритм модификации *mx*-файлов с пакетами относится к случаю, когда файлы *mx*-формата относятся к той же операционной платформе, что и планируемая их модификация.

Следующая процедура **RedMxFile** обеспечивает автоматизацию модификации файлов *mx*-формата, которая рассмотрена выше. Вызов **RedMxFile**[*x*, *y*, *r*, *f*] возвращает полный путь к *mx*-файлу, чье базовое имя (*FileBaseName*) имеет формат **FileBaseName**[*x*] <> "\$", являющемуся результатом применения к исходному *mx*-файлу *x* с пакетом операции *r* относительно объекта, определенного именем *y*, а именно:

**"delete"** - из файла *x* удаляется объект, определенный именем *y*, сам исходный файл не изменяется; вызов возвращает полный путь к модифицированному файлу; более того, если объект в файле отсутствует, возвращается путь к исходному файлу *x*;

**"add"** - в файл *x* добавляется объект, определенный именем *y*, тогда как сам исходный файл не изменяется; вызов возвращает полный путь к модифицированному файлу; в случае, если такой объект в файле существует, возвращается полный путь к исходному файлу *x*; четвертый аргумент *f* определяет *mx*-файл, содержащий пакет со справкой и определением дополняемого объекта *y*;

**"replace"** - в файле *x* заменяется объект, определенный именем *y*, сам исходный файл не изменяется; вызов возвращает полный путь к модифицированному файлу; более того, если такой объект в файле отсутствует, возвращается полный путь к исходному файлу *x*; четвертый аргумент *f* определяет *mx*-файл, содержащий пакет со справкой и

определением замещающего объекта *y*. Таким образом, возврат пути к обновленному файлу служит *индикатором* успешности вызова процедуры **RedMxFile**. При этом, для успешного выполнения операции "*add*" либо "*replace*" в текущем сеансе необходимо перед ней предварительное вычисление *определения y::usage* для объекта *y*, как весьма наглядно иллюстрируют примеры нижеследующего фрагмента.

```
In[62]:= RedMxFile[x_;/ FileExistsQ[x] && FileExtension[x] == "mx", y_;/ StringQ[y] &&
      SymbolQ[y], r_;/ MemberQ[{"add", "delete", "replace"}, r], f___] :=
      Module[{a, c, c1 = ContextFromFile[x], c2, save, t}, Get[x]; a = CNames[c1];
      save[z_] := Module[{p = DirectoryName[z] <> FileName[z] <> ".mx"},
      ToExpression["DumpSave[" <> ToString1[p] <> ", " <> ToString1[c1] <> ""]; p];
      If[r == "delete" && MemberQ[a, y], Unprotect[y]; ClearAll[y];
      c = save[x]; RemovePackage[c1]; c,
      If[r == "replace" && MemberQ[a, y] && {f} != {} &&
      FileExistsQ[f] && FileExtension[f] == "mx", c2 = ContextFromFile[f]; Get[f];
      c = ToString1[Definition[y]]; Map[Clear, Mapp[StringJoin, $ContextPath, y]];
      Quiet[ToExpression[c1 <> StringReplace[c, c2 -> c1]]];
      ToExpression[c1 <> y <> "::usage = " <> ToString1[Help]];
      c = save[x]; Map[RemovePackage, {c1, c2}]; c,
      If[r == "add" && {f} != {} && FileExistsQ[f] &&
      FileExtension[f] == "mx" && ! MemberQ[a, y],
      c2 = ContextFromFile[f]; Get[f]; c = ToString1[Definition[y]];
      Quiet[Map[Remove, Mapp[StringJoin, $ContextPath, y]]];
      Quiet[ToExpression[c1 <> StringReplace[c, c2 -> c1]]];
      ToExpression[c1 <> y <> "::usage = " <> ToString1[Help]];
      c = save[x]; Map[RemovePackage, {c1, c2}]; c, x]]]

In[63]:= RedMxFile["C:\\AVZ_Package\\Kherson.mx", "GSV", "delete"]
Out[63]= "C:\\AVZ_Package\\Kherson$.mx"
In[64]:= Get["C:\\AVZ_Package\\Kherson$.mx"]
In[65]:= CNames[ContextFromFile["C:\\AVZ_Package\\Kherson$.mx"]]
Out[65]= {"Ga", "Gs", "Vgs"}
In[66]:= Help = "A new help on GSV."
Out[66]= "A new help on GSV."
In[67]:= RedMxFile["C:\\AVZ_Package\\Kherson.mx", "GSV", "replace", "GSV.mx"]
Out[67]= "C:\\AVZ_Package\\Kherson$.mx"
In[68]:= Get["C:\\AVZ_Package\\Kherson$.mx"]
In[69]:= ?GSV
      A new help on GSV
In[70]:= Definition["GSV"]
Out[70]= GSV[x_Integer, y_, z_Integer] := Module[{a = 78}, (x*y)*a]
```

```
In[71]:= Help = "Help on GSV1."
Out[71]= "Help on GSV1."
In[72]:= RedMxFile["C:\\AVZ_Package\\Kherson.mx", "GSV1", "add", "GSV1.mx"]
Out[72]= "C:\\AVZ_Package\\Kherson$.mx"
In[73]:= Get["C:\\AVZ_Package\\Kherson$.mx"]
In[74]:= CNames[ContextFromFile["C:\\AVZ_Package\\Kherson$.mx"]]
Out[74]= {"Ga", "Gs", "GSV", "GSV1", "Vgs"}
In[75]:= ?GSV1
      Help on GSV1
In[76]:= DefFunc[GSV1]
Out[76]= GSV1[x_Integer, y_] := Module[{a = 47}, x*y*a]
```

Для выполнения операции "*add*" или "*replace*" предварительно следует создать файл *mx*-формата с пакетом, который содержит определение объекта, используемого для обновления (*обновление, дополнение*) основного *mx*-файла с пакетом. Нижеследующий фрагмент представляет типичное оформление подобных файлов для обновления. В то же время необходимо иметь в виду, что обновляемый и обновляющий *mx*-файлы данных должны быть созданы в одной и той же операционной системе.

```
In[2189]:= BeginPackage["avzransian78`"]
      GSV::usage = "A new help on GSV."
      Begin["`GSV`"]
      GSV[x_Integer, y_, z_Integer] := Module[{a = 78}, (x*y)*a]
      End[]
      EndPackage[]
Out[2189]= "avzransian78`"
Out[2190]= "A new help on GSV."
Out[2191]= "avzransian78`GSV`"
Out[2192]= "avzransian78`GSV`"
In[2195]:= DumpSave["GSV.mx", "avzransian78`"]
Out[2195]= {"avzransian78`"}
```

Более того, в результате вызова процедуры оба пакета удаляются из текущего сеанса. В целом, процедура **RedMxFile** допускает целый ряд расширений, которые оставляем заинтересованному читателю. Между тем, следует отметить, данная процедура в ряде отношений базируется на искусственных для типовой *процедурной парадигмы* приемах, обеспечивающих корректные вызовы процедуры в среде системы, являясь зависимой от версии. В частности, это еще раз говорит, что процедурный язык системы уступает *Maple*. В целом же, его возможности достаточны для широкого круга пользователей.

Довольно полезная процедура обеспечивает конвертацию пакета, расположенного в файле *mx*-формата, в файл *m*-формата. Вызов **MxFileToMfile**[*x*, *y*] возвращает путь к файлу *y* – результату конвертации *mx*-файла *x* с пакетом в файл *y* *m*-формата. Более того, вызов процедуры удаляет пакеты *x*, *y* из текущего сеанса. Следующий фрагмент



представляет исходный код процедуры с примером применения, а также содержимое исходного и отконвертированного файлов данных *x*, *y* с пакетом пользователя.

```
(* ::Package:: *)
(* ::Input:: *)
(*BeginPackage["Kherson`"]*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2 + y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_Integer, y_Integer] := x*y*)
(*End[]*)
(*EndPackage[]*)

In[2369]:= MxFileToMfile[x_/. FileExistsQ[x] && FileExtension[x] == "mx",
           y_/. StringQ[y] && FileExtension[y] == "m"] :=
Module[{a = ContextFromFile[x], b, c, k = 1}, Get[x]; b = CNames[a];
WriteString[y, "(" ::Package:: *)", "\n", "(" ::Input:: *)", "\n",
           "("BeginPackage["`" <> a <> "`"]*)", "\n"];
While[k <= Length[b], c = b[[k]] <> "::usage";
WriteString[y, "(" <> c <> " = " <> ToString1[ToExpression[a <> c]], ")"", "\n"]; k++];
k = 1; While[k <= Length[b], c = b[[k]];
WriteString[y, "("Begin["`" <> c <> "`"]*)", "\n", "(" <> PureDefinition[a <> c] <>
           ")", "\n", "("End[]*)", "\n"]; k++]; WriteString[y, "("EndPackage[]*)", "\n"];
Map[{Clear1[2, a <> # <> "::usage"], Clear1[2, a <> #]} &, b];
$ContextPath = MinusList[$ContextPath, {a}]; Close[y]]

In[2370]:= MxFileToMfile["C:/AVZ_Package/Kherson.mx", "C:\\AVZ_Package/Kiev.m"]
Out[2370]:= "C:\\AVZ_Package/Kiev.m"

(* ::Package:: *)
(* ::Input:: *)
(*BeginPackage["Kherson`"]*)
(*Ga::usage = "Help on Ga."*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs."*)
(*Begin["`Ga`"]*)
(*Ga[Kherson`Ga`x_Integer, Kherson`Ga`y_Integer] := Kherson`Ga`x*Kherson`Ga`y +
           Gs[Kherson`Ga`x, Kherson`Ga`y]*)
(*End[]*)
(*Begin["`Gs`"]*)
(*Gs[Kherson`Gs`x_Integer, Kherson`Gs`y_Integer] := Kherson`Gs`x^2 +
           Kherson`Gs`y^2*)
```

```
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[Kherson`Vgs`x_Integer, Kherson`Vgs`y_Integer] :=
                                   Kherson`Vgs`x*Kherson`Vgs`y*)
(*End[]*)
(*EndPackage[]*)
```

Тогда как процедура **MfileToMx** обеспечивает конвертацию пакета, расположенного в файле *m*-формата, в файл *mx*-формата. Вызов процедуры **MfileToMx[x]** возвращает путь к файлу – результату конвертации *m*-файла *x* с пакетом в файл *mx*-формата, чье имя совпадает с именем исходного файла *x* с заменой расширения "*m*" на "*mx*". Более того, вызов процедуры удаляет пакет *x* из текущего сеанса, если до вызова процедуры **MfileToMx** он не был в него загружен, и нет в противном случае. Фрагмент приводит исходный код процедуры **MfileToMx** с типичными примерами ее использования.

```
BeginPackage["avzransian`"]
GSV::usage = "Help on GSV."
Begin["`GSV`"]
GSV[x_Integer, y_, z_Integer] := Module[{a = 460}, a*x*y*z]
End[]
Avz::usage = "Help on Avz."
Begin["`Avz`"]
Avz[x_Integer, y_, z_] := Module[{}, x*y*z + 460]
End[]
Art::usage = "Help on Art."
Begin["`Art`"]
Art[x_Integer, y_, z_] := x + y + z + 78
End[]
EndPackage[]
```

```
In[1]:= MfileToMx[x_;/; FileExistsQ[x] && FileExtension[x] == "m"] :=
Module[{a = ContextFromFile[x], b, c = ToString1[x <> "x"], d},
If[MemberQ[$ContextPath, a],
ToExpression["DumpSave[" <> c <> "," <> ToString1[a] <> ""]; x <> "x",
b = ReadList[x, String];
d = Select[Map[StringReplace[#, {"(" -> "(", ")" -> ")"}] &, b[[3 ;; -1]]], # != "(" &];
Quiet[ToExpression[d]];
ToExpression["DumpSave[" <> c <> "," <> ToString1[a] <> ""];
Map[Clear1[2, a <> #] &, CNames[a]];
$ContextPath = MinusList[$ContextPath, {a}]; x <> "x"]];

In[2]:= MfileToMx["C:\\AVZ_Package\\Rans_Ian.m"]
Out[2]= "C:\\AVZ_Package\\Rans_Ian.mx"
In[3]:= $ContextPath
Out[3]= {"AladjevProcedures`", "PacletManager`", "QuantityUnits`", "WebServices`",
```

```

"System`", "Global`")
In[4]:= Definition[Art]
Out[4]= Null
In[5]:= Get["C:\\AVZ_Package\\Rans_Ian.mx"]
In[6]:= $ContextPath
Out[5]= {"avzransian`", "AladjevProcedures`", "PacletManager`", "QuantityUnits`",
"WebServices`", "System`", "Global`"}
In[7]:= CNames[ContextFromFile["C:\\AVZ_Package\\Rans_Ian.mx"]]
Out[7]= {"Art", "Avz", "GSV"}
In[8]:= Definition[Art]
Out[8]= Art[avzransian`Art`x_Integer, avzransian`Art`y_, avzransian`Art`z_] :=
avzransian`Art`x + avzransian`Art`y + avzransian`Art`z + 78
In[9]:= Definition[GSV]
Out[9]= GSV[avzransian`GSV`x_Integer, avzransian`GSV`y_, avzransian`GSV`z_Integer] :=
Module[{avzransian`GSV`a = 460}, avzransian`GSV`a*avzransian`GSV`x*
avzransian`GSV`y*avzransian`GSV`z]
In[10]:= ?Art
Help on Art
In[11]:= ?GSV
Help on GSV
In[12]:= Definition[Avz]
Out[12]= Avz[avzransian`Avz`x_Integer, avzransian`Avz`y_, avzransian`Avz`z_] :=
Module[{}, avzransian`Avz`x*avzransian`Avz`y*avzransian`Avz`z + 460]
In[13]:= ?Avz
Help on Avz

```

Начало фрагмента представляет исходный код простого пакета, который содержится в файле **"Rans\_Ian.m"**, взятого в качестве иллюстрационного примера использования процедуры **MfileToMx**, обеспечивающей конвертацию пакета, находящегося в файле *m*-формата **"Rans\_Ian.m"**, в файл *mx*-формата **"Rans\_Ian.mx"** с проверкой результата. В целом ряде приложений данная процедура представляет определенный интерес.

Для тестирования содержимого файла *mx*-формата с пакетом в разрезе имен средств, определения которых находятся в данном файле, достаточно полезной оказывается процедура **NamesFromMx**. Вызов **NamesFromMx[x]** возвращает список имен средств, определения которых находятся в *x* файле *mx*-формата с пакетом. В случае, если этот пакет не был загружен в текущий сеанс, то вызов процедуры **NamesFromMx** оставляет его незагруженным. Следующий фрагмент представляет как *исходный* код процедуры **NamesFromMx**, так и наиболее типичные примеры ее использования.

```

In[2210]:= NamesFromMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
Module[{a = ContextFromFile[x], b},
If[MemberQ[$ContextPath, a], CNames[a], Get[x]; b = CNames[a];
Map[Close1[2, a <> #] &, b]; $ContextPath = MinusList[$ContextPath, {a}]; b]]

```

```

In[2211]:= NamesFromMx["C:\\AVZ_Package\\AVZ_Package_1.mx"]
Out[2211]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts",
  "Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes",
  "Arity", "ArityBFM", "ArrayInd", "AssignL", "AssignToList", "AtomicQ",
  =====
  "WhatType", "WhatValue", "WhichN", "$CallProc", "$Line1", "$Load$Files$",
  "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc"}

In[2216]:= Length[%]
Out[2216]= 582

In[2212]:= NamesFromMx["C:\\AVZ_Package\\Kherson.mx"]
Out[2212]= {"Ga", "Gs", "GSV", "Vgs"}

In[2213]:= $ContextPath
Out[2213]= {"AladjevProcedures", "PacletManager", "QuantityUnits", "WebServices",
  "System", "Global"}

In[2214]:= Definition[GSV]
Out[2214]= Null

```

В то время как процедура **NamesFromMx1** в отличие от процедуры **NamesFromMx** не требует для получения списка *имен*, чьи определения находятся в файле *mx*-формата с пакетом, реальной загрузки в текущий сеанс этого файла. Вызов **NamesFromMx1[x]** возвращает список имен средств, определения которых находятся в файле *x* формата *"mx"* с пакетом. Фрагмент представляет исходный код процедуры с примерами.

```

In[3570]:= NamesFromMx1[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=
  Module[{c, d = {}, p, h = "", k = 1, j, m, n,
    a = ContextFromFile[x], b = ToString[ReadFullFile[x]]},
    b = StringJoin[Map[FromCharacterCode,
      Select[ToCharacterCode[b], # > 32 && # < 128 &]]];
    {n, m} = Map[StringLength, {a, b}];
    c = Map[#[[1]] + n &, StringPosition[b, a]] [[2 ;; -1]];
    While[k <= Length[c], For[j = c[[k]], j <= m, j++, p = StringTake[b, {j, j}];
      If[p == "", AppendTo[d, h]; h = ""; Break[], h = h <> p]; k++];
    Sort[MinusList[Select[d, SymbolQ[#] &], {"Private"}]]]

In[3585]:= BeginPackage["avzransian`"]
  GSV::usage = "Help on GSV."
  Begin["`GSV`"]
  GSV[x_Integer, y_, z_Integer] := Module[{a = 460}, a*x*y*z]
  End[]
  Avz::usage = "Help on Avz."
  Begin["`Avz`"]
  Avz[x_Integer, y_, z_] := Module[{}, x*y*z + 460]
  End[]

```

```
Art::usage = "Help on Art."
Begin["`Art`"]
Art[x_Integer, y_, z_] := x + y + z + 78
End[]
EndPackage[]
```

```
Out[3585]= "avzransian`"
```

```
=====
```

```
Out[3592]= "Help on Art."
```

```
Out[3593]= "avzransian`Art`"
```

```
In[3598]:= DumpSave["C:\\AVZ_Package\\Kiev.mx", "avzransian`"]
```

```
Out[3598]= {"avzransian`"}
```

```
In[3599]:= NamesFromMx1["C:\\AVZ_Package\\Kiev.mx"]
```

```
Out[3599]= {"Art", "Avz", "GSV"}
```

```
In[3600]:= Length[NamesFromMx1["C:\\AVZ_Package\\AVZ_Package_1.mx"]]
```

```
Out[3600]= 464
```

При этом, возвращаются только те имена средств, определения которых, получаемые по функции **Definition**, содержат контекст, ассоциированный с пакетом. Тогда как с другой стороны модификация процедуры **NamesFromMx1** позволяет получать более полный список имен средств, чьи определения находятся в файле *x* формата *"mx"* с пакетом. Следующий фрагмент представляет одну из таких модификаций, в качестве которой выступает процедура, чей вызов **NamesFromMx2[x]** возвращает список имен средств, определения которых находятся в файле *x* формата *"mx"* безотносительно их связи с контекстом. Между тем, процедура требует довольно значительных временных издержек на файлах формата *"mx"* с пакетом достаточно большого объема.

```
In[3372]:= NamesFromMx2[x_;/; FileExistsQ[x] && FileExtension[x] == "mx"] :=
Module[{a = ToString[ReadFullFile[x]], b},
  b = Select[ToCharacterCode[a], # == 255 || (# > 31 && # < 123 &&
    ! MemberQ[Flatten[{Range[37, 47], Range[91, 95]}], #]) &];
  b = ReduceList[b, 255];
  b = Select[Quiet[SplitList[b, 96]], # != {} &];
  b = Quiet[Map[FromCharacterCode, b]];
  b = DeleteDuplicates[Select[b, SymbolQ[#] &]];
  Select[b, ! MemberQ[{"Private", "System"}, #] &&
    StringFreeQ[#, {StringTake[ContextFromFile[x], {1, -2}], "ÿ"}] &]]

In[3373]:= NamesFromMx2["C:\\AVZ_Package\\Kiev.mx"]
Out[3373]= {"Art", "GSV", "Avz"}
```

В целях сокращения временных издержек алгоритм процедуры **NamesFromMx2** может быть модифицирован, используя следующие средства, расширяющие программную среду *Mathematica*. Фрагмент представляет исходные коды этих средств с примерами их применения. Вызов функции **Map11[F, {{x, y, z, ...}, {a, b, c, ...}, ...}, h]**, где *F* – символ, {*x, y, z, ...*}, {*a, b, c, ...*}, ... и *h* – произвольные выражения, возвращает результат вида

$\{F[x, h], F[y, h], F[z, h], \dots\}, \{F[a, h], F[b, h], F[c, h], \dots\}, \dots\}$

не требуя каких-либо дополнительных пояснений ввиду своей прозрачности.

```
In[163]:= Map11[x_Symbol, y_List, z_] := Map[If[ListQ[#], Map[x[#1, z] &, #], x[#, z]] &, y]
In[164]:= Map11[G, {x, y, z}, h]
Out[164]= {G[x, h], G[y, h], G[z, h]}
In[165]:= Map11[G, {{x, y, z}, {a, b, c}}, h]
Out[165]= {{G[x, h], G[y, h], G[z, h]}, {G[a, h], G[b, h], G[c, h]}}
In[166]:= SplitList1[x_List, y_List, z_List] := Module[{a = Map12[ToString, {x, y, z}],
    b = ToString[Unique["$"], c], c = Map11[StringJoin, a, b]; c = Map[StringJoin, c];
    c = SubsString1[c[[1]], {c[[2]], c[[3]]}, StringQ[#] &, 0];
    ToExpression[Map11[StringSplit, c, b]]]
In[167]:= SplitList1[{x, y, z, a, b, c, m, n, p, x, y, z, 42, 47, 67, m, n, p}, {x, y, z}, {m, n, p}]
Out[167]= {{a, b, c}, {42, 47, 67}}
```

Вызов процедуры **SplitList1**[*x*, *y*, *z*] возвращает подписки списка *x*, ограниченные его подписками *y* и *z*, исключая сами разграничительные подписки *y* и *z*.

Довольно простая процедура **VizContentsNB** обеспечивает вывод файла *nb*-формата в читабельном виде, что довольно удобно для оперативного просмотра содержимого, в частности, пакета, содержащегося в *nb*-файле без его загрузки в текущий сеанс. При этом, вызов **VizContentsNB**[*w*] в читабельном виде выводит на монитор содержимое *nb*-файла *w*, в противном случае возвращается **\$Failed**. Ниже фрагмент представляет исходный код процедуры **VizContentsNB** наряду с примером ее использования.

```
In[2119]:= VizContentsNB[x_] := Module[{}, If[StringQ[x] && FileExistsQ[x] &&
    FileExtension[x] == "nb", CopyToClipboard[Get[x]]; Paste[], $Failed]]
In[2120]:= VizContentsNB["C:\\AVZ_Package\\Kherson.nb"]
<Содержание файла nb-формата в читабельном виде>
```

Как отмечалось, *Mathematica* располагает большим числом глобальных переменных, описывающих, например, характеристики самого пакета, операционной платформы, полные пути к важнейшим его каталогам наряду с целым рядом других показателей текущего состояния системы. Таким образом, пользователь получает вполне реальную возможность довольно эффективно проводить разработку *своих* собственных средств, включая средства, расширяющие возможности самой системы. Действительно, на базе целого ряда таких глобальных переменных и ряда довольно развитых средств можно разрабатывать оригинальные средства, разработка аналогов которых в среде системы *Maple* требует, зачастую, более существенных усилий и нестандартных подходов. И наш опыт в этом направлении вполне подтверждает сказанное. Некоторые довольно простые примеры были приведены в [99] и, чаще всего, это касалось средств доступа; между тем, можно представить и несколько более наглядные примеры. Так, немалый интерес для продвинутого программирования в среде системы представляет и задача определения имени текущего документа {*mws*-файла, *nb*-файла}. В системе *Maple* нами

для данной цели была создана процедура *nbwname*, разработка которой потребовала нестандартного подхода. Тогда как разработка подобного средства для *Mathematica* оказалась намного более простой, не потребовав какого-либо нестандартного подхода, что наглядно иллюстрирует следующая достаточно простая процедура **NbName**, чей исходный код с примерами использования представлены следующим фрагментом.

```
In[3242]:= NbName[] := Module[{a, b, c, d, k = 1}, {a, d} = {ToString[Notebooks[]], {}};
                                {b, c} = Map3[StringPosition, a, {"<<", ">>"}];
    While[k <= Length[b], AppendTo[d, StringTake[a, {b[[k]][[2]] + 1, c[[k]][[1]] - 1}]; k++];
                                Select[d, SuffPref[#, ".nb", 2] &]]

In[3243]:= NbName[]
Out[3243]= {"78.nb", "460.nb", "AVZ_Package_1.nb", "Kherson.nb"}
In[3244]:= NbName[]
Out[3244]= {"SearchFile.nb", "LoadFile.nb", "ActiveProcess.nb", "Integral.nb", "Ver.nb"}
In[3245]:= AcNb[] := StringSplit[NotebookFileName[], {"\\", "/" }][[-1]]
In[3246]:= AcNb[]
Out[3246]= "AVZ_Package_1.nb"
```

Вызов процедуры **NbName[]** возвращает список *nb*-документов пользователя, которые загружены в текущий сеанс; при этом, порядок их в списке определяется порядком их загрузки в текущий сеанс так, что первый элемент определяет текущий документ. В свою очередь, вызов достаточно простой функции **AcNb[]** возвращает имя текущего документа либо пакета, ранее сохраненного в файле данных *nb*-формата.

По вызову **Needs["Context`"]** выполняется вызов **Get["Context`"]**, где по соглашению в качестве файла, загружаемого таким образом, выступает файл, содержащий пакет с контекстом "Context". Как правило, по вызову **Needs["Имя`"]** в текущий сеанс будет загружаться файл с именем «Имя.m», расположенный в текущем подкаталоге сеанса. Если контекст "Context" отсутствует в списке, определяемом переменной *\$Packages*, для загрузки в текущий сеанс используется вызов **Needs["Пакет", "Пакет.m | mx"]**; при этом, файл с пакетом должен располагаться в одном из каталогов, определенном переменной *\$Path*; при наличии в каталоге файлов с расширениями "m" и "mx", то в текущий сеанс загружается файл с расширением "mx". Для удобства загрузки пакета в текущий сеанс может быть использована процедура **Need**, в определенной степени обобщающая стандартную функцию **Needs**. Исходный код процедуры с примерами применения представлен в следующем достаточно прозрачном фрагменте, а именно:

```
In[221]:= Need[x_] := Module[{a = $Path[[1]], b = If[Length[{x}] > 1 && StringQ[{x}][[2]],
                                {x}[[2]], "Null"], c, p, d = {x}[[1]], f},
    If[! (StringQ[d] && StringTake[d, -1] == "`"), $Failed,
    If[b == "Null", Quiet[Check[Get[d], $Failed]],
    If[b != "Null" && ! MemberQ[{"m", "mx"}, FileExtension[b]], $Failed,
    If[MemberQ[$Packages, d], True,
```

```

CopyFile[b, f = a <> "\\\" <> StringTake[d, {1, -2}] <> "." <> FileExtension[b]];
p = Quiet[Check[Get[d], $Failed]]; DeleteFile[f];
If[! SameQ[p, $Failed], $Failed, True]]]]]]

In[222]:= $Packages
Out[222]= {"ResourceLocator`, "DocumentationSearch`, "GetFEKernelInit`, "Jlink`,
"PacketManager`, "WebServices`, "System`, "Global`}
In[223]:= Need["MyFunctions`, "D:\\AVZ_Package"]
Out[223]= $Failed
In[224]:= Need["AladjevProcedures`, "C:\\AVZ_Package\\AVZ_Package_1.mx"]
Out[224]= True
In[225]:= Need["Kherson`, "C:\\AVZ_Package\\Kiev.m"]
Out[225]= True
In[226]:= $Packages
Out[226]= {"Kherson`, "AladjevProcedures`, "GetFEKernelInit`, "ResourceLocator`,
"PacketManager`, "QuantityUnits`, "WebServices`, "System`, "Global`}
In[227]:= Definition[StrStr]
Out[227]= StrStr[x_] := If[StringQ[x], "\" <> x <> "\" , ToString[x]]

```

Вызов процедуры **Need[x]** загружает в текущий сеанс пакет, который соответствует *x*-контексту при условии, что соответствующий файл формата {«.m» | «.mx»} находится в одном из каталогов, определяемых системной переменной **\$Path**, с возвратом **True**; в противном случае вызов возвращает значение **\$Failed**. В то время как вызов **Need[x, y]** загружает в текущий сеанс пакет, который соответствует *x*-контексту при условии, что соответствующий файл формата {«.m» | «.mx»} находится или в одном из подкаталогов, определенных переменной **\$Path**, или определен аргументом *y*, с возвратом значения **True**; в противном случае возвращается значение **\$Failed**.

Таким образом, создав *nb*-документ с определениями объектов, снабдив их справками с последующими его вычислением и сохранением функцией {*Save* | *DumpSave*} в файле формата {«.m» | «.mx»} соответственно, получаем возможность в последующих сеансах загружать его в текущий сеанс функцией **Needs** или приведенной процедурой **Need** с получением доступа к содержащимся в нем программным объектам и к справкам по ним. Более того, с целью повышения эффективности загрузки пакета рекомендуется использовать файл *mx*-формата, в котором он был ранее сохранен по **DumpSave[x]**, где аргумент *x* определяет контекст, ассоциированный с сохраняемым пакетом.

Предложенные выше методы загрузки пакета в текущий сеанс посредством функций **Get["Пакет"]** (<< *Пакет*), и **Needs["Пакет", "Пакет.{m | mx}"]**, **Needs["Пакет", Dir]**, где *Dir* – каталог с пакетом, являются явными, но в целом ряде случаев целесообразно определять режим автоматической загрузки некоего пакета, что и обеспечивает вызов следующей функции, а именно:

```
DeclarePackage["Пакет", {"Имя1", "Имя2", ..., "ИмяN"}]
```

который определяет, что пакет *Пакет* должен быть загружен в текущий сеанс, если в нем будет обращение к любому имени из списка {"Имя1", "Имя2", ..., "ИмяN"}, как это



иллюстрирует следующий простой фрагмент относительно пакета *"MyFunctions"*:

```
In[2338]:= DeclarePackage["MyFunctions`", {"ProcQ", "Df", "Locals", "Uprocs", "SortNL"}]
Out[2338]= "MyFunctions`"
In[2339]:= $Packages
Out[2339]= {"ResourceLocator`", "DocumentationSearch`", "GetFEKernelInit`", "Jlink`",
            "PacletManager`", "WebServices`", "System`", "Global`"}
In[2340]:= Df[(Sin[x] + Cos[x])/Log[x]^2, Log[x]^2]
Out[2340]= -((Cos[x] + Sin[x])/Log[x]^4)
In[2341]:= Map[ProcQ, {Locals, Avz_71, Df, Mapp, SortNL}]
Out[2341]= {True, False, True, True, False}
In[2342]:= ?Df
Df[x, y] returns the result of differentiation of x on y
```

Таким образом, первый же вызов функций из упомянутого списка (*в нашем случае это функция Df*) загружает в текущий сеанс пакет *"MyFunctions"*, содержащий функцию *Df*; при этом, обеспечивая доступ ко всем остальным функциям/процедурам пакета. Если пользователь располагает довольно большим набором пакетов с определениями процедур/функций и иных объектов, то в ряде случаев представляется целесообразным в одном файле определить последовательность вызовов функций **DeclarePackage**, что позволит за одну загрузку определить декларации пакетов, которые будут загружены в текущий сеанс при обращении к содержащимся в них функциям/процедурам. По функциям **DeclarePackage** создаются символы с указанными в них именами, которым присваивается специальный *Stub*-атрибут; в случае обнаружения символа с атрибутом *Stub* автоматически загружается пакет, соответствующий контексту данного символа.

Отметим, что тесты на *допустимость* фактических аргументов в *заголовке* процедуры/функции допускают прямое использование определений процедур/функций, как это достаточно наглядно иллюстрирует следующий простой фрагмент, а именно:

```
In[999]:= G[x_ /; If[S[a_] := Module[{c = 6}, a + c]; S[x] < 78, True, False]] := Module[{}, x^2]
In[1000]:= Map[G, {71, 2013}]
Out[1000]= {5041, G[2013]}
In[1001]:= S[x_ /; If[EvenQ[x] && (g[y_ /; IntegerQ[y]] := Module[{c = 78}, y^2 + c];
                    g[x] > 460), True, False]] := Module[{a = 6}, a + x + g[x]^2]
In[1002]:= Map[S, {460, 2013}]
Out[1002]= {44807576150, 2019}
In[1003]:= Definition[g]
Out[1003]= g[y_ /; IntegerQ[y]] := Module[{c = 78}, y^2 + c]
In[1004]:= V[x_ /; If[EvenQ[x] && (g[y_ /; IntegerQ[y]] := Module[{c = 78}, y^2 + c];
                    g[x] > 460), True, False]] := Module[{a = 6}, {a + x + g[x]^2, Remove[g]}][[1]]]
In[1005]:= Map[V, {460, 2013}]
Out[1005]= {44807576150, V[2013]}
In[1006]:= Definition[g]
Out[1006]= Null
```

В целом ряде случаев представленный механизм оказывается достаточно полезным, в заголовке процедуры обеспечивая определение процедур/функций, сопутствующих основной процедуре/функции. Процедура/функция, чье определение закодировано в заголовке основной процедуры/функции, доступна в текущем сеансе, обеспечивая к ней доступ *наравне* со стандартными функциями и средствами пользователя, которые активированы в текущем сеансе. Один из подходов к определению такого механизма можно проиллюстрировать на примере довольно прозрачной конструкции формата:

$x\_;/ \text{If}[\text{Def1}; \text{Def2}; \dots ; \text{Defn}; \text{BF}[x, \dots], \text{True}, \text{False}]$

используемой для типирования формального  $x$ -аргумента. В заголовках процедур/функций в качестве выражений, тестирующих фактические аргументы, получаемые в точке их вызова, на допустимость, допускается последовательность определений из любого (*конечного*) числа процедур/функций, завершающаяся *булевой* функцией (*BF*), вызов которой возвращает лишь значения  $\{\text{True}, \text{False}\}$ . Как правило, ее аргументами являются значения  $x$  и вызовов всех или некоторых процедур/функций, определения которых находятся в вышеуказанном списке  $\{\text{Def1}, \text{Def2}, \dots, \text{Defn}\}$ .

Между тем, имеется ряд других достаточно интересных механизмов подобного типа для определения сопутствующих средств. Если же механизм определения подобных сопутствующих средств требуется только на момент вызова основной процедуры, в основной процедуре можно несложно обеспечить удаление данного сопутствующего средства из текущего сеанса путем его удаления в нужном месте главной процедуры, как довольно наглядно иллюстрирует последний пример предыдущего фрагмента.

