

*Международная Академия Ноосферы  
Балтийское отделение*

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

*Дополнительные функциональные  
средства для пакета Mathematica*

*Херсон – 2012*

ББК 22.18

Д 681

ISBN 978-966-2393-59-0

Д 681    Аладьев В.З., Ваганов В.А., Гринь Д.С. **Дополнительные функциональные средства для пакета Mathematica:** Монография / В.З. Аладьев, В.А. Ваганов, Д.С. Гринь.- Х.: Олди-Плюс, 2012. – 404 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 программирования в среде *Mathematica*, расширяя ее программную среду, позволяя более эффективно и просто программировать задачи различного назначения, прежде всего, носящие системный характер. Книга снабжена специальным пакетом с *Freeware* лицензией, содержащим более 320 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 22.18

ISBN 978-966-2393-59-0

© Аладьев В.З., Ваганов В.А.,  
Гринь Д.С., 2012

## Содержание

Предисловие	4
Глава 1. Краткий экскурс в историю систем компьютерной алгебры	11
Глава 2. Общие сведения по встроенному языку программирования пакета <i>Mathematica</i> ( <i>Math-языку</i> )	20
Глава 3. Базовые управляющие структуры программной среды <i>Mathematica</i>	31
3.1. Управляющие структуры ветвления программной среды <i>Mathematica</i>	34
3.2. Циклические управляющие структуры программной среды <i>Mathematica</i>	42
3.3. Специальные типы циклических управляющих структур <i>Mathematica</i>	45
Глава 4. Базовые механизмы типирования объектов в среде <i>Mathematica</i>	48
4.1. Механизмы типирования объектов в программной среде пакета <i>Mathematica</i>	50
4.2. Дополнительные средства тестирования типов объектов в среде <i>Mathematica</i>	57
Глава 5. Обработка особых и ошибочных ситуаций в среде <i>Mathematica</i>	70
Глава 6. Организация механизмов процедур и функций в <i>Mathematica</i>	77
6.1. Средства тестирования процедурных объектов в среде <i>Mathematica</i>	90
6.2. Локальные переменные процедур в среде пакета <i>Mathematica</i>	100
6.3. Глобальные переменные процедур и механизмы возврата результатов процедурами в среде пакета <i>Mathematica</i>	114
6.4. Некоторые полезные средства для работы с <i>Mathematica</i> -процедурами	119
6.5. Функциональные конструкции в среде пакета <i>Mathematica</i>	171
Глава 7. Дополнительные средства работы с базовыми структурами данных и с выражениями в программной среде пакета <i>Mathematica</i>	192
7.1. Дополнительные средства работы со <i>строчными</i> структурами в <i>Math</i> -языке	192
7.2. Дополнительные средства работы со <i>списочными</i> структурами в <i>Math</i> -языке	213
7.3. Дополнительные средства работы со структурами типа <i>последовательность</i>	225
7.4. Дополнительные средства работы с алгебраическими выражениями	228
7.5. Некоторые полезные процедурные средства в среде <i>Mathematica</i>	245
Глава 8. Организация работы с пакетами в среде <i>Mathematica</i>	262
Глава 9. Средства ввода/вывода программной среды <i>Mathematica</i>	297
9.1. Средства пакета <i>Mathematica</i> для работы с внутренними файлами	297
9.2. Средства <i>Mathematica</i> для работы с внешними файлами данных	304
9.3. Некоторые полезные средства доступа в среде пакета <i>Mathematica</i>	330
Глава 10. Организация программного обеспечения пользователя	349
10.1. Организация программного обеспечения пользователя в среде пакета	351
10.2. Программирование больших систем в среде пакета <i>Mathematica</i>	366
10.3. <i>Mathematica</i> или <i>Maple</i> – субъективная точка зрения	373
Литература	391
Перечень пользовательских процедур и функций для пакета <i>Mathematica</i> , рассматриваемых и упоминаемых в настоящей книге	398
Summary	403

## Предисловие

Системы компьютерной алгебры (СКА) 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. Системы такие как *Mathematica*, *Maple*, *REDUCE*, *MuPAD*, *Derive*, *Magma*, *Axiom*, *GAP*, *Maxima*, *MathPiper* и др. все более востребованы для преподавания [20,53,60,82] математически ориентированных дисциплин, в научных исследованиях и технологиях. Эти системы являются основными инструментами для ученых, исследователей, преподавателей и инженеров. Исследования на основе СКА-технологии, как правило, хорошо сочетают алгебраические методы с продвинутыми вычислительными методами. В этом смысле СКА – междисциплинарная область между математикой и информатикой, в которой исследования сосредотачиваются как на разработке алгоритмов для алгебраических (символьных) и численных вычислений и обработки данных, так 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/1*, *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 алгебры*), которые, в первую очередь, предназначены для решения задач математического характера, и, прежде всего, лидерам в данном классе пакетам *Mathematica* фирмы *Wolfram Research Inc.* и *Maple* фирмы *MathSoft Inc.* Книга является вполне естественным продолжением нашей 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 нами уделялось как опыту работы с описываемым средством, так и особенностям его применения, а также вытекающим

из этого рекомендациям пользователю. По мере возможности, предлагались наиболее эффективные технологии применения данных средств для решения тех либо других прикладных задач пользователя. Довольно детальная характеристика данной серии книг может быть найдена, например, 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 пакеты *Maple* и *Mathematica* соответственно фирм *MapleSoft Inc.* и *Wolfram Research Inc.* Эти пакеты, превосходя по целому ряду важных показателей упомянутые средства 5-го уровня, вместе с тем, наследуя целый ряд их стандартов, как пионеров-эталонов *ПС* такого типа, что достаточно легко прослеживается при более детальных их рассмотрении, использовании и апробации.

В среде отечественных пользователей *СКА* наибольшую популярность завоевал пакет *Maple*; в качестве некоторого косвенного подтверждения степени *популярности* обоих пакетов в *СНГ* может служить посещаемость форумов по таким пакетам на наиболее известном русскоязычном образовательном математическом сайте *www.exponenta.ru*. Сегодня на данном сайте форум *Maple* имеет 2385 тем и 10845 сообщений, тогда как форум *Mathematica* лишь 850 тем и 2825 сообщений. Естественно, к такой статистике следует относиться достаточно осторожно, ибо она не совсем верно отражает степень популярности обоих пакетов 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. Может быть, именно по этой причине, несмотря на двадцатилетний возраст *Mathematica*, ее достаточно мощные вычислительные возможности наряду с возможностью установки на такие операционные системы, как *Windows, Linux, Mac OS X* популярность данной *СКА* на постсоветском пространстве относительно невелика. В беседах, проведенных нами на целом ряде мастер-классов по *СКА*, многие слушатели считают *Mathematica* слишком сложной, а в функциональном отношении избыточной. И если с первым, в определенной мере можно согласиться, то вторым пакет *Maple* также злоупотребляет, пусть и не в такой степени. Об этом несколько детальнее речь будет идти ниже. Нами пару раз просматривался целый ряд тем и сообщений на указанном сайте в форумах по обоим пакетам и 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, пожалуй, имеются редкие исключения*), и если *Maple* им как-то удалось освоить на довольно низком уровне, более сложный для восприятия второй пакет для освоения требует существенно больших усилий. В то время как довольно квалифицированные пользователи *СКА Mathematica* и *Maple*, как правило, не посещают форумов данного уровня. Итак, приведенная 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 тот либо иной пакет, сколько даже суть задачи, предложенной им для решения в среде пакета*), хотя она и представляет существенный пласт пользователей *СКА*, правда, во многом только на время учебы по курсам, завязанным на указанные *СКА*. Замечено, во многих случаях вопрошающие на довольно низком уровне владеют постановочной стороной вопроса, а отвечающие не очень сведущи в программной среде данных пакетов. На ряде других форумов по *CAS* ситуация хоть и несколько лучше, однако и статистика на их основе представляется нам весьма ненадежной для анализов подобного рода.

Наш опыт эксплуатации данных *СКА* 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-язычном сокращении*). 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.

Пакет *Mathematica* наряду с уже упомянутым пакетом *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, пакет 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, пакет 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 сбоев и перезапуск остановленных процессов, средства *распараллеливания* встроенного языка пакета, 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*;
- пакет поддерживает встроенный процедурно-функциональный язык, обеспечивая пользователя довольно эффективным средством программирования своих задач.

При этом, *Mathematica* обладает даже большей функциональной *избыточностью*, чем *Maple*. Средства поддержки работы со *звуком* и простейшими *клеточными автоматами* в *Mathematica*, *Socket* в *Maple* и ряд других вполне можно отнести к «*архитектурным*» излишествам пакетов. Между тем, пакет *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 оба пакета не являются универсальными с общепринятой точки зрения

системами программирования, давая возможность пользователю создавать средства, выполнимые только в среде самих пакетов (*т.е. в полной мере невозможно создание exe- и com-файлов*), что существенно сужает область выполнимости данных средств, требуя наличия самих пакетов, для целого ряда пользователей достаточно дорогих.

Пакет *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 пакета – возможностях его 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 пакета и/или устраняющих их недостатки, либо дополняющих пакет новыми средствами. Именно в этом контексте и представляют интерес возможности пакета по созданию такого типа процедур или функций. Итак, программирование в среде пакета – 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 пакета, в программной среде самого пакета, т.е. реализованные средствами встроенного языка пакета (*Math-языка*). Именно в этом контексте и можно в довольно полной мере оценить *программную* среду пакета *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.