Между тем, следует отметить нежелательность использования *идентичных* контекстов для пакетов, в ряде случаев приводящую к непредсказуемым результатам. Фрагмент представляет два пакета с идентичным контекстом, но с разными составами средств.

```
In[2229]:= BeginPackage["avzransian`"]
          GSV::usage = "Help on GSV."
          Avz::usage = "Help on Avz1."
          Begin["`GSV`"]
          GSV[x_Integer, y_, z_Integer] := Module[{a = 460}, a*x*y*z]
          End[]
          Begin["`Avz`"]
          Avz[x_Integer, y_, z_] := Module[{}, x*y*z + 460]
          End[]
          EndPackage[]

Out[2229]= "avzransian`"
Out[2230]= "Help on GSV."
Out[2231]= "Help on Avz."
Out[2232]= "avzransian`GSV`"
Out[2233]= "avzransian`Avz`"
In[2239]:= $ContextPath
Out[2239]= {"avzransian`", "avzransian78`", "AladjevProcedures`", "PacletManager`",
            "QuantityUnits`", "WebServices`", "System`", "Global`"}
```

```

In[2240]:= BeginPackage["avzransian`"]
           GSV1::usage = "Help on GSV1."
           Avz::usage = "Help on Avz2."
           Begin["`GSV1`"]
           GSV1[x_Integer, y_, z_] := Module[{a = 6}, a*(x + y + z)]
           End[]
           Begin["`Avz`"]
           Avz[x_Integer, y_, z_] := Module[{a = 71}, x*y*z + a]
           End[]
           EndPackage[]

Out[2240]= "avzransian`"
Out[2241]= "Help on GSV1."
Out[2242]= "Help on Avz2."
Out[2243]= "avzransian`GSV1`"
Out[2244]= "avzransian`Avz`"
In[2250]:= $ContextPath
Out[2250]= {"avzransian`", "avzransian78`", "AladjevProcedures`", "PacletManager`",
           "QuantityUnits`", "WebServices`", "System`", "Global`"}
In[2251]:= CNames["avzransian`"]
Out[2251]= {"Avz", "GSV", "GSV1"}
In[2252]:= {GSV[42, 47, 67], GSV1[42, 47, 67]}
Out[2252]= {60838680, 936}
In[2253]:= DefFunc[Avz]
Out[2253]= Avz[x_Integer, y_, z_] := Module[{a = 71}, x*y*z + a]
In[2254]:= ?Avz
           Help on Avz2

```

Данная ситуация обусловлена механизмом контекстов, который состоит в следующем, а именно, при загрузке в текущий сеанс пакетов с одинаковым контекстом имеет место:

1. Все средства из таких пакетов выступают под **одним и тем же** контекстом;
2. Все одноименные средства (с **разными** заголовками) сохраняются в текущем сеансе;
3. Из одноименных средств (с **одинаковыми** заголовками) сохраняется средство, находящееся в последнем загруженном пакете;
4. Для всех одноименных средств (с **одинаковыми** либо **разными** заголовками) сохраняются в текущем сеансе справки лишь по последним загруженным средствам.

Что достаточно наглядно иллюстрирует предыдущий фрагмент. Между тем, данная особенность наряду с негативными моментами имеет и положительные. В частности, она может быть использована для пополнения файла *mx*-формата с пакетом новыми средствами подобно ранее рассмотренной процедуре **RedMxFile**, как иллюстрирует следующий достаточно прозрачный фрагмент, а именно:

```

In[225]:= AddMxFile[x_/, FileExistsQ[x] && FileExtension[x] == "mx", y_/, FileExistsQ[y]
           && FileExtension[y] == "mx", z_/, FileExtension[z] == "mx"] :=

```

```

Module[{a = ContextFromFile[x], b = ContextFromFile[y], c = 78},
  If[a != b, $Failed, If[MemberQ[$ContextPath, a], c = 460; Quiet[Get[y]],
    Quiet[{Get[x], Get[y]}]];
  ToExpression["DumpSave[" <> ToString1[z] <> "," <> ToString1[a] <> "]];
  If[c == 78, Map[Clear1[2, a <> #] &, CNames[a]];
  Quiet[$ContextPath = MinusList[$ContextPath, {a}]]]; z]

In[226]:= AddMxFile["File1.mx", "File2.mx", "File3.mx"]
Out[226]= "File3.mx"
In[227]:= $ContextPath
Out[227]= {"AladjevProcedures", "PacletManager", "QuantityUnits", "WebServices",
  "System", "Global"}
In[228]:= CNames["avzransian"]
Out[228]= {}
In[229]:= {avzransian`Avz[42, 47, 67], avzransian`GSV[42, 47, 67]}
Out[229]= {avzransian`Avz[42, 47, 67], avzransian`GSV[42, 47, 67]}
In[230]:= Get["File3.mx"]
In[231]:= $ContextPath
Out[231]= {"avzransian", "AladjevProcedures", "PacletManager", "QuantityUnits",
  "WebServices", "System", "Global"}
In[232]:= CNames["avzransian"]
Out[232]= {"Avz", "GSV", "GSV1"}
In[233]:= ?avzransian`Avz
Help on Avz2
In[234]:= Definition[Avz]
Out[234]= Avz[avzransian`Avz`x_Integer, avzransian`Avz`y_, avzransian`Avz`z_] :=
  Module[{}, avzransian`Avz`x*avzransian`Avz`y*avzransian`Avz`z + 460]
In[4226]:= SaveInMx[x_;/; FileExtension[x] == "mx", y_;/; SymbolQ[y] || ListQ[y] &&
  DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}, z_;/; ContextQ[z]] := Module[{b,
  a = Flatten[Select[Map[PureDefinition[#] &, Flatten[{y}]], ! SameQ[#, $Failed] &]]},
  Map[ToExpression[z <> #] &, a]; $ContextPath = AppendTo[$ContextPath, z];
  DumpSave[x, z]; ]
In[4227]:= Agn[x_, y_] := Module[{a = 78}, a*(x + y)]; Agn[x_] := x + 460
In[4228]:= SaveInMx["Tallinn.mx", {Avz, Agn}, "Kherson"]
In[4229]:= $ContextPath = MinusList[$ContextPath, {"Kherson"}]; Get["Tallinn.mx"]
In[4230]:= PureDefinition[Agn]
Out[4230]= {"Agn[x_, y_] := Module[{a = 78}, a*(x + y)]", "Agn[x_] := x + 460"}

```

Фрагмент приводит исходный код процедуры **AddMxFile**, использующей описанный выше механизм контекстов, вызов которой **AddMxFile[x, y, z]** возвращает полный путь к файлу **z** – результату дополнения файла **x** средствами файла **y**. При этом, все файлы

имеют *mx*-формат, а первые два содержат пакеты с одинаковыми контекстами. Более того, если первый файл загружен в текущий сеанс, то загруженным остается и второй файл, в противном случае первые два пакета выгружаются из текущего сеанса. Тогда как вызов процедуры **SaveInMx**[*x*, *y*, *z*] ничего не возвращает, сохраняя в *mx*-файле *x* с контекстом *z* определение символа или списка символов *y*, которые имеют контекст "Global". Исходный код процедуры приводит предыдущий фрагмент с примером ее применения. Процедура дополняет ранее представленные средства этого же плана.

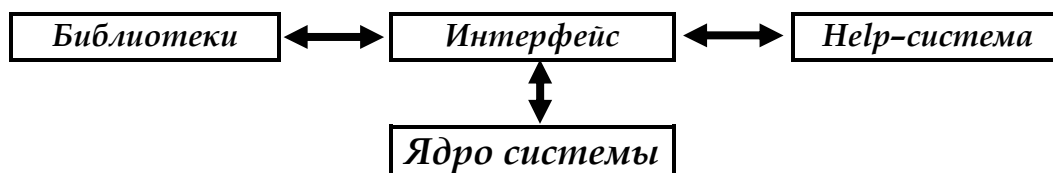
В целом ряде случаев возникает необходимость тестирования файла на предмет того, содержит ли он пакет. Данную задачу решает довольно простая функция, чей вызов **PackageFileQ**[*x*] возвращает **True**, если аргумент *x* определяет файл форматов {"cdf", "mx", "m", "nb"} с пакетом, в противном случае возвращается **False**. Данная процедура оказалась полезным средством для ряда средств нашего пакета [120]. Нижеследующий фрагмент представляет исходный код функции наряду с примерами ее применения.

```
In[230]:= PackageFileQ[x_] := If[StringQ[x] && FileExistsQ[x] && MemberQ[{"cdf", "m",  
                                "mx", "nb"}, FileExtension[x]], If[SameQ[ContextFromFile[x],  
                                $Failed], False, True], False]  
  
In[231]:= PackageFileQ["C:\\AVZ_Package\\AVZ_Package_1.m"]  
Out[231]= True  
In[233]:= PackageFileQ["C:\\AVZ_Package\\AVZ_Package.cdf"]  
Out[233]= False
```

Итак, вообще говоря, даже уже представленной информации наряду с информацией, представленной в наших книгах [98-101] вполне достаточно не только для понимания принципиальной организации пакетов в среде *Mathematica*, но также для написания собственных относительно несложных пакетов, содержащих определения процедур и функций, а также довольно сложных средств для нестандартной обработки пакетов. В данном контексте, если блокноты (*notebooks*) *Mathematica* с полным основанием можно отождествлять с документами системы *Maple* (*{mws | mw}*-файлами), то пакеты вполне можно ассоциировать с системными модулями *Maple*. Однако, если системные модули в организационном контексте размещаются в библиотеках, структурно аналогичных главной *Maple*-библиотеке, обеспечивая весьма высокую унификацию как пакетных, так и пользовательских средств, пакеты системы *Mathematica* располагаются отдельно, в общем случае, по различным каталогам, поддерживая контекстную взаимосвязь как с программной средой *Mathematica*, так и с другими пакетами как системными, так и пользовательскими. Между тем, в ряде случаев такая организация не представляется нам достаточно удобной при разработке приложений. Локализация пакетов в одном подкаталоге представляется нам более удобной как с точки зрения их программной обработки, так и временной эффективности при организации доступа к ним. Правда, это относится, прежде всего, к случаю разработки достаточно объемных программных проектов в среде системы *Mathematica*; в случае же довольно несложных приложений данное пожелание какого-либо принципиального значения не имеет. Приведенные в настоящей главе средства достаточно полезны при работе с пакетами пользователя.

## Глава 11. Организация программного обеспечения пользователя в среде системы компьютерной математики *Mathematica*

Прежде, чем рассматривать организацию программного обеспечения пользователей, целесообразно ознакомиться с файловой организацией обеих систем, которая может быть, в целом, с достаточной степенью общности представлена следующей схемой.



Данная схема является общей для современных *систем компьютерной математики (СКМ)*, наиболее отчетливо просматриваемой в таких наиболее известных и широко используемых системах, как *Maple* и *Mathematica*. Именно относительно данных двух систем и рассмотрим вкратце наполнение представленной выше схемы.

**Ядро** системы содержит множество высокореактивных функций и процедур, которые обеспечивают довольно представительный набор встроенных функций и операторов системы. Так, *ядро* пакета *Maple* обеспечивает наиболее фундаментальные операции, ответственные прежде всего за *интерпретацию* языка программирования системы и за представление ее структур данных. Ядро также обеспечивает основные операции со структурами данных и математических алгоритмов низкого уровня, точнее содержит набор наиболее массовых высокоэффективных программ, написанных на *C-языке*. В функциональном плане ядро обеспечивает поддержку самых разнообразных задач, например, ядро системы управляет памятью, файловым вводом/выводом (*более того, библиотека функций доступа к компонентам файловой системы компьютера заимствована непосредственно из соответствующей библиотеки C-языка*), обработкой запросов самой системы, запросов к математической библиотеке, наряду с обеспечением поддержки справочной базы данных, и телекоммуникации. Ядро выполняется как отдельный от пользовательского интерфейса процесс. Это позволяет единому пользовательскому интерфейсу использовать в параллельном или смешанном режимах несколько ядер, либо нескольким пользовательским интерфейсам использовать одно ядро, как имеет место в случае с *Maplets* – пакетным модулем, содержащим команды для создания и удобной визуализации различного рода *Maplet*-приложений.

Однопоточное выполнимое ядро на *Windows*-платформе – «*mserver.exe*» («*mserver*» на всех других платформах) или *SMP*-допустимое выполнимое ядро – «*mtserver.exe*» (либо «*mtserver*» на всех других платформах). Одно из данных ядер стартует автоматически в фоновом режиме при загрузке системы *Maple*, общаясь с интерфейсом пользователя по гнезду *TCP/IP*. Такая связь безопасна, ибо она является локальной для компьютера, на котором выполняется система *Maple*, к ней невозможно получить доступ с других компьютеров сети без особых хакерских ухищрений.

Базовые принципы, заложенные при проектировании ядра системы *Maple*, в полной мере наследуются и ядром системы *Mathematica*, однако их реализации различаются

и характеризуются специфическими чертами каждой из систем, наряду с вкусами и пристрастиями разработчиков. В частности, ядро *Mathematica* реализовано языками *Mathematica* и *C*, который зарекомендовал свою высокую эффективность в качестве языка системного программирования. Тогда как ядро *Maple* реализовано *C*-языком с использованием встроенного *Maple*-языка. Так как одним из ограничений функций, включаемых в ядро любой системы является требование их высокой реактивности, то функции, как правило, выполняются очень эффективным экстракодом относительно небольшого объема, а само ядро системы имеет достаточно ограниченный объем.

Поэтому все остальные процедуры и функции оформляются в виде *библиотек*, тогда как процедуры и функции, носящие более специальный характер, наряду с многими пользовательскими процедурами могут располагаться и в специальных документах в форматах, допускаемых той либо иной системой. Так, для *Maple* документы в файлах форматов *{mpl, mws, mw}*, тогда как для системы *Mathematica* в форматах *{m, nb, cdf}*, которые обеспечивают загрузку документов в текущий сеанс на выполнение. Данные документы наряду с библиотеками пользователя пишутся на встроенном языке той или иной системы, что делает возможным их подготовку обычными пользователями. Более того, расширение и функциональных, и прикладных возможностей систем на основе библиотек и документов не ограничено практически ничем. В частности, для системы *Maple* характерна возможность создания *библиотек* пользователей, имеющих организацию как аналогичную организации главной *Maple*-библиотеке системы, так и отличных от нее. Так, сама *главная* библиотека *Maple* содержит определения средств в количестве, зависящем от *релиза* системы (*Maple 8 – 22501, Maple 10 – 27986, Maple 11 – 32883*), тогда как наша библиотека для релизов  $6 \div 11$  содержит чуть более 850 средств, наш пакет *AVZ\_Package\_1* для *Mathematica* включает более 580 процедур и функций; при этом, все стандартные библиотеки пишутся, как правило, на встроенных языках, тогда как наши средства для обеих систем написаны исключительно на этих языках.

Наконец, *интерфейс* современных систем компьютерной математики, включая *Maple* и *Mathematica*, является характерным для всех *Windows*-приложений и обеспечивает присущие ему удобства работы, предоставляя пользователю удобную (*и привычную из других приложений*) возможность обращаться к ядру со своими запросами и получать результат решения непосредственно на экране, сохранять его в файле или на бумаге.

В процессе создания собственного программного обеспечения в среде той либо иной систем естественно возникает вопрос о его наиболее эффективной организации. При этом, используемая программная среда определяет метод организации программных средств, созданных в ней. Если среда позволяет создавать *exe*- или *com*-файлы, прямо выполняемые в операционной среде, то вопрос их организации решается с помощью известных общих подходов, с которыми можно ознакомиться, например, в [1-3]. Как правило, в качестве наиболее распространенного метода здесь выступают различные библиотеки той или иной организации. Совершенно иная картина имеет место в том случае, когда созданные средства могут использоваться только в среде самой системы. В таком случае вопросы организации пользовательского программного обеспечения принимают *специфический* характер, определяемый собственно средствами системы, предназначенными для этих целей. В этом контексте мы и попытались сравнить обе

системы *Maple* и *Mathematica*. Естественно, для пользователя, использующего систему только на внешнем уровне, т.е. в виде написания документов, решающих отдельные задачи, или использующего систему в качестве интеллектуального калькулятора для вычисления математических выражений, формирования на их основе графических объектов, организация программных средств не носит столь актуального характера, однако для пользователя, разрабатывающего довольно серьезные проекты на основе системы с использованием средств, отсутствующих в самой системе или недостаточно эффективных и требующих дополнительного программирования, данная проблема оказывается достаточно актуальной в разработке целого ряда важных приложений.

### 11.1. Организация программного обеспечения пользователя в среде системы компьютерной математики *Mathematica*

Система *Mathematica* не располагает средствами организации библиотек пользователя подобно случаю системы *Maple*, создавая определенные затруднения при организации программного обеспечения, разработанного пользователем в ее среде. Для сохранения определений объектов и результатов вычислений *Mathematica* использует различной организации файлы. При этом, в общем случае наиболее часто используются файлы текстового формата, которые не только легко загружаются в текущий сеанс системы, но и удобны для обработки другими известными средствами, например, текстовыми редакторами. Более того, текстовый формат обеспечивает простую переносимость на другие вычислительные платформы. Одной из основных предпосылок сохранения в файлах является возможность использования определений *Mathematica*-объектов в последующих сеансах системы. Основными операторами сохранения являются ">>" и ">>>", считывания – "<<", т.е. в определенном смысле они соответствуют операторам перенаправления ">", ">>" и "<" базовой операционной системы *MS DOS* [1-3,36], а именно: оператор перенаправления ">" создает либо обновляет уже существующий файл заданным содержимым, тогда как оператор перенаправления ">>" дописывает информацию в конец принимающего файла, сохраняя его предыдущее содержимое.

При использовании операторов ">>" либо ">>>" определения *Mathematica*-объектов сохраняются в файлах во входном формате системы *Mathematica*, обеспечивая весьма простое чтение их обратно в среду системы. Для сохранения в файлах определений *Mathematica*-объектов в других форматах используется форматирующая директива *OutputForm* для сохраняемого выражения. Файлы могут содержать любое количество выражений, однако каждое новое выражение начинается с новой строки, количество которых для выражения не ограничивается. В случае чтения файла оператором "<<" в текущий сеанс системы возвращается *вычисленное последнее* выражение, находящееся в файле; более того, вычисляются все выражения считанного файла, обеспечивая их доступность в текущем сеансе. При соответствующем кодировании завершения для последнего выражения в файле можно управлять визуализацией результата чтения файла в текущий сеанс. Наличие в считываемом файле ошибки возвращает при его считывании соответствующее диагностическое сообщение, которое вполне допускает программную обработку в процессе вычислений, хотя имеется и ряд особенностей.



Следует иметь ввиду, что определяя объект в *Mathematica*, чье определение зависит от других объектов, необходимо загружать в текущую сессию и такие объекты, чтобы обеспечить корректный доступ к основному объекту. Поэтому, сохранять следует все связанные определения. Функция **Save** решает эту задачу, выявляя определения всех объектов, от которых зависит сохраняемый объект и автоматически их сохраняет, но для эффективности исключает из этого списка определения объектов, имена которых имеют *Protected*-атрибут, предполагая доступность таких объектов в текущем сеансе. О других способах сохранения определений *Mathematica*-объектов в файлах разных форматов более говорить нецелесообразно, как не имеющих отношения к проблеме организации программного обеспечения пользователя, для решения которой в виде, определенном выше, вполне достаточно сохранения определений во входном *Input*-формате. Между тем, в случае достаточно сложных и объемных определений вполне приемлемым и эффективным может оказаться и сохранение по функции **DumpSave**, которая обеспечивает сохранение во внутреннем формате системы (бинарный *mx-файл*), наиболее оптимальный для считывания системой. Однако, в качестве недостатка этой функции является то, что сохраненные по ней файлы успешно считываемы только в операционной среде, в которой они были созданы, например, в среде *Windows XP*.

Между тем, здесь целесообразно сделать целый ряд весьма существенных замечаний по использованию указанных средств системы. Во-первых, если по функции **Save** (>>) мы и сохраняем определения *Mathematica*-объектов в заданных файлах, которые по функции **Get** (<<) впоследствии возможно загружать в текущий сеанс с последующей их активацией, позволяя использовать их в текущем сеансе. Между тем, такой вполне естественный подход работает лишь в случае вызова **Get**-функции в *Input*-режиме, но не дает результата при использовании **Get**-функции в теле процедуры. Поэтому для решения ниже описанной задачи нами был использован модифицированный метод «дисковых транзитов», впервые предложенный нами еще в 1978 для ЕС ЭВМ и первых отечественных ПК, и существенно расширяющий возможности программирования; метод эффективен не только для систем компьютерной математики. Для этой цели в конкретных случаях мы использовали функции **Import** и **ToExpression** системы.

Во-вторых, механизм обработки ошибочных и особых ситуаций представляет собой достаточно мощный инструмент программирования практически каждого довольно сложного алгоритма. Однако, в системе *Mathematica* такой механизм характеризуется рядом существенных недостатков, например, успешно используя в *Input*-режиме (*т. н. режиме ввода с консоли*) механизм подавления вывода об ошибочных ситуациях {**Off**, **On**}, в теле процедур такой механизм, вообще говоря, не работает, как иллюстрирует нижеследующий достаточно простой фрагмент, а именно:

```
In[2112]:= Import["D:\\Math_myLib\\ArtKr_2013.m"]
          Import::nffil: File not found during Import. >>
Out[2112]= $Failed
In[2113]:= Off[Import::nffil]
In[2114]:= Import["D:\\Math_myLib\\ArtKr_2013.m"]
Out[2114]= $Failed
```

```

In[2115]:= On[Import::nffil]
In[2116]:= F[x_] := Module[{a}, Off[Import::nffil]; a := Import[x]; On[Import::nffil]; a]
In[2117]:= F["D:\\Math_myLib\\ArtKr_2013.m"]
      Import::nffil: File not found during Import. >>
Out[2117]= $Failed

```

Таким образом, при создании собственных достаточно *сложных* процедур, в которых требуется решать вопросы блокировки вывода ряда ошибочных сообщений, средства системы *Mathematica* представляются нам недостаточно развиты. И, прежде всего, это относится к таким вопросам как разработка довольно сложных процедур и библиотек программных средств пользователя, содержащих средства массового использования.

Принимая во внимание схему организации библиотеки пользовательских процедур, рассмотренную в наших изданиях [97–99] относительно системы *Maple* с отличной от главной библиотеки организацией, представим реализацию подобной библиотеки пользователя для случая системы *Mathematica*. Для этого на первом шаге в файловой системе компьютера создается каталог, например, "C:\\Math\_myLib", который будет содержать *txt*-файлы с определениями процедур/функций пользователя (*по одной в каждом файле*). В принципе, можно помещать в *txt*-файлы любое число определений, но в этом случае предварительно нужно вызывать процедуру, имя которой носит *txt*-файл, после чего в текущем сеансе становятся доступными все процедуры/функции, определения которых находятся в данном файле. Это действительно удобно в случае, когда в единый файл помещаются *основная* процедура и все сопутствующие ей. После чего на втором шаге создаются и отлаживаются процедуры/функции с последующим сохранением их в заданном файле данных библиотечного подкаталога, например:

```

In[2942]:= NF[x_] := Sin[x]*Cos[x]
In[2943]:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]]

```

Тогда как сохранять их в файлах *txt*-формата возможно несколькими способами: (1) по функции **Save**, сохраняющей предварительно вычисленные определения в файле, заданном ее первым фактическим аргументом, например:

```

In[2944]:= Save["C:\\Math_myLib\\NF.txt", NF]
In[2945]:= Save["C:\\Math_myLib\\ArtKr.txt", ArtKr]

```

при этом, сохранение производится в *append*-режиме (*дописывания в конец файла*), или (2) создавая *txt*-файл с именем процедуры/функции, чье определение формируется в файле посредством простого текстового редактора, в частности, *Notepad*.

Следующий фрагмент приводит пример несложной процедуры **CALL**, вызов которой **CALL[G[x,...], d]** обеспечивает возврат результата вызова **G[x,...]** процедуры/функции **G** при условии, что ее определение находится в файле "**G.txt**" каталога с библиотекой, который определяется вторым необязательным аргументом **d**. При вызове процедуры с *одним* аргументом в качестве библиотечного каталога понимается текущий каталог, который определяется вызовом **Directory[]**. Более того, если файла с данным именем нет, то вызов процедуры **CALL** возвращает **\$Failed**. Если процедура/функция **G** уже активирована в текущем сеансе, то файл "**G.txt**" игнорируется и вызов **CALL[G[x,...]]**

относится к активному объекту *G*. Если файл данных "*G.txt*" содержит определения нескольких *одноименных* процедур/функций *G*, то вызов **CALL**[*G*[*x*,...], *d*] процедуры производится относительно ее последнего определения, как наглядно иллюстрируют примеры использования процедуры **CALL** для вызовов двух простых процедур.

```
In[2202]:= NF[x_] := Sin[x]*Cos[x]
In[2203]:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]]
In[2204]:= Save["C:\\Math_myLib\\NF.txt", NF]
In[2205]:= Save["C:\\Math_myLib\\ArtKr.txt", ArtKr]
In[2206]:= NF[x_] := Sin[x]*Cos[x] + x^3
In[2207]:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y
In[2208]:= Save["C:\\Math_myLib\\NF.txt", NF]
In[2209]:= Save["C:\\Math_myLib\\ArtKr.txt", ArtKr]

In[2210]:= CALL[x_, y___] := Module[{a = ToString[x, InputForm], b, c},
                                b = StringTake[a, Flatten[StringPosition[a, "["]][[1]] - 1];
                                If[! MemberQ[{"Null", $Failed}, Definition4[b]], x, c = If[{y} === {}, Directory[],
                                If[StringQ[y] && DirectoryQ[y], y, Directory[]]] <> "\\\" <> b <> ".txt";
                                If[! FileExistsQ[c], $Failed, ToExpression[ReadList[c, "String"]][[-1]]; x]]

In[2211]:= CALL[NF[42.47], "C:\\Math_myLib"]
Out[2211]= 76603.1
In[2212]:= CALL[ArtKr[42.71, 47.66], "C:\\Math_myLib"]
Out[2212]= 2035.56 + 8.97489 I
In[2213]:= Definition4["NF"]
Out[2213]= "NF[x_] := Sin[x]*Cos[x] + x^3"
In[2214]:= Definition4["ArtKr"]
Out[2214]= "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y"
```

Таким образом, подобная организация библиотеки пользователя обеспечивает весьма простой режим ее ведения, тогда как сама процедура **CALL** допускает расширения на достаточно широкие функции работы с пользовательской библиотекой. В частности, принцип модификации текстовых файлов с определениями процедур/функций не только весьма прост, но и позволяет сохранять историю модификации определений библиотечных средств, что в целом ряде случаев является достаточно актуальным.

Для настройки системы под свои нужды пользователь имеет возможность размещать в ее каталогах специальные файлы, загружаемые в текущий сеанс или при загрузке системы, или по необходимости. Эти файлы размещаются, как правило, в системных или в предопределенных пользовательских каталогах. При этом, пользователь может сам определить местоположение таких каталогов посредством переопределения ряда предопределенных переменных системы. В частности, файлы "*init.m*" конфигурации *ядра* системы могут содержать любые команды *Mathematica*. Такие *предопределенные* *\$*-переменные имеют *Protected*-атрибут, поэтому для их переопределения требуется предварительно отменить данный атрибут с последующим его восстановлением. Для

нашего случая интерес представляют две предопределенные переменные, а именно:

**\$BaseDirectory** – определяет *основной* подкаталог системы, в который помещаются все системные файлы, автоматически загружаемые при загрузке системы *Mathematica*;

**\$UserBaseDirectory** – определяет *основной* подкаталог системы, в который помещаются все пользовательские файлы, автоматически загружаемые при загрузке *Mathematica*.

Для нашего случая данные \$-переменные имеют следующие значения, а именно:

```
In[2127]:= $UserBaseDirectory
```

```
Out[2127]= "C:\\Documents and Settings\\Viktor\\Application Data\\Mathematica"
```

```
In[2128]:= $BaseDirectory
```

```
Out[2128]= "C:\\Documents and Settings\\All Users\\Application Data\\Mathematica"
```

Поэтому для удобства последующего использования процедуры **CALL** помещаем ее исходный код в файл *"init.m"*, размещаемый в одном из указанных каталогов. Этим обеспечивается автоматическая загрузка файла при каждой новой загрузке системы, активизируя в текущем сеансе процедуру **CALL**. Однако, в связи с тем, что процедура **CALL** использует прямо и опосредствованно 2 наши процедуры **Definition4**, **SuffPref**, их определения также необходимо включить в файл *"init.m"*. Поэтому в простейшем случае содержимое файла *"init.m"* принимает следующий вид, а именно:

```
(** User Mathematica initialization file **)
Definition4[x_String] := Module[{a}, a = Quiet[Check[Select[StringSplit[ToString[
InputForm[Quiet[Definition[x]]], "\n"], # != " " && # != x &], $Failed]];
If[a === $Failed, $Failed, If[SuffPref[a[[1]], "Attributes["], 1],
a = AppendTo[a[[2 ;; -1]], a[[1]]]; If[Length[a] != 1, a, a[[1]]];
SuffPref[S_String, s_ /; StringQ[s] || ListQ[s] && DeleteDuplicates[Map[StringQ, s]]
== {True}, n_ /; MemberQ[{1, 2, 3}, n]] := Module[{a, b, c, k = 1},
If[StringFreeQ[S, s], False, b = StringLength[S]; c = Flatten[StringPosition[S, s]];
If[n == 3 && c[[1]] == 1 && c[[-1]] == b, True, If[n == 1 && c[[1]] == 1, True,
If[n == 2 && c[[-1]] == b, True, False]]];
CALL[x_, y_] := Module[{a = ToString[x, InputForm], b, c},
b = StringTake[a, Flatten[StringPosition[a, "["]][[1]] - 1];
If[! MemberQ[{"Null", $Failed}, Definition4[b]], x, c = If[{y} === {}, Directory[],
If[StringQ[y] && DirectoryQ[y], y, Directory[]]] <> "\\ " <> b <> ".txt";
If[! FileExistsQ[c], $Failed, ToExpression[ReadList[c, "String"]][[-1]]; x]]];
```

которое легко модифицируется любым текстовым редактором, например, *Notepad*.

Представленный выше простой подход к организации средств пользователя в среде системы *Mathematica* является лишь одним из возможных, предоставляя возможность создания собственных *библиотек* процедур/функций с доступом на уровне системных средств. Действительно, достаточно несложно поместить исходный код (*определение*) отлаженной процедуры/функции в *txt*-файл с ее именем, сохранить его в выбранном подкаталоге библиотеки пользователя и поместить исходный код процедуры **CALL** в

файл «*init.m*», который находится (*размещается*) в одном из каталогов, определяемых переменными **\$BaseDirectory** и **\$UserBaseDirectory**. В результате чего каждая загрузка системы *Mathematica* делает доступными в текущем сеансе все процедуры/функции, чьи определения находятся в файлах библиотечного каталога пользователя, наравне со встроенными средствами системы, на основе вызовов процедуры **CALL**, например:

```
In[2172]:= AGN = Sqrt[CALL[NF[42.47]]^2 + CALL[ArtKr[19.89, 19.96]]^3]
```

```
Out[2172]= 77034.1
```

```
In[2173]:= Definition4["NF"]
```

```
Out[2173]= "NF[x_] := Sin[x]*Cos[x] + x^3"
```

```
In[2174]:= Definition4["ArtKr"]
```

```
Out[2174]= "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y"
```

Как уже отмечалось, один такой *txt*-файл вполне может содержать ряд определений одноименных процедур/функций, однако в текущем сеансе активизируется только последнее из определений, находящихся в файле. Тогда первый же вызов процедуры или функции активизирует ее в текущем сеансе, после чего обращаться к ней можно уже без использования процедуры **CALL**, т.е. традиционным для системы способом.

Отметим, что процедура **CALL** может быть расширена по различным направлениям в зависимости от нужд приложений пользователя. Например, представляет интерес модификация процедуры **CALL** на случай файлов *mx*-формата, который позволяет по функции **DumpSave["Name.mx", {N<sub>1</sub>, N<sub>2</sub>, ..., N<sub>k</sub>}]** эффективно сохранять в файлах данного типа определения больших процедур. Считывание таких файлов в текущий сеанс можно производить посредством функции **Get**; при этом, *mx*-формат обладает весьма существенным свойством: в отличие от файлов форматов {*m*, *nb*, *txt*} функция **Get** позволяет загружать и активизировать в текущем сеансе определения процедур и функций, содержащихся в загружаемом *mx*-файле, безотносительно точки их вызова - в *Input*-режиме или в теле процедуры. Именно данное свойство, порой, существенно упрощает использование вызовов функции **Get** в теле процедур, которые базируются на активизации определений в текущем сеансе системы, находящихся в загружаемых *mx*-файлах данных. Именно это свойство эффективно при организации библиотек.

```
In[2187]:= NF[x_] := Sin[x]*Cos[x] + x^3
```

```
In[2188]:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y
```

```
In[2189]:= CALLmx[y_, z_ /; MemberQ[{1, 2}, z], d___] := Module[{c = {}, h, k = 1, s,
    a = If[{d} === {}, Directory[], If[StringQ[d] && DirectoryQ[d], d, Directory[]]},
    b = Map[ToString, If[ListQ[y], y, {y}]],
    If[z == 1, While[k <= Length[b], s = b[[k]]; h = a <> "\\\" <> s <> ".mx";
        If[! MemberQ[{"Null", $Failed}, Definition4[s]],
            ToExpression["DumpSave[" <> ToString1[h] <> ", " <> ToString[s] <> ""];
            AppendTo[c, s]; k++; Prepend[c, a],
        While[k <= Length[b], s = b[[k]]; h = a <> "\\\" <> s <> ".mx";
            If[FileExistsQ[h], Get[h]; AppendTo[c, s]; k++; c]]
```

```

In[2190]:= CALLmx[{NF, ArtKr}, 1]
Out[2190]= {"C:\\Documents and Settings\\Viktor\\My Documents", "NF", "ArtKr"}
In[2191]:= Clear[NF, ArtKr]
In[2192]:= CALLmx[{NF, ArtKr}, 2]
Out[2192]= {"NF", "ArtKr"}
In[2193]:= AGN = Sqrt[NF[42.47]^2 + ArtKr[19.89, 19.96]^3]
Out[2193]= 77034.1

```

В качестве одного из таких расширений **CALL** представим процедуру **CALLmx**, вызов которой обеспечивает сохранение в библиотечном каталоге определений объектов в форме *mx*-файлов с возможностью их последующей загрузки в текущий либо другой сеанс системы. Вызов **CALLmx**[*y*, 1, *d*] процедуры возвращает список, первый элемент которого определяет библиотечный каталог, тогда как остальные – имена объектов из аргумента *y* (*отдельное имя или их список*), определения которых вычислены в текущем сеансе; необязательный аргумент *d* определяет каталог, в который будут помещаться вычисленные определения объектов *y* в форме *mx*-файлов с именами "*Name.mx*", где *Name* – имя объекта, определяемого аргументом *y*; в случае отсутствия аргумента *d* в качестве библиотечного выбирается каталог, определяемый вызовом **Directory[]**. В то время как вызов **CALLmx**[*y*, 2, *d*] обеспечивает загрузку в текущий сеанс объектов, чьи имена определены аргументом *y*, из библиотечного каталога, определяемого третьим аргументом *d* либо его отсутствием. Предыдущий фрагмент представляет исходный код процедуры **CALLmx** наряду с наиболее типичными примерами ее применения. Следует отметить, что в один файл целесообразнее всего помещать только основную процедуру и те функции, на которые имеются ссылки в ее теле, исключая ссылки на процедуры с *Protected*-атрибутом, которые в *Mathematica* по умолчанию полагаются встроенными. Это позволяет весьма просто оформлять данные процедурные файлы.