Книга состоит из **10** глав и затрагивает довольно обширный материал по *программной* среде пакета *Mathematica* в контексте его возможностей в *процедурно-функциональном* программировании. Между тем, ограниченный объем и сроки подготовки настоящей книги не позволили нам изложить рассматриваемые вопросы с одинаковой степенью полноты. Поэтому, порой, имеет место освещение лишь сути дела без обстоятельного 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, примеры, приведенные в книге, могут быть использованы в качестве довольно полезных средств при разработке собственных приложений в среде пакета *Mathematica*. Для удобства их использования именно в данном качестве читателю предоставляется возможность свободной загрузки содержащего их пакета *AVZ\_Package* для *Mathematica 5 – 8* [90]. В целом, содержимое глав книги может быть охарактеризовано следующим образом.

*Система компьютерной алгебры* (Computer Algebra System – CAS) представляет собой тип программного средства, предназначенного для манипулирования математическими формулами. Основной задачей CAS является автоматизация достаточно громоздких и сложных символьных преобразований. При этом, основное различие между CAS 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 этих систем изменяются в весьма широком диапазоне, все же основная их цель остается неизменной, а именно: обработка символьных выражений. 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 на содержательном уровне представляет общие сведения по встроенному языку программирования *СКМ Mathematica*.

В 3-й главе в сравнительном плане рассматриваются *базовые* управляющие структуры пакетов *Mathematica* и *Maple* в разрезе *управляющих* структур *ветвления, циклических* и *специальных* циклических. Дается определенная оценка механизмов управляющих структур пакетов *Mathematica* и *Maple*. *Управляющие* структуры данного типа играют одну из важнейших ролей в программировании различного рода алгоритмов, посему они играют далеко не последнюю роль и при сравнении программных сред пакетов. Концепция типа данных в языках программирования высокого уровня появилась как совершенно *естественное* отражение факта, что обрабатываемые программой данные и выражения могут иметь различные множества допустимых значений, сохраняться в оперативной памяти компьютера различным способом, обрабатываться различными командами процессора и т.д. Эта концепция является одной из основополагающих в современном программировании. Поэтому, учитывая важность типизации языковых объектов, следует данный аспект рассмотреть и в контексте рассматриваемого пакета *Mathematica*, чему и посвящена *четвертая* глава. При этом, наряду со стандартными средствами рассматриваются также и дополнительные средства тестирования типов объектов в среде *Mathematica*. И именно с точки зрения развитости поддерживаемых им механизмов типизации объектов *Mathematica* рассматривается в сравнительном аспекте с *однотипным* пакетом *Maple*, как одной из наиболее развитых и популярных на сегодня универсальных *CAS*. Между тем, наши книги [97-99] представляют более детальный сравнительный аспект обоих популярных пакетов *Mathematica* и *Maple*.

В процессе выполнения достаточно сложных алгоритмов возникают различного рода особые и ошибочные ситуации, которые для повышения робастности алгоритмов мы должны программно обрабатывать. От успешного решения данной задачи с учетом того, насколько мы сумели предусмотреть программную обработку всех возможных ситуаций, могущих возникать в процессе работы алгоритма, в значительной степени зависит его надежность, устойчивость, эффективность и конкурентноспособность. И в первую очередь это относится к средствам широкого (*массового*) использования, где робастность играет весьма существенную роль. Именно в данном контексте в *пятой* главе рассматриваются *основные* средства, предоставляемые пакетом *Mathematica* для реализации обработки особых и ошибочных ситуаций, которые в значительной мере можно отнести к вопросам робастности программных средств в целом.

Так как основной сравнительный анализ обоих пакетов лежит в области *процедурного* программирования, вопросы поддержки данного типа программирования пакетами представляют первостепенный интерес. Именно поэтому *шестая* глава данной книги посвящена вопросам организации механизмов процедур и функций в *Mathematica*.

Рассматриваются такие вопросы, как: определения процедур и функций, и их типы, формальные и фактические аргументы, локальные и глобальные переменные наряду с механизмами возврата процедурой результатов ее вызова и ряд других. Для пакета *Mathematica* представлен ряд полезных средств работы с процедурами и функциями различных типов. Глава рассматривает также ряд примеров оформления процедур и функций в среде пакета, рассмотрены элементы функциональной парадигмы в среде пакета *Mathematica*, которая существенно отличает его, в частности, от *Maple*.

Подобно пакету *Maple*, *Mathematica* поддерживает достаточно развитые средства для расширения *своих* функциональных возможностей. Действительно, 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 этом случае *Mathematica* предоставляет возможность написания на ее языке специального пакета, содержащего определения требуемых средств. Именно в данном аспекте *седьмая* глава и представляет ряд дополнительных средств работы с базовыми структурами данных и выражениями в программной среде *Mathematica*.

*Mathematica* предоставляет развитые средства для сохранения процедур, функций и других объектов в т.н. пакетах, организацию работы с которыми представляет *глава 8*.

*Языки* программирования *Mathematica* и *Maple* ориентированы, в первую очередь, на символьные вычисления, располагая относительно ограниченными возможностями в работе с данными, находящимися во *внешней* памяти *ПК*. Вместе с тем, ориентируясь, прежде всего, на решение задач в *символьном* виде, такие языки предоставляют набор средств для доступа к файлам данных, который может удовлетворить широкий круг пользователей пакета, существенно не использующих работы с файлами. Между тем, данный вопрос весьма актуален для задач, довольно активно использующих работу с файлами различных формата и организации. Поэтому в *девятой* главе рассмотрены средства *Mathematica* для работы как с внутренними, так и с внешними файлами. В главе приводится целый ряд средств, расширяющих стандартные средства доступа к файлам данных различного формата и организации.

Учитывая важность не только разработки программных средств, но и их сохранения в файлах специального формата для возможности последующего их использования, *десятая* глава рассматривает вопросы организации пользовательского программного обеспечения в среде *Mathematica*, некоторые полезные способы создания библиотек пользователя наряду с вопросами программирования больших систем в среде пакета. В заключение представлена наша субъективная точка зрения на предпочтительность в использовании пакетов *Mathematica* и *Maple* при решении тех либо иных задач. В книге приведено описание более 320 процедур, функций и глобальных переменных, в значительной степени расширяющих средства программной среды *Mathematica*.

В процессе подготовки настоящей книги нами использовались лицензионные версии 2.2, 7.0.0 – 8.0.4 (License # L4704-9074) *Mathematica*, предоставленные *Wolfram Research Inc.* Все вопросы, замечания и предложения по настоящей книге можно отправлять в любой из адресов, указанных на нашем web-сайте <http://www.aladjev.narod.ru/>.

## Глава 1. Краткий экскурс в историю компьютерной алгебры

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 *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*, ориентированном 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 *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**, **RERUCE**, **Mathematica** и **Maple**; причем книги [5-6] вообще были *первыми* массовыми книгами по **MatCAD**, **RERUCE** 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 напомнить изречение К. Пруткова – «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 подобных 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*.

*Axiom* для математика 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 *невычисленным*. Тогда как *Axiom* возвращает интеграл *невычисленным* 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 коду, написанному 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.

Более того, за период **1988 – 2011** было создано **20** версий пакета, из которых *первой* из доступных для массового пользователя выступает версия *Mathematica 1.0* (1988), тогда как текущей на сегодня является версия *Mathematica 8.0.4* (2011). При подготовке этой книги нами использовались, в основном, версии *Mathematica 7.0, 8.0.0 – 8.0.4* пакета.

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.

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, ВИБРАН, ДИРАК, КЕНТАВР, МАРС, MLR, ПОЛИНАЛ, СПРИНТ, GRATOS, CompuNER* и др. Но в силу целого ряда причин конкурентоспособных отечественных *CAS* в обозримом будущем не предвидится, что делает особенно актуальным освоение отечественными учеными, преподавателями и учащимися новейших западных систем компьютерной алгебры. К таковым относятся системы *Mathematica* и *Maple* – признанные мировые лидеры среди систем компьютерной математики и ориентированные, прежде всего, на массовый класс *ПК*, вполне доступных весьма широкому кругу пользователей.

## Глава 2. Общие сведения по встроенному языку программирования пакета *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.

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 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 иллюстрирует простой фрагмент:

```
> n, m := 0, 0: A: n := n + 1: if n <= 72 then m := m + n^2; goto(A) end if: m;  => 1
Error, return out of context
> G := proc() local n, m; n, m := 0, 0: A: n := n + 1: if n <= 72 then m := m + n^2; goto(A)
end if: m end proc: G();  => 127020
> n, m := 0, 0: A: n := n + 1: if n <= 72 then m := m + n^2; goto(A) else return m end if:
Error, return out of context
```

```
> G := proc() local n, m; n, m := 0, 0; A: n := n + 1; if n <= 72 then m := m + n^2; goto(A)
    else return m end if end proc: G(); ⇒ 127020
```

Из данного фрагмента легко заметить, что если *goto*-функция и *return*-предложение не допустимы во входном языке пакета, то в процедурных выражениях *Maple*-языка их использование вполне корректно. Итак, по большому счету языками реализации пакета *Maple* являются язык *C* и собственный встроенный *Maple*-язык, подмножество которого и составляет собственно входной язык программирования пакета.

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. В настоящей книге рассматривается в сравнительном контексте наиболее полная лингвистическая компонента пакета – его *встроенный Maple-язык* программирования (или просто *Maple-язык*). Вместе с тем, все иллюстрационные фрагменты этой книги во многом являются довольно типичными предложениями входного *Maple*-языка пакета, однако акцент сделан на *Math*-языке.

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, состоящее в следующем, 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 лишь его симуляция в достаточно ограниченных пределах. Следующий простой фрагмент

довольно наглядно иллюстрирует реализацию в среде *Maple* функции **Select** пакета *Mathematica*, которая использует в качестве второго фактического аргумента чистую функцию (*pure function*) короткого формата, а именно:

```
> L := [k$ k = 1 .. 42]: x := 68: map(proc(x) if type(x, prime) then x end if end proc, L);
      [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
> select(isprime, L); => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
In[547]:= L = Range[42]; Select[L, PrimeQ[#] &] => {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41}
```

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 – 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 иллюстрирует следующий достаточно простой фрагмент, представляющий эквивалентную реализацию в среде *Mathematica* в *Input*-режиме ранее представленного фрагмента для среды *Maple*:

```
In[523]:= {n, m} = {0, 0}; Label[A]; n = n + 1; If[n <= 72, m = m + n^2; Goto[A], m]
Out[523]= 127020
```

```

In[524]:= {n, m} = {0, 0}; Label[A]; n = n + 1; If[n <= 72, m = m + n^2; Goto[A], Return[m]]
Out[524]= Return[127020]
In[525]:= P[] := Module[{n = 0, m = 0, A}, Label[A]; n = n + 1; If[n <= 72, m = m + n^2;
                                Goto[A], Return[m]]]
In[526]:= P[]
Out[526]= 127020
In[527]:= F[n_, m_] := If[n + m <= 72, m^2 + n^2, Return[{m, n}]]
In[528]:= {F[14, 22], F[42, 47]}
Out[528]= {680, {47, 42}}
In[628]:= RETURN[x_, res_ /; ! HowAct[res]] := If[SuffPref[ToString[x], "Return["], 1],
                                res = ToExpression[StringTake[ToString[x], {8, -2}]]; Abort[]; Return[x]]
In[629]:= {n, m} = {0, 0}; Label[A]; n = n + 1; If[n <= 72, m = m + n^2; Goto[A],
                                RETURN[Return[m], res]; 2011]; res
Out[629]= 127020
In[630]:= F1[n_, m_] := If[n + m <= 72, m^2 + n^2, RETURN[Return[{m, n}]]]
In[631]:= {F1[14, 22], F1[42, 47]}
Out[631]= {680, {47, 42}}

```

Из представленного фрагмента следует, что **Goto**-функция может использоваться и в *Input*-режиме пакета, и в процедуре; при этом, в процедуре метка может определяться локальной, тогда как в случае пакета *Maple* это недопустимо. Правда, использование **Return**-функции в *Input*-режиме пакета хоть и не вызывает ошибочной ситуации, но ее вызов возвращается *невывчисленным*, если **Return** использована вне тела процедуры/функции. Таким образом, если вне контекстное использование **return**-предложения в среде *Maple* инициирует ошибочную ситуацию, то в среде *Mathematica* аналогичное использование **Return**-функции возвращает ее вызов *невывчисленным*. Между тем, эту ситуацию вполне можно исправить обрамлением вызова **Return**-функции **RETURN**-функцией, как это иллюстрируют последние примеры представленного фрагмента.

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 пакетом 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[1537]:= Sin[42.47]
```

```
Out[1537]= -0.998289
```

```
In[1538]:= Sin[42.47]
```

```
Out[1538]= -0.998289
```

```
In[1539]:= Sin[42. 47]
```

```
Out[1539]= 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 настоящей книге *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*.

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, даже достаточно опытного 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 [90], содержащий более 320 средств, как дополняющих средства пакета, так и устраняющих некоторые его недостатки наряду с расширением некоторых его стандартных средств. Представленные в [90] 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 достаточно полезных и эффективных.

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 читать *Math*-тексты без загрузки лицензионного пакета *Mathematica*, тогда как для *Maple* необходима загрузка самого лицензионного пакета, что существенно затрудняет просмотр даже доступных кодов. Правда, на основе файлов с документами *{mws, mw}* *Maple* и *m*-файлов с пакетами, в принципе, не составляет особого труда создать перлюстраторы программных кодов.

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.

Сравнение систем программирования обоих пакетов производилось на основе *Maple 8-11* и *Mathematica 7,8*. 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 пакетов без привлечения средств, обеспечивающих интерфейс с внешними по отношению к пакетам программными средами. Итак, сравнение языков обеих пакетов проводилось лишь в рамках их встроенных средств, исключая средства, позволяющие использовать программные средства извне, что повышает уровень объективности; т.е.

производилось сравнение замкнутых программных сред обоих пакетов [97-99].

Пакет *Mathematica* в плане *процедурного* программирования в определенной степени уступает пакету *Maple*, хотя и декларирует его как *особую* свою отличительную черту. Между тем, именно пакет *Maple* поддерживает встроенный язык ярко выраженного *процедурного* типа, тогда как пакет *Mathematica* использует смешанный процедурно-функциональный язык, в основе которого лежит комбинирование элементов таких языков как *C++* и *APL*. Для более детального ознакомления с возможностями среды пакета *Mathematica* ниже рассматриваются его основные элементы процедурного и функционального программирования. При этом, нами не рассматриваются средства программирования собственно *Math*-документов, а лишь средства, обеспечивающие процедурно-функциональное программирование на уровне процедур и функций.

В книге представлено несколько *версий* целого ряда средств, что обусловлено рядом причин. Прежде всего, такой подход преследовал цель проиллюстрировать разные приемы, полезные в практическом программировании; во-вторых, данный подход довольно хорошо иллюстрирует методику оптимизации программных реализаций при появлении новых средств как стандартных, так и нестандартных. И наконец, в целом ряде случаев последующие *версии*, порой, довольно существенно расширяют функциональные возможности предыдущих версий. Все эти моменты достаточно поучительны при освоении программирования в среде *Mathematica*, учитывая его встроенный *Math*-язык, 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 пакета, пытайтесь 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 пакета, 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 [90], распространяемый по лицензии *Freeware*, который может быть полезен как в учебных целях, так и при программировании прикладных задач различного типа. Его функциональные средства расширяют программную среду пакета *Mathematica*, в ряде случаев устраняя ошибки и ограничения стандартных средств пакета.

## Глава 3. Базовые управляющие структуры пакета Mathematica

Для описания произвольного вычислительного алгоритма структур, базирующихся на сугубо последовательных операциях, совершенно недостаточно без средств по управлению собственно вычислительным процессом. Настоящая глава служит задаче рассмотрения управляющих структур встроенного языка пакета *Mathematica*.

Современное *структурное* программирование сосредоточивает внимание на одном из наиболее подверженных ошибкам факторов: *логике программы* и включает 3 основные компоненты, т.е. *нисходящее проектирование*, *модульное программирование* и *структурное кодирование*. Первые две компоненты достаточно детально рассмотрены в книгах [1-3], кратко остановимся здесь только на третьей компоненте программирования.

В задачу *структурного кодирования* входит получение корректной программы (*модуля*) на основе простых управляющих структур. В качестве подобных базовых выбираются *управляющие структуры следования, ветвления, организации циклов и вызовов функций (процедур, программ)*; при этом, все перечисленные структуры допускают только один *вход* и один *выход*. При этом, первые из 3 указанных *управляющих структур (следования, ветвления и организации циклов)* составляют тот минимальный базис, на основе которого возможно создавать любой сложности корректную программу с одним *входом*, одним *выходом*, без зацикливаний и недостижимых команд. Детальное обсуждение базисов управляющих структур программирования можно найти, в частности, в книгах [1-3] и в другой доступной литературе по основам современного программирования.

*Следование* отражает принцип последовательного выполнения предложений той или иной программы, пока не встретится предложение, изменяющее последовательность. Типичной управляющей структурой *следования* является последовательность из ряда простых предложений присваивания в том либо ином языке программирования.

*Ветвление* характеризует собой выбор одного из всех возможных путей последующих вычислений; типичными предложениями, обеспечивающими данную управляющую структуру, являются предложения формата «*IF A THEN B ELSE C*». Структура «*цикл*» реализует *повторное* выполнение группы предложений до тех пор, пока выполняется некоторое *логическое условие*; типичными предложениями, обеспечивающими такую управляющую структуру, являются предложения *DO, DO\_WHILE, DO\_UNTIL*. Итак, базисные структуры определяют соответственно *последовательную (следование), условную (ветвление) и итеративную (цикл)* передачи управления в программах [1-3]. Показано, что корректная *структурированная* программа теоретически любой сложности может быть написана с использованием только управляющих структур следования, *WHILE*-циклов и *IF*-операторов ветвления. Однако расширение набора указанных средств, в первую очередь, за счет обеспечения вызовов функций и механизма процедур может весьма существенно облегчить программирование, не нарушая *структурированности* программ и повышая уровень их модульности и робастности (*надежности*).

При этом, сочетания (*итерации, вложения*) корректных структурированных программ, полученные на основе указанных управляющих структур, не нарушают принципа их структурированности и корректности. Итак, любых сложности и размера программы

можно получать на основе соответствующего сочетания расширенного базиса (*вызовы функций и механизм процедур, цикл, следование, ветвление*) управляющих структур. Такой подход позволяет нам отказаться в программах от использования *меток* и *безусловных переходов*. Структура подобных программ четко прослеживается от начала (*сверху*) и до конца (*вниз*) при отсутствии передач управления на *верхние* уровни. Итак, именно в свете сказанного именно языки такого типа представляют собой довольно удобное лингвистическое обеспечение в деле разработки эффективных структурированных программ, сочетающее лучшие традиции *структурно-модульной* технологии, которая ориентирована в данном случае на математическую сферу приложений и достаточно большое количество программистски непрофессиональных пользователей из целого ряда прикладных областей, включая и не совсем математической направленности.