В свою очередь, процедуру **CALLmx** можно расширить на справочную информацию о библиотечных средствах, поддерживаемых данной процедурой. Данная методика достаточно проста и состоит в следующем. *Mathematica* предоставляет пользователю *usage*-механизм, позволяющий обеспечивать справками средства пользователя, суть которого хорошо иллюстрирует следующий весьма простой фрагмент, а именно:

```

In[2116]:= P::usage = "Help on procedure P"; F::usage = "Help on function F";
In[2117]:= DumpSave["HelpBase.mx", {P, F}]
Out[2117]= {P, F}
In[2118]:= Clear[P, F]
In[2119]:= Get["HelpBase.mx"]
In[2120]:= ?P
      Help on procedure P
In[2121]:= ??F
      Help on function F
In[2122]:= Information[P]
      Help on procedure P

```

Таким образом, определять справочную информацию можно и по несуществующим программным средствам, что впоследствии позволяет получать ее вызовами формата  $\{?N|??N| \text{Information}[N]\}$ , где  $N$  – имя программного средства. Между тем, для случая программирования пакета справки (*usage*) по всем объектам, чьи определения входят в пакет, обеспечивают корректность их экспорта, а также возможность получения по ним *справок* после загрузки пакета в текущий сеанс. Итак, справочную базу возможно создавать и модифицировать независимо от описываемых ею программных средств, что достаточно удобно при проектировании и разработке больших проектов в среде системы *Mathematica*. Взаимосвязь определений средств пользовательского пакета и организация справок по ним рассматривалась нами несколько выше.

```

In[2161]:= NF[x_] := Sin[x]*Cos[x] + x^3
In[2162]:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y
In[2163]:= Help[] := {NF::usage = "Help on function NF",
                     ArtKr::usage = "Help on function ArtKr"}

In[2164]:= CALLmxH[y_, z_;/; MemberQ[{1, 2}, z], help_, d_] := Module[{h, k = 1, s, t,
    a = If[{d} === {}, Directory[], If[StringQ[d] && DirectoryQ[d], d, Directory[]]],
    b = Map[ToString, If[ListQ[y], y, {y}]], c = {}},
    t = a <> "\\\" <> ToString[help] <> ".mx";
    If[z == 1, While[k <= Length[b], s = b[[k]]; h = a <> "\\\" <> s <> ".mx";
        If[! MemberQ[{"Null", $Failed}, Definition4[s]],
            ToExpression["DumpSave[" <> ToString1[h] <> "," <> ToString[s] <> "]];
            ToExpression["DumpSave[" <> ToString1[t] <> "," <> ToString1[help] <> "]];
            AppendTo[c, s]; k++; Prepend[c, a],
        While[k <= Length[b], s = b[[k]]; h = a <> "\\\" <> s <> ".mx";
            If[FileExistsQ[h], Get[h]; Get[t]; help[]; AppendTo[c, s]; k++; c]]

In[2165]:= CALLmxH[{NF, ArtKr}, 1, Help]
Out[2165]= {"C:\\Documents and Settings\\Viktor\\My Documents", "NF", "ArtKr"}
In[2166]:= Clear[NF, ArtKr, Help]
In[2167]:= CALLmxH[{NF, ArtKr}, 2, Help]
Out[2167]= {"NF", "ArtKr"}
In[2168]:= AGN = Sqrt[NF[42.47]^2 + ArtKr[19.89, 19.96]^3]
Out[2168]= 77034.1
In[2169]:= ??NF
Help on function NF
In[2170]:= ?ArtKr
Help on function ArtKr
    
```

Предыдущий фрагмент представляет процедуру **CALLmxH**, которая является одной из возможных простых модификаций процедуры **CALLmx**, поддерживающей работу со справочной базой библиотеки пользователя. Так, для справочной базы используется списочная структура следующего простого формата, а именно:

**Help[]** := {F1::usage = "Help on object F1", ..., Fn::usage = "Help on object Fn"}

Тогда как формат вызова принимает вид **CALLmxH[{F1, ..., Fn}, {1 | 2}, Help, d]**, когда в отношении 1, 2 и 4 аргументов полностью сохраняет силу все сказанное относительно формата вызова **CALLmx[{F1, ..., Fn}, {1 | 2}, d]** процедуры **CALLmx**, тогда как аргумент **Help** определяет справочную базу библиотеки. При таком подходе вызов процедуры **CALLmxH[{F1, ..., Fn}, 1, Help, d]** (где аргумент **d** необязателен) создает набор *mx*-файлов с определениями средств библиотеки пользователя и *mx*-файл со справочной базой. Все последующие загрузки библиотеки по вызову **CALLmxH[{F1, ..., Fn}, 2, Help, d]** не только обеспечивают в текущем сеансе доступность определений средств {F1, ..., Fn}, находящихся в библиотеке пользователя, но и к ее справочной базе. Приведенные во фрагменте примеры иллюстрируют сказанное наряду с исходным кодом **CALLmxH**.

В качестве еще одного простого примера ведения пользовательских библиотек можно привести процедуру **UserLib[L, S]**, поддерживающую ряд полезных функций, чей код с примерами типичного применения представляет следующий фрагмент, а именно:

```
In[4224]:= UserLib[L_ /; FileExistsQ[L], f_ /; ListQ[f]] := Module[{a, b = "", c, d = 0},
  If[f[[1]] === "print" && f[[2]] === "all" && ! EmptyFileQ[L], FilePrint[L],
  If[f[[1]] === "print" && f[[2]] != "all" && ! EmptyFileQ[L],
    a = OpenRead[L]; While[b != "EndOfFile", b = Read[a, String];
    If[SuffPref[b, f[[2]]] <> "[", 1, Print[b]; d = 1; Continue[],
    If[d == 1, If[b === "\"$$$$$", Close[a]; Break[], Print[b];
    Continue[], Continue[]]; If[d == 1, Null,
    Print[f[[2]] <> " is absent in Library " <> StrStr[L]],
  If[f[[1]] === "add", PutAppend[ToExpression["Definition[" <> f[[2]] <> "]", L];
  PutAppend["$$$$$", L],
  If[f[[1]] === "load" && f[[2]] === "all" && ! EmptyFileQ[L], Get[L],
  If[f[[1]] === "load" && f[[2]] != "all" && ! EmptyFileQ[L],
    a = OpenRead[L]; c = ""; While[b != "EndOfFile", b = Read[a,
    String]; If[SuffPref[b, f[[2]]] <> "[", 1, c = c <> b; d = 1; Continue[],
    If[d == 1, If[b === "\"$$$$$", Close[a]; Break[], c = c <> b;
    Continue[], Continue[]]; If[d == 1, ToExpression[c],
    Print[f[[2]] <> " is absent in Library " <> StrStr[L]],
  If[f[[1]] === "names" && f[[2]] === "list" && ! EmptyFileQ[L],
    a = OpenRead[L]; c = {}; While[b != "EndOfFile", b = Read[a, String];
    If[Quiet[StringTake[b, {1, 1}]] != " " && b != "\"$$$$$",
    c = Append[c, StringTake[b, {1, Flatten[StringPosition[b, "["][[1]] - 1]}],
    Continue[]]; Close[a]; Return[c, Defer[UserLib[L, f]]]]];

In[4225]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "Lib"}]
Out[4225]= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "Lib"}]
In[4226]:= UserLib["D:\\Math_MyLib\\Library.txt", {"names", "list"}]
Out[4226]= UserLib["D:\\Math_MyLib\\Library.txt", {"names", "list"}]
```



```

In[4227]:= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "all"}]
Out[4227]= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "all"}]
In[4228]:= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "Lib"}]
Out[4228]= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "Lib"}]
In[4229]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "all"}]
Out[4229]= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "all"}]
In[4230]:= UserLib["D:\\Math_MyLib\\Library.txt", {"add", "Lib1"}]
=====
In[4261]:= UserLib["D:\\Math_MyLib\\Library.txt", {"add", "Lib8"}]
In[4262]:= UserLib["D:\\Math_MyLib\\Library.txt", {"names", "list"}]
Out[4262]= {"Lib1", "Lib2", "Lib3", "Lib4", "Lib5", "Lib6", "Lib7", "Lib8"}
In[4263]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "Lib7"}]
In[4264]:= UserLib["D:\\Math_MyLib\\Library.txt", {"Names", "List"}]
Out[4264]= UserLib["D:\\Math_MyLib\\Library.txt", {"Names", "List"}]
In[4265]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "ArtKr"}]
      ArtKr is absent in Library "D:\\Math_MyLib\\Library.txt"
In[4266]:= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "ArtKr"}]
      ArtKr is absent in Library "D:\\Math_MyLib\\Library.txt"

```

Достаточно простая процедура **UserLib**[*L*, *f*] поддерживает ряд массовых функций по ведению простой библиотеки пользователя, расположенной в файле *L* *txt*-формата. В качестве второго фактического аргумента *f* выступает двух-элементный список, для которого допустимыми парами значений элементов являются нижеследующие:

**{"names", "list"}** – вывод списка имен процедур/ функций, чьи определения находятся в библиотечном файле; в случае пустого файла вызов возвращается невычисленным;

**{"print", "all"}** – вывод на экран *всего* содержимого библиотечного файла *L*; тогда как в случае пустого файла вызов процедуры **UserLib** возвращается невычисленным;

**{"print", "Name"}** – вывод на экран определения процедуры/функции с именем *Name*, находящейся в библиотечном файле *L*; для пустого файла вызов процедуры **UserLib** возвращается невычисленным; при отсутствии в библиотечном файле *L* требуемого средства вызов процедуры возвращает **Null**, т.е. ничего; в таком случае вызов **UserLib** печатает сообщение следующего общего вида *"Name is absent in Library L"*;

**{"add", "Name"}** – сохранение в библиотечном файле *L* в *append*-режиме процедуры/ функции с именем *Name*; определение сохраняемого средства *предварительно* должно быть вычислено в текущем сеансе системы в *Input*-режиме;

**{"load", "all"}** – загрузка в текущий сеанс *всех* средств, определения которых находятся в библиотечном файле *L*; тогда как в случае пустого файла вызов процедуры **UserLib** возвращается невычисленным;

**{"load", "Name"}** – загрузка в текущий сеанс процедуры/функции с именем *Name*, чье определение находится в библиотечном файле *L*; тогда как в случае пустого файла *L* вызов процедуры **UserLib** возвращается невычисленным; в случае отсутствия в файле *L* требуемого средства вызов процедуры **UserLib** возвращает **Null**, т.е. ничего, выводя сообщение следующего общего вида *"Name is absent in Library L"*.

В остальных случаях вызов процедуры **UserLib** возвращается *невывчисленным*. Имеется хорошая возможность расширить процедуру рядом полезных функций такими, как: удаление из библиотеки определений указанных средств или старых их версий и т.д.

Списочная структура *Mathematica* позволяет достаточно легко симулировать работу со структурами других систем компьютерной математики, в частности, *Maple*. Так, в системе *Maple* одной из наиболее важных является табличная структура, достаточно широко используемая как для организации структур данных, так и для организации библиотек программных средств. Подобная *табличная* организация используется для организации пакетных модулей *Maple* и ряда средств нашей Библиотеки [45]. Тогда как для симулирования основных операций с *табличной* организацией, аналогичной системе *Maple*, в среде *Mathematica* может быть использована процедура **Table1**[*L*, *x*], рассматривающая в качестве *таблицы* список *L* типа *ListList*, 2-элементные подспски {*x*,*y*} которого отвечают {*index*, *entry*} *Maple*-таблиц соответственно. В качестве второго *x*-аргумента выступают: (1) список {*a*, *b*}, (2) слово {"*index*" | "*entry*"} или (3) выражение иного типа. Результатом вызова **Table1**[*L*, *x*] процедуры является список типа *ListList*, получаемый из исходного списка *L* следующим образом. В случае (1) при наличии в *L* подсписка с *первым* элементом *a* он заменяется на список {*a*, *b*}, в противном случае он дополняет *L*; если аргумент *x* имеет вид {*a*, *Null*}, при наличии в *L* подсписка с *первым* элементом *a* подсписк удаляется. Для случая (2) возвращается список {*indices* | *entries*} соответственно списка *L*; тогда как в случае (3) вызов процедуры возвращает вход для *x*-индекса, если такой в данной таблице реально существует. На остальных кортежах фактических аргументов вызов **Table1**[*x*,*y*] возвращает **\$Failed**. Следующий фрагмент представляет исходный код процедуры с типичными примерами ее использования.

```
In[2212]:= Table1[L_, x_] := Module[{a = {}, c = L, d = {}, k = 1, b = Length[L]},
    If[ListListQ[L] && Length[L][[1]] == 2,
    For[k, k <= b, k++, AppendTo[a, L[[k]][[1]]]; AppendTo[d, L[[k]][[2]]];
    {a, d} = Map[DeleteDuplicates, {a, d}];
    If[x === "index", a, If[x === "entry", d, If[ListQ[x] && Length[x] == 2,
    If[! MemberQ[a, x[[1]]], AppendTo[c, x],
    Select[Map[If[#1[[1]] === x[[1]] && ! SameQ[x[[2]], Null], x,
    If[#[[1]] === x[[1]] && x[[2]] === Null, Null, #]] &, L], ! SameQ[#, Null] &]],
    Quiet[Check[Select[Map[If[#[[1]] === x, #[[2]] &, L],
    ! SameQ[#, Null] &][[1]], $Failed]]], $Failed]]

In[2213]:= Tab1 := {{a, a72}, {b, b42}, {c, c47}, {Kr, d17}, {Art, h24}}
In[2214]:= Table1[Tab1, "entry"]
Out[2214]= {a72, b42, c47, d17, h24}
In[2215]:= Table1[Tab1, "index"]
Out[2215]= {a, b, c, Kr, Art}
In[2216]:= Table1[Tab1, {ArtKr, 2013}]
Out[2216]= {{a, a72}, {b, b42}, {c, c47}, {Kr, d17}, {Art, h24}, {ArtKr, 2013}}
In[2217]:= Table1[Tab1, {Kr, 2013}]
```

```

Out[2217]= {{a, a72}, {b, b42}, {c, c47}, {Kr, 2013}, {Art, h24}}
In[2218]:= Table1[Tab1, Art]
Out[2218]= h24
In[2219]:= Table1[Vsv, ArtKr]
Out[2219]= $Failed
In[2220]:= Table1[Tab1, {Vsv, Agn}]
Out[2220]= {{a, a72}, {b, b42}, {c, c47}, {Kr, d17}, {Art, h24}, {Vsv, Agn}}

```

На основе табличной организации, поддерживаемой процедурой **Table1**, достаточно несложно можно определять пользовательские библиотеки процедур и функций. На указанной основе в качестве одного из таких подходов приведем пример библиотеки **LibBase**, структурная организация которой имеет формат списка типа *ListList* и чьи элементы имеют длину 2. Ниже приведен принципиальный вид такой библиотеки:

```

LibBase := {{Help, {"O1::usage = \"Help on O1. \", ..., \"On::usage = \"Help on On. \"}},
            {O1, PureDefinition[O1]}, {O2, PureDefinition[O2]}, ..., {On, PureDefinition[On]}}

```

Первый 2-элементный подсписок **LibBase** своим первым элементом содержит **Help**, в то время как второй представляет список справок (*usage*) в строчном формате по всем объектам, чьи определения находятся в библиотеке **LibBase**; при этом, фактического присутствия их в библиотеке не требуется. Остальные элементы библиотеки являются 2-элементными подсписками формата *{Oj, PureDefinition[Oj]}*, где *Oj* – имя *j*-объекта, и **PureDefinition[Oj]** – его определение, представленное в строчном формате.

```

In[2403]:= LibBase := {{Help, {"NF::usage = \"Help on function NF.\",
                                "ArtKr::usage = \"Help on function ArtKr.\""}},
                        {"NF", PureDefinition[NF]}, {"ArtKr", PureDefinition[ArtKr]}}
In[2404]:= DumpSave["LibBase.mx", LibBase]
Out[2404]= {{{Help, {"NF::usage = \"Help on function NF.\",
                    "ArtKr::usage = \"Help on function ArtKr.\""}},
             {"NF", PureDefinition[NF]}, {"ArtKr", PureDefinition[ArtKr]}}}
In[2405]:= TabLib[lib_ /; FileExistsQ[lib] && FileExtension[lib] == "mx", x_, y_] :=
Module[{a = Get[lib], b, c},
  If[MemberQ[{"index", "entry"}, x], Table1[LibBase, x],
    Map[ToExpression, LibBase[[1]][[2]]];
  If[ListQ[x] && Length[x] == 2, c = If[SameQ[x[[2]], Null], x,
    {x[[1]], PureDefinition[x[[1]]]}; b = Table1[LibBase, c];
  If[! SameQ[b, $Failed], LibBase = b;
  ToExpression["DumpSave[" <> ToString1[lib] <> ", " <> "LibBase"]"],
  If[StringQ[x] && ! StringFreeQ[x, "::usage = "],
  c = Quiet[LibBase[[1]][[2]] = AppendTo[LibBase[[1]][[2]], x];
  LibBase = ReplacePart[LibBase, {1, 2} -> c];
  ToExpression["DumpSave[" <> ToString1[lib] <> ", " <> "LibBase"]"],
  If[Table1[LibBase, x] === $Failed, $Failed, b = Table1[LibBase, x];
  If[! SameQ[b, $Failed], ToExpression[b; x[y]], $Failed]]]]

```

```

In[2406]:= TabLib["LibBase.mx", "index"]
Out[2406]= {Help}
In[2407]:= TabLib["LibBase.mx", "entry"]
Out[2407]= {"NF::usage = \"Help on function NF.\",
            \"ArtKr::usage = \"Help on function ArtKr.\""}
In[2408]:= NF[x_] := Sin[x]*Cos[x] + x^3
In[2409]:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y
In[2410]:= TabLib["LibBase.mx", {ArtKr, PureDefinition[ArtKr]}]
Out[2410]= {{{Help, {"NF::usage = \"Help on function NF.\",
                    \"ArtKr::usage = \"Help on function ArtKr.\""},
              {ArtKr, "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y"}}}
In[2411]:= TabLib["LibBase.mx", {NF, PureDefinition[NF]}]
Out[2411]= {{{Help, {"NF::usage = \"Help on function NF.\",
                    \"ArtKr::usage = \"Help on function ArtKr.\""},
              {ArtKr, "ArtKr::usage = \"Help on function ArtKr.\""},
              {NF, "NF[x_] := Sin[x]*Cos[x] + x^3"}}}
In[2412]:= TabLib["LibBase.mx", "index"]
Out[2412]= {Help, ArtKr, NF}
In[2413]:= Clear[ArtKr, LibBase, NF]
In[2414]:= TabLib["LibBase.mx", ArtKr, 78.42, 460.2013]
Out[2414]= 36091.6
In[2415]:= TabLib["LibBase.mx", NF, 460.2013]
Out[2415]= 9.74638*10^7
In[2416]:= Table1["LibBase.mx", ArtKr]
Out[2416]= ArtKr[]
In[2417]:= Table1["LibBase.mx", {NF, Null}]
Out[2417]= {"NF::usage = \"Help on function NF.\",
            \"ArtKr::usage = \"Help on function ArtKr.\""},
            {ArtKr, "ArtKr::usage = \"Help on function ArtKr.\""}
In[2418]:= TabLib["LibBase.mx", Avz42]
Out[2418]= $Failed
In[2119]:= TabLib["LibBase.mx", "Avz::usage = \"Help on function Avz.\""]
Out[2119]= {{{Help, {"NF::usage = \"Help on function NF.\",
                    \"ArtKr::usage = \"Help on function ArtKr.\",
                    \"Avz::usage = \"Help on function Avz.\""},
              {ArtKr, "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y"}}}
In[2120]:= LibBase
Out[2120]= {{{Help, {"NF::usage = \"Help on function NF.\",
                    \"ArtKr::usage = \"Help on function ArtKr.\",
                    \"Avz::usage = \"Help on function Avz.\""},
              {ArtKr, "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y"}}}

```

```
In[2421]:= ??NF
```

```
Help on function NF
```

```
In[2422]:= ?ArtKr
```

```
Help on function ArtKr
```

Основные операции с библиотекой, организованной таким образом, обеспечиваются процедурой **TabLib**, исходный код которой с примерами применения представлены предыдущим фрагментом. Вызов **TabLib**[*x*, *y*] в зависимости от второго аргумента *y* возвращает либо текущее содержимое библиотеки, находящейся в *mx*-файле *x*, либо имена объектов, находящихся в библиотеке, либо их определения, а именно:

**TabLib**[*x*, "index"] – возвращает список *имен* объектов, определения которых находятся в библиотеке *x*, включая имя справочной базы *Help* библиотеки;

**TabLib**[*x*, "entry"] – возвращает список определений объектов, которые содержатся в библиотеке *x*, включая и справочную базу *Help* библиотеки;

**TabLib**[*x*, {*N*, *Df*}] – возвращает содержимое библиотеки *x* после расширения ее новым определением объекта с именем *N*, если *Df* отлично от значения *Null*; при этом, старое определение *N*-объекта обновляется;

**TabLib**[*x*, {*N*, *Null*}] – возвращает содержимое библиотеки *x* в результате удаления из нее определения объекта с именем *N*; при этом, справка (*usage*) по *N* сохраняется;

**TabLib**[*x*, *N*, *y*, *z*, ...] – возвращает результат вызова *N*[*y*, *z*, ...] процедуры/функции из библиотеки *x*; если объект *N* отсутствует в библиотеке, то возвращается **\$Failed**;

**TabLib**[*x*, *N*] – если *N* – справка (*usage*) по объекту *N*, она дополняет справочную базу библиотеки *x* с возвратом обновленного содержимого библиотеки *x*.

В остальных случаях вызов возвращает невычисленным либо возвращает **\$Failed**. В качестве *исходной* библиотеки **LibBase**, предназначенной для наполнения ее нужным содержимым, может быть определена списочная структура следующего формата:

**LibBase := {{Help, {}}}**

Следующий простой фрагмент весьма наглядно иллюстрирует сказанное, а именно:

```
In[2127]:= LibBase := {{Help, {}}}
```

```
In[2128]:= DumpSave["Tablib.mx", LibBase]
```

```
Out[2128]= {{{Help, {}}}}
```

```
In[2129]:= TabLib["Tablib.mx", "Avz::usage = \"Help on function Avz.\""]
```

```
Out[2129]= {{{Help, {"Avz::usage = \"Help on function Avz.\""}}}}
```

```
In[2130]:= Avz[x_, y_, z_] := Sin[x + y + z] + 78*x*y*z
```

```
In[2131]:= TabLib["Tablib.mx", {Avz, Df}]
```

```
Out[2131]= {{{Help, {"Avz::usage = \"Help on function Avz.\""},
               {Avz, "Avz[x_, y_, z_] := Sin[x + y + z] + 78*x*y*z"}}}}
```

```
In[2132]:= Clear[Avz]; TabLib["Tablib.mx", Avz, 42.71, 47.66, 67.46]
```

```
Out[2132]= 1.07109*10^7
```

```
In[2133]:= Information[Avz]
```

```
Help on function Avz
```

Естественно, для реального использования **TabLib** в качестве готового программного средства для организации пользовательских библиотек она требует *расширения* своих функциональных возможностей, между тем, она представлена как иллюстративный пример одного из *возможных* подходов к решению задачи организации программных средств пользователя. Оставляем данную задачу для заинтересованного пользователя программной среды *Mathematica* в качестве достаточно полезного упражнения.

В принципе, представленная *библиотечная* организация, обеспечиваемая процедурой **TabLib** и базирующаяся на табличной организации, которая поддерживается **Table1**, представляет собой своего рода аналог *Maple*-пакета табличного типа. Естественно, представленная библиотека имеет лишь базовый набор функций, который, впрочем, обеспечивает ее вполне удовлетворительное функционирование. Между тем, на базе предложенного подхода вполне реально создавать быстрые относительно небольшие библиотеки пользовательских процедур и функций, весьма удобные в эксплуатации. Более того подобные довольно простые средства могут сослужить хорошую службу в деле поддержки и ведения библиотек пользовательских процедур/функций, которые имеют простой текстовый формат, мобильны и весьма просто редактируемы такими средствами, как обычные текстовые редакторы и процессоры, например, *Notepad*.

Заинтересованный читатель может разработать собственные средства организации в среде *Mathematica* своих программных средств, используя предложенные нами либо иные подходы. Однако остается открытым вопрос организации для пользовательских библиотек удобной справочной базы. Здесь имеется ряд подходов, в качестве одного из них можно предложить создание *mx*-файлов (*BaseHelp*), содержащих описания всех процедур/функций *библиотеки* пользователя, находящихся в библиотечном каталоге. Работа с подобной *справочной* базой обеспечивается процедурой **UsageBase**, исходный код которой с примерами использования приведены в следующем фрагменте.

```
In[2287]:= BaseHelp := {}
In[2288]:= DumpSave["BaseHelp.mx", BaseHelp]
Out[2288]= {}
In[2289]:= Clear[BaseHelp]

In[2290]:= UsageBase[x_;/; FileExistsQ[x] && FileExtension[x] == ".mx", y___] :=
Module[{a = Get[x], b, c = 6, d, k = 1},
  If[{y} == {}, Map[ToExpression, BaseHelp];,
  If[Length[{y}] == 1 && SameQ[y, "?"], BaseHelp,
  If[Length[{y}] == 1 && SymbolQ[y],
    BaseHelp = Select[BaseHelp, StringTake[#1,
      {1, Flatten[StringPosition[#, "::usage"]][[2]]] != ToString[y] <> "::usage" &];
    ToExpression["DumpSave[" <> ToString1[x] <> ", " <> "BaseHelp"]];,
  If[Length[{y}] == 1 && StringQ[y] && ! StringFreeQ[y, {"::usage=", "::usage="}],
    If[BaseHelp == {}, AppendTo[BaseHelp, y],
    b = StringTake[y, {1, Flatten[StringPosition[y, "::usage"]][[2]]];
    For[k, k <= Length[BaseHelp], k++, d = BaseHelp[[k];
```

```

If[StringTake[d, {1, Flatten[StringPosition[d, "::usage"]][[2]]}] === b,
BaseHelp[[k]] = y; c = 78; Break[]]; If[c != 78, AppendTo[BaseHelp, y]];
ToExpression["DumpSave[" <> ToString1[x] <> ", " <> "BaseHelp"]; ]]]]

In[2291]:= UsageBase["BaseHelp.mx", "Avz::usage = \"Help on function Avz.\""]
In[2292]:= BaseHelp
Out[2292]= {"Avz::usage = \"Help on function Avz.\""}
In[2293]:= UsageBase["BaseHelp.mx", "?"]
Out[2293]= {"Avz::usage = \"Help on function Avz.\""}
In[2294]:= Clear[BaseHelp]; Get["BaseHelp.mx"]; BaseHelp
Out[2294]= {"Avz::usage = \"Help on function Avz.\""}
In[2295]:= UsageBase["BaseHelp.mx", Avz]
In[2296]:= Clear[BaseHelp]; Get["BaseHelp.mx"]; BaseHelp
Out[2296]= {}
In[2297]:= UsageBase["BaseHelp.mx", "Avz::usage = \"Help on function Avz.\""]
In[2298]:= Clear[BaseHelp]; Get["BaseHelp.mx"]; BaseHelp
Out[2298]= {"Avz::usage = \"Help on function Avz.\""}
In[2299]:= UsageBase["BaseHelp.mx", "Avz::usage = \"Help on procedure Avz.\""]
In[2300]:= Clear[BaseHelp]; Get["BaseHelp.mx"]; BaseHelp
Out[2300]= {"Avz::usage = \"Help on procedure Avz.\""}
In[2301]:= UsageBase["BaseHelp.mx", "Agn::usage = \"Help on procedure Agn.\""]
In[2302]:= Clear[BaseHelp]; Get["BaseHelp.mx"]; BaseHelp
Out[2302]= {"Avz::usage = \"Help on procedure Avz.\"",
            "Agn::usage = \"Help on procedure Agn.\""}
In[2303]:= UsageBase["BaseHelp.mx", "?"]
Out[2303]= {"Avz::usage = \"Help on procedure Avz.\"",
            "Agn::usage = \"Help on procedure Agn.\""}
In[2304]:= UsageBase["BaseHelp.mx"]
In[2305]:= ??Avz
            Help on procedure Avz
In[2306]:= ?Agn
            Help on procedure Agn

```

На основе списочной организации, поддерживаемой системой, достаточно несложно можно определять справочные базы для пользовательских библиотек. На этой основе в качестве одного из таких подходов приведем пример *справочной* базы **BaseHelp**, чья структурная организация имеет формат списка с элементами нижеследующего вида. Ниже представлен принципиальный вид такой справочной базы, а именно:

**BaseHelp** := {"O1::usage = \"Help on O1. \", ..., "On::usage = \"Help on On. \"}

Таким образом, *справочная* база представляет собой список справок (*usage*) в строчном формате по всем *Oj*-объектам, чьи определения находятся в библиотеке пользователя; более того, в качестве исходной справочной базы **BaseHelp**, предназначенной для ее

наполнения нужным содержимым, может быть определен пустой список, а именно:

**BaseHelp** := {}

Базовые операции с базой справок, организованной таким образом, обеспечиваются процедурой **UsageBase**, чей исходный код с примерами применения представлены в предыдущем фрагменте. Вызов **UsageBase[x, y]** в зависимости от второго аргумента *y* возвращает либо текущее содержимое справочной базы, находящейся в *mx*-файле *x*, либо удаляет ее записи, либо обновляет или активирует их, а именно:

**UsageBase[x, "?"]** – возвращает содержимое *справочной базы BaseHelp* в формате списка, которая находится в *mx*-файле данных *x*;

**UsageBase[x, y]** – возвращает **Null**, т.е. ничего, удаляя из справочной базы, заданной *mx*-файлом данных *x*, справку по объекту с именем *y*, если такая существует;

**UsageBase[x, y]** – возвращает **Null**, т.е. ничего, дополняя справочную базу, заданную *mx*-файлом *x*, справкой *y* вышеуказанного формата, или заменяет в справочной базе справку на справку *y*, если имена *Obj*-объектов в обоих справках совпадают;

**UsageBase[x]** – возвращает **Null**, т.е. ничего, активируя в текущем сеансе справочную базу, заданную *mx*-файлом данных *x*.

Процедура **UsageBase** допускает расширение своих функциональных возможностей, тогда как данный подход может успешно применяться при организации библиотек. Естественно, предложенный довольно простой способ организации справочной базы библиотеки пользователя далеко не единственный. Используя стандартные средства системы можно создавать подобные наряду с более развитыми средствами обеспечения пользовательских справочных баз данных. Однако это требует определенных усилий. Так, возможно создавать справочные базы на основе пакетов, сохраняемых в файлах *mx*-формата, оформляемых соответствующим образом, например, пакет со *справками* по средствам пользовательской библиотеки может иметь следующий вид, а именно:

```
In[2150]:= BeginPackage["HelpBase`"]
           G::usage = "Help on function G."
           V::usage = "Help on function V."
           S::usage = "Help on function S."
           Art::usage = "Help on procedure Art."
           Kr::usage = "Help on procedure Kr."
           EndPackage[]

Out[2150]= "HelpBase`"
Out[2151]= "Help on function G."
Out[2152]= "Help on function V."
Out[2153]= "Help on function S."
Out[2154]= "Help on procedure Art."
Out[2155]= "Help on procedure Kr."

In[2157]:= DumpSave["C:\\Math_myLib\\HelpBase.mx", "HelpBase`"]
Out[2157]= {"HelpBase`"}
```



Новый сеанс с системой Mathematica

=====

```
In[1947]:= Get["C:\\Math_myLib\\HelpBase.mx"]
```

```
In[1948]:= ?G
```

```
Help on function G
```

```
In[1949]:= ?V
```

```
Help on function V
```

```
In[1950]:= ?S
```

```
Help on function S
```

```
In[1951]:= ??Art
```

```
Help on procedure Art
```

```
In[1952]:= Information[Kr]
```

```
Help on procedure Kr
```

При этом, для всей библиотеки можно оформить только один справочный *mx*-файл, по мере необходимости загружая его в текущий сеанс по функции **Get** с получением в последующем доступа ко всей справочной информации, находящейся в файле, как наглядно иллюстрируют примеры предыдущего фрагмента. Более того, справочный файл рекомендуется размещать в том же каталоге, что и файлы с библиотекой.

Вполне определенный интерес организации справочной базы данных для библиотек пользователя может представить и следующая процедура **Usages**, чей исходный код с наиболее типичными примерами применения представлен следующим фрагментом. Данная процедура обеспечивает ведение *справочной* базы независимо от библиотеки, что достаточно удобно в целом ряде случаев организации программных средств.

```
In[2012]:= G::usage = "Help on function G."
```

```
V::usage = "Help on function V."
```

```
S::usage = "Help on function S."
```

```
Art::usage = "Help on procedure Art."
```

```
Kr::usage = "Help on procedure Kr."
```

```
In[2018]:= Usages[x_String, y___] := Module[{a, b, h = ""}, If[! FileExistsQ[x], Put[x];
    If[{y} == {} && ! EmptyFileQ[x], While[! SameQ[h, EndOfFile],
        Quiet[ToExpression[h = Read[x, Expression]]]; Close[x];
    If[{y} == {} && EmptyFileQ[x], $Failed,
    If[Quiet[Check[ListQ[y], False]] && {y} != {} && ListSymbolQ[y],
        a = DeleteDuplicates[Select[y, Head[#:usage] === String &]];
    If[a != {}, PutAppend[Sequences[Map[ToString[#] <> "::usage = "
        <> "\"\" <> #:usage <> "\"\" &, a]], x], $Failed],
    If[! Quiet[Check[ListQ[y], False]],
        b = DeleteDuplicates[Reverse[ReadList[x, Expression]]];
    Put[Sequences[Select[b, ! SuffPref[#1, Map[ToString[#] <>
        "::usage" &, Flatten[{y}]], 1] &]], x], $Failed]]]]
```

```
In[2019]:= Usages["C:\\Math_myLib\\HelpBase.m", {Art, Kr, G, V, Art, Kr, Vsv}]
In[2020]:= Usages["C:\\Math_myLib\\HelpBase.m", G, V]
In[2021]:= Usages["C:\\Math_myLib\\HelpBase.m", Art, Kr]
In[2022]:= Usages["C:\\Math_myLib\\HelpBase.m", {G, V}]
          Новый сеанс с системой Mathematica
          =====
In[17]:= Usages["C:\\Math_myLib\\HelpBase.m"]
In[18]:= ?? G
          Help on function G
In[19]:= Information[V]
          Help on function V
In[20]:= ?S
          Help on function S
```

Для начального наполнения справочной базы данных в текущем сеансе вычисляются все известные на данный момент справки (*usage*) по средствам, которые планируются для включения в библиотеку пользователя, как иллюстрирует *первый Input*-параграф предыдущего фрагмента. Затем вызовом **Usages**[*x*, *y*] обеспечивается сохранение в *x*-файле *ASCII*-формата всех справок, относящихся к программным средствам, которые определены списком *y*. При этом, сохранение производится в режиме дописывания в конец *x*-файла; если указанный файл *x* отсутствует, то создается пустой *x*-файл. В то время как вызов процедуры **Usages**[*x*, *y*, *z*, ...], когда в качестве аргументов, начиная со второго, выступают имена {*y*, *z*, ...} программных средств, удаляет из справочной базы справки по данным средствам. Наконец, вызов **Usages**[*x*] активирует в текущем сеансе все справки, содержащиеся в справочной базе *x*, делая их доступными безотносительно наличия библиотеки средств, описываемых этими справками. Более того, успешный вызов процедуры **Usages** возвращает *Null*, т.е. ничего, иначе возвращается **\$Failed**, в частности, в случае вызова **Usages**[*x*] при отсутствующем либо пустом файле *x*.