В дальнейшем изложении под «*предложением*» *Math*-языка понимается конструкция следующего весьма простого вида, а именно:

### *Math (Mathematica)-выражение*

где в качестве выражения допускается любая корректная с точки зрения *Math*-языка конструкция, например:  $A = \text{Sqrt}[70 + x]; 42 \neq 70; \text{Sin}[x] + x; \text{Tallinn} := 6; \text{GrGu} = 2012$  и др. В рассмотренных ниже примерах приведено достаточно много разных примеров относительно простых предложений в рамках управляющей структуры *следования*, в целом довольно прозрачной и особых пояснений не требующей. Предложения языка кодируются друг за другом, каждое в отдельной строке или несколько в одной строке; завершаются в общем случае «*;*»-разделителями, выполняясь строго последовательно, если управляющие структуры ветвления или цикла не определяют другого порядка. В дальнейшем предложения *Math*-языка будем называть согласно их определяющим назначениям, например, предложение *присваивания*, *вызова функции*, *If-предложение*, *While-предложение*, *Do-предложение* и т.д. Сделаем лишь одно замечание к разделителю «*;*», а именно. Если последнее в группе предложение завершается этим разделителем, то результат его выполнения на печать не выводится, сохраняясь в рабочей области пакета, тогда как при его отсутствии результат одновременно выводится и на печать:

```
In[227]:= A = Sqrt[69 + Sin[69.42]]; Sin[x] + x; Tallinn = 2012
```

```
Out[227]= 2012
```

```
In[228]:= A = Sqrt[69 + Sin[69.42]]; Sin[x] + x; Grodno = 2012;
```

```
In[229]:= Grodno
```

```
Out[229]= 2012
```

Наиболее широко употребительно определение предложения *присваивания* на основе одноименного (=)-оператора, допускающего множественные присваивания на основе списочной структуры, как это иллюстрирует следующий простой пример, а именно:

```
In[317]:= {x, y, z} = {69, 64, 44}
```

```
Out[317]= {69, 64, 44}
```

```
In[318]:= {x, y, z}
```

```
Out[318]= {69, 64, 44}
```

Наряду с (=)-оператором присваивания *Math*-язык использует также и (*:=*)-оператор отложенного присваивания, суть которого состоит в том, что реальное присваивание

$x:=y$  значения  $y$  объекту  $x$  производится только при конкретном использовании  $x$ , как это имеет место, например, при определении процедуры или функции. При каждом использовании  $x$  каждый раз производится замена  $x$  на  $y$ , например:

```
In[319]:= {x, y, z} := {42, 47, 67}
```

```
Out[319]= {Null, Null, Null}
```

```
In[320]:= {x, y, z}
```

```
Out[320]= {42, 47, 67}
```

```
In[321]:= x := 450.75
```

```
In[322]:= x
```

```
Out[322]= 450.75
```

```
In[323]:= y := 2012
```

```
In[324]:= {%, y}
```

```
Out[324]= {Null, 2012}
```

При этом, в результате ( $:=$ )-присваивания возвращается *Null*-значение, т.е. ничего. В *Math*-языке допускаются множественные присваивания в форме списка следующего вида  $\{x1=a1, \dots, xn=an\}$  либо  $\{x1:=a1; \dots; xn:=an\}$ , как иллюстрирует следующий весьма простой и достаточно наглядный фрагмент, а именно:

```
In[333]:= a := 68; b := 63; c := 43; d := 47; {x, y, z} = {42, 47, 67};
```

```
In[334]:= {{a, b, c, d}, {x, y, z}}
```

```
Out[334]= {{68, 63, 43, 47}, {42, 47, 67}}
```

```
In[335]:= x1 = 42; x2 = 47; x3 = 43; x4 = 14; x5 = 22;
```

```
In[336]:= {x1, x2, x3, x4, x5}
```

```
Out[336]= {42, 47, 43, 14, 22}
```

```
In[337]:= x1 = 42; x2 = 47; x3 = 43; x4 = 14; x5 = 22;
```

```
In[338]:= {x1, x2, x3, x4, x5}
```

```
Out[338]= {42, 47, 43, 14, 22}
```

```
In[339]:= {y1 = 42, y2 = 47, y3 = 43, y4 = 14, y5 = 22}
```

```
Out[339]= {42, 47, 43, 14, 22}
```

```
In[340]:= {z1 := 42, z2 := 47, z3 := 43, z4 := 14, z5 := 22}
```

```
Out[340]= {Null, Null, Null, Null, Null}
```

```
In[341]:= {z1, z2, z3, z4, z5}
```

```
Out[341]= {42, 47, 43, 14, 22}
```

```
In[342]:= {z1 := 42, z2 = 47, z3 := 43, z4 = 15, z5 := 23}
```

```
Out[342]= {Null, 47, Null, 15, Null}
```

```
In[343]:= {z1 := 42; g = 64, z2 = 47, z3 := 43, z4 = 15; v = 69; s := 44, z5 := 23}
```

```
Out[343]= {64, 47, Null, Null, Null}
```

```
In[344]:= {z1, g, z2, z3, z4, v, s, z5}
```

```
Out[344]= {42, 64, 47, 43, 15, 69, 44, 23}
```

В качестве элементов списка *Math*-языка могут использоваться также и конструкции формата *имя*{:= | :}*выражение*; т.е. *последовательности* предложений языка. Детальнее с вопросами организации *последовательностных* структур можно ознакомиться в [100].

### 3.1. Управляющие структуры ветвления в среде Mathematica

Условные структуры ветвления. Достаточно сложные алгоритмы вычислений и/или управляющие (в первую очередь) не могут обойтись сугубо последовательной схемой, а включают различные конструкции, которые изменяют *последовательностный* порядок выполнения алгоритма в зависимости от наступления тех или иных условий: *условные* и *безусловные* переходы, *циклы* и *ветвления* (данного типа структуры в целом ряде случаев называются *управляющими*). Так, для организации *управляющих* структур ветвящегося типа *Math*-язык располагает достаточно эффективным средством, обеспечиваемым т. н. **If-предложением (функцией)**, имеющим следующие три формата кодирования:

- (1) **If**[<Логическое условие>, <ПП>]
- (2) **If**[<Логическое условие>, <ПП>, <ПП1>]
- (3) **If**[<Логическое условие>, <ПП>, <ПП1>, <ПП2>]

В качестве *логического условия (ЛУ)* всех 3 форматов **If**-предложения выступает любое допустимое *булево* выражение, образованное на основе операторов *отношения* {< | <= | > | >= | == (Equal) | === (SameQ) | != (Unequal)}, логических операторов {**And** (&&), **Or** (| |), **Not** (!)}, логических констант {**True**, **False**} и/или функций, которое возвращает логическое {**True** | **False**}-значение. *Последовательность предложений (ПП)* представляет собой управляющую структуру типа следования, чьи предложения завершаются **{;}**-разделителем; при этом, в последнем предложении *ПП* кодирование **{;}**-разделителя необязательно, либо другие сочетания типов корректных предложений, включая **If**-предложения и предложения цикла. Таким образом, *Math*-язык допускает вложения **If**-предложений, как это иллюстрирует нижеследующий простой фрагмент.

*Первый* формат **If**-предложения несет следующую смысловую нагрузку: если результат вычисления *ЛУ* возвращает **True**-значение, тогда возвращается результат выполнения последнего предложения из *ПП*, в противном случае возвращается **Null**-значение, т.е. ничего. *Второй* формат **If**-предложения несет следующую смысловую нагрузку: если результат вычисления *ЛУ* возвращает **True**-значение, тогда возвращается результат выполнения последнего предложения из *ПП*, иначе – последнего предложения *ПП1*. *Третий* формат **If**-предложения несет следующую смысловую нагрузку: если пакет не может определить истинность *ЛУ*, то возвращается результат выполнения последнего предложения из *ПП2*. При этом, если подобная ситуация возникает для первого либо второго формата **If**-предложения, вызов **If**-функции возвращается невычисленным, как это иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[869]:= If[a == b, c]
Out[869]= If[a == b, c]
In[870]:= If[a == b, c, d]
Out[870]= If[a == b, c, d]
In[871]:= If[a == b, c, d, G]
Out[871]= G
```

Итак, третий формат **If**[*ЛУ*, *ПП*, *ПП1*, *ПП2*] **If**-функции ориентирован на возможность получения *ЛУ* неопределенного значения, как иллюстрирует последний пример. **If**-

предложение допускает произвольный *уровень* вложенности, тогда как в качестве *ППк* могут выступать любые последовательности корректных предложений *Mathematica*. В контексте синтаксиса **If**-функции могут участвовать в образовании выражений:

```
In[1011]:= If[70 == 70, 450 + 75*If[42 == 47, If[42 == 47, c], d], b]
Out[1011]= 450 + 75 d
In[1012]:= If[V == V, If[42 == 42, Sqrt[If[70 == 70, 19.47] + 19.42], d], b, h]
Out[1012]= 6.23618
```

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[944]:= 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[945]:= a = {}; For[k = 1, k <= 100, k++, Iff[PrimeQ[k], "a = Append[a, k]"]; a
Out[945]= {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 пакета *Maple* воспринимаются несколько проще.

В частности, *Maple* допускает условное **If**-предложение следующего формата

**if ЛУ1 then V1 elif ЛУ2 then V2 elif ЛУ3 then V3 elif ЛУ4 then V4 ... else V<sub>k</sub> end if** (1)

смысл которого достаточно прозрачен и рассмотрен, например, в [8,42-44,99]. Данное предложение весьма удобно при программировании целого ряда условных структур. В целях определения аналогичной структуры может быть применена процедура **Ifk**, чей исходный код и примеры использования представляет следующий фрагмент:

```
In[887]:= 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[888]:= Ifk[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]
Out[888]= If[a, b, If[c, d, If[g, s, If[m, n, If[q, p, h]]]]]
In[889]:= Ifk[{False, b}, {False, d}, {False, s}, {True, n}, {False, p}, h]
Out[889]= n
In[890]:= Ifk[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]
Out[890]= If[g, p, h]
In[891]:= 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[892]:= Ifk1[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]
Out[892]= Ifk1[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]
In[893]:= Ifk1[{False, b}, {False, d}, {False, s}, {True, n}, {False, p}, h]
Out[893]= n
In[894]:= Ifk1[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]
Out[894]= Ifk1[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]
In[895]:= Ifk1[{a, b, g}, {c, d}, {g, s}, {m, n}, {q, p}, h]
Out[895]= Ifk1[{a, b, g}, {c, d}, {g, s}, {m, n}, {q, p}, h]
In[896]:= Ifk1[{True, b}]
Out[896]= Ifk1[{True, b}]
In[897]:= Ifk1[{False, b}, agn]
Out[897]= agn

```

Вызов процедуры **Ifk** использует произвольное число фактических аргументов более одного, в качестве которых выступают 2-элементные списки формата  $\{LUj, Vj\}$ , кроме последнего (1). В качестве последнего аргумента допустимо 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 формата (1). При этом, в качестве некоторого полезного расширения служит 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, подобно *Maple*-предложению (1) процедуры **Ifk** и **Ifk1** оказываются довольно полезными для программирования *ветвящихся* алгоритмов. В дополнение к используемым стандартным функциональным средствам обе процедуры используют процедуру **StringMultiple**, обеспечивающую кратную конкатенацию строк. В данном отношении для работы со строчными выражениями довольно полезной оказывается простая процедура **StringMultiple**, чей вызов **StringMultiple[s, p]** возвращает строку – результат *p*-кратной конкатенации *s*-строки. Фрагмент представляет исходный код.

```
In[882]:= StringMultiple[s_;/ StringQ[s], p_;/ IntegerQ[p] && p >= 1] :=
Module[{c = "", k = 1}, For[k, k <= p, k++, c = c <> s]; c]

In[883]:= StringMultiple["agn", 6]
Out[883]= "agnagnagnagnagnagn"
In[884]:= StringMultiple["agn", 0]
Out[884]= StringMultiple["agn", 0]
In[885]:= StringMultiple["agn", 1]
Out[885]= "agn"
```

В качестве несложного обобщения предыдущей является процедура **StringMultipleD**, вызов которой на 2 фактических аргументах эквивалентен вызову **StringMultiple[s, p]**, тогда как вызов процедуры **StringMultipleD[s, p, h]** на трех аргументах (*3-й должен быть строкой*) возвращает результат *p*-кратной конкатенации *s*-строки, разделенной *h*. В случае получения недопустимого кортежа *фактических* аргументов вызов процедуры возвращается невычисленным, как иллюстрирует следующий простой фрагмент.

```
In[1778]:= StringMultipleD[s_;/ StringQ[s], p_] := Module[{a = {s, p}, c = "", k = 1, b},
b = Length[a];
If[b == 2 && IntegerQ[a[[2]]] && a[[2]] >= 1, StringMultiple[a[[1]], a[[2]]],
If[b == 3 && IntegerQ[a[[2]]] && a[[2]] >= 1 && StringQ[a[[3]]],
For[k, k <= a[[2]], k++, c = c <> s <> a[[3]]];
StringTake[c, {1, StringLength[c] - StringLength[a[[3]]}],
Defer[StringMultipleD[s, p]]]]]

In[1781]:= StringMultipleD["agn", 6, " "]
Out[1781]= "agn agn agn agn agn agn"
In[1782]:= StringMultipleD["agn", 6]
Out[1782]= "agnagnagnagnagnagn"
In[1783]:= StringMultipleD["agn", 6, 450.75]
Out[1783]= StringMultipleD["agn", 6, 450.75]
```

Используя описанный подход, достаточно несложно реализовать в среде *Math*-языка любую конструкцию *Maple*-языка, описывающую ветвящийся алгоритм.

В определенной мере к **If**-конструкциям можно отнести и **Which**-функцию формата

**Which[ЛУ1, V1, ЛУ2, V2, ЛУ3, V3, ..., ЛУk, Vk]**

которая возвращает результат вычисления первого *Vj*-выражения, булево выражение *ЛУj* (*j=1..k*) для которого принимает значение *True*, например:

```
In[886]:= G[x_] := Which[-Infinity <= x < 75, Sin[x], 75 <= x < 420, Cos[x],
                                     420 <= x <= Infinity, x^2]
```

```
In[887]:= {G[65], G[75.450], G[450], G[2011], G[-20.11]}
```

```
Out[887]= {Sin[65], 0.99866, 202500, 4044121, -0.952226}
```

Пример иллюстрирует определение через **Which** кусочно-определенной функции. Если какое-нибудь из вычисляемых условий *ЛУj* не возвращает *{True | False}*, то вызов функции возвращается *невывчисленным*, тогда как в случае значения *False* для всех *ЛУj* (*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[911]:= 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[912]:= WhichN[a, b, c, d, f, g, h, r]
Out[912]= Which[a, b, c, d, f, g, h, r]
In[913]:= f = 75; WhichN[False, b, f == 75, SV, g, h, r, t]
Out[913]= SV
In[914]:= WhichN[True, b, False, d, True, g, h]
Out[914]= WhichN[True, b, False, d, True, g, h]
```

В качестве еще одного полезного средства ветвления следует отметить переключатель (*функцию*) **Switch**, имеющую следующий общий формат кодирования, а именно:

**Switch**[*G*, *F1*, *V1*, *F2*, *V2*, *F3*, *V3*, ..., *Fk*, *Vk*]

Вызов функции **Switch** возвращает результат вычисления *Vj*, для первого из которых было обнаружено совпадение значения вычисленного *G*-выражения с шаблоном *Fj*. В случае отсутствия таких совпадений вызов функции возвращается невычисленным; в случае использования в качестве *последнего Fk* шаблон «\_» обеспечивается возврат *Vk*, если до него указанных совпадений обнаружено не было. При этом, в теле функции **Switch** возможно использование функций выхода **Break**, **Return** и **Throw**. Следующий простой фрагмент иллюстрирует применение функции-переключателя **Switch**:

```
In[887]:= Switch[Art, c, Sin[x], Art, Sin[x] + Cos[y], Kr, (a + b)]
```

```
Out[887]= Cos[y] + Sin[x]
```

```
In[888]:= Switch[Art, c, Sin[x], ArtKr, Sin[x] + Cos[y], _ (a + b)]
```

```
Out[888]= a + b
```

```
In[889]:= Switch[Art, Ian, Sin[x], ArtKr, Sin[x] + Cos[y], VS, (a + b)]
```

```
Out[889]= Switch[Art,
    Ian, Sin[x],
    ArtKr, Sin[x] + Cos[y],
    VS, a + b]
```

**Безусловные переходы.** Передачи управления *безусловного* типа определяются в языке, как правило, *goto*-конструкциями, по которым управление передается в точку *Label* программы (*процедуры*), указанную соответствующей меткой (*Label*). Встроенный язык *Math* для организации ветвления алгоритмов наряду с представленным предложением *If* допускает использование безусловных переходов на основе функции *Goto*, которая кодируется в форме *Goto[W]*, *безусловно* передавая управление в точку, именованную конструкцией *Label[W]*. Как правило, *Goto*-функция используется, главным образом, в процедурных конструкциях, однако в отличие от встроенной *goto*-функции пакета *Maple* она может использоваться также во входной конструкции *Mathematica*, как это иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[880]:= x= 1; Label[a + b; Sin[c]; AVZ]; Print[x++]; If[x != 31, Goto[AVZ], Null]
1
.....
30
In[881]:= x= 1; Label[a + b; Sin[42]+x; Sqrt[16]]; Print[x++]; If[x != 31, Goto[4], AVZ]
1
.....
30
Out[881]= AVZ
```

Более того, фрагмент иллюстрирует также тот факт, что в качестве метки может быть произвольное допустимое выражение, включая последовательность выражений, чье последнее выражение определяет собственно саму метку; при этом, метка может быть как глобальной, так и локальной переменной относительно модуля, как это наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[882]:= ArtKr[x_/; IntegerQ[x]] := Module[{prime, agn},
      If[PrimeQ[x], Goto[prime], If[OddQ[x], Goto[agn], Goto[Sin]]];
      Label[prime]; Print[x^2]; Goto[Sin];
      Label[agn]; Print[NextPrime[x]]; Goto[Sin];
      Label[Sin]; Null]

In[883]:= ArtKr[47]
2209
In[884]:= ArtKr[42]
In[885]:= ArtKr[121]
127
```

При этом, во избежание недоразумений метку рекомендуется определять *локальной*, ибо вычисленная вне процедуры *глобальная* метка всегда допустима для процедуры, тогда как вычисленная в теле процедуры может исказить вычисления *вне* процедуры. Кратность вхождения идентичных *Goto*-функций в процедуру вполне естественна и определяется реализуемым ею алгоритмом задачи, тогда как с соответствующими им метками *Label* подобная ситуация, как правило, недопустима, не распознаваясь при вычислении определения процедуры и даже на уровне ее выполнении, между тем, в значительной степени, порой, искажая запланированный алгоритм. В данном случае

управление получает точка, помеченная *первой* такой меткой, как довольно наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[911]:= Kr[x_ /; IntegerQ[x]] := Module[{prime, agn, y}, If[PrimeQ[x], Goto[prime],
      If[OddQ[x], Goto[agn], Goto[agn]]]; Label[45; prime]; y = x^2; Goto[agn];
      Label[agn]; y = NextPrime[x]; Label[agn]; y]
In[912]:= Map[Kr, {11, 450, 121}]
Out[912]= {13, 457, 127}
```

Следует иметь в виду, что отсутствие **Label[a]** для соответствующей функции **Goto[a]** в процедуре на стадии вычисления ее определения не распознается, а лишь в момент выполнения с реальным обращением к такой **Goto[a]**, как иллюстрирует фрагмент:

```
In[920]:= Svet[x_] := Module[{prime, agn}, If[PrimeQ[x], Goto[prime], Goto[agn]];
      Label[prime]; Print[x^2]; Goto[Sin]; Label[Sin]; Null]
In[921]:= Svet[2011]
4044121
In[922]:= Svet[2012]
Goto::nolabel: Label agn$2036 not found. >>
Out[922]= Hold[Goto[agn$2036]]
```

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 четырех-элементный список, чей первый элемент представляет список всех используемых процедурой **Goto**-функций, второй – список *всех* меток (с учетом их кратности), третий – список *некорректных* **Goto**-функций (для **Goto** *отсутствует* **Label**) и четвертый – список всех меток, чья кратность больше 1. В следующем фрагменте представлены исходный код с примерами ее применения.