Как правило, справочные базы данных целесообразно размещать в тех же каталогах, в которых размещены и ассоциированные с ними библиотеки пользователя любой из возможных организаций, примеры которых представлены выше. Между тем, данные справочные базы совместно с их библиотеками для обеспечения простого доступа к ним возможно размещать в специальных каталогах файловой системы *Mathematica*. Типичными в этом отношении выступают подкаталоги, определяемые системными переменными **\$InstallationDirectory**, **\$UserBaseDirectory**, **\$Path** или вызовом **Directory[]**. С другой стороны, средства *Mathematica* позволяют создавать и размещать справочные базы данных в любом подкаталоге файловой системы компьютера, для чего требуется запрограммировать соответствующий алгоритм обработки данной справочной базы, базирующийся как на собственно самой задаче поиска требуемой справки, так и на средствах *Mathematica* для работы с файлами форматов {*m*, *mx*, *nb*, *txt*}. Между тем, на этом вопросе мы не останавливаемся, отсылая заинтересованного читателя к нашим книгам [98-101]. В целом же, созданию справочных баз для пользовательских библиотек следует уделять не меньше внимания, чем собственно самим библиотекам, ибо даже

весьма развитая библиотека без удобной и достаточно подробной справочной базы представляется не вполне приемлемой для использования пользователем, отличным от ее разработчика; такой программный продукт не может рассматриваться *массовым*.

Достаточно простая процедура **HelpPrint** обеспечивает вывод на печать содержимого справочной базы загруженного в текущий сеанс пакета пользователя, где 1-й записью выводится контекст, приписанный пакету. Фрагмент представляет как исходный код процедуры **HelpPrint**, так и пример ее использования. Вызов процедуры **HelpPrint[n]** выводит справки в строчном формате по всем средствам пользовательского пакета с соответствующим контекстом. Механизм использования процедуры довольно прост, а именно: в процессе работы с *Mathematica* в режиме документа на некотором шаге, в частности, **Out[n]** выполняется *загрузка* в текущий сеанс пакета, находящегося в файле формата {"cdf", "m", "mx", "nb"} либо перезагрузка загруженного пакета, после этого следует обеспечить вызов **HelpPrint[n+1]**, в результате которого на выходе получаем контекст, ассоциированный с пакетом наряду с упомянутой выше информацией.

```
In[1942]:= HelpPrint[n_Integer] := Module[{a = Out[n], b, k = n + 1}, Print[ToString[a]];
      While[k < Infinity, b = ToString[ToExpression["%" <> ToString[k]]];
      If[! StringFreeQ[b, a], Return["End of HelpBase on context " <> a], Print[b]]; k++]]
=====

In[2013]:= G[x_Integer, y_Integer, z_Integer] := Plus[x, y, z]
In[2014]:= G[42, 71, 2013]
Out[2014]= 2126



Активизация в текущем сеансе пакета из файла AVZ_Package_1.nb



In[2842]:= HelpPrint[2015]
      "AladjevProcedures`"
      "The call UprocQ[x] returns the False if x is not a procedure; otherwise,
      two-element list of the format {True, {"Module\" | \"Block\" |
      \"DynamicModule\"}} is returned."
      =====

Out[2842]= "End of HelpBase on context AladjevProcedures`"

In[2947]:= HelpBasePac[x_ /; ContextQ[x], y___] := Map[If[{y} != {}, Information[#],
      ToExpression["?" <> #]] &, NamesContext[x]]

In[2947]:= HelpBasePac["AladjevProcedures`", agn]


The call AcNb[] returns full name of the current document earlier saved as a nb-file.


AcNb[] := StringSplit[NotebookFileName[], {"\\", "/" }][[-1]]
=====
```

С другой стороны, вызов функции **HelpBasePac[w]** выводит содержимое справочной базы всех средств, снабженных справками (*usage*), пакета с контекстом *w* при условии его доступности в текущем сеансе. Тогда как вызов **HelpBasePac[w, p]**, *p* – произвольное выражение, дополнительно выводит определения всех таких средств пакета *w*. В ряде случаев функция может быть полезной для перлюстрации пользовательских пакетов.

Для получения справочной информации по средствам, которые находятся в пакетах, можно использовать процедуру **UsagesMNb**, чей исходный код наряду с типичными примерами использования, представлены нижеследующим фрагментом.

```
In[2167]:= UsagesMNb[x_;/FileExistsQ[x] && MemberQ[{"m", "nb"}, FileExtension[x]] :=
Module[{a, b, c}, If[FileExtension[x] == "m",
a = Select[ReadList[x, String], ! StringFreeQ[#, "::usage="] &];
a = Map[StringTake[#, {3, -3}] &, a];
a = Map[If[SymbolQ[StringTake[#, {1, Flatten[StringPosition[#,
 "::usage="]][[1]] - 1]], #] &, a]; Select[a, ! SameQ[#, Null] &],
c = "$.m"; b = ContextFromFile[x];
ToExpression["Save[" <> StrStr[c] <> ", " <> StrStr[b] <> ""];
b = Select[Quiet[ReadList["$m", Expression]], ! MemberQ[{Null, {Temporary}}, #] &];
DeleteFile["$m"]; b]]

In[2168]:= UsagesMNb["C:\\AVZ_Package\\AVZ_Package_1.mx"]
Out[2168]= UsagesMNb["C:\\AVZ_Package\\AVZ_Package_1.mx"]
In[2169]:= UsagesMNb["C:\\AVZ_Package\\AVZ_Package_1.m"]
Out[2169]= {"UprocQ::usage="The call UprocQ[x] returns the False if x is not a procedure;
otherwise, two-element list of the format {True, {"Module\"|\"Block\"|
\"DynamicModule\"}} is returned.\""}, "SymbolQ::usage="The call SymbolQ[x]
returns the True if x is a symbol; otherwise, the False is returned.\""}, ...
=====}

In[2170]:= UsagesMNb["C:\\AVZ_Package\\AVZ_Package_1.nb"]
Out[2170]= {"The call MdP[x] returns the simple 2-element list (if x defines a procedure or
function activated in the current session; in the absence of such object $Failed
is returned); the nested list, whose 2-element sublists have the structure,
described above (if x defines the list of procedures/functions activated in the
current session), the enclosed list of the above-mentioned format (if x is empty,
defining the list of all procedures/functions activated in the current session); in
the absence of the procedures/functions activated in the current session, the
call MdP[] returns the empty list, i.e. {}.\", ...
=====}
```

Вызов **UsagesMNb[x]** возвращает список справок (*usages*) по программным средствам пользовательского пакета, находящегося в файле *x* формата {"m", "nb"}. Эти справки возвращаются в строчном формате. При этом, для файла *x* *m*-формата возвращается список справок, содержащих префикс "Name::usage=", тогда как для файла данных *x* *nb*-формата возвращается список справок в строчном формате без данного префикса. Если для пакета из файла *x* *m*-формата не требуется его активации в текущем сеансе, то для пакета из файла *x* *nb*-формата данная активация требуется. В отличие от двух предыдущих процедур **HelpPrint**, **HelpBasePac** процедура **UsagesMNb** обеспечивает не только возможность перлюстрации справочных баз пакетов, но и их обработку.

Наш опыт разработки пользовательских средств широкого назначения рекомендует в качестве их оформления *пакеты*, содержащие *справочную* информацию и *определения* объектов (*процедуры, функции, переменные и др.*); эти пакеты сохраняются в мобильных файлах форматов *{m, nb}*, тогда как для загрузки используются бинарные файлы *mx*-формата с пакетами. Примером служит наш пакет *AVZ\_Package\_1* [120], эффективное использования которого в целом предполагает после загрузки в текущий сеанс файла *"AVZ\_Package\_1.mx"* с пакетом по функции **Get** выполнить следующий вызов:

**LoadMyPackage["AVZ\_Package\_1.mx", "AladjevProcedures`"]**

что в *значительной* степени не только повысит эффективность использования средств, находящихся в нем, но и в целях корректного выполнения некоторых его средств, чьи вызовы в режиме *Input*-параграфа корректны, тогда как в результате загрузки пакета пользователя нет. Причина этого рассмотрена в наших предыдущих книгах [98–101]. Более того, данный подход позволит получать по функции **Definition** определения из пакета в оптимальном формате без входящих в них контекстов, приписанных пакету. В целом же, для удобства загрузки пакета пользователя, расположенного в *mx*-файле *x*, в текущий сеанс может быть использована также процедура, вызов **LoadPackage[x]** которой возвращает *Null*, т.е. ничего, загружая пакет в текущий сеанс *Mathematica* с активизацией всех содержащихся в нем определений в режиме, аналогичном режиму *Input*-параграфа системы *Mathematica*. Следующий фрагмент представляет исходный код процедуры **LoadPackage** наряду с типичными примерами ее использования.

```
In[1]:= LoadPackage[x_;/ FileExistsQ[x] && FileExtension[x] == "mx"] := Module[{a,
                                     Quiet[ToExpression["Off[shdw::Symbol]"]; Get[x];
                                     a = ToExpression["Packages[[[1]]"]];
    ToExpression["LoadMyPackage[" <> "\"\" <> x <> "\"\" <> ", " <> "\"\" <> a <> "\"\" <> ""]];
    ToExpression["On[shdw::Symbol]"]]]

In[2]:= LoadPackage["C:\\AVZ_Package\\AVZ_Package_1.mx"]
In[3]:= Definition[StrStr]
Out[3]= StrStr[x_] := If[StringQ[x], "\"\" <> x <> "\"\"", ToString[x]]
```

Следует иметь ввиду, в случае загрузки описанным образом в текущий сеанс другого пакета пользователя требуется наличие в текущем сеансе или пакета *AVZ\_Package\_1*, или активизированной процедуры **LoadPackage**. Данное средство весьма удобно при необходимости обработки определений пакета в оптимальном формате.

Кратко рассмотрев на принципиальном уровне возможности системы *Mathematica* в плане предоставляемых ею средств по организации пользовательского программного обеспечения, попытаемся представить свою точку зрения на два достаточно важных аспекта. Однако, прежде сделаем одно весьма существенное замечание, поясняющее суть наших доводов. Система *Mathematica* *обеспечивает* возможность создания в своей среде приложений в виде документов, содержащих и *оформительские*, и *вычисляемые* элементы, т.е. данные документы можно сохранять в файлах формата, допускаемого системой, и затем, загружая в текущий сеанс и вычисляя, обеспечивать доступ ко *всем* содержащимся в них *определениям* в текущем сеансе. Как правило, к числу подобных

документов относятся решения достаточно сложных прикладных задач, которые не имеет смысла тесно ассоциировать с *системой* в связи с их востребованностью по мере необходимости, возникающей в целом ряде случаев достаточно эпизодически.

Однако совершенно иным образом обстоит дело со средствами, являющимися весьма часто используемыми многими пользователями, которые отсутствуют в пакете либо в процессе своего применения показали недостаточную эффективность, а в некоторых случаях содержат и ошибки. Наш опыт работы с обоими анализируемыми системами *Mathematica* и *Maple* подтверждает сказанное. В качестве примера возможно привести следующий. Несколько лет тому назад на форумах по *Mathematica* и *Maple* довольно активно обсуждалась возможность дифференцирования произвольного выражения не по переменной, а по некоторому подвыражению первого (*в частности, на форуме mapleprimes разработчиков системы Maple; [www.mapleprimes.com/search/?term=Diff](http://www.mapleprimes.com/search/?term=Diff)*). Так, для дифференцирования выражений системы *Maple* и *Mathematica* располагают стандартными процедурой *diff* и функцией *D* соответственно.

Очевидно, что представленные средства достаточно широко используются и в случае использования выражений (*отличных от обычных переменных*) в качестве переменных, по которым производится дифференцирование, указанные средства оказываются и вовсе беспомощными. Для устранения указанного недостатка были созданы простые процедуры, успешно решающие задачу дифференцирования выражения по любому из входящих в него подвыражений. Исходные коды данных средств с их типичными примерами применения представлены в нашей библиотеке по *Maple* [45] и пакете для системы *Mathematica* [90,120], а также и в настоящей книге.

В частности, приведен вариант процедуры *Df*, довольно существенно использующий процедуру *Subs*, рассмотренную выше и расширяющую возможности стандартной процедуры *subs*, которая имеет целый ряд недостатков; в этом отношении созданная нами процедура *Subs* свободна от ряда недостатков процедуры *subs*. Представленная выше процедура *Df* не столь функциональна, ввиду того, что она приведена с целью иллюстрации сказанного, тогда как выполняемая ею функция хорошо усматривается, исходя из ее текста. Заинтересованный читатель вполне сможет ее функционально расширить по своему собственному усмотрению, например, возможностью кратного дифференцирования по разным подвыражениям дифференцируемого выражения. В конце фрагмента представлены еще две реализации процедуры *Df*, расширяющие функцию *D* пакета и использующие *другие* подходы, полезные для ряда приложений. В отличие от *первого* варианта процедуры, вторые возвращают *упрощенные* результаты дифференцирования произвольных выражений.

Однако не только повышение эффективности пакетных средств либо расширения их функциональности обуславливают необходимость их доработки с замещением их на пользовательские. Так мы поступали в случае как с *Maple*, так и системой *Mathematica* посредством создания библиотеки для *Maple* [45] и пакетов для *Mathematica* [90,120]. К сожалению, далеко не всегда такой подход оказывается достаточно продуктивным, хотя и весьма необходимым для расширения программной среды обеих систем.

Итак, пользовательские средства для рассматриваемых 2 систем можно подразделить условно на две группы, а именно: (1) документы, описывающие алгоритмы довольно

сложных задач и являющиеся конкретными приложениями, и (2) часто используемые средства, чья сфера приложений достаточно обширна с реальной целесообразностью оформления их на уровне стандартных средств системы. Типичным примером такого подхода являются процедуры **Subs** и **Df** или некоторые их функциональные аналоги.

Естественно, оформлять такие средства можно довольно просто, сохраняя, например, в файлах *m*x-формата, однако здесь налицо явные неудобства, а именно: нужно знать имя процедуры и ее описание (*справку*), затем нужно загрузить файл с определением нужной процедуры в текущий сеанс, обеспечивая к ней доступ. Однако, если данных средств достаточно много, то подобное ручное манипулирование этими средствами достаточно трудоемкая и неблагодарная работа. Именно поэтому, создавая средства, представляющие достаточно большой интерес для широкого числа пользователей, и претендующих на статус стандартных, мы должны максимально подобным образом имплантировать такие средства в структуру системы, используя ее возможности. Этот механизм представлен довольно детально в справке по системе и вполне понятен для подавляющего большинства достаточно опытных пользователей *Mathematica*.

Говоря об организации пользовательских библиотек в системе *Mathematica*, следует упомянуть про достаточно развитый механизм *LibraryLink* подключения библиотек, обеспечивающий весьма эффективную как по реактивности, так и по использованию памяти возможность подключения внешних пользовательских библиотек. Механизм дает возможность загружать в ядро системы *динамические* библиотеки, что позволяет в случае необходимости оперативно вызывать требуемые функции из таких библиотек. Более того, данный механизм поддерживает обмен с подключенными библиотеками данными не только C-подобных форматов, но и форматов таких, как: целые числа и действительные числа, строки, упакованные массивы и, в целом, любые *Mathematica*-выражения наряду с отправкой *сообщений* с последующим возвратом в среду системы. Работа с механизмом *LibraryLink* поддерживается на уровне следующих функций:

**LibraryFunctionLoad**[*L*, *F*, *targs*, *rtype*] – загружает заданную своим именем функцию *F* из указанной *динамической* библиотеки *L* в ядро системы *Mathematica*, где аргумент *targs* определяет список допустимых типов аргументов вызываемой функции и *rtype* определяет тип возвращаемого функцией *F* результата; загруженная таким способом *F*-функция пользователя становится доступной в текущем сеансе системы;

**LibraryLoad**[*L*] – загружает в ядро системы *динамическую* библиотеку *L*, не активируя какую-либо из содержащихся в ней функций пользователя;

**LibraryFunctionInformation**[*F*] – возвращает справочную информацию по *F*-функции, загруженной в текущий сеанс системы вызовом функции **LibraryFunctionLoad**;

**LibraryFunctionUnload**[*F*] – выгружает из памяти компьютера *библиотечную* функцию *F*, после чего функция становится недоступной в текущем сеансе системы;

**LibraryUnload**[*L*] – выгружает *все* загруженные в текущий сеанс функции библиотеки *L*, после чего выгружается и вся библиотека *L* в целом;

**FindLibrary**[*L*] – отыскивает *динамическую* библиотеку *L*, доступную для загрузки в текущий сеанс системы вызовом функции **LibraryFunctionLoad**; при этом, поиск такой библиотеки производится согласно информации *глобальной* переменной *\$LibraryPath*, значение которой определяет *список* подкаталогов системы, в которых будет делаться

поиск заданной динамической библиотеки *L*. При необходимости расширить список переменной **\$LibraryPath** на требуемый подкаталог с динамическими библиотеками пользователя можно воспользоваться весьма простой процедурой **ModLibraryPath[x]**, вызов которой дописывает в конец списка, определяемого переменной **\$LibraryPath**, подкаталог *x* с пользовательскими динамическими библиотеками, возвращая **Null**, т. е. ничего, и сохраняя исходные атрибуты системной переменной **\$LibraryPath**.

```
In[4216]:= $LibraryPath
Out[4216]= {"C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\SystemFiles
           \\Autoload\\PacletManager\\LibraryResources\\Windows", .....,
           "C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\SystemFiles
           \\Links\\TetGenLink\\LibraryResources\\Windows"}
In[4217]:= Attributes[$LibraryPath]
Out[4217]= {ReadProtected}
In[4218]:= ModLibraryPath[x_ /; DirQ[x]] := Module[{a = $LibraryPath,
           b = Attributes[$LibraryPath]}, ClearAttributes[$LibraryPath, b];
           a = Insert[a, x, -1]; $LibraryPath = a; SetAttributes[$LibraryPath, b]]
In[4219]:= ModLibraryPath["C:\\Math_MyLib"]; $LibraryPath
Out[4219]= {"C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\SystemFiles
           \\Autoload\\PacletManager\\LibraryResources\\Windows", .....,
           "C:\\Program Files\\Wolfram Research\\Mathematica\\9.0\\SystemFiles
           \\Links\\TetGenLink\\LibraryResources\\Windows", "C:\\Math_MyLib"}
In[4220]:= Attributes[$LibraryPath]
Out[4220]= {ReadProtected}
```

По первому вызову библиотечной функции *W* по **LibraryFunctionLoad** вызывается ее инициализационная функция, настраивающая структуру для поддержки *вызываемой* функции; при этом, данная функция может использоваться для выполнения нужной инициализации, специфичной непосредственно для собственно самой библиотеки. Как только динамическая библиотека загружена, можно вызывать ее функции через вызовы **LibraryFunction**, возвращаемые **LibraryFunctionLoad**; при этом, можно также загружать и дополнительные функции пользователя из динамической библиотеки.

При отсутствии надобности в загруженной *динамической* библиотеке, она может быть выгружена либо выгрузкой по **LibraryFunctionUnload** всех ее загруженных функций, либо выгрузкой *динамической* библиотеки в целом по **LibraryUnload**; в данном случае вызывается специальная деинициализирующая функция. Между тем, нужно иметь в виду, что операция *выгрузки* динамических библиотек полностью не поддерживается на всех операционных платформах, а именно. При попытке выгрузить *динамическую* библиотеку в любом случае вызывается специальная деинициализирующая функция и все функции такой библиотеки будут недоступны в *Mathematica*. Между тем, лишь платформы, полностью поддерживающие выгрузку динамических библиотек, могут обеспечить *модификацию* библиотек с последующей их перезагрузкой в ядро системы. В настоящее время обеспечена полная поддержка выгрузки динамических библиотек



лишь на платформах *Windows* и *Mac OS X 10.5*. Детальнее с принципами *LibraryLink* подключения библиотек к *Mathematica* можно ознакомиться в документации.

Наряду с механизмом *LibraryLink* подключения библиотек система *Mathematica* имеет и другие механизмы связи с *внешними* по отношению к системе программами с целью обмена между ними структурированными данными. Так, в качестве такого развитого механизма выступает *MathLink*, поддерживаемый почти на *всех* базовых платформах, который обеспечивает стандарт связи, позволяющий реализовать высоко уровневую связь между *Mathematica* и внешними программами. Для возможности использовать механизм *MathLink* внешняя программа должна включать некоторый специальный исходный код и *MathLink*-библиотеку, обычно поставляемую с системой *Mathematica*. Детально с механизмом *MathLink* также можно ознакомиться в *справке* по системе. Но вопросы поддержки библиотек средствами системы нами здесь не рассматриваются.

И еще на одном очень немаловажном аспекте следует акцентировать ваше внимание. Система *Maple* является *намного более* открытой системой как ввиду ранее сказанного, так и по причине доступности для пользователя исходных текстов средств из главной библиотеки системы. Такая возможность весьма важна по двум причинам: (1) освоение программистской *парадигмы*, используемой системой, (2) возможность модификации средств системы с целью устранения возможных ошибок (*коих в системах немало*) или их расширения. Тогда как в случае системы *Mathematica* с подобными вопросами дело обстоит значительно сложнее. Для организации пользовательских библиотек нужно создавать собственные средства (*некоторые примеры приведены выше*), которые в полной мере нельзя считать интегрированными в систему. Тогда как на уровне документов 2 системы по возможностям создания средств пользователя, практически, *эквивалентны*. Более того, в системе *Mathematica* исходные коды ее стандартных средств скрыты от пользователя, возможно, по причине ну уж очень оригинальных и инновационных (*быть может, на уровне открытий*) используемых алгоритмов. Хотя в свете достаточно большого числа ошибок, с каждой новой версией сыплющихся как из рога изобилия, большая открытость системы позволила бы пользователям принять *посильное* участие в устранении обнаруженных ошибок, если у разработчиков до этого не доходят руки, хотя их устранение и должно было бы быть задачей более приоритетной, чем новые версии и новые средства, порой, и совершенно излишние для задач, на которые была изначально ориентирована система, точнее на область приложений *CAS*. В частности, в систему включены средства для *симуляции 1-мерных клеточных автоматов (однородных структур)*. Проблематика действительно весьма интересная и нашла свое отражение в *Mathematica*, на наш взгляд, из-за интереса к ней основателя *Wolfram Research*; между тем, во-первых, к системам компьютерной математики данная тематика имеет весьма относительное отношение, а во-вторых, на сегодня имеются действительно довольно развитые средства симуляции клеточных автоматов с развитыми функциями, посему включение в *Mathematica* средства симуляции простых клеточных автоматов особого смысла не имеет и не соответствует основным задачам *CAS*. В определенной мере это можно отнести к средствам музицирования и некоторым другим. А про попытки все охватить очень хорошо сказал *К. Прутков* словами «не объять необъятное». Между тем, и *Maple* грешит подобными моментами, в частности, включением в состав пакетного

модуля *Sockets*, декларируемого как средство для сетевой обработки, возможностями весьма ограниченного относительно средств, ориентированных на подобную работу.

Таким образом, с учетом сказанного, а также опыта использования обеих систем мы отдаем в этом вопросе предпочтение системе *Maple*, которая располагает удобными и эффективными средствами по организации *массовых* пользовательских программных средств, полностью интегрированных в функциональном и в справочном отношении в среду системы, предоставляя возможность идентично *именованным* средствам иметь различные уровни доступа в текущем сеансе. Разумеется, наш вывод в определенной степени носит сугубо субъективный характер и не претендует на истину в последней инстанции, однако он базируется на многолетнем опыте работы с обоими системами.

В то же время вопрос организации пользовательских библиотек в среде *Mathematica* в определенном отношении более multifunctional. Например, встроенный пакет *CCompilerDriver* позволяет работать с установленными *C*-компиляторами, давая возможность программировать *исполняемые* библиотеки и *объектные* файлы на основе исходных кодов, написанных на *C*-языке. Более детально с вопросами организации в среде *Mathematica* библиотек пользователя можно ознакомиться, например, в [116].

Сложность в общем случае является достаточно интуитивно-субъективным понятием и его исследование представляет собой весьма сложную фундаментальную проблему современного естествознания да и познания вообще. Именно поэтому использование его ниже носит сугубо интуитивный характер и будет основываться на субъективных представлениях читателя. За свою историю человечество создала значительное число довольно сложных проектов в самых различных областях, к числу которых с полным основанием можно отнести и современные вычислительные системы (*ВС*) с системой программного обеспечения, разработанного для них. Поэтому обеспечение высокого качества разрабатываемых довольно сложных программных проектов представляется нам не только чрезвычайно важной, но и весьма трудной задачей, во многом носящей *многоаспектный* характер. Поэтому именно по данной причине особенно в последние годы данному аспекту программной индустрии уделяется особое внимание.

Решение указанной задачи можно обеспечивать двумя основными путями, а именно: (1) исчерпывающее тестирование готового *программного средства (ПС)* с последующим устранением всех замеченных ошибок и оптимизация его по заданным критериям; и (2) обеспечение высокого качества на всех этапах разработки средства. Но так как для большинства достаточно сложных *ПС* первый подход не вполне приемлем, наиболее реальным является второй, при котором вся задача разбивается на отдельные *объекты (модули)*, имеющие хорошо обозримые структуру и функции, достаточно небольшие размеры и сложность, *структурно-функциональное* объединение (*композиция*) которых позволяет решать исходную задачу. При подобном модульном подходе *сложность ПС* сводится к значительно меньшей сложности составляющих его компонент, каждая из которых выполняет четкие функции, которые в сочетании с другими компонентами обеспечивают требуемое функционирование *ПС* в целом. Метод программирования, когда программа разбивается на группы модулей, каждый со своей контролируемой структурой, четкими функциями и хорошо определенным интерфейсом с внешней программной средой, называется *модульным программированием*.

Так как *модульный* является единственной альтернативой монолитного (*в виде единой программы*) подхода, то основной вопрос состоит не в целесообразности разбивать или нет большую программу на модули, но в том – каков должен быть критерий данного разбиения. Современная практика программирования знает и использует целый ряд методов организации многомодульных *ПС*, когда разбиение на модули основывается на их объемных характеристиках в строках исходного текста, выделении однотипных операций и т.д. Между тем, наиболее развитым представляется нам критерий, в чьей основе лежит хорошо известный принцип «*черного ящика*». Этот подход предполагает на стадии проектирования *ПС* представлять его в виде совокупности функционально связанных *модулей*, каждый из которых реализует одну из допустимых функций. При этом, сам способ взаимодействия модулей должен в максимально возможной степени скрывать принципы функционирования и организации отдельного модуля. В таком случае модульная организация ведет к выделению *модулей*, которые характеризуются легко *воспринимаемой* структурой и вполне могут проектироваться и разрабатываться различными проектировщиками и программистами. При этом, более важный аспект состоит в том, что многие *требуемые* модификации сводятся к изменению алгоритмов функционирования отдельных модулей без серьезного изменения *общей* структурно-функциональной организации *ПС* в целом. С рядом основных вопросов современной концепции модульного программирования можно ознакомиться, например, в книгах [1-3] и в цитируемой в них весьма обширной литературе различного назначения.

Технология модульного программирования охватывает *макроуровень* разработки *ПС* и позволяет решать важные задачи программной индустрии. В данном направлении одним из основных методов, обеспечивающих *модульность* программ, представляются механизмы *процедур* и *функций*, относительно *Math*-языка рассматриваемые нами в настоящей главе, и с которыми дополнительно можно ознакомиться как в книгах [97-101,104,107,109-115,117], так и в развитой справочной базе системы *Mathematica*.

Выделение в одной большой задаче *локальных* подзадач с довольно большой частотой использования и применимости позволяет не только существенно упростить вопрос с повышением ее *модульности*, но собственно повысить эффективность и прозрачность разрабатываемых программных средств. Наряду с этим модульный подход позволяет сделать доступными отдельно оформленные виды и типы часто используемых работ для многих приложений. В этом отношении достаточно развитый механизм *процедур* и *функций* системы *Mathematica* во многих отношениях отвечает данному решению.

Таким образом, система *Mathematica* в довольно значительных пределах располагает вполне определенными широкими возможностями по использованию *нестандартных* приемов программирования. Правда, вопрос лишь в том, насколько они оправданы в каждом конкретном случае и нельзя ли обойтись стандартной техникой. Более того, наряду с возможностью применения ряда *нестандартных Mathematica* предоставляет и целый ряд недокументированных средств, на которых акцентировать внимание не имеет смысла. Между тем, с рядом наиболее полезных недокументированных средств системы можно ознакомиться в наших книгах [98-101], некоторые из них приведены и в настоящей книге, а также использованы при программировании ряда средств из нашего пакета *AVZ\_Package\_1* [120], содержащего более 580 средств различного типа.

## 11.2. Программирование больших систем в среде *Mathematica*

Построение достаточно *больших* систем программного обеспечения в таких системах, как *Mathematica* и *Maple* предполагает следование *общим* принципам, используемым при построении, практически, любой большой системы программного обеспечения. Детали могут быть специфичны для программной среды (*Mathematica*, *Maple* и др.), однако многие из таких принципов являются общими. Между тем, имеются методы, для которых *Mathematica* и *Maple* являются наиболее подходящими. Пользователь в своих интересах должен знать о них как можно больше и использовать их. Принципы в определенной степени выведены из нашего опыта разработки достаточно больших проектов в программной среде обеих систем – среде языков *Mathematica* и *Maple*. И первая, и вторая системы довольно эффективно используют *императивную парадигму* программирования, типичную для наиболее массово используемых языков и систем программирования, в то время как первая эффективно использует и *функциональную парадигму*, весьма существенно расширяющую возможности ее программной среды. В концептуальном отношении обе парадигмы имеют весьма существенные отличия, а именно: *императивное* программирование – парадигма, в отличие от *декларативного* программирования, описывающая процесс вычисления в виде инструкций, которые изменяют состояние программы – это последовательность команд, которые должен выполнить компьютер. В этом отношении императивные языки программирования отличны от функциональных и логических языков программирования. В частности, функциональные языки не представляют собой последовательность инструкций и не имеют глобального состояния, тогда как *логические* языки программирования обычно определяют цель вычисления, а не сам способ его реализации. Между тем, именно на основе *функционального* программирования механизм *чистых* функций обеспечивает возможность высокоэффективного *распараллеливания* вычислений, что играет весьма важную роль при проектировании сложных проектов вычислительного характера.

Одним из основных и хорошо известных методов эффективной разработки больших программных систем является представление системы в виде *набора* взаимосвязанных компонент меньшего размера и, возможно, меньшей сложности. Такой набор можно параллельно программировать большим числом разработчиков. Между тем, в данном контексте на первый план выходит не сам процесс собственно программирования, но задача качественного проектирования архитектуры программируемой системы, когда максимально естественным путем выделяются составляющие систему компоненты с определением между ними строго определенного интерфейса, что дает возможность впоследствии с большей надежностью и с уменьшением временных затрат проводить программирование системы в целом наряду с разработкой документации по ней. Все преимущества такого подхода хорошо апробированы и не требуют здесь какого-либо обоснования. Наряду с удобством программирования многокомпонентных систем, их большей прозрачностью, возможностью параллельной работы большим коллективом программистов и другими важными преимуществами, компоненты таким способом организованных систем могут быть использованы и независимо в разработке целого ряда других систем, позволяя создавать некоторый инструментарий программиста в

той либо иной прикладной, либо системной области.

Между тем, разработка и создание многокомпонентных систем может производиться различными путями, из которых в качестве *классического* можно вполне полагать путь, состоящий из таких последовательных этапов, как проектирование системы в составе ряда *взаимосвязанных* подсистем с четко определенными между ними интерфейсами, четкая постановка заданий программистам различного уровня, программирование и отладка подсистем в независимом режиме, их документирование, с последующим их объединение в единую систему с тестированием и документированием. Естественно, все это в самых общих чертах без учета конкретных особенностей программируемой системы. Между тем, существуют и другие подходы к созданию достаточно больших программных систем. Типичным примером такого подхода является Библиотека [45] средств для программной среды системы *Maple*, которая содержит *более 850* процедур и модулей, и ориентирована на достаточно широкие приложения. Она создавалась в качестве приложения, ориентированного не только на решение целого ряда весьма широко используемых задач, но, прежде всего, с целью устранения недостатков либо расширения *функциональных* возможностей стандартных средств системы *Maple*. Эта библиотека создавалась на протяжении ряда лет, включая немало средств как общего, так и специального назначения, которые зарекомендовали себя в качестве достаточно хорошего *дополнения* к программной среде системы. Более того, немало из них были в дальнейшем включены в виде *аналогов* в последующие релизы системы. К сожалению, зачастую это были, порой, весьма запоздалые решения, а в целом ряде случаев имели место и существенно более ограниченные функциональные возможности.

Библиотека изначально проектировалась как набор средств, расширяющих средства *Maple* для решения широкого круга задач и, прежде всего, работы с файлами данных, для которых система располагает довольно *ограниченными* возможностями. В основу ее проектирования был положен наш опыт использования и разработки программных систем в других программных и операционных средах. Между тем, в процессе весьма детального апробирования системы выявился ряд довольно существенных ее ошибок и недоработок, для устранения которых и были созданы соответствующие средства в составе упомянутой Библиотеки [45]. Эти средства наряду со стандартными зачастую используют и средства самой Библиотеки, предполагая для получения полноценного доступа к библиотеке логически сцепить библиотеку с *главной* библиотекой системы.

Подобная методика расширения программной среды была предложена для системы *Mathematica* созданием пакета, содержащего более *580* процедур, функций, которые наряду со стандартными зачастую используют и средства самого пакета, предполагая для получения *полноценного* доступа к ним загрузку соответствующего файла *{mx,nb}*-формата в текущий сеанс или обеспечение контекстной связи с *m*-файлом. В основу проектирования положен наш опыт применения и разработки программных систем в других программных и операционных средах, а также в среде систем *Mathematica* и *Maple*, функционально однотипных. В частности, в процессе довольно интенсивного использования пакета для решения различных задач выявился целый ряд достаточно существенных ошибок и недоработок, для устранения которых и были нами созданы соответствующие средства в составе вышеупомянутого пакета *AVZ\_Package\_1* [120].

В любом случае вопросу разработки архитектуры проекта следует уделить внимания даже больше, чем собственно программированию. Наш опыт разработки достаточно сложных программных систем со всей очевидностью подтверждает факт, разработка качественной архитектуры, учитывающей все основные особенности системы, много важнее и сложнее, чем этап программирования. При наличии хорошей архитектуры с достаточно продуманной детализацией составляющих ее компонент уже довольно несложно программировать систему коллективом программистов различного уровня под руководством системных аналитика и программиста. Немало найдется довольно сносных программистов для программирования большинства компонент системы, да и исправлять их программы намного проще, чем изменять архитектуру системы. При этом, весьма значительную роль следует отвести разработке документации, удобной как для пользователя, так и для последующего *ведения* системы, например, в случае ее модификации под изменившиеся условия эксплуатации и новые версии системы.

И *Mathematica*, и *Maple* предоставляет ряд подходов для разработки больших систем программного обеспечения в своей среде. В частности, *Mathematica* для данной цели предлагает использовать организацию системы в виде набора пакетов (*m-файлов*) и/или документов (*nb-файлов*), загружаемых в текущий сеанс по мере надобности. Возможно также использование библиотек, однако они, как правило, применяются для хранения оптимизированных и хорошо отлаженных функций, чьи определения не так велики по размеру. Механизм *LibraryLink* позволяет динамическим библиотекам загружаться в ядро системы с обеспечением непосредственного доступа к ним. Имеется ряд других механизмов организации программной системы пользователя, включая и сочетания в довольно широких пределах всех вышеперечисленных библиотечных механизмов.

Со своей стороны, система *Maple* в качестве механизмов организации пользовательской программной системы предлагает документы (*{mws | mw}-файлы*), пакетные модули (*m-файлы*) и библиотеки. Однако, манипулировать с отдельными компонентами *Maple*-документа существенно менее удобно (*а в целом ряде случаев и вовсе невозможно*), чем с аналогичными конструкциями в среде системы *Mathematica*, обладающей довольно развитым инструментарием для этих целей. С целью облегчения программирования пакетов и повышения эффективности данного процесса *Mathematica* предоставляет специальное автоматизированное рабочее место (*APM - Workbench*), созданное именно для этих целей. Отметим, что *Maple* подобным инструментарием не располагает. При этом, в отличие от *Mathematica*, пакетные модули *Maple* располагаются в библиотеках, организационно подобных *главной* библиотеке системы. При этом, довольно большие проекты с развитой документацией в *Maple* предпочтительнее хранить в документах.

В среде *Mathematica* несколько выгоднее писать относительно небольшие и простые процедуры, используя простой и довольно развитый синтаксис *Math*-языка, который базируется на процедурно-функциональной парадигме. В данном контексте следует довольно грамотно использовать все преимущества программистского стиля, который поддерживается *Math*-языком процедурно-функционального типа. Между тем, не *все* интересные возможности пакета целесообразно использовать при программировании системы, в частности, механизм перезагрузки *BeginPackage* и *EndPackage*. Подобные средства не только ухудшают прозрачность системы, но и сужают ее применимость. В

принципе, аналогичное замечание справедливо для *Maple*-языка, довольно серьезно ограничивая применение целого ряда его средств, особенно при условии достаточно высокого уровня несовместимости релизов данной системы, как отмечено в [9-13,35].

Хорошо известно, что любая программная система, ориентированная на длительное использование, разработанная коллективом программистов, которая в процессе своей эксплуатации предполагает модификации, должна быть снабжена системой ведения своих исходных кодов. Среди такого типа систем можно отметить такие, как *Rational ClearCase*, *CVS* и *SVN*. Данные системы обеспечивают такие функции, как: хранение различных версий файлов с исходными кодами компонент системы, информация по причинам модификации конкретного файла, группы теггирования файлов простых версий, сравнение различных версий файлов и объединение модификаций. Системы ведения исходного кода очень важны для разработчиков. По данной причине система *Mathematica* в своем составе содержит клиент для *CVS* с последующим обновлением на клиент для *SVN*. Подобные средства ведения исходных кодов в *Maple* отсутствуют.

Вопрос подготовки разноуровневой качественной документации не менее важен, чем разработка архитектуры системы и ее программная реализация. Любая программная система и, в первую очередь, коммерческая предполагает хорошее документирование своих функциональных возможностей, предоставляемых пользователю. В противном случае даже хорошая система будет малоприменимой для использования в достаточно широком кругу пользователей с недостаточной подготовкой. В этом контексте система *Mathematica* обеспечивает систему документирования, работающую с приложениями и связанную с Центром документации пакета (через *Help*), поддерживая возможность создания документации в *HTML*-формате. Хорошо продуманная и разносторонняя документация для пользователя делает систему более профессиональной, облегчая ее использование даже пользователю, не совсем хорошо знакомому с системой. Наряду с документацией пользователя целесообразно создавать и документацию разработчика системы, оформляемую в различных формах (комментарии в исходных кодах, описания алгоритмов, история модификаций и т.д.). Для пользователя, создающего свои системы в *Mathematica*, система предоставляет ряд полезных средств (документ, рабочее место). В то же самое время рабочая документация собственно по самой системе пользователю недоступна даже в плане ознакомления с кодами всех его функций. Тогда как *Maple* в данном отношении является намного более открытой системой, обеспечивая доступ к исходным кодам подавляющего большинства своих функциональных средств, кроме встроенных функций и предложений *Maple*-языка. Такая организация программной среды системы обеспечивает наряду с ее справочной базой не только более детальную информацию по ее функциональным возможностям, но и предоставляет достаточно хорошую возможность ознакомления с методикой программирования, используемой системой, что весьма важно осваивающему программирование в ее среде. Между тем, с другой стороны, справочная база *Mathematica* представляется нам более развитой.

Вопрос тестирования программной среды составляет одну из важнейших компонент в ее разработке, существенно определяя ее последующую робастность. Так как сложная система, как правило, имеет многокомпонентную архитектуру, тестирование следует проводить на двух уровнях: (1) блочный уровень и (2) системный уровень. На блочном

уровне производится тестирование отдельных блоков системы, тогда как системный предполагает тестирование системы в целом. Более того, предполагается, что процесс эксплуатации системы достаточно длителен и в течение его вопросом тестирования и исправления замеченных ошибок и недостатков кроме службы сопровождения будет заниматься достаточно большая армия пользователей системы. Вот здесь то и должна быть налажена четкая *обратная* связь с пользователями, разнопланово тестирующими систему. К большому сожалению, такая обратная связь не прослеживается для обеих систем, а многочисленные ошибки, зачастую, не устранимые из версии в версию, не делают чести разработчикам, в целом, достаточно полезных средств. Между тем, обе системы предоставляют довольно развитые средства не только для отладки программ пользователя, но и для обеспечения их средствами, которые позволяют пользователю обеспечивать довольно высокий уровень их *робастности*. Некоторые из них отмечены в данной книге, другие с различной степенью детализации представлены в [44,45,97].

В завершение еще раз акцентируем внимание на таком полезном средстве создания в среде *Mathematica* программных средств, как *автоматизированное рабочее место (АРМ)* пакета, представляющее собой интегрированную среду разработки для *Mathematica*. *АРМ* базируется на широко используемой платформе *IDE* по имени *Eclipse*; при этом, *АРМ* можно использовать непосредственно или можно установить *Mathematica* инструменты в *Eclipse*. *АРМ* обеспечивает достаточно полезный инструментарий для разработки больших приложений, включая *отладчик*, *профайлер* и *блочный отладчик*, и поддерживает проекты, которые содержат различные типы ресурсов (*коды*, *Java классы*, *nb-документы* и *документация*). Более того, *АРМ* содержит специальный редактор для кодов *Mathematica* и располагает большой информацией как о пакетах *Mathematica*, так и о ее приложениях, что существенно облегчает работу над большим проектом. В дополнение к этому *АРМ* поддерживает интеграцию с *CVS* и может быть достаточно легко настроен на работу с рядом других систем ведения исходных кодов. Более того, *АРМ* обеспечивает поддержку разработок в среде других языков типа *Java*, *Python*, *C* и *C++*, позволяя интегрировать их в среду *Mathematica*. В качестве последней важной особенности *АРМ* является появление инструментария для *параллельных* вычислений. В качестве некоторых особенностей, помогающих при разработке больших проектов в программной среде системы *Mathematica*, отметим лишь некоторые, а именно:

- *Организация проекта*; требуемая работа собирается в проекты, каждый из которых содержит различные типы ресурсов (*коды*, *Java классы*, *nb-документы*, *документацию*). В отличие от ряда других систем проектирования работа с проектами в *АРМ* удобнее и проще, снабжая пользователя специальными редакторами для открытых документов, специализированного поиска и составления отчетов для различных компонент. Более того, обеспечивается тесная интеграция с графическим интерфейсом *Mathematica*;
- *Запуск и выполнение программного кода*; *АРМ* располагает развитым интерфейсом для запуска и управления выполнением кода; обеспечивается инициализация пакета под конкретный проект с сохранением необходимой информации в случае разбиения проекта на подпроекты; *АРМ* обеспечивает достаточно удобное переключение между разными версиями пользовательского пакета;
- *Отладчик и профайлер*; обеспечивается интерфейс с отладчиком и профайлером; в



одно и то же время можно вести отладку *Mathematica* и *Java* кодов;

- **Блочный отладчик**; обеспечивает интерфейс с блочным отладчиком *MUnit* наряду с поддержкой клиента к *CVS*, расширяется на многие другие виды управления кодом;
- **Ошибки и сообщения**; обеспечивает нахождение и отчет по многим классам ошибок во всех файлах с исходным кодом проекта независимо от их режима открытия, что во многом облегчает отладку проекта, содержащего много файлов программ и данных.

Наряду с перечисленными *Mathematica* предоставляет еще немало возможностей для успешной разработки больших проектов, однако зачастую эти возможности в полной мере пользователями не используются. В частности, *Mathematica* содержит целый ряд инструментов визуализации, часть которых носит специальный характер, например, вычерчивание графов. Все они могут быть весьма полезны для получения достаточно наглядной картины выполнения приложения, в то время как интерактивный режим обеспечивает чрезвычайно удобный метод *блочной* отладки исходных кодов проекта.

Даже не взирая на возможности современных компьютеров, проблема оптимального использования их основных ресурсов все еще весьма актуальна. И, в первую очередь, это касается *оперативной* памяти, чьи размеры играют во многих случаях решающую роль, особенно при выполнении символьных вычислений, требующих памяти весьма большого объема. Если при длительных числовых вычислениях еще можно каким-то образом смириться с большими *временными* издержками, то недостаток оперативной памяти может попросту не дать возможность выполнить символьные вычисления. В этом отношении система *Maple* достаточно неустойчива по данному показателю, что в значительной степени ограничивает его возможности в решении целого ряда задач, связанных даже с относительно несложными *символьными* вычислениями. И причина этого кроется именно в недостаточно эффективных механизмах управления памятью, используемых системой. Наш опыт довольно *разностороннего* использования системы *Maple* версий 4 ÷ 11 со всей определенностью подтверждает сказанное. Более того, для решения ряда задач символьной обработки нам пришлось увеличивать оперативную память, что позволило существенно расширить круг решаемых такого типа задач без какой-либо переделки их исходных кодов. Средствами же оптимизации используемой текущим сеансом оперативной памяти система *Maple* не располагает.

С другой стороны, система *Mathematica* достаточно эффективно управляет памятью, на каждом шаге собирая "*муссор*" и освобождая от нее память. Функция **MemoryInUse** возвращает количество байтов, используемых в текущий момент ядром *Mathematica*, тогда как функция **Share** пытается *минимизировать* объем памяти, занимаемой всеми выражениями текущего сеанса. Следующий простой фрагмент достаточно наглядно иллюстрирует результат совместного использования обоих упомянутых функций.

```
In[3393]:= MemoryInUse[]
Out[3393]= 22103428

In[3394]:= V := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^2 + Sin[x]*Cos[y]];
Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &; Fn := Compile[{x}, x^2 + Sin[x]]
Sv[x_ /; StringQ[x], y_] := Module[{a = StringLength[x], b = 78}, (x^2 + y)/(a + b)]
```

```

In[3395]:= MemoryInUse[]
Out[3395]= 27947812
In[3396]:= Share[]
Out[3396]= 5745216
In[3397]:= MemoryInUse[]
Out[3397]= 22204404
In[3362]:= M[x_/, SameQ[x, "avz"], y_] := Module[{a, b, c}, y]; M[x_, y_, z_] := x + y + z;
          M[x_/, x == "avz"] := Module[{a, b, c}, x]; L[x_] := x
          M[x_/, IntegerQ[x], y_String] := Module[{a, b, c}, x];
          M[x_, y_] := Module[{a, b, c}, "agn"; x + y]; M[x_String] := x;
          M[x_/, ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y];
In[3363]:= Headings[M]
Out[3363]= {"Block", "M[x_/, ListQ[x], y_]"}, {"Function", "M[x_, y_, z_]", "M[x_String]"},
          {"Module", "M[x_/, x == \"avz\", y_]\", "M[x_/, x == \"avz\"\",
          "M[x_/, IntegerQ[x], y_String]\", "M[x_, y_]"}
In[3364]:= MemoryInUse[]
Out[3364]= 25349092
In[3365]:= Remove[M, V]; Share[]
Out[3365]= 1165784

```

В *Maple* вызов функции *gc()* делает попытку собрать в текущем сеансе ‘муссор’, однако опыт показывает, что во многих важных ситуациях *реального* освобождения памяти не производится. Использование с этой целью механизма *Java OpenMaple* также не дает ощутимого результата. Итак, по таким важным критериям как *память* и *реактивность* *Mathematica* более предпочтительна, особенно при разработке достаточно сложных проектов. Мы располагаем рядом примеров сложных процедур, *запрограммированных* идентичными алгоритмами, которые по требуемому объему памяти выполняются в *Mathematica*, но не выполняются в среде *Maple* на одном и том же компьютере. Итак, довольно эффективный механизм оптимизации используемой оперативной памяти обеспечивает *Mathematica* существенные преимущества при разработке проектов.

Исходя из вышесказанного, наряду с нашим опытом разработки достаточно больших и сложных проектов в среде обеих систем, большее предпочтение для этих целей мы отдаем системе *Mathematica*, которая для целого ряда *классов* задач располагает более развитым набором функциональных средств, включая средства, облегчающие работу по созданию программных систем и средств как прикладного, так и системного типа, наряду с существенно большим *продолжением* *Math*-языка, подверженным весьма *малым* изменениям при переходе от одной версии системы к другой. Тогда как в случае с системой *Maple* довольно частые модификации входного языка создают, зачастую, весьма существенные затруднения при разработке *продолженного* кода, требуя его последующей модификации и снижая по данному показателю уровень робастности проектов. Прежде всего, это относится к проектам, существенно использующим ряд специфических средств, подверженных модификациям с ростом номеров релизов. В целом, можно констатировать – *Maple* предпочтительнее для задач обучения *СКМ*, в то время как *Mathematica*, прежде всего, для разработки довольно сложных проектов.

### 11.3. Maple или Mathematica – субъективная точка зрения

**Модульное программирование** – наиболее очевидная и очень широко используемая в технологии программирования парадигма. Модульное программирование базируется на понятии *модуля* – логически взаимосвязанного *набора* функциональных элементов, оформленных в виде отдельных программных *модулей*, в качестве которых мы будем рассматривать *процедуры* и *функции*. В процессе проектирования модули возникают естественным образом в процессе разбиения решаемой задачи на подзадачи, или при выделении фрагментов некоторой задачи в отдельную подзадачу. С другой стороны, возникают относительно небольшие задачи массового характера как системного, так и прикладного характера, которые вполне естественно оформлять в форме процедур либо функций и разместить в отдельных библиотеках, обеспечивая доступ к ним при необходимости. К процедурам/функциям выдвигаются следующие требования:

- **логическая завершенность**: процедура/функция должна реализовывать логически законченный алгоритм решения некоторой конкретной задачи;
- **ограниченность**: процедура/функция должна быть ограничена в размерах, иначе ее следует разбить на логически завершенные части, вызывающие друг друга;
- **замкнутость**: процедура/функция по возможности не должна использовать какие-либо *глобальные* данные, имея связь с *внешней* средой через программный интерфейс;
- **универсальность**: процедура/функция должна быть универсальной, а параметры ее процесса обработки и сами данные для обработки должны передаваться извне;
- **принцип «черного ящика»**: процедура/функция должна иметь хорошо продуманный программный интерфейс, определяемый набором формальных аргументов наряду с возвращаемым ею результатом либо выполненным действием, через который модуль подключается к другим программным средствам в точке своего вызова.

Каждая процедура/функция состоит из *спецификации (заголовка)* и *тела*. Спецификация определяет интерфейсные правила при вызове процедуры/функции, а *тело* – способ реализации решения задачи в виде алгоритма, написанного на том либо ином языке программирования. При этом, принципы модульного программирования во многом сходны с принципами нисходящего проектирования. Сначала определяются состав и подчиненность функций, затем – *набор* программных модулей, реализующих данные функции. Более того, состав и вид программных модулей, их назначение и характер использования в программной среде в довольно значительной степени определяются имеющимися в распоряжении программиста инструментальными средствами. Так, в нашем случае относительно рассматриваемых пакетов в данном качестве могут быть:

- средства встроенного языка программирования (*Math-язык* либо *Maple-язык*);
- библиотеки программных средств различного назначения и организации;
- стандартные процедуры вычислений и обработки информации различного типа;
- стандартные средства отладки программных средств пользователя;
- пользовательские средства различного назначения и организации, и др.

Выделение *отдельной* процедуры/функции и ее программирование следует начинать с четкого представления алгоритма решаемой задачи, включая в него максимальную

по возможности обработку особых и ошибочных ситуаций, возникающих в процессе ее выполнения, определения удобного интерфейса с внешней программной средой – кортеж формальных аргументов с возможностью тестирования соответствующих им фактических аргументов при *вызове* наряду с набором возвращаемых результатов как основного, так и дополнительных. Не меньшую важность имеет и документирование создаваемой процедуры/функции, что во многом обеспечивает ее жизнеспособность. *Документирование* необходимо как: (1) средство передачи информации между самими разработчиками, (2) средство управления разработкой, а также (3) средство передачи пользователям информации, необходимой для применения и сопровождения. Более того, *документирование* разрабатываемых программных средств весьма целесообразно не выделять в отдельный этап, а проводить в течение всего процесса их разработки.

Именно в контексте модульного программирования и рассматриваются нами пакеты *Mathematica* и *Maple*, на сегодня являющиеся несомненными лидерами в классе *CAS*. Однако перед переходом к их сопоставлению именно в качестве среды для *модульного* программирования все же еще раз вкратце дадим свое видение на оба пакета в целом.

Являясь в целом *лидерами* в своем классе, обе системы попеременно выходят в лидеры по той либо иной группе составляющих их функций, зачастую беря пример со своего конкурента. Между тем, даже на сегодня обе системы имеют свои *коньки*, по которым они превосходят конкурента. Например, если система *Maple* располагает лучшими на сегодня средствами для аналитического решения диффуравнений, то *Mathematica* в определенной мере превосходит *первую* в задачах интегрального исчисления. В то же время обе системы просто нашпигованы различного рода ошибками, недоработками и другими неприятными моментами, пожалуй, в несколько большем количестве, чем *ОС Microsoft* и *Linux* вместе взятые, и в намного большем, чем следовало бы ожидать от современной системы компьютерной математики. Как правило, появляющиеся все новые версии обеих систем не только не устраняют большинство застарелых ошибок, но добавляют немало новых, вызывающих много нареканий у пользователя. Именно по этой причине пользователь вынужден *своими* силами устранять некоторые из них. Однако это при условии, что он располагает достаточной квалификацией, которой в большинстве случаев не хватает. Так, в случае с системой *Maple* наша Библиотека [45] преследовала, в первую очередь, именно такие цели, предоставив целый ряд средств как устраняющих ошибки системы *Maple*, так и расширяющих набор ее стандартных средств или их функциональность. Нечто в подобном роде можно сказать и о наших пакетах *AVZ\_Package* [90] и *AVZ\_Package\_1* [120] для системы *Mathematica*, которые в значительной степени преследуют цель расширения набора ее стандартных средств или их функциональность наряду с иллюстрацией целого ряда приемов и подходов к разработке приложений различного характера в программной среде *Mathematica*. При этом, пользователь указанных пакетов сможет найти в них целый ряд полезных программных средств для разработки своих собственных приложений.

В этом контексте представляется достаточно интересным и проект по сбору ошибок в системе *Maple*, которыми она изобилует подобно *Mathematica* [98,99]. Неплохо было бы адаптировать его также на систему *Mathematica*. Из нашего опыта использования *Mathematica* можно ожидать такую же картину. Между тем, подобные базы ошибок

имеют смысл, а сам проект будет вполне жизнеспособным лишь в том случае, если он и полученные им массивы ошибок будут адекватно восприняты разработчиками этих систем. В противном же случае подобные базы ошибок наряду с многочисленными ошибками, анонсированными на соответствующих форумах, так и останутся гласом вопиющего в пустыне, не оказывая какого-либо влияния на работу по их устранению в последующих релизах указанных средств. К сожалению, пока отмеченная картина представляется нам именно таковой. Тогда как наш опыт сотрудничества с *MapleSoft* по подготовке и написанию серии книг по системе *Maple*, выявлению ряда довольно принципиальных ошибок, устранению их разработанными нами средствами наряду с расширением набора ее стандартных средств либо их функциональности, большая часть которых помещена в нашу библиотеку [45], не оказали сколь-нибудь серьезного влияния на активность разработчиков системы по устранению *обнаруженных* ошибок, зачастую, переходящих из версии в версию; между тем, как вместо такой важнейшей работы по доведению до *кондиции* функционирующего средства с завидной частотой появляются все новые его версии как с дополнительными новыми ошибками, так и с немалым уровнем несовместимости между версиями. В частности, на наш взгляд, не совсем обоснованным является использование разных текстовых сообщений, которые иницируются возникающими одними и теми же ошибочными ситуациями, что не позволяет корректно их обрабатывать уже созданными программными средствами в среде *Maple* предыдущих версий, т.е. нарушается один из краеугольных принципов качественного программного обеспечения – совместимость «*снизу-вверх*». Примеры такого рода приводились в наших книгах [42-45], требуя перепрограммирования ряда средств из библиотеки [45], разработанных в предыдущих версиях системы *Maple*. Но в принципе, фирмы-разработчики помимо отделений маркетинга должны включать и серьезные отделения тестирования (*типа ОТК*) своей продукции, которые на основе результата своей работы совместно с откликами пользователей должны осуществлять четкую *обратную связь* с непосредственными разработчиками программного продукта с целью устранения выявленных недостатков. К сожалению, или данная служба вовсе отсутствует, или носит в значительной степени *формальный* характер, ибо результаты ее деятельности *особо* не прослеживаются, судя по *новым* версиям систем со *старыми* ошибкам, которые не имеют какого-либо разумного объяснения и оправдания.

Между тем, можно сколь угодно критиковать *обе* системы за их весьма *многочисленные* ошибки и недоработки, однако на сегодня они являются несомненными лидерами в своем классе *ПС*, играя довольно существенную роль при решении широкого спектра задач, прежде всего математического характера, с использованием компьютеров. Нам остается лишь надеяться, что работа службы сопровождения обеих систем будет все же налажена должным образом и будет серьезно учитывать в своей работе *рекламации* пользователей и соответствующим образом доводить *обе* системы «до ума». На сегодня же мы видим только гонку версий систем, обусловленную не столько настоятельной необходимостью, сколько сугубо коммерчески-конкурентными соображениями.

Немалую роль в степени продвижения обеих систем на рынке программных средств помимо их возможностей в той или иной области играет служба маркетинга систем. Так, служба маркетинга *Mathematica* работает намного агрессивнее и эффективнее,

чем аналогичная служба системы *Maple*. Судя же по качеству обеих систем у *первой* в данной службе работает сотрудников, не менее, чем разработчиков, и квалификация их повыше, и платят им поболее. Неодинаково и отношение разработчиков систем к многочисленным пользователям; так, если сетевая конференция для системы *Maple comp.soft-sys.math.maple* не модерируется, в то время как аналогичная конференция *comp.soft-sys.math.mathematica* не только модерируется, но также и цензурируется в том отношении, что фильтруются посты, с относительно резкой критикой системы. В плане поддержки своих пользователей от обеих систем следовало бы ожидать много большего, между тем, многочисленные пользователи *Maple* получают превосходные помощь и поддержку от большого числа пользователей системы на вышеупомянутой конференции и специальных форумах пользователей. Между тем, сравнивая число постов по *обоим* системам на обеих конференциях и на ряде других форумов по этим системам, с *большой* долей уверенности можно сказать, что пользователи *Mathematica* имеют от нее намного больше неприятностей, чем пользователи системы *Maple*. Более того, из нашего личного опыта можно сделать вывод, что система *Maple* *намного* более дружелюбна к своим пользователям, чем система *Mathematica*. Между тем, не взирая на многочисленную критику в адрес обеих систем, они на сегодня – лидеры в классе *систем компьютерной алгебры* и даже в несколько более широком понимании – *систем компьютерной математики*. Именно поэтому вполне резонно сравнить программные среды обеих систем на предмет их предпочтительности при разработке тех или иных пользовательских средств. Как показывает наш опыт использования данных систем в различных приложениях, практически, любая задача решается в среде любой из них с той или иной степенью эффективности. Именно в контексте использования данных систем в качестве среды программирования наиболее массовых задач и, прежде всего, оформляемых в виде процедур и функций, и лежит круг интересов данного раздела.

В качестве рассматриваемых *объектов*, программируемых средствами языков *Maple* и *Math* систем *Maple* и *Mathematica* соответственно, рассматриваются как процедуры, так и функции, принцип организации которых с достаточной степенью детализации рассмотрен выше. Между тем, целесообразно на общем уровне напомнить структуру рассматриваемых программных *модульных* объектов, реализующих некоторую задачу пользователя, а именно: *функции*, возвращающей только один результат, и *процедуры*, возвращающей один или более результатов, или выполняющей некоторые действия.

*Заголовок* процедуры/функции определяет не только ее формальные аргументы, но и интерфейс с внешней по отношению к ней программной средой. Именно на данных аргументах строится *алгоритм* вычисления, написанный на языке программирования и составляющий тело процедуры/функции. Тогда как реализация собственно самого интерфейса происходит в точке вызова процедуры/функции передачей фактических аргументов, соответствующих ее *формальным* аргументам. При этом, вызов полагается *корректным*, если типы *фактических* и *формальных* аргументов являются *совместимыми* по присваиванию; в противном случае инициируется ошибочная ситуация. Поэтому, данному вопросу уделяется особое внимание при разработке процедур/функций. В обеих системах для этой цели используются механизмы *типирования*, рассмотренные с достаточной степенью детализации в наших предыдущих книгах [98–101].

Система *Maple* поддерживает довольно развитый механизм *типизации* объектов, для которого характерно наличие весьма развитого набора встроенных как простых, так и составных *типов*, распознаваемых системой, наряду с двумя механизмами, которые поддерживают создание *пользовательских* типов, также распознаваемых системой. Но здесь имеются и недостатки такого механизма тестирования *фактических* аргументов, получаемых при вызове процедуры/функции, на предмет их соответствия заданным типам *формальных* аргументов. Прежде всего, используемый тип должен быть заранее определен путем вычисления его определения в текущем сеансе или предварительно сохранен в библиотеке, логически сцепленной с *главной* библиотекой системы *Maple*. В то же время имеется немало ситуаций, когда нецелесообразно отдельно определять *тип* (а в более широком смысле *тест*) с сохранением его вне процедуры, использующей его. Данная возможность системой *Maple* не предусмотрена, тогда как *Mathematica* в этом отношении имеет довольно серьезные преимущества, поддерживая механизмы не только *отдельного* определения типов объектов, распознаваемых системой, но дает возможность определения типов формальных аргументов непосредственно вместе с определением заголовка процедуры/функции. Итак, возможность задания тестов для определения допустимости получаемых при вызове фактических аргументов прямо в заголовке *Mathematica*-процедур имеет немаловажное преимущество перед *Maple*-процедурами, для которых в качестве теста можно указывать только оформленный в пакете *тип*, требуя при необходимости использования пользовательского определять его отдельным средством. Более того, во втором случае налицо некоторое ухудшение *мобильности* процедур, когда дополнительно к самой процедуре требуется указывать и используемый ею тип, который, вообще говоря, может иметь весьма ограниченную применимость, т.е. его актуальность непосредственно связана с данной процедурой.

Большим подспорьем в вопросе определения тестов в заголовках процедур/функций выступает функциональная составляющая *Math*-языка, поддерживающая достаточно эффективный механизм программирования чистых функций. Естественно, в системе *Maple* тоже допустимо использование элементов *функционального* программирования, однако в существенно более ограниченных пределах, к тому же это требует, порой, и немалых усилий со стороны программиста. Более того, в *Mathematica* в качестве теста на допустимость фактического аргумента можно использовать произвольную булеву функцию, т.е. функцию, возвращающую на фактических аргументах значение *False/True*, как это иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[2595]:= GS[x_Integer /; If[x > 78 | | x <= 0, Print["Actual argument <" <> ToString[x] <>
"> is inadmissible"]; False, True]] := Module[{}, x^2]
In[2596]:= {GS[17], GS[24]}
Out[2596]= {289, 576}
In[2597]:= GS[460]
Actual argument <460> is inadmissible
Out[2597]= GS[460]
```

С другой стороны, если *несовместимость* типов *формальных* и *фактических* аргументов при вызове процедуры/функции в среде *Maple* инициирует ошибочную ситуацию, допускающую несложную программную обработку, то в среде *Mathematica* подобная

ситуация возвращает вызов процедуры/функции невычисленным, что в ряде случаев не столь очевидно для программной обработки, как это иллюстрируют последние *два* примера предыдущего фрагмента, требуя нестандартных подходов. Выше в качестве иллюстрационного примера были представлены средства для этой цели; представлен целый ряд примеров процедур и функций, реализованных с учетом вышесказанного в среде обеих систем. Они достаточно наглядно иллюстрируют возможности обеих программных сред в вопросе создания средств, тестирующих *фактические* аргументы, получаемые в точках вызова процедур/функций, на предмет их допустимости. Выше и в книгах [98–101] нами приведен целый ряд приемов задания тестирующих средств в программной среде системы *Mathematica* в заголовках блоков, функций и модулей.

Локальными переменными процедур называются переменные, которые описываются в блоках *деклараций* функций или процедур. Их значения не доступны и невидимы вне области самих процедур, однако они играют чрезвычайно важную роль в описаниях реализуемых процедурами алгоритмов. Отметим, механизм управления начальными значениями или значениями по *умолчанию* для аргументов и локальных переменных процедур представляется нам более развитым именно в среде *Mathematica*; так, если в *Maple* локальным переменным начальные значения не присваиваемы, тогда как их типирование не имеет никакого смысла, как наглядно иллюстрирует пример:

```
> GS := proc(x::integer, y::integer) local a; a := 42.47; a*(x + y) end proc;
> GS(42, 47);      => 3779.83
> GS := proc(x::integer, y::integer) local a::integer; a := 42.47; a*(x + y) end proc;
> GS(42, 47);      => 3779.83
```

то в случае системы *Mathematica* локальные переменные в качестве начальных значений могут получать любые допустимые выражения, что в целом ряде случаев упрощает и само описание реализуемого процедурой алгоритма. Немало примеров данного типа было представлено в настоящей книге, здесь же мы лишь представим весьма простой фрагмент, иллюстрирующий как сказанное, так и некоторые особенности механизма начальных значений для локальных переменных в среде *Math*-языка, а именно:

```
In[3533]:= Art[x_/: {G[y_] := y^2; If[x + 2013 <= G[x], True, False]}[[1]] := x^2
In[3534]:= Art[78]
Out[3534]= 6084
In[3535]:= Kr[x_/: If[x > 78 || x <= 0, False, True]] := Module[{a}, x^2 + a + b
In[3536] := {a, b} = {17, 24}; Kr[42]
Out[3536]= 1788 + a$11271
In[3537]:= ArtKr[x_/: If[x > 72 || x <= 0, False, True]] := Module[{a = 460}, x^2+a+b]
In[3538] := {a, b} = {17, 24}; Art[78]
Out[3538]= 6084
```

Данные особенности не единственные и вполне применимы для некоторых довольно изощренных приемов программирования процедур, посему читателю рекомендуется проработать их отдельно. Как известно, наряду с локальными процедуры допускают использование и *глобальных* переменных, которые доступны как внутри процедур, так



и изне их. При этом, если в *Maple*-процедурах в их описательной секции глобальные переменные декларируются явно, то в *Mathematica*-процедурах в качестве *глобальных* выступают все переменные, не объявленные в процедуре локальными. Тогда как оба пакета допускают использованием единого механизма *синхронизации* использования *глобальных* переменных процедур с одноименными переменными вне процедур. Этот несложный механизм описан на конкретных примерах в настоящей книге выше. При этом, выше нами представлены расширенные механизмы работы как с формальными аргументами процедур/функций, так и с локальными переменными функций. Более того, в программной среде *Maple* подобные полезные механизмы недопустимы.

Механизмы возврата процедурой как основного, так и дополнительных результатов в обоих пакетах, практически, *эквивалентны* по своим возможностям, как было показано выше, и достаточно развиты. Основным отличием является лишь то, что *Maple* может в качестве результата возвращать через основной выход множество результатов в виде последовательности, списка либо множества значений, то *Mathematica* делает это же только через *списочную* структуру. Обусловлено это тем важным обстоятельством, что пакет *Mathematica* ориентирован на научные вычисления и, возможно, превосходит по функциональности все имеющиеся на сегодня алгебраические системы. Одной из базовых концепций пакета является «*все есть список*». *Mathematica* оптимизирована под функции **Map**, **Apply** и им подобные пользовательские, играющие чрезвычайно важную роль в реализации пакетом функциональной парадигмы.

*Тело* процедуры описывает алгоритм решаемой задачи, для чьего программирования могут использоваться все средства встроенного языка того либо иного пакета, наряду со средствами, присущими сугубо процедурной структуре. В качестве процедурных средств в пакете *Maple* выступает ряд *процедурных* переменных, имеющих смысл лишь при использовании их в теле процедуры, в частности, *\_passed (args)*, *\_npassed (nargs)*, *\_nresults*, *procname*, *\_params*, *\_nparams*, *return*. Данные переменные дают возможность проще программировать многие алгоритмы, погружаемые в тело процедур, тогда как пакет *Mathematica* подобными процедурными средствами не располагает, определяя вполне конкретные неудобства при программировании целого ряда типов процедур. Между тем, средства *Math*-языка предоставляют возможность создавать аналогичные средства, в значительной степени эквивалентные процедурным средствам *Maple*. С рядом из них можно познакомиться выше и в нашем пакете *AVZ\_Package\_1* [120]. Что в определенной степени уравнивает возможности обеих систем в программировании процедурных объектов, позволяя легче погружать *Maple*-процедуры в *Mathematica*.

Между тем, *процедурная парадигма* системы *Mathematica* с учетом предоставляемых ее *Math*-языком функциональных средств и вышесказанного, представляется нам более перспективной, чем процедурная среда программирования *Maple*. И действительно, функциональные средства *Mathematica* во многих отношениях более развиты, чем в *Maple*, особенно при разработке целого ряда средств системного характера, тогда как собственно сам механизм *процедурного* программирования *второй* системы, по нашему мнению, более предпочтителен. Вышеупомянутые процедурные средства, созданные в среде *Math*-языка в значительной степени уравнивают возможности обеих систем в смысле организации процедур, оставляя, между тем, большую читабельность *Maple*-

процедур относительно *Mathematica*-процедур, что несложно усмотреть на примере процедур уже достаточно небольшого размера. С подобного типа примерами можно ознакомиться выше, где приводятся аналоги процедур на языках и *Maple*, и *Math*. Но не все так однозначно – при всей наглядности и прозрачности *Maple*-процедур, они, между тем, в большинстве своем уступают (*и, порой, довольно существенно*) по времени выполнения и размеру исходного кода аналогичным *Mathematica*-процедурам. Итак, в противовес большей *читабельности Maple*-процедур наряду с лучшей процедурной технологичностью *Mathematica*-процедуры обладают большей эффективностью как по временным показателям, так и по размеру исходного кода. А в целом, *Mathematica* располагает более развитой функциональной поддержкой программирования часто используемых средств и средств, расширяющих средства собственно самой системы.

В контексте работы с *внешними* файлами данных, без чего немислимо использование обеих систем в достаточно сложных проектах, *Maple* имеет немало весьма серьезных ограничений, многие из которых рассматривались в наших книгах [28-38,42-44]. Для устранения ряда из них и расширения возможностей системы в этом направлении в более поздние версии *Maple* был добавлен пакетный модуль *FileTools* и улучшен ряд характеристик средств доступа, целый ряд средств доступа к файлам данных имеется и в нашей библиотеке [45], однако работа с файлами, прежде всего большого объема, все еще является *узким* местом *Maple*. В данном контексте *Mathematica* намного более привлекательна, хотя и здесь имеются достаточно серьезные проблемы со средствами доступа к файлам данных, на которых акцентироваться не считаем целесообразным.

Весьма важным показателем качества программного средства является *робастность*, т.е. *устойчивость* средства к нежелательным, но вполне возможным ошибочным либо особым ситуациям. Для обработки таких ситуаций *Maple*-язык располагает довольно развитым механизмом *try*-предложения, обеспечивающего достаточно эффективную обработку такого типа ситуаций, как возникающих незапланированно, так и заранее программируемых. Данный механизм позволяет программировать процедуры весьма высокого уровня робастности. Так, если рассматривать возможности обеих пакетов по обработке *фактических* аргументов, типы которых недопустимы, вызывая ошибочные ситуации в процессе выполнения процедуры, то средства типизации объектов *Maple* оказываются весьма удобным фильтром при вызове процедур еще до выполнения их тела, инициируя *ошибочные* ситуации, допускающие программную обработку. Тогда как система *Mathematica* в значительной степени *превосходит Maple* по тестированию фактических аргументов. Вполне естественно, что все особые ситуации возникающие в процессе выполнения процедур, целесообразно программно обрабатывать, для чего обе эти системы имеют вполне достаточно средств. Поэтому в этом отношении вполне можно констатировать, что между системами имеет место определенный паритет.

Со стороны продвинутого пользователя чрезвычайно важным представляется вопрос по разработке средств, отсутствующих в системе или по тем или иным причинам для нее неподходящими. В данном случае пользователь создает собственные средства, как правило, в форме *модулей* (*процедур или функций*). Не взирая на то, что *Maple*-язык по многим показателям весьма хорошо отвечает задачам *модульного* программирования, прежде всего, при организации процедур (*преимущества относительно данного аспекта*

рассматривались выше), все же более высокий уровень функциональности *Math*-языка наряду с его смешанной парадигмой *процедурной* и *функциональной*, а также на основе шаблонов и подстановок дает нам основание полагать, что в среде *Mathematica* более эффективна реализация средств пользователя, *расширяющих* стандартные системные средства. При этом, как правило, такие средства обладают относительно аналогичных *Maple*-средств и, зачастую, значительно лучшими *временными* характеристиками при их выполнении. На приведенных выше примерах в этом весьма несложно убедиться.

И еще в данном контексте на один довольно показательный аспект следует обратить внимание. *Maple* и *Mathematica* из одной группы программных средств, занимающие в ней *лидирующие* позиции и при определенных соглашениях функционально вполне сопоставимые. Именно поэтому естественным возникает вопрос создания *конверторов* из среды одной системы в среду другой, и наоборот. В системе *Mathematica* подобные средства не представлены, что, на наш взгляд, вполне обоснованно. Тогда как у *Maple* в качестве такого средства выступает внешний модуль **MmaTranslator**, включающий ряд средств для обеспечения конвертирования выражений, операций и *Mathematica*-документов в среду *Maple*, что может служить в качестве косвенного подтверждения отдачи приоритета в пользу своего конкурента. Декларируется, что данный модуль позволяет конвертировать *Mathematica*-вход в *Maple*-вход и *Mathematica*-документ (*nb*) в *Maple*-документ (*mw*); более того, конвертирование *Mathematica*-программ не поддерживается. Но не все так безоблачно; так, производится конвертирование очень простых *nb*-файлов в одноименные *mw*-файлы, располагаемые в том же каталоге, что и *nb*-файлы, тогда как достаточно сложные *nb*-файлы конвертируются некорректно или и вовсе при их конвертации инициируется *ошибочная* ситуация. При этом, польза от данного типа *mw*-файлов довольно сомнительна. Относительно конвертирования *Mathematica*-входа в *Maple*-вход также не приходится особо обольщаться, а именно. Простые примеры [99] иллюстрируют результат *конвертирования* в *Maple* корректного *Mathematica*-входа, который интерпретировать в среде *Maple* весьма затруднительно.

Точнее, без знания по меньшей мере основ *Mathematica*-языка такая интерпретация в среде *Maple* уже достаточно простых *Mathematica*-входов (*тем более их выполнение*) и вовсе не представляется возможной, сводя на нет результаты данной интерпретации. А так как конвертор не работает с *Mathematica*-кодами в целом (*что представляется с точки зрения замысла наиболее актуальным*), то даже конвертация простых выражений в этом свете не представляется успехом. Итак, если целью *конвертора* являлось решение вопроса совместимости *Maple* с *Mathematica*, то эта задача весьма далека от решения. Что же касается заимствования более эффективных *Mathematica*-кодов, значительно проще перепрограммировать их с учетом особенностей встроенного *Maple*-языка, но и здесь возникает целый ряд проблем, возникающих, прежде всего, из-за различных *программных* парадигм, используемых обеими системами. Это еще одно соображение в пользу того, что для своей работы наиболее эффективно освоение обеих систем.

Достаточно важным моментом при разработке набора процедур и функций является *вопрос* их организации и *имплантирования* в программную среду системы. И в данном отношении мы отдаем определенные предпочтения системе *Maple* и вот почему. Для сохранения *ПС* пользователя, созданных в среде любой из рассматриваемых систем, в

целом, существует два уровня – *документальный* и *библиотечный*. *Документальный* уровень поддерживают файлы *ASCII*-формата, содержащие определения объектов и, возможно, иную информацию (*функции, процедуры, таблицы, списки и т.д.*); для *Maple* это *{mws, mw}*-файлы, для *Mathematica* – *nb*-файлы. Для получения доступа к таким объектам, чьи определения содержатся в таких файлах, такие файлы с документами должны быть загружены в текущий сеанс и выполнены (*вычислены*), активизировав *все* содержащиеся в них определения объектов и позволив обращаться к ним наравне со стандартными системными средствами. Если таких файлов достаточно много, можно создавать довольно простые *диспетчеры*, загружающие в текущий сеанс определения требуемых объектов (*или целиком нужные файлы*) с их последующим вычислением. И в целом ряде случаев такой механизм вполне приемлем и, в частности, для *Maple* нами были созданы средства поддержки подобной диспетчеризации [10-14,28-38,45,90].

Второй уровень – *библиотечный* – поддерживается стандартными средствами, являясь наиболее эффективным со многих точек зрения. В *Mathematica* его основу составляет механизм работы с пакетами, находящимися в *{m, mx}*-файлах, в то время как в *Maple* в основе лежат библиотеки, структурно аналогичные главной библиотеке *Maple*. Оба механизма рассматривались нами несколько выше и, с нашей точки зрения, наиболее естественна и эффективна организация библиотек в *Maple*, поддерживающая весьма простые механизмы создания и ведения библиотек пользователя и имплантирования их в программную среду системы. Хорошо продумана и *справочная база*, которая легко настраивается на справочные базы библиотек пользователя. Только посредством *GUI* пользователь получает возможность *обновлять* свои справочные базы, чьи справочные страницы можно оформлять аналогичными справочным *страницам Maple*. В качестве неплохого примера организации пользовательских библиотек можно привести нашу Библиотеку [45]. Итак, не отдавая каких-либо существенных предпочтений той либо другой системе, все же *Maple*, по нашему мнению, предоставляет пользователю более дружелюбный и простой механизм для ведения собственных библиотек наряду с их имплантированием (*как разовым, так и фиксированным*) в программную среду *Maple*. В данном контексте, *пользователь*, обладающий даже не очень большим опытом работы с *Maple*, довольно легко осваивает механизм *ведения* собственных библиотек, которые снабжаются необходимыми справочными базами, в чем мы неоднократно убеждались в процессе проведения целого ряда мастер-классов по *Maple* в *СНГ*, в других странах. Итак, механизм интеграции пользовательских средств в среду *Maple* более удачен.

И последнее относительно справочных систем обоих пакетов. В обоих она довольно развита и предоставляет информацию по каждому средству пакета в таких разрезах как: форматы вызовов, описание средства, примеры по его применению и ссылки на связанные разделы пакета. Между тем, если в справке по средству, имени которого в базе нет, *Maple* выводит список всех имен с указанным именем в качестве префикса, то *Mathematica* подобного не делает, отсылая к сайту разработчиков либо выводя все страницы справки, содержащие данный префикс. Данный подход не представляется нам корректным, исходя из сути поиска в такой базе данных. Ведь даже недостаточно опытный пользователь, работая в среде математического пакета и базируясь на своих математических познаниях, вполне может пытаться применить для поиска нужного

средства известные ему имена. Например, для поиска использовать фразу «*search*», по которой в *Maple* выводятся имена средств с этим префиксом, независимо от регистра, тогда как *Mathematica* на эту же фразу выводит 264 ссылки, некоторые из которых не только не соответствуют смыслу такого поиска, например, «*NMinimize[f,x] – minimizes f numerically with respect to x*», но и засоряют запрос излишней информацией. При этом, по цепочке «*Help -> Topic search*» главного меню *Maple* можно получать полный список средств, чьи имена начинаются с заданного символа. А так как большинство данных имен для пользователя, знакомого с математикой, поможет многое ему предположить о сути данного средства, то такой подход следует приветствовать. Нечто подобное мы можем обнаружить и у *Mathematica*, но сделано это непродуманно и особого смысла не имеет ввиду обилия получаемой на запрос информации и нужной, и ненужной. В то же время и *Maple* позволяет осуществлять поиск по фразе по всей справочной базе, но это лишь как вспомогательное средство. Наконец, справочная система *Maple* дает возможность пользователю не только просто создавать и редактировать собственные справочные базы, организованные по аналогии с *системной*, но и помещать справки в системную справочную базу, что в целом ряде случаев представляется нам довольно существенной возможностью при разработке больших проектов в среде *Maple*.

Встроенный язык системы *Maple* является *универсальным* чисто процедурным языком программирования, который синтаксически довольно близкий к наиболее массовым современным императивным языкам программирования. Это существенно облегчает его освоение массовым пользователем, в первую очередь, знакомому с языками такого класса, в *Maple* можно начинать работу, имея минимальные сведения по *Maple*-языку. Этот же аспект играет и весьма немаловажную роль, когда возникает вопрос о выборе предпочтения – система *Maple* или система *Mathematica*, т.е. налицо субъективность.

Являясь *эквивалентными* в смысле *универсальной* вычислимости, системы *Mathematica* и *Maple* достаточно существенно разнятся по целому ряду важных аспектов. Прежде всего, программы, написанные в среде языка *Maple*, на наш взгляд, значительно более читабельны, чем аналогичные программы *Mathematica*. И прежде всего, обусловлено это, как уже отмечалось, определенной непривычностью языка *Mathematica*, которую иногда называют «*архаичностью*» – синтаксис и организация основных конструкций языка существенно отличаются от современных языков программирования. Конечно, это в значительной мере вопрос *привычки*, но система *Mathematica* является далеко не *универсальным* средством, чтобы под него серьезно подстраиваться, и подобного типа разноречие с общеустоявшимся вряд ли можно считать положительным явлением.

С другой стороны, встроенный *Math*-язык системы *Mathematica* является *процедурно-функциональным*, синтаксически достаточно непривычным как для традиционного программиста, так и для математика, хотя во многом данная ситуация определяется и *привычкой*. Действительно, при довольно активном использовании системы *Mathematica* наряду с другими пакетами и/или системами программирования контраст на уровне синтаксиса играет довольно существенную роль, являясь причиной многочисленных синтаксических ошибок и излишней траты нервной энергии у пользователей. Между тем, с одной стороны использование *квадратных* скобок там, где механически тянется рука закодировать круглую скобку, с другой стороны, длинные имена функций для

знакомому с основами английского позволяет легче понимать их суть и осуществлять ассоциативный поиск средств по именам в справке по системе *Mathematica*.

Следует четко представлять принципиальную разницу между *привычностью* языка и его *выразительными* возможностями. Вполне естественно, для пользователя, который достаточно рано ознакомился программированием и, прежде всего, традиционным, более естественным представляется процедурный язык программирования, который по целому ряду характеристик весьма существенно напоминает известные ему языки программирования ярко выраженного *процедурного* типа, например, *C*, *Pascal* и др. И в этом отношении пользователю на самом деле легче осваивать, а затем привыкать, к языку программирования *Maple*. Тогда как, в целом, со средой программирования в *Mathematica* ситуация обстоит несколько иным образом. Являясь языком *смешанного* типа, а именно, *процедурно-функциональным*, для его эффективного использования требуются большие усилия со стороны пользователя, имеющего дело с *традиционным* процедурным опытом. Прежде всего, что до *процедурных* средств *Mathematica*, то они, как уже акцентировалось нами внимание выше, существенно уступают среде *Maple*, тогда как *функциональная* составляющая среды *Mathematica*, являясь весьма мощным инструментом программирования, между тем, не совсем привычна пользователю, не имеющему сколько-нибудь заметного опыта работы с функциональными языками. В то же время, хорошо освоив *Math*-язык, пользователь понимает, насколько он удобен, гибок и функционален, предоставляя достаточно широкий спектр программистских парадигм, включая: (1) *процедурное* (модульное) и (2) *функциональное* программирование, (3) программирование на основе шаблонов и правил подстановок, и в определенной степени объектно-ориентированное программирование. Ради справедливости нужно отметить то обстоятельство, что и *Maple*-язык поддерживает определенные элементы *функционального* программирования (некоторые примеры были приведены в [99]), однако его возможности довольно ограничены. Подобно *Math*-языку, и *Maple*-язык также в определенной мере поддерживает объектно-ориентированное программирование. В то же время, несмотря на всю непривычность *Math*-языка, эффективно написанный на нем алгоритм весьма *реактивен* и далеко не всегда уступает в скорости выполнения программам, написанным на других языках. Из нашего опыта такого типа возможно привести ряд примеров, когда на больших процедурах, реализованных в *Math*-языке, получали почти *полуторный* временной выигрыш, чем на *Java 6* и *Turbo Delphi*. Тогда как *Maple*-язык во *временном* отношении выполнения программ не настолько хорош. В данном контексте следует отметить и такую весьма важную составляющую систем, как обучающая. Именно доступность для пользователя исходных кодов стандартных средств пакета на его входном языке может предоставить превосходные примеры для освоения эффективной *технологии* программирования пользовательских средств и не только. Более продвинутый пользователь получает возможность модификации таких средств с целью их расширения либо улучшения. И в этом отношении, если средства среды *Mathematica* скрыты от пользователя, возможно, содержат очень оригинальные алгоритмы, достойные премии *Филдса* или на худой конец *Тьюринга*, то стандартные средства среды *Maple*, исключая ограниченное число *встроенных* функций, доступны для просмотра пользователем. Это дает прекрасную возможность не только осваивать

программирование в среде пакета на примере качественно разработанных процедур и модулей, но и модифицировать их под *нужды* квалифицированным пользователем, устранять возможные ошибки, брать за *прототип* для создания собственных средств. Имеется ряд других соображений в пользу такой открытости системы [10-14,28-38]. В отличие от *Maple*, *Mathematica* с учетом вышесказанного в определенной мере более сложна для освоения, однако освоив ее, пользователь получает немало преимуществ и на уровне готовых программных средств, и на уровне процедурно-функционального языка программирования, обеспечивая пользователя достаточно мощным аппаратом разработки собственных проектов в различных областях и, прежде всего, достаточно сложных проектов физико-математической и инженерной ориентации.

В определенной степени среду *Maple* возможно рассматривать как наиболее удачное на сегодня средство в деле освоения систем компьютерной алгебры в университетах, а также преподавания информатики, что делает ее более естественная организация, встроенный язык программирования и достаточно дружелюбный пользовательский интерфейс, тогда как среда *Mathematica* с ее не совсем привычным для пользователя, обладающего опытом работы с традиционными *процедурными* языками, *Math*-языком программирования, несколько менее дружелюбным *интерфейсом* пользователя (*GUI*), изрядной долей излишеств (*правда, и Maple в данном отношении не безгрешен*), однако в целом ряде случаев обладающий большей (*порой, значительной*) функциональностью и реактивностью можно рекомендовать как достаточно удовлетворительное средство в серьезных научных исследованиях и разработке больших и сложных проектов. На наш взгляд, как *Maple*, так и *Mathematica* являются несомненными лидерами среди *CMS* общего назначения; но существуют *CMS* более специально ориентированные, в своей предметной области значительно превосходящие по возможностям первые две.

Естественно, обе рассмотренные системы и *Maple*, и *Mathematica* универсальны в том отношении, что пользователь может либо непосредственно средствами пакета, либо используя его встроенный язык запрограммировать любой алгоритм, однако здесь мы акцентировали внимание именно на эффективности и наличии средств у пакета для наиболее типичных задач (*в значительной степени реализуемых в виде процедур либо функций*), с которыми сталкивается каждый пользователь при программировании и/или организации собственных программных средств. И здесь следует также отметить, немалую роль при использовании данных пакетов играет такой момент как *привычка* и *предыдущий* опыт программирования. В целом же, практически, любую задачу с той либо другой степенью эффективности можно запрограммировать как в *Mathematica*, так и в *Maple*; между тем, если реализующие их *Maple*-процедуры по читабельности и прозрачности превосходят соответствующие им *Mathematica*-процедуры, вторые в определенной степени несколько менее читабельны, однако они могут выполняться, зачастую, существенно быстрее и их *исходный* код более компактен. Высокий уровень компактности исходных кодов процедур и выражений иного типа обеспечивается не в последнюю очередь именно ранее отмеченной «*архаичностью*» (*непривычностью для пользователя, работающего с императивными языками*) синтаксиса базовых управляющих структур, которая позволяет достаточно широко использовать композицию функций. В качестве простого примера приведем функционально эквивалентную композицию

функций в программной среде систем *Mathematica* и *Maple*, реализующую простой вычислительный алгоритм, а именно:

```
In[2725]:= Log[{a, k} = {0, 1}; For[k, k <= 200, k++, a = a + If[PrimeQ[k], k, 0]]; a + 460.78]
Out[2725]= 8.45271
In[2726]:= Log[Sum[If[PrimeQ[k], k, 0], {k, 200}] + 460.78]
Out[2726]= 8.45271
> ln((proc(x) local a, k; a := 0; for k to 200 do a := a + `if`(type(k, prime), k, 0) end do;
a + x end proc)(460.78));      => 8.452714402
> evalf(ln(add(`if`(type(k, prime), k, 0), k=1..200) + 460.78));      => 8.452714402
```

Между тем, говоря о *Maple*-языке, как средстве для освоения парадигмы *процедурного* программирования, следует отметить и то весьма немаловажное обстоятельство, что *Math*-язык вполне пригоден в качестве средства освоения парадигмы *функционально-процедурного* программирования, и в обоих случаях, в целом, парадигмы *модульного* программирования. Более того, полученные навыки программирования в среде этих двух систем составят хорошую основу для расширения *процедурной* парадигмы для среды *Mathematica*, с одной стороны, и *функциональной* парадигмы для среды *Maple*, с другой стороны. При этом, *Maple*-язык, не будучи ориентированным на парадигму функционального программирования, располагает, между тем, средствами, которые позволяют реализовывать в его среде целый ряд важных как управляющих структур, так и конструкций, присущих сугубо функциональным языкам программирования. С другой стороны, средства *Math*-языка поддерживают возможность реализации как управляющих структур, так и конструкций, которые специфичны для расширенного *процедурного* программирования. Поэтому, пользователь имеет хорошую возможность расширять парадигму программирования *Maple*-языка в сторону повышения уровня ее *функциональной* составляющей, тогда как *Math*-языка в сторону повышения уровня ее *процедурной* составляющей. Правда, связано это, зачастую, с довольно сложными приемами, включая *нестандартные*, и алгоритмами наряду с довольно *существенными* временными издержками и, в первую очередь, это относится именно к среде *Maple*. Тогда как в случае с системой *Mathematica* в виду большей, в целом, реактивности ее программной среды издержки подобного типа заметны не столь существенно.

Естественно, более существенный уровень использования парадигмы, не присущей в определяющей степени той или иной из рассматриваемых *двух* систем, предполагает *расширение*, порой существенное, стандартных средств систем *Maple* или *Mathematica* соответствующим набором *дополнительных* пользовательских средств, как это и было проиллюстрировано рядом специальных *процедур* и *функций*, запрограммированных в среде обоих пакетов, с которыми можно ознакомиться в [45,90]. Между тем, следует иметь в виду, что свести парадигмы языков программирования к единой достаточно высокого уровня парадигме *функционально-процедурного* программирования, на наш взгляд, не представляется возможным по причине довольно существенных различий используемых обоими системами изначальных парадигм программирования.

В подтверждение ранее сказанному следует еще раз подчеркнуть, наш богатый опыт проведения мастер-классов по системам компьютерной математики для докторантов



и специалистов физико-математического профиля и, в первую очередь, математиков однозначно показывает, парадигма *процедурного* программирования, поддерживаемая средой *Maple*, довольно легко и быстро усваивается теми, кто ранее имел некий опыт работы с языками *процедурного* типа. Тогда как парадигму *процедурно-функциональную* и, прежде всего, *функциональную* парадигму, поддерживаемую системой *Mathematica*, достаточно легко и быстро осваивают те, кто ранее не имел серьезного опыта работы с процедурными языками программирования. Посему, несколько большую простоту освоения программной среды *Maple* вполне можно относить на счет уже имеющегося у осваивающих опыта программирования, как правило, в среде *императивных* языков и ни в коем разе ввиду т.н. «архаичности» *встроенного Math-языка* среды *Mathematica*.

Тут же следует отметить, парадигма функционального программирования наиболее близка именно математикам и специалистам ряда других направлений с достаточно серьезной математической подготовкой, одновременно позволяя создавать довольно компактные и эффективные во временном отношении программы различного типа, которые с массовым использованием *чистых* функций допускают довольно высокий уровень распараллеливания на многоядерных *ПК* и кластерах. Правда, применение *чистых* функций оказывается эффективным при наличии 2 основных предпосылок, а именно: (1) язык должен располагать средствами как определения чистых функций, что в целом не столь проблематично, так и приписывания для них статуса «*чистая*» с целью последующей их обработки компилятором (*интерпретатором*) именно в таком качестве, и (2) компилятор (*интерпретатор*) с такого языка должен располагать вполне эффективным механизмом идентификации *чистых* функций на предмет выявления возможности *распараллеливания* вычисления выражений, содержащих такие функции.

И, если *Math-язык* располагает достаточно развитыми средствами как тестирования, так и идентификации чистых функций (*в частности, мы представили в книге несколько выше довольно простое средство тестирования т.н. &-функций* [120]), что обуславливает весьма неплохие предпосылки для распараллеливания вычислений, использующих такие функции, то *Maple-язык* подобными средствами не располагает (*напомним, что сравнительный анализ систем Maple и Mathematica проводился нами* [97-101], *в основном, на основе их релизов 8 ÷ 11 и 7.0 ÷ 9.0.1.0 соответственно*). При этом, при рассмотрении в настоящей книге ряда средств *Mathematica* с целью акцента на его свойствах дается и сравнение с соответствующими средствами системы *Maple*.

И еще раз на одном чрезвычайно важном аспекте пакетов *Maple* и *Mathematica* нужно акцентировать внимание, а именно на стабильности их *программных* сред в контексте совместимости разрабатываемых в них средств пользователя в зависимости от версии пакета. В отличие от *Mathematica*, *Maple* характеризуется довольно высоким уровнем несовместимости «*снизу-вверх*», прослеживаемым нами на протяжении его релизов с четвертого по 11-й. Итак, немало средств нашей Библиотеки [45] приходилось заново перепрограммировать под *новые* релизы, тогда как *Mathematica* в данном отношении представляется нам намного более стабильным – практически все созданные средства в ее среде еще релиза 2.2 успешно выполняются в последних его релизах 7 и 8, 9, а это весьма важный показатель для любого пролонгированного программного средства. В процессе работы с *Mathematica* мы убедились в пролонгированности ее среды.

В связи со *сравнительными* оценками обеих систем, представленными нами в книгах [98–101], в контексте использования их в качестве программной среды, прежде всего, для задач *модульного* программирования наша точка зрения на данные системы, как признанных лидеров в классе *CMS*, состоит в следующем, а именно:

– *CMS Mathematica, поддерживая довольно развитый процедурно-функциональный язык программирования, наилучшим образом отвечает, в первую очередь, довольно серьезным научно-техническим исследованиям аналитически-численного характера в достаточно больших проектах, не исключая задач образовательного характера и, в частности, совершенствования преподавания математически-ориентированных дисциплин в университетах, освоении CMS. В целом, можно отметить, Maple более удобна в преподавании, Mathematica в научных исследованиях. Опыт использования обеих CMS в преподавании и разработке разного рода приложений в таких областях, как математика, физика и инженерные науки подтверждают нашу точку зрения.*

В частности, использование среды *Mathematica* имеет существенные преимущества в разработке довольно сложных проектов, использующих работу с большими файлами данных и базами данных/знаний. *Maple* в этом отношении имеет значительно более слабые возможности. На наш взгляд, достаточно серьезными преимуществами среда *Mathematica* обладает также в разработке системных средств, расширяющих средства системы, носящие массовый характер или устраняющих какие-либо их недостатки, а также расширяющих их *функциональные* возможности. Однако, это не самый худший вариант, к большому сожалению, *обе* системы страдают весьма большим количеством неустраняемых из релиза в релиз ошибок, а также дополнением новых ошибок к уже существующим. Из нашего немалого опыта следует отметить, что и в таком контексте система *Maple* выглядит несколько более впечатляюще, чем система *Mathematica*.

И если на *уровне* студенческих задач и относительно несложных проектов в контексте упомянутой простоты можно отдать некоторое предпочтение системе *Maple*, уже для разработки серьезных проектов, требующих достаточно развитых функциональных средств высокой реактивности наряду с предпочтениями *функциональной* парадигмы, в целом, более предпочтительным представляется нам именно среда *Mathematica*. И в данном контексте довольно заметно, что *Maple* со временем пытается заимствовать немалое число новаций, навеянных именно системой *Mathematica*.

Не взирая на достаточно жесткую конкуренцию рассмотренных выше систем *Maple* и *Mathematica*, в пределе стремящуюся к оформлению некоего *общего* знаменателя, как по основным функциональным средствам, так и по целому ряду «*архитектурных излишеств*», наши рекомендации по эффективной применимости обоих пакетов еще очень долгое время будут сохранять свою актуальность. В то время как специалисту, довольно серьезно использующему в своей профессиональной деятельности *системы компьютерной математики*, еще раз рекомендуется освоить *обе* рассмотренных среды, что позволит достаточно эффективно разрабатывать, практически, любые различных назначения и сложности математически-ориентированные проекты. На этом важном моменте постоянно нами делается акцент, апробированный многолетним опытом по разработке достаточно сложных проектов, весьма существенно использующих *CAS*. В целом ряде важных приложений *обеих* систем *Mathematica* и *Maple* эта рекомендация

зарекомендовала себя с самой наилучшей стороны.

Между тем, весьма немаловажную роль при выборе того либо иного пакета играет и, порой весьма существенную, привычки и предыдущий опыт программирования. Так, если имеющий опыт работы с функциональными языками вполне может отдать свое предпочтение *Mathematica*, то имеющему опыт работы с процедурными языками во многом более предпочтительным может оказаться *Maple*. Оба рассмотренных пакета, являясь не только несомненными лидерами среди современных универсальных *CMS*, но и остро конкурирующими средствами в этой нише программных средств, весьма активно пытаются перенять все *плюсы* и *новации* своего конкурента, что по большому счету постепенно выравнивает их возможности для пользователей среднего уровня и ниже. Тогда как для продвинутых пользователей, не обремененных традиционными парадигмами, в вопросе выбора могут лежать приведенные выше соображения.

Наш опыт разработки довольно сложных программных продуктов, включая средства системного характера, на основе сравнительного анализа сложности реализации их в среде обеих систем со всей определенностью говорит в пользу предпочтения системы *Mathematica*, но с одним весьма существенным «*но*», а именно. Преимущества среды *Mathematica* сказываются только в том случае, если пользователь достаточно искушен в ее программной среде, хорошо владея средствами встроенного *Mathematica*-языка, прежде всего, функциональной парадигмой программирования. Иначе недостаточно квалифицированному пользователю придется по душе существенно более простой в освоении и использовании встроенный язык среды *Maple*. Еще раз отметим, здесь мы акцентируем внимание именно на *программном* аспекте систем, но не на режиме их использования в качестве высоко интеллектуального математического калькулятора или написания простейших документов и процедур/функций, когда определяющую роль, зачастую, играет привычность используемого системой синтаксиса. Именно по последнему показателю среда *Maple* и отличается в лучшую сторону и привычностью используемого ею языком синтаксиса, и вполне достаточной прозрачностью наряду с хорошей структурированностью исходных кодов ее программных средств. Тогда как по этим показателям система *Mathematica* довольно существенно уступает *Maple*.

В случае использования режима интенсивного программирования, когда в процессе его процедура/ функция либо документ программируются достаточно интуитивно, в результате чего часто приходится вновь возвращаться к *предыдущим* состояниям кода, довольно существенным недостатком *Mathematica* является наличие у нее функции **Undo** с глубиной лишь на *один* шаг, что предполагает, в частности, достаточно частое сохранение изменений в файлах, делая процесс программирования неудобным, ибо, в частности, интуитивное программирование предполагает частое редактирование и кодов документов, и кодов процедур/функций, и без возможности по функции **Undo** с достаточной глубиной (*подобно, например, в Word*) весьма неудобно выполнять такого рода работу, предполагая частые сохранения текущего документа. В частности, для пакета *Maple* функция **Undo** имеет глубину порядка **20** шагов. Более того, для целого ряда выражений *E* допустимы вызовы процедуры **Undo(E)**, возвращающие *предыдущее* состояние при вычислениях *E*. Данная процедура может повторяться до тех пор, пока выражение не придет в свое исходное до обработки состояние.

Более того, ввиду идентификации пакетом процедур/функций не по их именам, а по заголовкам, при отладке процедур/функций (*когда требуются, зачастую, многократные перевычисления версий их кодов*) в интерактивном режиме во многих случаях глубина реального обновления слишком мала, требуя периодической очистки предыдущих определений и перевычисления последнего определения, иначе активным остается одно из предыдущих определений. К сожалению, определяется это лишь тогда, когда вызовы новой версии средства оказываются идентичными одной из его предыдущих версий, хотя и должны быть отличными. Здесь оказывают помощь лишь сказанное и достаточный программистский опыт пользователя в среде системы. Тогда как в среде системы *Maple* такие ситуации не имеют места, упрощая процесс программирования.

Также уместно отметить, что характерной чертой организации *Math*-языка является то, что чтение листингов процедур/ функций, написанных на данном языке, вообще говоря, требует *большего* профессионализма, чем чтение листингов подобных средств на *встроенном* языке среды *Maple*. Однако, при получении определенного навыка при работе с программной средой системы данная проблема *нивелируется*, тогда как сами коды средств получаются более компактными и эффективными при выполнении.

В данном разделе можно было бы дать и более развернутый *сравнительный* анализ по обоим системам, однако было решено акцентировать отдельное внимание только на аспектах, имеющих непосредственное отношение к рассматриваемой тематике. Тогда как элементы более детального *сравнительного* анализа можно найти в наших книгах [98–101]; между тем, подобная работа вполне заслуживает определенного внимания и в данном направлении экспертные оценки специалистов, использующих обе системы *CMS*, были бы востребованными. К сожалению, серьезных работ в этом направлении мы не встречали, что обусловлено рядом причин, здесь не обсуждаемых.

Естественно, наши оценки – не являются истиной в последней инстанции, между тем, они нашли свое подтверждение в процессе как нашего опыта программирования как средств широкого применения и системных в среде систем *Maple* и *Mathematica*, так и лежат в русле оценок ряда других специалистов по *CMS* и опытных программистов. Надеемся, что представленные выше соображения окажут определенное влияние на решение вопроса о приоритетности той либо другой системы для конкретной сферы приложений. Более того, рекомендуется работать с обеими средами, которые весьма неплохо дополняют друг друга при решении широкого круга достаточно сложных и объемных задач, требующих и аналитических, прежде всего, и численных методов. И иметь в виду, что даже значительный опыт работы с обоими системами не исключает ряда неприятных моментов, серьезно раздражающих пользователя, когда совместная работа с обеими системами, порой, приводит к ошибкам, обусловленным различиями *синтаксисов* их языков, исповедуемыми *парадигмами* программирования, и др. Между тем, преимущества, получаемые от такого *симбиоза*, весьма существенно превалируют над вполне естественными неудобными и раздражающими *издержками*. В заключение отметим, что настоящую книгу вполне можно рассматривать и в качестве своего рода расширенного справочного руководства по нашему пакету *AVZ\_Package\_1* [120]. При этом, книгу можно рассматривать также в качестве руководства по применению ряда полезных и эффективных приемов *процедурно-функционального* программирования.

## Литература

1. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Основы информатики.– Гомель: Изд-во Salcombe Eesti, 1997, 396 с., ISBN 5-14-064254-5.
2. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Основы информатики: Учебное пособие.– Москва: Изд-во ФилинЪ, 1998, ISBN 5-89568-068-2.
3. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Основы информатики: Учебное пособие. 2-е изд.– Москва: Изд-во ФилинЪ, 1999, 520 с.
4. Аладьев В.З., Гершигорн Н. Вычислительные задачи на персональном компьютере.– Киев: Изд-во Техника, 1991, 248 с.
5. Аладьев В.З., Тупало В.Г. Алгебраические вычисления на компьютере.– Москва: Изд-во Минтопэнерго, 1993, 251 с., ISBN 5-942-00456-8.
6. Аладьев В.З., Хунт Ю., Шишаков М.Л. Математика на персональном компьютере.– Гомель: Изд-во ФОРТ, 1996, 498 с.
7. Аладьев В.З., Шишаков М.Л. Введение в среду пакета *Mathematica 2.2*.– Москва: Изд-во ФилинЪ, 1997, ISBN 5-89568-004-6.
8. Аладьев В.З., Ваганов В.А. и др. Введение в среду математического пакета *Maple V*.– Минск: Изд-во IAN Press, 1998, 452 с., ISBN 14-064256-98.
9. Аладьев В.З. др. Программирование в среде математического пакета *Maple V*.– Гомель: TRG & Salcombe Press, 1999, 470 с.
10. Аладьев В., Ваганов В., Хунт Ю., Шишаков М. Рабочее место для математика.– Гомель–Таллинн: International Academy of Noosphere, 1999, 605 с.
11. Аладьев В.З., Богдэвичус М.А. Решение математических и физико-технических задач с пакетом *Maple V*.– Вильнюс: Technics Press, 1999, 686 с., ISBN 9986-05-398-6.
12. Аладьев В.З., Шишаков М.Л. АРМ математика.– М.: Изд-во БИНОМ, 2000 + CD.
13. Аладьев В., Богдэвичус М. *Maple 6*: Решение математических, статистических и инженерно-физических задач.– М.: Изд-во БИНОМ, 2001, ISBN 5-93308-085-X.
14. Aladjev V.Z., Bogdevicius M.A. *Interactive Maple: Solution of Mathematical, Engineering, Statistical and Physical Problems*.– Tallinn-Vilnius: Academy of Noosphere, 2002, CD.
15. Aladjev V.Z., Bogdevicius M.A. Use of package *Maple V* for solution of physical and engineering problems // Int. Conf. TRANSBALTICA– 99.– Vilnius: Technics Press, 1999.
16. Aladjev V.Z., Hunt U.J. Workstation for mathematicians // Int. Conf. TRANSBALTICA– 99.– Vilnius: Technics Press, April 1999.
17. Aladjev V.Z., Hunt U.J. Workstation for mathematicians // Internat. Conf. «Perfection of Mechanisms of Management», Institute of Modern Knowledge, 1999, Grodno, Byelorussia.
18. Aladjev V.Z., Shishakov M.L. Programming in Package *Maple V* // 2<sup>nd</sup> Internat. Conf. «Computer Algebra in Fundamental and Applied Researches and Education».– Minsk, 1999.
19. Aladjev V.Z., Shishakov M.L. A Workstation for mathematicians // 2<sup>nd</sup> Internat. Conf. «Computer Algebra in Fundamental and Applied Researches and Education».– Minsk, 1999.
20. Aladjev V.Z., Shishakov M.L., Trokhova T.A. Educational computer laboratory of the engineer // Proc. 8<sup>th</sup> Byelorussia Mathemat. Conf., vol. 3, Minsk, Byelorussia, 2000.
21. Aladjev V.Z., Shishakov M.L., Trokhova T. Modelling in program environment of the

- mathematical package *Maple* // Int. Conf. on Math. Mod. MKMM-2000.- Herson, 2000.
22. *Aladjev V.Z., Shishakov M.L., Trokhova T.A.* A workstation for solution of systems of differential equations // 3<sup>rd</sup> Internat. Conf. «Differential Equations and Applications».- Saint-Petersburg, Russia, 2000.
23. *Aladjev V.Z., Shishakov M.L., Trokhova T.A.* Computer laboratory for engineering researches // Internat. Conference ACA-2000.- Saint-Petersburg, Russia, 2000.
24. *Aladjev V.Z., Bogdevicius M., Hunt U.J.* A Workstation for mathematicians / Lithuanian Conf. TRANSPORT-2000.- Vilnius: Technics Press, April 2000, Lithuania.
25. *Аладьев В.З.* Компьютерная алгебра // Альфа, № 1.- Беларусь: Гродно: Изд-во Гродненский государственный университет, 2001.
26. *Aladjev V.Z.* Modern computer algebra for modeling of the transport systems // Intern. Conf. TRANSBALTICA-2001.- Vilnius: Technics Press, April 2001, Lithuania.
27. *Aladjev V.Z., Shishakov M.L., Trokhova T.* Workstation for the engineer-mathematician // Proc. of the GSTU, № 3, 2000.- Byelorussia: Gomel: Gomel State University.
28. *Aladjev V.Z., Bogdevicius M.A.* Special Questions of Operation in Environment of the Mathematical Maple Package.- Vilnius: Vilnius Gediminas Technical University, 2001.
29. *Aladjev V.Z., Vaganov V.A., Grishin E.P.* Additional Functional Tools of Mathematical Package Maple 6/7.- Tallinn: International Academy of Noosphere, 2002.
30. *Аладьев В.З.* Эффективная работа с *Maple 6/7*.- М.: Изд-во БИНОМ, 2002 + CD.
31. *Аладьев В.З., Лиопо В., Никитин А.В.* Математический пакет *Maple* в физическом моделировании.- Гродно: Гродненский госуниверситет, 2002, ISBN 3-093-31831-3.
32. *Aladjev V.Z., Vaganov V.A.* Computer Algebra System *Maple*: A New Software Library.- Tallinn: Изд-во International Academy of Noosphere, 2002, 420 p. + CD.
33. *Aladjev V.Z., Bogdevicius M.A., Prentkovskis O.* New Software for Mathematical Package Maple of Releases 6, 7 and 8.- Vilnius: Vilnius Gediminas Technical University, 2002.
34. *Aladjev V.Z.* Computer Algebra Systems: A New Software Toolbox for *Maple*.- Palo Alto: CA: Fultus Publishing, 2004, ISBN 1-59682-000-4.
35. *Aladjev V.Z.* Computer Algebra Systems: A New Software Toolbox for *Maple*.- Palo Alto: CA: Fultus Publishing, 2004, Adobe Acrobat eBook.
36. *Aladjev V. et al.* Electronic Library of Books and Software for Experts, Scientists, Teachers and Students in Natural and Social Sciences.- Palo Alto: CA: Fultus Publishing Co., 2005, CD.
37. *Aladjev V.Z., Vaganov V.A.* Systems of Computer Algebra: A New Software Toolbox for *Maple*.- Tallinn: International Academy of Noosphere, 2003, 270 p., ISBN 9985-9277-6-1.
38. *Aladjev V.Z., Bogdevicius M.A., Vaganov V.A.* Systems of Computer Algebra: A New Software Toolbox for *Maple*. 2<sup>nd</sup> ed.- Tallinn: International Academy of Noosphere, 2004.
39. *Aladjev V.Z., Bogdevicius M.A.* Computer algebra system *Maple*: A new software toolbox // 4<sup>th</sup> Intern. Conf. TRANSBALTICA-2003.- Vilnius: Technics Press, April 2003.
40. *Aladjev V.Z.* Computer Algebra System *Maple*: A New Software Library // Intern. Conf. «Computer Algebra Systems and Their Applications», CASA-2003, Saint-Petersburg, 2003.
41. *Aladjev V., Bogdevicius M., Vaganov V.* Systems of Computer Algebra: A New Software Toolbox for package *Maple* // Int. Conf. on Soft. Engin. Res. and Practice, 2004, Las Vegas.
42. *Аладьев В.З.* Системы компьютерной алгебры. *Maple*: Искусство программирования,

- Москва: Изд-во БИНОМ, 2006, 792 с.
43. Аладьев В.З. Основы программирования в *Maple*.– [www.aladjev-maple.narod.ru](http://www.aladjev-maple.narod.ru), Таллинн: Международная Академия Ноосферы, 2006, ISBN 9985-9508-1-X.
  44. Аладьев В.З., Бойко В.К., Ровба Е. Программирование и разработка приложений в *Maple*.– Гродно: Изд-во ГрГУ, <http://www.aladjev-maple-book.narod.ru/>, 2007, 456 с.
  45. Аладьев В.З. Библиотека версии 2.2215 для пакета *Maple*.– [www.aladjev.narod.ru](http://www.aladjev.narod.ru), [www.aladjev-maple-book.narod.ru](http://www.aladjev-maple-book.narod.ru), 2011; <http://yadi.sk/d/P1FQaYmW619C7>.
  46. Голоскоков Д. Уравнения математической физики. Решение задач в системе *Maple*.– Санкт-Петербург: Изд-во Питер, 2004.
  47. Васильев А.Н. *Maple 8*. Самоучитель.– Москва: Изд-во Диалектика, 2003.
  48. Кирсанов М. Задачи по теоретической механике с решениями в *Maple 11*.– Москва: Изд-во Физматлит, 2010, 264 с.
  49. Коробов В.И., Очков В.Ф. Химическая кинетика: Введение с *Mathcad/Maple/MCS*.– Москва: Изд-во Горячая линия–Телеком, 2009.
  50. Кирсанов М.Н. Графы в *Maple*.– Москва: Изд-во Физматлит, 2007.
  51. Эдвардс Ч.Г., Пенни Д.Э. Дифференциальные уравнения и краевые задачи: Моделирование и вычисление с помощью *Mathematica*, *Maple* и *MATLAB*.– Киев: Изд-во Диалектика, 2007.
  52. Очков В. Физические и экономические величины в *Mathcad* и *Maple*.– Москва: Изд-во Финансы и Статистика, 2002.
  53. Матросов А. *Maple 6*: Решение задач высшей математики и механики.– Санкт-Петербург: Изд-во БХВ–Петербург, 2001.
  54. Манзон Б. *Maple V Power Edition*.– Москва: Изд-во ФилинЪ, 1998.
  55. Прохоров Г., Леденев М., Колбеев В. Пакет символьных вычислений *Maple*.– Москва: Изд-во Петит, 1997.
  56. Говорухин В., Цибулин В. Введение в *Maple*. Математический пакет для всех.– Москва: Изд-во Мир, 1997.
  57. Шмидский Я. К. *Mathematica 5*.– Москва: Изд-во Диалектика, 2004.
  58. Капустина Т. Компьютерная система *Mathematica 3.0* для пользователя.– Москва: Изд-во Солон, 1999.
  59. Воробьев Е. Введение в систему *Mathematica*.– Москва: Изд-во ФиС, 1998.
  60. Кулешов А.А. и др. Электронное учебное пособие по высшей математике на базе системы *Mathematica*.– Беларусь: Минск: Электронная книга БГУ, 2001.
  61. [www.ecsoman.edu.ru/text/16089443](http://www.ecsoman.edu.ru/text/16089443), [www.aladjev.narod.ru](http://www.aladjev.narod.ru), [www.aladjev-maple.narod.ru](http://www.aladjev-maple.narod.ru)
  62. Nelson H. F. Beebe. A Bibliography of Publications about the *Maple* Symbolic Algebra Language.– Salt Lake City: Univ. of Utah, Department of Mathematics, 119 с., 2010.
  63. <http://reduce-algebra.sourceforge.net/bibl/bib.html> – библиография публикаций по физике с использованием системы компьютерной алгебры *REDUCE*.
  64. Еднерал В.Ф., Крюков А.П., Родионов А.Я. Язык аналитических вычислений *REDUCE*.– Москва: Изд-во МГУ, 1989, 176 с.
  65. Климов Д.М., Руденко В.М. Методы компьютерной алгебры в задачах механики.– Москва: Изд-во Наука, 1989, 215 с.



66. Теоретическая механика. Вывод и анализ уравнений движения на ЭВМ / Под ред. В.Г. Веретенникова.- Москва: Изд-во Высшая школа, 1990, 174 с.
67. Методы аналитических преобразований в системе виртуальных машин ЕС ЭВМ для задач динамики ЛА / Под ред. проф. А.Г. Сокольского.- Москва: МАИ, 1988.
68. Программирование на языке R-ЛИСП.- Москва: Изд-во Радио и связь, 1991, 192 с.
69. Hearn A.C. REDUCE User's Manual: Version 3.3, RAND Publication CP78, The Rand Corporation, Santa Barbara, California, 4/1987.
70. Hearn A.C. REDUCE: The First Forty Years // A3L Conference in Honor of the 60th Birthday of Volker Weispfenning, Santa Monica, April 2005.
71. Дэвенпорт Дж. и др. Компьютерная алгебра.- Москва: Изд-во Мир, 1991, 352 с.
72. Дэвенпорт Дж. Интегрирование алгебраических функций / Пер с англ.- Москва: Изд-во Мир, 1985, 192 с.
73. [http://saither.ucoz.ru/news/symbolic\\_integration/2010-08-28-49](http://saither.ucoz.ru/news/symbolic_integration/2010-08-28-49)
74. Компьютерная алгебра: Символьные и алгебраические вычисления / Пер. с англ.- Москва: Изд-во Мир, 1986, 392 с.
75. Reference Manual *muLisp-85*.- Honolulu: Hawaii, Software House, Inc., 1985, 137 p.
76. Stein W. et. al. Sage Tutorial.- Charleston: CreateSpace Press, 2009.
77. Joyner D. Differential Calculus and Sage.- Charleston: CreateSpace Press, 2009.
78. Очков В.Ф. *Mathcad 14* для студентов и инженеров: Русская версия.- Санкт-Петербург: Изд-во BHV-Петербург, 2009.
79. Аладьев В.З. Система компьютерной математики *Maple*: Программа мастер-класса.- <http://www.aladjev-maple.narod.ru/Program.pdf>
80. Очков В.Ф. *Mathcad 14* для студентов и инженеров.- Санкт-Петербург: Изд-во BHV-Петербург, 2007.
81. [http://en.wikipedia.org/wiki/Comparison\\_of\\_computer\\_algebra\\_systems](http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems) - Сравнение современных систем компьютерной алгебры различного назначения.
82. Шишаков М.Л., Трохова Т.А., Стрижак И.В. Обучение алгоритмизации и программированию в системах компьютерной математики // Информатизация образования, № 2, с. 59-62, 2005.
83. Jenks R.D., Sutor R.S. AXIOM: The Scientific Computation System.- N.Y.: NAG and Springer Verlag, 1992.
84. Кулябов Д.С., Кокотчикова М.Г. Аналитический обзор систем символьных вычислений // Вестник РУДН, № 1-2, с. 38-45, 2007.
85. Aladjev V.Z. Classical Cellular Automata: Homogeneous Structures.- CA: PaloAlto: Fultus Books, 2010, 478 p., ISBN 1-59682-222-8.
86. <http://www.haskell.org> - Сайт по вопросам функционального программирования.
87. Wadler P. Why no one uses functional languages // ACM SIGPLAN Notices, 1998.
88. Dominus M.J. Higher-Order Perl.- Massachusetts: Morgan Kaufmann Press, 2005.
89. Arantes R.D. A Computational Reference Guide on Experimental Mathematics, Algorithmic Number Theory and Symbolic Computing.- Rio de Janeiro: Federal University, 2004.
90. Aladjev V.Z. A Package of Procedures and Functions for *Mathematica*.- Tallinn, 2012; The package can be freely downloaded from [www.aladjev.narod.ru/MapleLib.htm](http://www.aladjev.narod.ru/MapleLib.htm)



91. Аладьев В.З., Бойко В.К., Ровба Е.А. Классические однородные структуры: Теория и приложения.– Гродно: Изд-во Гродненского госуниверситета, 2008, 488 с.
92. Аладьев В.З. Классические однородные структуры: Клеточные автоматы.– СА: Palo Alto: Fultus Books, 2009, 535 p., ISBN 1-59682-137-X.
93. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Вопросы математической теории классических однородных структур.– Гомель: BELGUT, 1996, ISBN 5-063-56078-5.
94. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Математическая теория классических однородных структур.– Таллинн-Гомель: TRG & VASCO & Salcombe Eesti Ltd., 1998.
95. Aladjev V.Z., Haritonov V.N. *General Theory of Statistics*.– СА: Palo Alto: Fultus Books, 2004, 256 p., ISBN 1-59682-012-8.
96. Аладьев В.З., Харитонов В.Н. Общая теория статистики.– Palo Alto: Fultus Books, 2006, 256 с., ISBN 1-59682-086-1, Adobe Acrobat eBook (pdf), ISBN 1-59682-081-0.
97. Aladjev V.Z., Vaganov V.A. *Modular Programming: Mathematica vs Maple, and vice versa*.– СА: USA, Palo Alto, Fultus Corporation, 2011, ISBN 978-1-59682-268-9, 418 p.
98. Aladjev V.Z., Bezrukavii A.S., Haritonov V.N., Hodakov V.E. *Programming: Maple or Mathematica?*.– Ukraine: Herson, Oldi-Plus Press, 2011, ISBN 978-966-2393-46-0, 474 p.
99. Аладьев В.З., Бойко В.К., Ровба Е.А. Программирование в пакетах *Mathematica* и *Maple*: Сравнительный аспект.– Беларусь: Гродно: Гродненский госуниверситет, 2011, 517 с., ISBN 978-985-515-481-6.
100. Аладьев В.З., Гринь Д.С., Ваганов В.А. Дополнительные функциональные средства для пакета *Mathematica*.– Украина: Херсон: Изд-во Олди-Плюс, 2012, ISBN 978-966-2393-59-0, 404 с.; <http://computersystems.webs.com>
101. Аладьев В.З., Гринь Д.С. Расширение функциональной среды системы *Mathematica*.– Херсон: Изд-во Олди-Плюс, 2012, ISBN 978-966-2393-72-9, 552 с.
102. Mangano S. *Mathematica Cookbook*.– СА: Sebastopol: O'Reilly Media, Inc., 2010, 828 p.
103. Kendrick D. et al. *Computational Economics*.– Princeton University Press, 2006, 406 p.
104. Hasbrouck J. *Empirical market microstructure: The institutions, economics, and econometrics of securities trading*.– Oxford University Press, 2007, 198 p., ISBN 978-019-530-164-9.
105. Roe T. et al. *Multisector Growth Models: Theory and Applications*.– Springer, 2009, 329 p.
106. Wellin P. et al. *An Introduction to Programming with Mathematica, 3<sup>rd</sup> ed.*.– Cambridge University Press, 2005, 550 p., ISBN 052-184-678-1.
107. Sisson P. *College Algebra, 2<sup>nd</sup> ed.*.– Hawkes Learning Systems, 2008, ISBN 9781932628272.
108. Blower D. *Information Processing: Boolean Algebra, Classical Logic, Cellular Automata, and Probability Manipulation, vol. 1.*.– CreateSpace, 2011, 528 p., ISBN 978-146-093-838-6.
109. Gregor J., Tier J. *Discovering Mathematics: A Problem-Solving Approach to Mathematical Analysis with Mathematica and Maple*.– Springer, 2010, 254 p., ISBN 978-0857290540.
110. Панкратьев Е.В. Элементы компьютерной алгебры.– М.: Изд-во МГУ, 2007, 243 с.
111. Hege Hans-Christian, Polthier K. *Visualization and Mathematics III*, 2011, 457 p.
112. Alberty R. *Applications of Mathematica*.– Wiley Press, 2011, ISBN 9780470639320, 456 p.
113. Shiskowski K., Frinkle K. *Principles of Linear Algebra with Mathematica*.– Wiley, 2011.
114. Terras A. *Zeta Functions of Graphs, A Stroll through the Garden*.– Cambridge Univ., 2011.
115. Kilian A. *Programmieren mit Wolfram Mathematica*.– Springer, 2010, ISBN 97836420467.

116. *Hollis S. CalcLabs with Mathematica for Multivariable Calculus.*– Brooks/Cole, 2012, 274 p
117. *Anmong Xu. Introduction to Scientific Computing: Numerical Analysis With Mathematica.*– China Machine Press, 2010, ISBN 978-711-131-091-4.
118. *Core Language: Tutorial Collection.*– Wolfram Research Inc., 2008, 358 p.
119. *Hastings K.J. Introduction to Probability with Mathematica.*– CRC Press, 2010, 451 p.
120. *Aladjev V.Z. A Package of Procedures and Functions for Mathematica.*– Tallinn, 2013; the package can be freely downloaded from web-site <http://yadi.sk/d/quWOB3fe9kRnf>
121. *Wellin P.R. Programming with Mathematica: An Introduction*, ISBN 9781107009462, 2013
122. *Blinder S.M. Guide to Essential Math, A Review for Physics, Chemistry and Engineering Students*, second edition, ISBN 9780124071636, 2013.
123. *Koberlein B., Meisel D.D. Astrophysics through Computation: With Mathematica Support*, ISBN 9781107010741, 2013.
124. *Ruskeepaa H. Mathematica Navigator: Mathematics, Statistics, and Graphics.*– Elsevier Academic Press, Second Edition, ISBN 012603642X, 2004.
125. *Kilian A. Programmieren mit Wolfram Mathematica.*– Springer, (in Germany), ISBN 9783642046711, 2010.
126. *Boccara N. Essentials of Mathematica: With Applications to Mathematics and Physics.*– Springer, ISBN 9780387495132, 2007.
127. *Shifrin L. Mathematica Programming: An Advanced Introduction.*– Brunel Univ., 2008.
128. *Wu Frederick. Manipulate@Mathematica.*– Tsinghua Univ. Press, 2010 (in Chinese).
129. *Trott M. The Mathematica GuideBook for Numerics.*– Springer-Verlag, 2006.
130. [http://www.elbook.bsu.by/PRODUCTS/elbook\\_list.html](http://www.elbook.bsu.by/PRODUCTS/elbook_list.html).– Электронные книги: электронное учебное пособие по высшей математике на основе системы *Mathematica*, *Mathematica usage in Calculus*, справочное пособие по *Mathematica* и другие.
131. <http://www.elbook.bsu.by/OurMath/glalg.htm>.– сайт научно-методического центра "Электронная книга БГУ"; представлены материалы по системе *Mathematica*.
132. <http://www.calc101.com>.– предоставление платных вычислительных сервисов студентам и техническим профессионалам на основе *webMathematica*.
133. *Глушко В.П., Глушко А.В. Курс уравнений математической физики с использованием пакета Mathematica. Теория и технология решения задач.*– Санкт-Петербург: Лань, 2010, ISBN 978-5-8114-0983-9.
134. *Шапошников Н.Н., Кристалинский Р.Е. Решение вариационных задач строительной механики в системе Mathematica.*– Санкт-Петербург: Лань, 2010.
135. *Wagon S. Mathematica® in Action: Problem Solving Through Visualization and Computation*, 3rd ed., 2010, 574 p., ISBN 978-0-387-75477-2.
136. *Tam P.T. A Physicist's Guide to Mathematica.*– Academic Press: 2011, 641 p.
137. *Bunker G. Mathematica Quickstart.*– Illinois Institute of Technology, 2010.
138. *Mathematica 9 Documentation Center: The complete reference for Mathematica 9*, <http://reference.wolfram.com/mathematica/guide/Mathematica.html>
139. *Gockenbach M.S. Mathematica Tutorial: To accompany Partial Differential Equations: Analytical and Numerical Methods*, 2nd edition, SIAM, 2010.
140. *Nickerson H. Adding OO Capabilities to Mathematica.*– Univ. of Colorado at Boulder, 2011.

***Books on Computer Algebra Systems, published by members of the  
Baltic Branch of International Academy of Noosphere during 1995 – 2013***

1. *Aladjev V.Z., Hunt Ü.J., Shishakov M.L.* Mathematics on Personal Computer.– Gomel: BELGUT Press, 1996, 498 p., ISBN 3-420-614023-3 (in Russian with English summary).
2. *Aladjev V.Z., Shishakov M.* Introduction into Mathematical Package *Mathematica 2.2.*– Moscow: Filin Press, 1997, 363 p., ISBN 5-89568-004-6 (in Russian with English summary).
3. *Aladjev V.Z., Hunt Ü.J., Shishakov M.L.* Basics of Computer Informatics: Textbook.– Tallinn–Gomel: Russian Academy of Noosphere & TRG, 1997, 396 p., ISBN 5-14-064254-5.
4. *Aladjev V.Z., Hunt Ü.J., Shishakov M.L.* Basics of Computer Informatics: Textbook.– Moscow, Filin Press, 1998, 496 p., ISBN 5-89568-068-2 (in Russian with English summary).
5. *Aladjev V.Z., Hunt Ü.J., Shishakov M.L.* Basics of Computer Informatics: Textbook, Second edition.– Moscow, Filin Press, 1999, 545 p. (in Russian with English summary).
6. *Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.* Introduction into Environment of Mathematical Package *Maple V.*– Minsk: International Academy of Noosphere, the Baltic Branch, 1998, 452 p., ISBN 14-064256-98 (in Russian with extended English summary).
7. *Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.* Programming in Environment of Mathematical Package *Maple V.*– Minsk–Moscow: Russian Ecology Academy, 1999, 470 p., ISBN 4-10-121298-2 (in Russian with extended English summary).
8. *Aladjev V.Z., Bogdevicius M.A.* Solution of Physical, Technical and Mathematical Problems with *Maple V.*– Tallinn–Vilnius, TRG, 1999, 686 p., ISBN 9986-05-398-6.
9. *Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.* Workstation for Mathematician.– Tallinn–Minsk–Gomel–Moscow: Russian Academy of Natural Sciences, 1999, 608 p., ISBN 3-42061-402-3 (in Russian with extended English summary).
10. *Aladjev V.Z., Shishakov M.L.* Workstation of Mathematician.– Moscow: Laboratory of Basic Knowledge, 2000, 752 p. + CD, ISBN 5-93208-052-3 (in Russian with English summary).
11. *Aladjev V.Z., Bogdevicius M.A.* *Maple 6*: Solution of Mathematical, Statistical, Physical and Engineering Problems.– Moscow: Laboratory of Basic Knowledge, 2001, 850 p. + CD, ISBN 5-93308-085-X (in Russian with extended English summary).
12. *Aladjev V.Z., Bogdevicius M.A.* Special Questions of Operation in Environment of the Mathematical Package *Maple.*– Vilnius: International Academy of Noosphere, the Baltic Branch & Vilnius Gediminas Technical University, 2001, 208 p. + CD with Library.
13. *Aladjev V.Z., Bogdevicius M.A.* Interactive *Maple*: Solution of Mathematical, Statistical, Engineering and Physical Problems.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2001–2002, CD with Booklet, ISBN 9985-9277-1-0.
14. *Aladjev V.Z., Vaganov V.A., Grishin E.P.* Additional Software of Mathematical Package *Maple* of releases 6 and 7.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2002, 314 p. + CD with Library, ISBN 9985-9277-3-7 (in Russian with English summary).
15. *Aladjev V.Z.* Effective Operation in Mathematical Package *Maple.*– Moscow: Laboratory of Basic Knowledge, 2002, 334 p. + CD, ISBN 5-93208-118-X.
16. *Aladjev V.Z., Liopo V.A., Nikitin A.V.* Mathematical Package *Maple* in Physical Modeling.– Grodno: Grodno State University, 2002, 416 p., ISBN 3-093-31831-3.

17. *Aladjev V.Z., Vaganov V.A.* Computer Algebra System *Maple*: A New Software Library.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2002, CD with Booklet, ISBN 9985-9277-5-3 (in Russian with extended English summary).
18. *Aladjev V.Z., Bogdevicius M.A., Prentkovskis O.V.* A New Software for Mathematical Package *Maple* of Releases 6, 7 and 8.– Vilnius: Vilnius Gediminas Technical University & International Academy of Noosphere, the Baltic Branch, 2002, 404 p., ISBN 9985-9277-4-5, 9986-05-565-2 (in Russian with extended English summary).
19. *Aladjev V.Z., Vaganov V.A.* Systems of Computer Algebra: A New Software Toolbox for *Maple*.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2003, 270 p. + CD, ISBN 9985-9277-6-1 (in Russian with extended English summary).
20. *Aladjev V.Z., Bogdevicius M.A., Vaganov V.A.* Systems of Computer Algebra: A New Software Toolbox for *Maple*. Second edition.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2004, 462 p., ISBN 9985-9277-8-8 (in Russian with English summary).
21. *Aladjev V.Z.* *Computer Algebra Systems*: A New Software Toolbox for *Maple*.– CA: Palo Alto: Fultus Corporation, 2004, 575 p., ISBN 1-59682-000-4.
22. *Aladjev V.Z.* *Computer Algebra Systems*: A New Software Toolbox for *Maple*.– CA: Palo Alto: Fultus Corporation, 2004, Adobe Acrobat eBook, ISBN 1-59682-015-2.
23. *Aladjev V.Z. et al.* Electronic Library of Books and Software for Scientists, Experts, Teachers and Students in Natural and Social Sciences.– CA: Palo Alto: Fultus Corporation, 2005, CD, ISBN 1-59682-013-6 (in Russian language and in English language).
24. *Aladjev V.Z., Bogdevicius M.A.* *Maple*: Programming, Physical and Engineering Problems.– CA: Palo Alto: Fultus Corporation, 2006, 404 p., ISBN 1-59682-080-2, Adobe Acrobat eBook (pdf), ISBN 1-59682-081-0, <http://writers.fultus.com/aladjev/index.html>
25. *Aladjev V.Z.* *Computer Algebra Systems. Maple: Art of Programming*.– Moscow: BINOM Press, 2006, 792 p., ISBN 5-93208-189-9 (in Russian with extended English summary).
26. *Aladjev V.Z.* *Foundations of programming in Maple: Textbook*.– Tallinn: International Academy of Noosphere, 2006, 300 p., (pdf), ISBN 9985-9508-1-X, 978-9985-9508-1-4.
27. *Aladjev V.Z., Boiko V.K., Rovba E.A.* *Programming and applications elaboration in Maple*: Monograph.– Grodno: GRSU, Tallinn: International Academy of Noosphere, 2007, 456 p., ISBN 978-985-417-891-2, ISBN 978-9985-9508-2-1 (in Russian with English summary).
28. *Aladjev V.Z., Vaganov V.A.* *Modular Programming: Mathematica vs Maple*, and vice versa.– CA: USA, Palo Alto, Fultus Corporation, 2011, 418 p., ISBN 978-1-59682-268-9.
29. *Aladjev V.Z., Bezrukavyi A.S., Haritonov V.N., Hodakov V.E.* *Programming: Maple or Mathematica?*– Ukraine: Herson, Oldi-Plus Press, 2011, 474 p., ISBN 978-966-2393-46-0.
30. *Aladjev V.Z., Boiko V.K., Rovba E.A.* *Programming in the Packages Mathematica and Maple: Comparative Aspect*.– Belarus: Grodno, Grodno State University, 2011, 517 p., ISBN 978-985-515-481-6 (in Russian with extended English summary).
31. *Aladjev V.Z., Grinn D.S., Vaganov V.A.* *The extended functional means for the package Mathematica*.– Ukraine: Kherson: Oldi-Plus Press, 2012, ISBN 978-966-2393-59-0, 404 p.
32. *Aladjev V.Z., Grinn D.S.* *Extension of functional environment of system Mathematica*.– Ukraine: Kherson: Oldi-Plus Press, 2012, ISBN 978-966-2393-72-9, 552 p.
33. *Aladjev V.Z., Grinn D.S., Vaganov V.A.* *The selected system problems in Mathematica software*.– Ukraine: Kherson: Oldi-Plus Press, 2013, ISBN 978-966-289-012-9, 556 p.

## Перечень пользовательских процедур и функций для системы Mathematica, рассматриваемых в настоящей книге. Некоторые полезные рекомендации по использованию средств, которые представлены в монографии и нашем пакете AVZ\_Package\_1

Ниже представлена сводная таблица всех основных программных средств, которые с той либо иной детализацией были рассмотрены в настоящей книге. Данные средства сгруппированы по их основному функциональному назначению; между тем, данная группировка в значительной мере условна, ибо многие средства многофункциональны. Представленные средства использовались как в рамках самого пакета AVZ\_Package\_1, так и при программировании целого ряда приложений различного назначения. При этом, целый ряд средств имеют несколько модификаций, отличающихся как набором функций, так и используемыми алгоритмами. Причины этого носят как временной, так и познавательный характер, а именно. Первое обусловлено различным временем создания однотипных средств при условии появления новых средств, тогда как второе служит целям ознакомления с некоторыми полезными приемами программирования в среде системы Mathematica. Нижеследующая таблица представляет перечень всех средств, рассмотренных в настоящей книге и содержащихся в пакете AVZ\_Package\_1; при этом, для удобства читателя каждое средство идентифицировано как его именем, так и номером страницы, содержащей его исходный код с примерами использования.

Программные средства пакета AVZ_Package_1 [120]						
№	Функция, процедура, глобальная переменная	стр.		№	Функция, процедура, глобальная переменная	стр.
Средства для работы со строчными структурами и символами						
1	Border	99		36	StrToList	112
2	CharacterQ	112		37	StringDependAllQ	413
3	Characters1	71		38	StringDependQ	100
4	DelSubStr	94		39	StringJoin1	262
5	DelSuffPref	102		40	StringLevels	110
6	ExprOfStr	97		41	StringMultiple	110
7	ExprOfStr1	98		42	StringPosition1	96
8	ExprQ	111		43	StringReplace1	90
9	ExtrExpr	99		44	StringReplace2	90
10	ExtrName	101		45	StringReplace3	335
11	ExtrVarsOfStr	276		46	StringReplaceS	101
12	InsertN	93		47	StringSplit1	103
13	IsMonotonic	100		48	StringSplit2	104
14	IsPermutation	100		49	StringStringQ	110

15	<i>LeftFold</i>	115	50	<i>StringTake1</i>	92
16	<i>ListStrToStr</i>	111	51	<i>StringTake2</i>	93
17	<i>LongestCommonSubSequence</i>	100	52	<i>StringTrim1</i>	109
18	<i>LongestCommonSubString</i>	100	53	<i>SubDelStr</i>	92
19	<i>MaximalPalindromicSubstring</i>	100	54	<i>SubStr</i>	97
20	<i>OverLap</i>	100	55	<i>SubStrSymbolParity</i>	105
21	<i>PrefCond</i>	103	56	<i>SubStrSymbolParity1</i>	106
22	<i>PrefixQ</i>	153	57	<i>SubsBstr</i>	104
23	<i>RedSymbStr</i>	108	58	<i>SubsDel</i>	91
24	<i>RightFold</i>	115	59	<i>SubsPosSymb</i>	111
25	<i>SortQ</i>	114	60	<i>SubsStr</i>	104
26	<i>SortRevStr</i>	114	61	<i>SubsStrLim</i>	107
27	<i>SortString</i>	95	62	<i>SubsStrLim1</i>	108
28	<i>Spos</i>	89	63	<i>SubsString</i>	429
29	<i>StrDelEnds</i>	93	64	<i>SubsString1</i>	451
30	<i>StrExprQ</i>	84	65	<i>SuffPref</i>	89
31	<i>StrFromStr</i>	102	66	<i>SuffixQ</i>	99
32	<i>StrOfSymb1Q</i>	92	67	<i>SymbolQ</i>	46
33	<i>StrStr</i>	89	68	<i>SymbolQ1</i>	46
34	<i>StrSub</i>	103	69	<i>ToString1</i>	95
35	<i>StrSymbParity</i>	106	70	<i>ToString2</i>	96
<i>Средства для работы со списочными структурами</i>					
71	<i>ArrayInd</i>	70	99	<i>MaxNestLevel</i>	126
72	<i>AssignL</i>	308	100	<i>MinusList</i>	131
73	<i>AssignToList</i>	124	101	<i>MinusList1</i>	131
74	<i>BinaryListQ</i>	123	102	<i>MultEntryList</i>	141
75	<i>DelEl</i>	128	103	<i>NestListQ</i>	122
76	<i>ElemLevelsL</i>	125	104	<i>NestListQ1</i>	122
77	<i>ElemLevelsN</i>	125	105	<i>NestQL</i>	122
78	<i>ElemOnLevels</i>	142	106	<i>ParVar</i>	131
79	<i>ElmsList</i>	143	107	<i>PosIntListQ</i>	123
80	<i>GroupIdentMult</i>	128	108	<i>PosListTest</i>	458
81	<i>IntegerListQ</i>	123	109	<i>PosSubList</i>	132
82	<i>ListAppValue</i>	127	110	<i>ReduceList</i>	144
83	<i>ListAssign</i>	121	111	<i>ReplaceSubLists</i>	434



*Избранные системные задачи в программной среде Mathematica*

84	ListAssignP	120		112	SelectPos	123
85	ListExprHeadQ	123		113	SortLpos	135
86	ListListGroup	129		114	SortNL	135
87	ListListQ	122		115	SortNL1	135
88	ListNumericQ	123		116	SortNestList	136
89	ListOp	118		117	Split1	142
90	ListPosition	124		118	SplitList	144
91	ListRulesQ	79		119	SplitList1	470
92	ListStrList	50		120	SubLists	125
93	ListStringQ	448		121	SubListsMin	427
94	ListSymbolQ	123		122	SubsList	434
95	ListToSeq	48		123	SymbolToList	132
96	ListToString	130		124	Table1	490
97	ListsAssign	127		125	ToList	67
98	MaxLevel	126		126	TransListList	130
<i>Средства для работы с последовательностными структурами</i>						
127	SEQ	83		133	SeqToString	49
128	SeqDel	49		134	SeqUnion	49
129	SeqIns	48		135	SequenceQ	49
130	SeqQ	48		136	Sequences	50
131	SeqToList	48		137	Sq	50
132	SeqToList1	48				
<i>Средства для работы с процедурами (Module, Block) и функциями</i>						
138	\$CallProc	305		202	\$InBlockMod	336
139	\$ProcName	314		203	Locals	260
140	\$ProcType	304		204	Locals1	263
141	\$TestArgsTypes	251		205	Locals2	264
142	\$TypeProc	305		206	LocalsGlobals	272
143	ActBFMuserQ	215		207	LocalsGlobals1	273
144	ActCsProcFunc	332		208	LocalsGlobalsM	274
145	Args	243		209	MdP	228
146	ArgsBFM	245		210	ModToPureFunc	194
147	ArgsTypes	246		211	ModuleQ	204
148	Arity	247		212	ModuleQ1	198
149	ArityBFM	248		213	ModuleQ2	199

150	<i>Avg</i>	299	214	<i>NamesProc</i>	320
151	<i>BFMSubsQ</i>	315	215	<i>OptDefinition</i>	179
152	<i>BlockFuncModQ</i>	202	216	<i>OptRes</i>	182
153	<i>BlockFuncModVars</i>	275	217	<i>PartProc</i>	317
154	<i>BlockModQ</i>	203	218	<i>ProcActCallsQ</i>	323
155	<i>BlockQ</i>	198	219	<i>ProcBMQ</i>	206
156	<i>BlockQ1</i>	198	220	<i>ProcBody</i>	316
157	<i>BlockToModule</i>	201	221	<i>ProcCalls</i>	300
158	<i>CallListable</i>	282	222	<i>ProcContent</i>	324
159	<i>CallsInProc</i>	331	223	<i>ProcFuncBlQ</i>	200
160	<i>ClearAllAttributes</i>	241	224	<i>ProcFuncCS</i>	325
161	<i>CompActPF</i>	306	225	<i>ProcFuncTypeQ</i>	212
162	<i>CompActPF1</i>	306	226	<i>ProcQ</i>	205
163	<i>CompileFuncQ</i>	210	227	<i>ProcQ1</i>	205
164	<i>CsProcsFuncs</i>	332	228	<i>ProcsAct</i>	319
165	<i>CsProcsFuncs1</i>	332	229	<i>PureFuncQ</i>	211
166	<i>DefFunc</i>	176	230	<i>QBlockMod</i>	264
167	<i>DefFunc3</i>	178	231	<i>QFunction</i>	208
168	<i>DefOpt</i>	178	232	<i>QFunction1</i>	209
169	<i>DefOptimum</i>	181	233	<i>QmultiplePF</i>	223
170	<i>Default1</i>	288	234	<i>RealProcQ</i>	197
171	<i>DefaultValues1</i>	290	235	<i>RedundantLocals</i>	266
172	<i>Defaults</i>	285	236	<i>RedundantLocalsM</i>	266
173	<i>DefaultsM</i>	287	237	<i>RemProcOnHead</i>	227
174	<i>DefaultsQ</i>	291	238	<i>RemProcOnHead1</i>	230
175	<i>DelRestPF</i>	326	239	<i>RemovePF</i>	330
176	<i>DelRestPF1</i>	326	240	<i>RenBlockFuncMod</i>	297
177	<i>DuplicateLocalsQ</i>	268	241	<i>RenameH</i>	334
178	<i>ExpArgs</i>	303	242	<i>ReplaceProc</i>	294
179	<i>ExpLocals</i>	265	243	<i>ReplaceProcBody</i>	317
180	<i>ExtrCall</i>	321	244	<i>ScanLikeProcs</i>	301
181	<i>ExtrNames</i>	272	245	<i>StandHead</i>	234
182	<i>ExtrProcFunc</i>	234	246	<i>StructProcFunc</i>	115
183	<i>FunCompose</i>	325	247	<i>SubProcs</i>	309
184	<i>FuncBlockModQ</i>	199	248	<i>SubProcs1</i>	310



*Избранные системные задачи в программной среде Mathematica*

185	<i>FuncToPure</i>	192		249	<i>SubProcs2</i>	311
186	<i>FunctionQ</i>	211		250	<i>SubProcs3</i>	312
187	<i>GlobalToLocal</i>	278		251	<i>SubsProcQ</i>	314
188	<i>GlobalToLocalM</i>	279		252	<i>SubsProcs</i>	317
189	<i>Globals</i>	270		253	<i>SyntCorProcQ</i>	295
190	<i>Globals1</i>	271		254	<i>SysFuncQ</i>	208
191	<i>Globals2</i>	272		255	<i>SysFuncQ1</i>	208
192	<i>GotoLabel</i>	328		256	<i>SysUserSoft</i>	296
193	<i>HeadName</i>	226		257	<i>TestArgsCall</i>	253
194	<i>HeadPF</i>	222		258	<i>TestArgsTypes</i>	249
195	<i>HeadToCall</i>	227		259	<i>TestArgsTypes1</i>	250
196	<i>HeadingQ</i>	219		260	<i>TestFactArgs</i>	253
197	<i>HeadingQ1</i>	220		261	<i>TestProcCalls</i>	322
198	<i>HeadingQ2</i>	221		262	<i>ToDefOptPF</i>	181
199	<i>HeadingQ3</i>	221		263	<i>UprocQ</i>	207
200	<i>Headings</i>	224		264	<i>Uprocs</i>	320
201	<i>HeadingsPF</i>	225				
<i>Средства для работы с пакетами и документами (т- и пб-файлами)</i>						
265	<i>AcNb</i>	471		303	<i>MfilePackageQ</i>	413
266	<i>ActUcontexts</i>	433		304	<i>MfileToMx</i>	466
267	<i>AddMxFile</i>	475		305	<i>ModLibraryPath</i>	504
268	<i>Aobj</i>	341		306	<i>MxFileToMfile</i>	465
269	<i>CALL</i>	483		307	<i>MxPackNames</i>	449
270	<i>CALLmx</i>	485		308	<i>MxToTxt</i>	431
271	<i>CALLmxH</i>	487		309	<i>MxToTxt1</i>	432
272	<i>CNames</i>	400		310	<i>MxToTxt2</i>	432
273	<i>ContMxFile</i>	417		311	<i>NamesCS</i>	436
274	<i>ContOfContex</i>	437		312	<i>NamesContext</i>	438
275	<i>ContentOfMfile</i>	428		313	<i>NamesFromMx</i>	467
276	<i>ContentOfMfile1</i>	429		314	<i>NamesFromMx1</i>	468
277	<i>ContextActQ</i>	440		315	<i>NamesFromMx2</i>	469
278	<i>ContextFromFile</i>	431		316	<i>NamesMPackage</i>	455
279	<i>ContextMXfile</i>	430		317	<i>NamesNbPackage</i>	454
280	<i>ContextMfile</i>	407		318	<i>NamesNbPackage1</i>	455
281	<i>ContextNBfile</i>	408		319	<i>NbCallProc</i>	65

282	<i>ContextQ</i>	439		320	<i>NbName</i>	471
283	<i>Contexts1</i>	440		321	<i>Need</i>	471
284	<i>CurrentPackageQ</i>	420		322	<i>Npackage</i>	437
285	<i>DefFromM</i>	428		323	<i>PackNames</i>	450
286	<i>DefFromPackage</i>	422		324	<i>PackageFileQ</i>	477
287	<i>DefInPackage</i>	413		325	<i>PackageMxCont</i>	420
288	<i>DefWithContext</i>	443		326	<i>PackageQ</i>	418
289	<i>DumpSaveP</i>	418		327	<i>Packages</i>	420
290	<i>ExtrFromNBfile</i>	456		328	<i>RedMfile</i>	458
291	<i>ExtrOfMfile</i>	426		329	<i>RedMxFile</i>	463
292	<i>ExtrPackName</i>	453		330	<i>ReloadPackage</i>	441
293	<i>FindFileContext</i>	406		331	<i>ReloadPackage1</i>	442
294	<i>FindFileContext1</i>	406		332	<i>RemovePackage</i>	415
295	<i>FullCalls</i>	423		333	<i>RestoreDelPackage</i>	396
296	<i>FullCalls1</i>	424		334	<i>SystemPackages</i>	408
297	<i>HelpBasePac</i>	499		335	<i>ToContextPath</i>	411
298	<i>HelpPrint</i>	499		336	<i>UpdateContextPaths</i>	449
299	<i>LoadMyPackage</i>	444		337	<i>UpdatePackages</i>	448
300	<i>LoadNameFromM</i>	452		338	<i>UpdatePath</i>	448
301	<i>LoadPackage</i>	438		339	<i>VizContentsNB</i>	470
302	<i>MathPackages</i>	397		340	<i>VizContext</i>	440
<i>Средства, расширяющие стандартные функции системы Mathematica либо ее программную среду в целом</i>						
341	<i>\$Line1</i>	37		422	<i>Map13</i>	159
342	<i>ActRemObj</i>	47		423	<i>Map14</i>	160
343	<i>Affiliate</i>	45		424	<i>Map15</i>	161
344	<i>AtomicQ</i>	84		425	<i>Map16</i>	161
345	<i>Attributes1</i>	176		426	<i>Map17</i>	79
346	<i>AttributesH</i>	281		427	<i>Map2</i>	156
347	<i>AttributesQ</i>	280		428	<i>Map3</i>	156
348	<i>BitGet1</i>	146		429	<i>Map4</i>	157
349	<i>BitSet1</i>	146		430	<i>Map5</i>	157
350	<i>Bits</i>	145		431	<i>Map6</i>	157
351	<i>CallQ</i>	335		432	<i>Map7</i>	157
352	<i>CatN</i>	99		433	<i>Map8</i>	158

*Избранные системные задачи в программной среде Mathematica*

353	<i>Clear1</i>	43		434	<i>Map9</i>	158
354	<i>ClearOut</i>	38		435	<i>MapInSitu</i>	149
355	<i>ClearValues</i>	41		436	<i>MapInSitu1</i>	149
356	<i>ComplexQ</i>	319		437	<i>MapInSitu2</i>	149
357	<i>ContextDef</i>	402		438	<i>Mapp</i>	155
358	<i>ContextSymbol</i>	434		439	<i>Mapp1</i>	155
359	<i>ContextToSymbol</i>	434		440	<i>MaxParts</i>	168
360	<i>CountOptions</i>	292		441	<i>MemberLN</i>	141
361	<i>D1</i>	56		442	<i>MemberQ1</i>	140
362	<i>DO</i>	166		443	<i>MemberQ2</i>	140
363	<i>Decomp</i>	150		444	<i>MemberQ3, MemberQ4</i>	140
364	<i>Def1</i>	175		445	<i>MemberT</i>	141
365	<i>DefAttributesH</i>	281		446	<i>MixCaseQ</i>	112
366	<i>DefOnHead</i>	231		447	<i>Names1</i>	152
367	<i>DefOp</i>	39		448	<i>Nvalue</i>	42
368	<i>Definition1</i>	172		449	<i>Nvalue1</i>	42
369	<i>Definition2</i>	173		450	<i>OP</i>	73
370	<i>Definition3</i>	173		451	<i>ObjType</i>	153
371	<i>Definition4</i>	302		452	<i>Op</i>	73
372	<i>DeleteOptsAttr</i>	293		453	<i>PalindromeQ</i>	100
373	<i>Df</i>	76		454	<i>PartialSums</i>	137
374	<i>Df1</i>	77		455	<i>PatternQ</i>	236
375	<i>Df2</i>	77		456	<i>PosIntQ</i>	123
376	<i>Diff</i>	79		457	<i>ProtectedQ</i>	66
377	<i>DumpSave1</i>	433		458	<i>PureDefinition</i>	233
378	<i>DumpSave2</i>	435		459	<i>Range1</i>	138
379	<i>EquExprPatt</i>	238		460	<i>Range2</i>	138
380	<i>EquExprPatt1</i>	239		461	<i>Range3</i>	138
381	<i>ExprComp</i>	85		462	<i>Range4</i>	139
382	<i>ExprOnLevels</i>	58		463	<i>RemoveNames</i>	152
383	<i>ExprPatternQ</i>	237		464	<i>Rename</i>	62
384	<i>ExpressionQ</i>	84		465	<i>RepStandFunc</i>	241
385	<i>ExprsInStrQ</i>	84		466	<i>Replace1</i>	55
386	<i>FormalArgs</i>	335		467	<i>Replace2</i>	56
387	<i>FreeQ1</i>	60		468	<i>Replace3</i>	61

388	<i>FreeQ2</i>	60		469	<i>Replace4</i>	81
389	<i>FullFormF</i>	59		470	<i>ReplaceAll1</i>	77
390	<i>GC</i>	135		471	<i>ReplaceAll2</i>	81
391	<i>GV</i>	83		472	<i>ReplaceOut</i>	39
392	<i>Gather1</i>	134		473	<i>RevRules</i>	113
393	<i>Gather2</i>	134		474	<i>RhsLhs</i>	67
394	<i>GenRules</i>	147		475	<i>RhsLhs1</i>	68
395	<i>GenRules1</i>	148		476	<i>Rule1</i>	169
396	<i>GroupNames</i>	151		477	<i>RuleQ</i>	294
397	<i>Head1</i>	51		478	<i>SaveInMx</i>	476
398	<i>Head2</i>	214		479	<i>SetAttributes1</i>	242
399	<i>Head3</i>	214		480	<i>SortLS</i>	137
400	<i>HowAct</i>	45		481	<i>Subs</i>	75
401	<i>Iff</i>	163		482	<i>SymbolGreater</i>	113
402	<i>Ifk</i>	164		483	<i>SymbolLess</i>	114
403	<i>Ifk1</i>	164		484	<i>SyntaxLength1</i>	52
404	<i>Ind</i>	73		485	<i>SyntaxQ1</i>	52
405	<i>Index</i>	70		486	<i>SystemQ</i>	223
406	<i>IndexQ</i>	72		487	<i>ToStringRule</i>	55
407	<i>IndexedQ</i>	70		488	<i>ToStringRule1</i>	55
408	<i>Indices</i>	72		489	<i>Tuples1</i>	133
409	<i>Int</i>	78		490	<i>TwoHandQ</i>	69
410	<i>Int1</i>	78		491	<i>TypeActObj</i>	154
411	<i>Integral</i>	80		492	<i>UnDef</i>	44
412	<i>Integrate1</i>	56		493	<i>UnDefVars</i>	54
413	<i>Levels</i>	58		494	<i>UnDefVars1</i>	54
414	<i>ListStrQ</i>	120		495	<i>UnevaluatedQ</i>	171
415	<i>ListableC</i>	119		496	<i>VarExch</i>	62
416	<i>ListableQ</i>	117		497	<i>VarExch1</i>	63
417	<i>LocObj</i>	151		498	<i>VarsValues</i>	41
418	<i>Map1</i>	156		499	<i>WhatObj</i>	45
419	<i>Map10</i>	159		500	<i>WhatType</i>	74
420	<i>Map11</i>	470		501	<i>WhatValue</i>	43
421	<i>Map12</i>	159		502	<i>WhichN</i>	165
<i>Средства для работы с файловой системой компьютера</i>						
503	<i>Adrive</i>	338		532	<i>FileOpenQ1</i>	354

*Избранные системные задачи в программной среде Mathematica*

504	<i>Adrive1</i>	339		533	<i>FileQ</i>	384
505	<i>Attrib</i>	365		534	<i>FilesDistrDirs</i>	387
506	<i>Attrib1</i>	367		535	<i>FindFile1</i>	382
507	<i>Attribs</i>	372		536	<i>FindFileObject</i>	387
508	<i>BootDrive</i>	377		537	<i>FindSubDir</i>	386
509	<i>CDir</i>	364		538	<i>FreeSpaceVol</i>	377
510	<i>ClearRecycler</i>	395		539	<i>IsFile</i>	384
511	<i>Close1</i>	375		540	<i>IsFileOpen</i>	379
512	<i>CloseAll</i>	350		541	<i>LoadExtProg</i>	369
513	<i>Closes</i>	349		542	<i>LoadFile</i>	344
514	<i>CopyDir</i>	364		543	<i>MathematicaDF</i>	345
515	<i>DelAllAttribs</i>	373		544	<i>Memory</i>	391
516	<i>DelDirFile</i>	374		545	<i>Nobj</i>	340
517	<i>DelDirFile1</i>	376		546	<i>OpenFiles</i>	378
518	<i>DeleteFile1</i>	395		547	<i>PathToFileQ</i>	388
519	<i>DirEmptyQ</i>	380		548	<i>ReadFullFile</i>	388
520	<i>DirFD</i>	380		549	<i>RenDirFile</i>	391
521	<i>DirFull</i>	381		550	<i>Save1</i>	342
522	<i>DirName</i>	364		551	<i>SearchDir</i>	383
523	<i>DirQ</i>	363		552	<i>SearchFile</i>	371
524	<i>EmptyFileQ</i>	385		553	<i>SetDir</i>	338
525	<i>ExtProgExe</i>	389		554	<i>SetDir1</i>	339
526	<i>FileDirStForm</i>	393		555	<i>SetPathSeparator</i>	392
527	<i>FileExistsQ1</i>	363		556	<i>StandPath</i>	393
528	<i>FileFormat1</i>	347		557	<i>StreamFiles</i>	378
529	<i>FileFormat2</i>	356		558	<i>StreamsU</i>	349
530	<i>FileFormat3</i>	357		559	<i>TypeFilesD</i>	381
531	<i>FileOpenQ</i>	353		560	<i>Ver</i>	372
<i>Некоторые средства специального назначения</i>						
561	<i>ClearCS</i>	302		567	<i>Un</i>	256
562	<i>Cost</i>	86		568	<i>Unique1</i>	256
563	<i>Nconcat</i>	168		569	<i>UsageBase</i>	494
564	<i>ReductRes</i>	298		570	<i>Usages</i>	497
565	<i>TabLib</i>	491		571	<i>UsagesMNb</i>	500
566	<i>Try</i>	162		572	<i>UserLib</i>	488

Довольно детальное описание средств представленного списка может быть найдено в пакете для *Mathematica* [120]. При этом, следует отметить, что исходные коды средств, приведенные в данной книге, в целом ряде случаев содержат вызовы нестандартных средств, которые не нашли отражения в настоящей монографии, но представлены в [120]. Поэтому для их детального анализа требуется знакомство с данными средствами, по меньшей мере, на уровне справок по ним. Между тем, основной алгоритм многих процедур книги достаточно хорошо просматривается и без знакомства с подобными средствами. Тогда как реальное использование данных средств возможно лишь после загрузки в текущий сеанс, в частности, файла "*AVZ\_Package\_1.nb*" с последующим его вычислением, имеющего лицензию *FreeWare* (для работы в среде *Mathematica* версий 8 ÷ 9 [120]). Наряду с иллюстративными целями представленные в данной монографии средства вполне могут быть использованы и в качестве достаточно полезных средств, расширяющих программную среду *Mathematica*, которые достаточно существенно облегчают программирование весьма широкого круга задач, прежде всего, носящих системный характер. Большинство этих средств ориентировано именно на массовый характер использования при программировании довольно сложных задач, применяя хорошо апробированную концепцию процедурно-функциональной парадигмы.

В целях упрощения сравнительного анализа систем *Mathematica* и *Maple* была избрана следующая методика – проводилось программирование процедур и функций одного и того же назначения в среде обеих систем с использованием как стандартных, так и созданных нами средств, отсутствующих в стандартных поставках. Именно на такого рода парах проверялась эффективность и удобство разработки программных средств. С нашей точки зрения, целый ряд представленных в этой связи средств представляет определенный интерес в практическом программировании различных приложений, прежде всего, системного характера. Было выявлено, что не взирая на большие, порой, наглядность и привычность исходных кодов однотипных процедур/функций *Maple*, программная среда *Mathematica* располагает как более развитыми и эффективными, так и более реактивными средствами разработки как таких, так и целого ряда других приложений. Особо следует отметить значительно большую реактивность системы, что во многих приложениях играет определяющую роль. Стоит также отметить и больше возможностей у программной среды *Mathematica* для развития собственно ее самой в рамках аксиоматики составляющих ее стандартных средств. Ряд примеров такого типа был представлен и в настоящей книге. Между тем, данная книга не преследует целей детального сравнительного анализа программных сред систем *Mathematica* и *Maple*, а ряд приведенных реализаций в обеих средах носят специальный иллюстративный характер, тогда как с рядом основных аспектов сравнительного анализа программных сред обеих систем можно ознакомиться в наших предыдущих книгах [97-101], которые посвящены сравнительным аспектам процедурно-функционального программирования.

Упомянув наши издания по системам компьютерной математики, целесообразно здесь вкратце пояснить основной принцип их организации и назначения. Прежде всего, по данным показателям в массе своей наши книги ориентированы на уровень достаточно хорошо подготовленного читателя, имеющего опыт работы с системами *Mathematica* и *Maple*. Хотя они и могут представить интерес для начинающих в плане ознакомления

с встроенными языками программирования обоих систем. Прежде всего это касается представленных в книге достаточно представительного набора процедур и функций, которые носят не только сугубо иллюстративный характер, преследуя ряд других не менее важных целей. Прежде всего, средства по обоим средам, в основе своей, носят парный характер (*реализация одного и того же алгоритма в среде обоих систем*). Сам же подбор средств был выбран таким образом, чтобы проиллюстрировать возможности программных сред обоих систем на примерах, прежде всего, системного характера, носящих достаточно массовый характер наподобие массовых стандартных системных средств. Данные средства достаточно прозрачны по назначению и хорошо обозримы, их *исходные* коды хорошо читаемы достаточно подготовленным пользователем. Тогда как для читателя, недостаточно опытного в работе с *исходными* кодами программных средств, для лучшего их восприятия может понадобиться их некое предварительное редактирование. И, если *исходный* код *Maple*-языка довольно хорошо структурирован и легко воспринимается при минимуме комментирования, в случае с *Mathematica* эта задача представляется нам несколько более сложной. В любом случае навык работы с исходным кодом в среде обоих систем представляется весьма важной предпосылкой качественного освоения программирования в их среде; где под *программированием* мы понимаем программирование продуктов, отличных от простых процедур. И здесь на ум приходит тот опыт работы с исходными кодами программ на ассемблере, который во многом способствовал появлению в свое время программистов высокого класса. В определенной степени можно сказать, чтение *листингов* программ определяет *уровень* программиста, которое сродни умению читать *партитуру* хорошим дирижером. При этом, представленные здесь примеры иллюстрируют целый ряд достаточно полезных как стандартных, так и нестандартных приемов программирования процедур, вместе с тем допустимых программной средой системы, в целом ряде случаев позволяющих программировать средства более компактными, прозрачными и эффективными.

Наконец, ряд представленных средств предназначен как для *расширения* стандартных средств пакетов, так и для устранения их недостатков и ошибок. Данные средства не только более четко акцентируют различия обоих систем, но и их общие проблемы. И в данном отношении они позволяют с различных позиций взглянуть на те либо иные как преимущества, так и недостатки обоих программных сред. При этом, необходимо отметить, что массовой оптимизации процедур не проводилось, процедуры в многих случаях писались, как говорится с «*листа*»; с другой стороны, немало процедур было оптимизировано, используя как стандартные, так и созданные системные средства. В данном контексте здесь имеется прекрасное экспериментальное поле для повышения профессионализма пользователя в работе с программной средой *Mathematica*.

Включение исходных кодов процедур и функций, представленных в данной книге, с их краткой характеристикой непосредственно в сам текст книги позволяет работать с ними без компьютера, учитывая привычку немалого числа пользователей старшего поколения работать с листингами программ перед тем как выйти на компьютер, что в целом ряде случаев способствовало более качественному программированию. Более того, многие листинги представленных средств имеют достаточно небольшой объем, позволяя анализировать их вне среды *Mathematica* в предположении, что читатель в

достаточной степени знаком с ее программной средой. Между тем, даже небольшие в объеме исходного кода средства зачастую являются полезными в программировании различных приложений, например, системного характера. Тогда как другие требуют для своего понимания серьезной проработки, включая знакомство с нашим пакетом.

Как показывает наш опыт, программирование в таком режиме несколько медленнее, чем непосредственно за компьютером, однако он позволяет лучше сосредоточиться на объекте программирования и лучше продумать проблему и путь ее решения, скорее, чем метод ее решения в интерактивном режиме. Даже при наличии ПК мы привыкли основной костяк программы писать на бумаге и лишь затем переходить к отладке на ПК в интерактивном режиме. На наш взгляд, такой подход позволяет более вдумчиво писать программы; при этом, следуя давней привычке писать оптимальные коды для последующего их выполнения на довольно ограниченных вычислительных ресурсах, которыми обладали компьютеры 20–30 лет тому назад. Впрочем, во многом это дело привычки, однако не стоит забывать, что старое не всегда хуже нового, и приобретая новые возможности, мы, зачастую, теряем старые навыки, важные для работы. Вот и в этом случае, получив весьма удобное средство общения, мы, порой, упускаем из вида эффективность программного кода, создавая его без особых затруднений в режиме диалога с единственной целью получить требуемый результат, игнорируя качество.

Конечно, не существует единственного самого лучшего способа создания программ. Для решения задач разного рода и уровня сложности требуется применять и разные технологии и парадигмы программирования. Так, в простейших случаях достаточно освоить азы структурного написания программ. Для создания же сложных проектов требуется не только свободно владеть языком программирования в полном объеме, но также иметь представление о принципах проектирования и отладки программ, возможностях стандартной и других библиотек той либо иной программной среды, и т. д. Как правило, чем сложнее задача, тем больше времени требуется на освоение инструментов, необходимых для ее решения. В данном контексте представленные в настоящей книге программные средства (*процедуры/функции*), как содержащие целый ряд полезных и эффективных приемов программирования в среде *Mathematica*, так и расширяющие ее программную среду, предоставляют возможность более просто и эффективно программировать различные задачи. Средства в процессе применения пакета обновляются с учетом как новых средств, так и оптимизации существующих.

Для системы *Mathematica* исходные коды более 580 процедур/функций находятся в 4 файлах *AVZ\_Package\_1.nb*, *AVZ\_Package\_1.cdf*, *AVZ\_Package\_1.m*, *AVZ\_Package\_1.mx*; первые 2 файла с *Mathematica* документом можно прямо загружать в текущий сеанс системы по цепочке функций *GUI: Evaluation → Evaluation Notebook*, получая доступ ко всем средствам, содержащимся в нем, наряду с краткой справкой по ним, а также и получать исходный код средств на экран ПК. Третий файл с пакетом *AVZ\_Package.m* загружается в текущий сеанс *Mathematica* описанным выше способом. Такой подход позволяет удовлетворить пользователя с различными используемыми платформами. Файл *AVZ\_Package.mx* является бинарным файлом, оптимизированным под загрузку на платформе, на которой он был создан функцией *DumpSave*, рассмотренной выше. В предлагаемой версии архива *AVZ\_Package.zip* файл "*AVZ\_Package\_1.mx*" с пакетом



был создан на платформе *Ms Windows XP*, однако это ограничение несущественно, т. к. архив *AVZ\_Package.zip* содержит три остальных файла, легко читаемые на любой платформе, и уже на их основе несложно создать файл "*AVZ\_Package\_1.mx*" с пакетом на текущей платформе вышеописанным способом, оптимизирую и упрощая загрузку в текущий сеанс пакета. Оценка требуемой памяти под данный пакет (по состоянию на 22.11.2013 на платформе *Windows XP Version 5.1.2600*) дает следующий результат:

```
In[1]:= MemoryInUse[]
Out[1]= 22 103 084
In[2]:= Get["C:\\AVZ_Package\\AVZ_Package_1.mx"]
In[3]:= MemoryInUse[]
Out[3]= 30 457 580
In[4]:= % - %%%
Out[4]= 8 354 496
```

т.е. в рабочей области *Mathematica* наш пакет *AVZ\_Package\_1* занимает почти 8 МВ, а количество программных средств, определения которых составляют данный пакет, на текущий момент при условии его загрузки в текущий сеанс *Mathematica* доступно на основе следующих чрезвычайно простых вычислений, а именно:

```
In[2941]:= Get["C:\\AVZ_Package\\AVZ_Package_1.mx"]
In[2942]:= Length[CNames["AladjevProcedures`"]]
Out[2942]= 582
```

И последнее. Все представленные в книге средства прошли тестирование в наиболее типичных для них режимах выполнения – результаты их вызовов сопровождают все их исходные коды. Там же приводится краткая характеристика каждого конкретного средства с допустимыми выражениями в качестве его фактических аргументов. В то же время, при вызовах некоторых средств возможно возникновение ошибочных либо особых ситуаций с выводом соответствующих сообщений. В основе данных ситуаций вполне может лежать тот факт, что вызвавшие их средства используют наши средства, подвергнутые с той либо иной целью модификации. Наиболее часто используемые в программировании средства пакета *AVZ\_Package* были апробированы после данного типа модификаций, однако вполне возможно, что некоторые были упущены из вида. В принципе, особо страшного здесь нет, т.к. исходные коды достаточно прозрачны, не объемны и вполне по силам довольно опытному пользователю исправить их с учетом сказанного либо запрограммировать самому на основе алгоритма, реализуемого этим средством, благо постановочная составляющая имеется, а этого уже немало. В то время как в качестве довольно полезного практического упражнения на программирование в среде *Mathematica* это представляется нам довольно важным. Наш опыт проведения различного уровня мастер-классов по системам и *Mathematica*, и *Maple* подтверждает целесообразность применения совместно со стандартными средствами обеих систем и некоторого инструментария пользователя, созданного в процессе программирования своих приложений. Надеемся, что наша методика предоставления материала книги в определенной степени найдет понимание у читателя. В заключение отметим, данная книга завершает нашу серию изданий по системам компьютерной математики ввиду возобновления исследований по одному из разделов математической кибернетики.

## Summary

***V.Z. Aladjev, D.S. Grinn, V.A. Vaganov. The selected system problems in Mathematica software.– Ukraine: Kherson, Oldi-Plus Press, 2013, 556 p.***

The computer mathematics has found application in many fields of science such as physics, mathematics, chemistry, computer sciences, engineering, education, computational biology, technology, etc. The computer mathematics systems (CMS) such as *Mathematica* & *Maple* are becoming more and more popular in teaching, research and industry. Researchers use well-known system *Mathematica* as enough essential tool for solving problems related to their investigations. The system is ideal feature for formulating, solving, and exploring the different mathematical models. Its symbolic manipulation facilities extend greatly over a range of the problems that can be solved with its help. At that, educators in universities and colleges have revitalized traditional curricula by introducing problems and exercises, which widely use *Mathematica*'s interactive mathematics and physics. At the same time students can concentrate on the more fundamental concepts rather than on plural tedious algebraic manipulations. Finally, engineers & experts in industries use the system *Mathematica* as an efficient facility replacing many traditional resources such as reference books, spreadsheets, calculators, and programming languages. These users easily solve mathematical problems, creating projects and consolidating their computations into professional report. Meanwhile, our experience with system *Mathematica* of releases 7÷9 enabled us not only to estimate its advantages in regard to other similar CMS, first of all, system *Maple*, but has also revealed a number of faults and shortcomings that were eliminated by us. Finally, *Mathematica* does not support a number of functions important for procedural programming. Therefore, our package *AVZ\_Package* substantially solves this problem. The package contains more than 580 procedures, functions and global variables that both eliminate restrictions of a number of standard functions of the *Mathematica*, and expand its software environment with new means. In this context, this package can serve as a tool of modular programming, especially useful in numerous applications where some nonstandard evaluations have to accompany programming. The archive *AVZ\_Package.zip* with the given package that owns the license *FreeWare* can be freely downloaded from the web-site <http://yadi.sk/d/quWOB3fe9kRnf>.

The given book considers certain principal questions of procedure-functional programming in *Mathematica*, not only for the decision of applied problems, but, first of all, for creation of the software expanding frequently used facilities of the system and/or eliminating their defects or expanding the system with new facilities. The software represented in this book contains a series of useful and effective receptions of programming in *Mathematica*, and expands its software which enables more simply and effectively to programme in system *Mathematica* the problems of various purpose. Contents of the presented book in context of its chapters and sections can be presented as follows, namely:

### Preface

**Chapter 1.** Brief excursus into the history of computer mathematics systems

**Chapter 2.** The general information concerning the builtin programming language of the *Mathematica* system (*the Math-language*)

- Chapter 3.** Means of work in a mode of the current *Mathematica* document
- Chapter 4.** Additional means of expressions processing in the *Mathematica* software
- Chapter 5.** Additional means of processing of symbols and strings in the *Mathematica*
- Chapter 6.** Additional means of processing of sequences and lists in the *Mathematica*
- Chapter 7.** The additional means expanding the standard *Mathematica* software
- Chapter 8.** Problems of procedural programming in the *Mathematica* software
- 8.1. Definition of procedures in the *Mathematica* software
- 8.2. Definition of functions and pure functions of the user in the *Mathematica* software
- 8.3. Means of testing of procedures and functions in the *Mathematica* software
- 8.4. Headings of procedures and functions in the *Mathematica* software
- 8.5. Formal arguments of procedures and functions; the means of processing them in the *Mathematica* software
- 8.6. Local variables of modules and blocks; the means of manipulation by them in the *Mathematica* software
- 8.7. Global variables of modules and blocks; the means of manipulation by them in the *Mathematica* software
- 8.8. Attributes, options and values by default for arguments of the user blocks, functions and modules; additional means of processing of them in *Mathematica* software
- 8.9. Some additional facilities for work with blocks, functions and modules in the *Mathematica* software
- Chapter 9.** Means of input-output of the *Mathematica* software
- 9.1. Facilities of the *Mathematica* system for work with internal datafiles
- 9.2. Facilities of the *Mathematica* system for work with external datafiles
- Chapter 10.** The manipulation organization with packages in the *Mathematica* software
- 10.1. Concept of the context, and its use in the *Mathematica* software
- 10.2. Definition of packages, and their use in the *Mathematica* software
- 10.3. Additional means of manipulation with packages in the *Mathematica* software
- Chapter 11.** The organization of the user software in the *Mathematica* software
- 11.1. The organization of the user software in the *Mathematica* software
- 11.2. Programming of large systems in the *Mathematica* software
- 11.3. System *Mathematica* or *Maple* – a subjective point of view

## References

*Monographs, textbooks and books on systems Mathematica and Maple, that have been published by members of the International Academy of Noosphere during 1995 – 2013*

*Table of the user means for the Mathematica system, that are considered and mentioned in the book; some useful recommendations on use of the means, presented in the given book*

The presented monograph, is mostly for people who want the more deep understanding in the *Mathematica* programming, and particularly those *Mathematica* users who would like to make a transition from a user to a programmer, or perhaps those who already have some limited experience in *Mathematica* programming but want to improve their possibilities in the system. Expert *Mathematica* programmers will probably find an useful information too.

*В.З. Аладьев, В.А. Ваганов, Д.С. Гринь*

---

*Аладьев Виктор Захарович  
Ваганов Вячеслав Алексеевич  
Гринь Дмитрий Сергеевич*

# *Избранные системные задачи в программной среде Mathematica*

## *Монография*

Оригинал-макет издания подготовлен в Балтийском отделении  
Международной Академии Ноосферы (Таллинн, Эстония)

Формат 7,44 x 9,69. Бумага Офс.  
Тираж 3000.

Издательство ЧП «Олди-плюс»  
e-mail: oldi-ks@i.ua  
73033. г. Херсон, а/я № 15  
Лицензия сер. № 2 от 16.08.2000 г.