```
In[91]:= GotoLabel[P_ /; ProcQ[P]] := Module[{a = ToString1[DefFunc[P]], b = {}, c = {}, d,
      h, p, f = Union[CharacterRange["a", "z"], CharacterRange["A", "Z"]], k = 1, j, g, l},
      If[P === GotoLabel, Return[{{}, {}, {}, {}], {g, l} = {{}, {}];
      {d, h} = Map3[StringPosition, a, {"Goto[", "Label["];
      d = Select[d, ! MemberQ[f, StringTake[a, {#[[1]] - 1, #[[1]] - 1}]] &];
      h = Select[h, ! MemberQ[f, StringTake[a, {#[[1]] - 1, #[[1]] - 1}]] &];
      For[k, k <= Length[h], k++, p = StringTake[a, h[[k]]];
      For[j = h[[k]][[2]] + 1, j < Infinity, j++, p = p <> StringTake[a, {j, j}];
      If[Quiet[ToExpression[p]] === $Failed, Continue[], l = Append[l, p]; Break[]];
      For[k = 1, k <= Length[d], k++, p = StringTake[a, d[[k]]];
      For[j = d[[k]][[2]] + 1, j < Infinity, j++, p = p <> StringTake[a, {j, j}];
      If[EvenQ[StringCount[p, {"[", "["}], g = Append[g, p]; p = {}; Break[]; Continue[]];
      {g, l} = {DeleteDuplicates[StringReplace[Map[ToString,
      Map[ToExpression, StringReplace[g, "Goto[" -> "Goto1["],
      "Goto1[" -> "Goto["], Map[ToString, Map[ToExpression, l]]];
```

```

p = DeleteDuplicates[StringReplace[l, "Label[" -> "Goto["]];
{g, 1, Select[g, ! MemberQ[p, #] &], MinusList[l, DeleteDuplicates[l]]]
In[92]:= ArtKr[x_./IntegerQ[x]] := Module[{prime, agn}, If[PrimeQ[x], Goto[45; prime],
If[OddQ[x], Goto[agn], Goto[Sin]]]; Label[45; prime]; Print[x^2]; Goto[Sin];
Print[NextPrime[x]]; Goto[Sin]; Label[45; prime]; Null]
In[93]:= Kr[x_./IntegerQ[x]] := Module[{prime, agn, y}, If[PrimeQ[x], Goto[prime],
If[OddQ[x], Goto[agn], Goto[agn]]]; Label[45; prime]; y = x^2; Goto[agn];
Label[agn]; y = NextPrime[x]; Label[agn]; y]
In[94]:= GotoLabel[ArtKr]
Out[94]= {"Goto[prime]", "Goto[agn]", "Goto[Sin]", {"Label[prime]", "Label[prime]"},
{"Goto[agn]", "Goto[Sin]", {"Label[prime]"}}}
In[95]:= GotoLabel[Kr]
Out[95]= {"Goto[prime]", "Goto[agn]", {"Label[prime]", "Label[agn]", "Label[agn]"},
{}, {"Label[agn]"}}}
In[96]:= Map[GotoLabel, {Nproc, ExtrExpr}]
Out[96]= {{{{"Goto[g]", {"Label[g]"}, {}, {}}, {"Goto[b]", {"Label[b]"}, {}, {}}}}
In[97]:= Map[GotoLabel, {GotoLabel, TestArgsTypes}]
Out[97]= {{{}, {}, {}, {}}, {"Goto[h]", "Goto[p]", {"Label[h]", "Label[p]"}, {}, {}]}
In[98]:= Map[GotoLabel, {SearchDir, StrDelEnds, OP}]
Out[98]= {{{{"Goto[b]", {"Label[b]"}, {}, {}}, {"Goto[p]", "Goto[Fin]", {"Label[1]", "Label[3]",
"Label[2]", "Label[Fin]", {"Goto[p]"}, {}, {"Goto[420]", {"Label[420]"}, {}, {}}}}

```

Отметим, что наличие в возвращаемом вызовом процедуры **GotoLabel[P]** вложенного списка с *третьим* непустым подсписком не обязательно говорит о наличии функции **Goto[V]**, для которой отсутствует **Label[V]**. Это может быть, например, при генерации значения *V* в зависимости от некоторого условия. Так, вызов **GotoLabel[StrDelEnds]** в качестве третьего подсписка возвращает {"Goto[p]"}, что можно было бы воспринять в качестве некорректности данной **Goto**-функции. Однако, все дело в том, что значение фактического *p*-аргумента в вызове процедуры **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. Следует отметить, с нашей точки зрения *Goto*-функция пакета *Mathematica* предпочтительнее относительно *goto*-функции из пакета *Maple* в плане эффективности применения в *процедурном* программировании. Рассмотрев средства организации ветвления вычислительных алгоритмов, переходим к основным средствам организации *циклических* конструкций в среде *Mathematica*.

## 3.2. Циклические управляющие структуры пакета Mathematica

Одна из базовых *циклических* структур пакета основывается на **For**-функции, которая имеет 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 несколько простых наиболее типичных примеров на применение данной функции, а именно:

```
In[512]:= For[k = 1; h = 1, k < 4, k = k + 1, h = h^3 + h^2 + h + k; Print[h]]
```

4

86

643541

```
In[513]:= 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 иллюстрирует последний пример фрагмента.

В качестве другого достаточно широко используемого средства в пакете *Mathematica* для организации циклических вычислений является функция **Do**[...], которая имеет следующие форматы кодирования, а именно:

**Do**[*W*, {*n*}] – вычисляет *W*-выражение *n* раз;

**Do**[*W*[*k*], {*k*, *n*}] – последовательно вычисляет выражение *W*[*k*] при изменении индекса *k* от единицы до *n* с шагом единица;

**Do**[*W*[*k*], {*k*, *t*, *n*}] – последовательно вычисляет выражение *W*[*k*] при изменении от *k=t* индекса цикла *k* до *n* с шагом единица;

**Do**[*W*[*k*], {*k*, *t*, *n*, *s*}] – последовательно вычисляет выражение *W*[*k*] при изменении от *k=t* индекса цикла *k* до *n* с шагом *s*;

**Do**[*W*[*k*], {*k1*, *t1*, *n1*}, {*k2*, *t2*, *n2*}, ..., {*kp*, *tp*, *np*}] – вычисляет выражение *W*[*k1*, *k2*, ..., *kp*] при изменении переменных цикла *kj* от *tj* до *nj* (*j=1..p*) с шагом единица.

Приведем несложные примеры на использование указанных форматов функции **Do**:

```
In[219]:= t := 0; Do[t = t + 1, {1000000}]; t
```

```
Out[219]= 1000000
```

```
In[220]:= t := 0; Do[t = t + k^2, {k, 1000}]; t
```

```
Out[220]= 333833500
In[221]:= t := 0; Do[t = t + k^2, {k, 1000, 10000}]; t
Out[221]= 333050501500
In[222]:= t := 0; Do[t = t + k^2, {k, 1000, 10000, 20}]; t
Out[222]= 16700530000
In[223]:= t := 0; Do[t = t + k^2 + j^2 + h^2, {k, 10, 100}, {j, 10, 100}, {h, 10, 100}]; t
Out[223]= 8398548795
```

Приведенные примеры на все форматы **Do**-функции цикла достаточно прозрачны и особых пояснений не требуют. Аналогично случая **For**-цикла для **Do**-цикла **Break[]** и **Continue[]** служат для обеспечения выхода из цикла и продолжения выполнения цикла соответственно, как это иллюстрирует следующий весьма простой пример, а именно:

```
In[548]:= t := 0; Do[t = t + k^2; If[k < 10000, Continue[], Break[]], {k, 1000, 1000000, 20}]; t
Out[548]= 16700530000
```

Между тем, в отличие от *Maple*, пакет *Mathematica* не имеет аналога весьма полезных циклических конструкций типа (1.b) и (1.d) [99], позволяющих выполнять циклические вычисления по подвыражениям заданного выражения, обеспечивая возможность на их основе создавать интересные конструкции, как иллюстрирует простой фрагмент.

```
> NF := proc(F::symbol, L::list(symbol), p::list(symbol)) local z, k; z := NULL;
    for k in L do z:=(z@k) end do: parse(cat("", F, ":=", convert(p, string)[2..-2], ") -> ",
    convert(%(op(p)), string)), statement) end proc:
    G := (x, y) -> R(F(H(T(V(S(x, y))))))
    (x, y) -> R(F(H(T(V(S(x, y))))))
> NF1(F, [R, Gs, S, Art, Kris], [x, y, z]);    => (x, y, z) -> R(Gs(S(Art(Kris(x,y,z))))))
> NF1(F, [arccos, tan, ln, cos, sin], [x]);    => x -> arccos(tan(ln(cos(sin(x))))))
> F(19.42);    => 1.726131477
```

Данный фрагмент демонстрирует применение циклической **for in**-конструкции для создания процедуры  $NF(F, L, p)$ , возвращающей вложенную  $F$ -функцию на основе как списка  $L$  имен функций, составляющих ее уровни, так и списка  $p$  переменных самой внутренней ее функции. При этом, если **in** указывается относительно произвольного выражения  $expr$ , кроме *rtable*, *table* или последовательности, переменная цикла будет принимать значения подвыражений, определяемых по  $op(expr)$ . Поэтому для данных выражений целесообразно выбирать соответственно конструкции следующего вида:

**map(op, [entries(convert(expr, table))]), map(op, [entries(expr)])** и **op([expr])**

Нет в *Mathematica* и прямого эквивалента встроенной *op*-функции, которую можно с точностью до аксиоматики пакетов определить следующей процедурой, а именно:

```
In[1272]:= Op[x_] := Module[{a, b}, a := {}; If[ListQ[x], a = x,
    Do[a = Insert[a, Part[x][[b]], -1], {b, Length[x]}]; a]
In[1273]:= Op[Sin[x] + Cos[x]]
Out[1273]= {Cos[x], Sin[x]}
In[1274]:= Op[Sin[x]*Cos[x]]
Out[1274]= {Cos[x], Sin[x]}
```

```

In[1275]:= Op[{1, 2, 3, 4, 5, 6}]
Out[1275]= {1, 2, 3, 4, 5, 6}
In[1276]:= Op[Sqrt[a + b]]
Out[1276]= {1 + a, 1/2}
In[1277]:= Op[Sqrt[a + b] + Sin[x] - c/d]
Out[1277]= {Sqrt[1 + a], -(c/d), Sin[x]}
In[1278]:= Op[(x + y*Cos[x])/(y + x*Sin[y])]
Out[1278]= {x + y Cos[x], 1/(y + x Sin[y])}
In[1279]:= Map[Op, {Sin[x], Cos[a + b], 1/(a + b)}]
Out[1279]= {{x}, {1 + a}, {1 + a, -1}}
In[1280]:= Op[f[g[a, b], h[c, d, e, j, k, l]]]
Out[1280]= {g[a, b], h[c, d, e, j, k, l]}

```

Несложно убедиться, что полученные результаты вызовов **Op**-функции идентичны с аналогичными вызовами **op**-функции в среде *Maple* с учетом того, что *Mathematica* не поддерживает структуру типа последовательность, которая заменяется списком. В этом же контексте представим функцию **DO[x, y, k]**, возвращающую список значений циклического вычисления выражения *x* по переменной цикла *k*, которая принимает значения из списка **Op[y]**. Данная конструкция в определенном отношении является аналогом циклической **for\_in**-конструкции программной среды пакета *Maple*.

```

In[125]:= 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[126]:= DO[k^2 + Log[k], f[g[a, b], h[c, d, e, j, k, l]], k]
Out[126]= {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[127]:= t := 0; b := {}; While[t < 100, t = t + 1; If[PrimeQ[t], AppendTo[b, t], Null]]; b
Out[127]= {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* обоих пакетов, когда наиболее важные вычислительные конструкции пакета *Mathematica* с той либо иной эффективностью симулируются конструкциями *Maple*, и наоборот. И действительно, в принципе, это вполне ожидаемый результат, ибо встроенные языки обоих пакетов универсальны и в этой связи с той или 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 25.296 с., то для *Mathematica 8* для этой же конструкции требуется уже 260.891 с., т.е. более чем

# (Maple 11)

```
In[892]:= n = 10; t = TimeUsed[]; For[k1 = 1, k1 <= n, k1++,  
  For[k2 = 1, k2 <= n, k2++,  
    For[k3 = 1, k3 <= n, k3++,  
      For[k4 = 1, k4 <= n, k4++,  
        For[k5 = 1, k5 <= n, k5++,  
          For[k6 = 1, k6 <= n, k6++,  
            For[k7 = 1, k7 <= n, k7++,  
              For[k8 = 1, k8 <= n, k8++, 75]]]]]]]; TimeUsed[] - t  
Out[893]= 260.891
```

### 3.3. Специальные типы циклических управляющих структур в программной среде пакета Mathematica

[illegible]

Функция `NestList[F, h, n]` возвращает список результатов применения  $F$  к выражению  $h$ , начиная с 0-вложенности до вложенности глубины  $n$ , например:

```
In[2023]:= NestList[S, y, 6]
Out[2023]= {y, S[y], S[S[y]], S[S[S[y]]], S[S[S[S[y]]]], S[S[S[S[S[y]]]]], S[S[S[S[S[S[y]]]]]]}
In[2024]:= NestList[Tan, 6.8, 6]
Out[2024]= {6.8, 0.56834, 0.638629, 0.742415, 0.917528, 1.30655, 3.69585}
```

Итак, функция **NestList** является расширением предыдущей функции **Nest** подобно тому как процедура **NestList** является расширением упомянутой процедуры **Nest** для *Maple* [99], однако, в отличие от одноименной функции процедура **NestList** обладает несколько более ограниченными возможностями, прежде всего, для создания чистых функций. В свою очередь функция **FoldList** возвращает список значений формата:

```
FoldList[f, x, {a, b, c, d, ...}] ≡ {x, f[x, a], f[f[x, a], b], f[f[f[x, a], b], c], f[f[f[f[x, a], b], c], d], ...}
In[1042]:= FoldList[F, x, {1, 2, 3, 4}]
Out[1042]= {x, F[x, 1], F[F[x, 1], 2], F[F[F[x, 1], 2], 3], F[F[F[F[x, 1], 2], 3], 4]}
In[1043]:= FoldList[Plus, 6.8, {4.2, 4.7, 6.7, 68, 63, 43, 14, 22}]
Out[1043]= {6.8, 11, 15.7, 22.4, 90.4, 153.4, 196.4, 210.4, 232.4}
```

Данная процедура, в частности, позволяет получать кумулятивные суммы элементов заданного списка, как иллюстрирует последний пример фрагмента. При этом, **Fold**-функция возвращает последний элемент списка, полученного по **FoldList**, например:

```
In[2044]:= Fold[G, x, {a, b, c, d, h, g, s, w, p, q}]
Out[2044]= G[G[G[G[G[G[G[G[G[x, a], b], c], d], h], g], s], w], p], q]
```

Тогда как в пакете *Maple* одноименная процедура **FoldList** может служить некоторым аналогом для функции **FoldList**; при этом, функция **Fold** очень просто реализуется на основе достаточно простой конструкции, представленной в нашей книге [99].

Функция **MapIndexed** располагает двумя форматами кодирования, а именно:

**MapIndexed**[F, V]    и    **MapIndexed**[F, V, L]

Вызов функции **MapIndexed**[F, V] первого формата возвращает список, элементами в котором являются применения F к подвыражениям V-выражения; более того, вторым аргументом F-функции выступает список с номером местоположения подвыражения

```
In[2045]:= MapIndexed[G, {a, b, c, d, f, g, h}]
Out[2045]= {G[a, {1}], G[b, {2}], G[c, {3}], G[d, {4}], G[f, {5}], G[g, {6}], G[h, {7}]}
In[2046]:= MapIndexed[G, x*Sin[x] + y*Cos[y]]
Out[2046]= G[y Cos[y], {1}] + G[x Sin[x], {2}]
```

Вызов функции **MapIndexed**[F, V, L] второго формата возвращает список, элементами в котором являются применения F ко всем подвыражениям V-выражения на уровнях, определенных 3-м фактическим L-аргументом; при этом, используется стандартная спецификация уровней, например:

```
In[2047]:= MapIndexed[G, x*Sin[x] + y*Cos[y], {2}]
Out[2047]= G[y, {1, 1}] G[Cos[y], {1, 2}] + G[x, {2, 1}] G[Sin[x], {2, 2}]
```

Тогда как в *Maple* аналогом *первому* формату функции **MapIndexed** может выступать процедура **MapIndexed**, с которой можно познакомиться в [99]. Подобным же образом

в пакете *Maple* можно получить аналоги и второго формата функции **MapIndexed**. В свою очередь первый формат функции **Map** полностью аналогичен встроенной *map*-функции *Maple*. Тогда как функция **MapAll**[*F*, *x*] возвращает результат применения *F* к каждому подвыражению выражения *x*. Простой фрагмент иллюстрирует сказанное:

```
In[2048]:= Map[G, {a, b, c, d, e, f, g, h}]
Out[2048]= {G[a], G[b], G[c], G[d], G[e], G[f], G[g], G[h]}
In[2049]:= Map[G, x*Cos[x] + y*Sin[y]]
Out[2049]= G[x Cos[x]] + G[y Sin[y]]
In[2050]:= MapAll[G, x*Cos[x] + y*Sin[y]]
Out[2050]= G[G[G[x] G[Cos[G[x]]]] + G[G[y] G[Sin[G[y]]]]]
```

Другие форматы таких средств, а также *map2*-функции *Maple* либо функционально пересекаются, либо допрограммируемы до возможностей друг друга. Способствует в данном вопросе как применение базовых встроенных функций *map*, *map2* и *op* языка *Maple*, так и наших процедур *expLS*, *map3* .. *map6*, *Nest*, *NestList*, *OP*, *op1* и др. [44,45].

Функция **TakeWhile**[*L*, *C*] возвращает подсписок списка *L*, начиная с его начала, чьи элементы удовлетворяют заданному логическому условию (*C*), например:

```
In[2051]:= TakeWhile[{1.4, 2.2, 4.3, 4.8, 6.3, 6.8, 68, 43, 48, 72, 14, 22}, # <= 70 &]
Out[2051]= {1.4, 2.2, 4.3, 4.8, 6.3, 6.8, 68, 43, 48}
```

Возвращаемый подсписок включает элементы исходного списка, чьи значения < 70. В свою очередь, вызов функции **NestWhile**[*F*, *x*, *ЛУ*] возвращает вложенное выражение, образуемое применением *F* к выражению *x*, пока результат не удовлетворит заданное логическое условие (*ЛУ*), как иллюстрирует следующий весьма простой пример:

```
In[2052]:= NestWhile[Tan, 13, # > 0 &]
Out[2052]= Tan[Tan[Tan[Tan[Tan[Tan[Tan[Tan[13]]]]]]]]]
```

Данная функция располагает еще 4-мя форматами кодирования, обеспечивающими более сложные вычисления. Между тем, все форматы вполне успешно симулируются процедурами пакета *Maple*, например, первому формату соответствует одноименная процедура *NestWhile*, приведенная в нашей книге [99]. Наряду с вышеприведенными пакет *Mathematica* располагает и другими довольно интересными средствами такого типа, однако по ряду причин здесь нами не рассматриваемыми.

При сравнительном рассмотрении *управляющих* структур *ветвления* и *цикла*, которые поддерживаются обоими пакетами, были выделены две основные группы, а именно: базовые и дополнительные средства обеспечения указанных управляющих структур. Предложение **if** пакета *Maple* и **If**-функция пакета *Mathematica* представляют собой наиболее типичное средство обеспечения ветвящихся алгоритмов. В процессе работы с обоими средствами сформировалась точка зрения, что оба 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 оба лидирующих пакета вполне можно рассматривать эквивалентными.

## Глава 4. Механизм типирования объектов в среде 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.

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.

*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, следует этот аспект рассмотреть и относительно пакета *Mathematica*. Отметим, в нашей книге [99] достаточно детально с точки зрения развитости механизма *типизации* рассматриваются оба пакета *Maple* и *Mathematica*, как наиболее развитые и популярные на сегодня универсальные CAS.

## 4.1. Механизм типирования объектов в среде пакета Mathematica

Сразу же следует отметить, что механизм типизации объектов, которым располагает *Mathematica*, существенно уступает аналогичному механизму пакета *Maple*, но лишь относительно встроенных типов тестирования объектов. Между тем, средства пакета *Mathematica* позволяют тестировать типы наиболее важных объектов. В частности, по функции **FileType** обеспечивается проверка быть каталогом либо файлом, например:

```
In[522]:= FileType["D:\\Math_myLib"]
Out[522]= Directory
In[523]:= FileType["D:\\Math_myLib\\ArtKr.mx"]
Out[523]= File
In[524]:= FileExistsQ["D:\\Math_myLib\\ArtKr.mx"]
Out[524]= True
In[525]:= FileExistsQ["D:\\Math_myLib"]
Out[525]= True
```

Однако это средство уступает нашим процедурам *isFile* и *isDir* для пакета *Maple* [45], обеспечивающим тестирование файлов и каталогов соответственно, например, по

```
> isFile("D:\\Math_myLib\\ArtKr.mx", h), h;  =>  true, close
```

не только тестируется собственно наличие файла, но и режим его открытости, что в целом ряде случаев весьма важно. Имеются и другие интересные средства [41-45] для тестирования состояния каталогов и файлов, включая их типы. Для тестирования у *Mathematica* имеется и функция **FileExistsQ**, которая возвращает *True*, если объектом является файл или каталог, что с точки зрения файловой системы вполне корректно, тогда как для пользователя, работающего с файлами – это не одно и то же, что вполне наглядно иллюстрирует следующий достаточно простой пример, а именно:

```
In[327]:= 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[327]= 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 обеих процедур и примеры их применения.

```
In[824]:= IsFile[x_] := If[FileExistsQ[x], If[! DirectoryQ[x], True, False], False]
In[825]:= IsFile["D:\\Math_myLib"]
Out[825]= False
In[826]:= IsFile["D:\\Math_myLib\\ArtKr.mx"]
Out[826]= True
In[959]:= 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[960]:= Map[FileQ, {"c:/Temp/Lugupeetud.docx", "Rans.Ian", "G:/New_Book/Art.Kr"}]
Out[960]= {True, True, False}
```

Для дифференцированного тестирования файлов используется и функция **FileType**:

```
In[1027]:= {FileType["D:\\Math_myLib"], FileType["D:\\Math_myLib\\ArtKr.mx"]}
Out[1027]= {Directory, File}
```

Пакет *Mathematica* располагает также рядом других подобных тестирующих средств, рассматриваемых несколько ниже.

В отличие от 209 типов, например, пакета *Maple 11*, тестируемых процедурой *type* (не считая значительного набора пользовательских типов, подключенных к пакету посредством библиотеки [45]), *Mathematica 8* 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* пакета *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 таких средств.

Из упомянутых *Q*-функций можно отметить следующие, наиболее используемые:

**AlgebraicIntegerQ, ArrayQ, AtomQ, DigitQ, DirectoryQ, EvenQ, ExactNumberQ, ExistsRealQ, FileExistsQ, ForAllRealQ, FreeQ, ImpliesQ, ImpliesRealQ, IntegerQ, IntervalMemberQ, LetterQ, MatchQ, MatrixQ, MemberQ, NumberQ, OddQ, OrderedQ, PolynomialQ, PrimePowerQ, PrimeQ, SameQ, SquareFreeQ, StringFreeQ, StringMatchQ, StringQ, SyntaxQ, TautologyQ, TrueQ, ValueQ, VectorQ**

Даже обладая достаточно ограниченными познаниями в английском, математике и программировании, несложно понять назначение приведенных функций. Вместе с тем, ниже будет представлен ряд нестандартных тестирующих средств данного типа наряду с другими, имеющими аналогичное назначение.

Пакет *Mathematica* располагает механизмом специального *типирования* выражений в шаблонах на основе функций **Head** либо **Part**, позволяющих получать не только тип выражения, но и применять к его подвыражениям функции, в зависимости от типов первых. В частности, функции **Head** и **Part** позволяют получать *общий* тип выражения и в этом отношении они, практически, аналогичны процедурам *whattype* и *op* пакета *Maple* соответственно; процедуры *whattype* и *op* пакета *Maple* имеют более широкие возможности по определению типов *Maple*-объектов, чем **Head** и **Part** по определению типов *Mathematica*-объектов [99], что обусловлено, скорее всего, более ограниченным набором типов, распознаваемых пакетом *Mathematica*. Следующие простые примеры достаточно наглядно иллюстрируют использование функций **Head** и **Part**, а именно:

```
In[424]:= Head[x + y]
Out[424]= Plus
In[425]:= Part[x + y, 0]
Out[425]= Plus
In[426]:= Head[{1, 2, 3, 4, 5}]
Out[426]= List
In[427]:= Part[{1, 2, 3, 4, 5}, 0]
Out[427]= List
In[428]:= Head[Cos[x]]
Out[428]= Cos
In[429]:= Part[Cos[x], 0]
Out[429]= Cos
In[430]:= Head[Type]
Out[430]= Symbol
In[431]:= Part[Type, 0]
Out[431]= Symbol
```

Довольно полезной при анализе и обработке выражений дополнительно к функции **Head** представляется функция **Part**, допускающая шесть форматов кодирования, из которых рассмотрим лишь 3, по остальным можно обратиться к справке по пакету:

**Expr[[k]]** или **Part[Expr, k]** – возвращает  $k$ -ю часть выражения *Expr*; при этом, при  $k < 0$  отчет производится справа налево, при  $k = 0$  имеет место **Part[Expr, k]  $\equiv$  Head[Expr]**;  
**Expr[[k1, k2, ..., kp]]** или **Part[Expr, k1, k2, ..., kp]** – эквивалентно **Expr[[k1]][[k2]] ... [[kp]]**

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**; ниже использование функций **Head**, **Level** и **Part** в целом ряде наших функций и процедур оказывается достаточно эффективным.

```
In[1017]:= 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[1018]:= Decomp[{a*Cos[x] - n*Sin[y]/(Log[h] - b), ProcQ[c, d]}]
Out[1018]= {a, x, Cos, Times, -1, n, b, h, Log, Plus, Power, y, Sin, c, d, ProcQ}
```

Для модулей (*процедур*) не существует стандартных средств тестирования на тип, что является достаточно существенным недостатком. Между тем, использование *шаблонов* в сочетании с типами, которые распознаются **Head**, **Part**, в определении значений для переменных (*фактических аргументов*) имеет и свои положительные черты. В данном случае *Mathematica* дает возможность представлять *классы* выражений. Итак, главное преимущество шаблонов состоит в том, что многие операции пакета применимы как к отдельному выражению, так и к некоторому классу выражений. Например, общего вида шаблон *\_h* представляет *класс* выражений (*значений*), на которых вызовы функций **Head**, **Part** возвращают *h*-значение (*при этом, пустой шаблон '\_' идентифицирует любое выражение, допустимое пакетом*). Следующие типичные шаблоны являются одними из допустимых пакетом, а именно:

<i>x_</i>	выражения произвольного, допустимого пакетом типа
<i>x_h</i>	выражения общего типа <i>h</i>
<i>x_Integer</i>	выражения общего типа <i>Integer</i>
<i>x_List</i>	выражения общего типа <i>List</i>
<i>x_Symbol</i>	выражения общего типа <i>Symbol</i>

На основе подобных *шаблонов* возможно не только избирательно применять функции и операции к выражениям, как это иллюстрирует весьма простой фрагмент:

```
In[1044]:= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2} /. X_Real -> 0
Out[1044]= {0, 2, 3, 0, 5, 6, 7, 0, 9, 0, 11, 0, 0}
```

```
In[1045]:= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2} /. X_Real -> Round[x]
Out[1045]= {2, 2, 3, 5, 5, 6, 7, 9, 9, 11, 11, 12, 13}
In[1046]:= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2} /. X_Real -> Rationalize[x]
Out[1046]= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2}
In[1047]:= Map[Rationalize, {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2}]
Out[1047]= {3/2, 2, 3, 23/5, 5, 6, 7, 89/10, 9, 109/10, 11, 123/10, 66/5}
```

но и в качестве еще одного применения шаблонов можно отметить использование их в определении процедур и функций для указания типов, которые приемлемы для их фактических аргументов. Между тем, из примера следует не только сказанное, но и определенное ограничение имеющее место быть, а именно: не все функции пакета работают в такой конструкции. Что же до второго способа использования шаблонов, то здесь целесообразно сделать некоторые довольно существенные замечания.

Определяя функцию  $G[x, y]$  следующим образом, а именно:

```
In[1351]:= G[x_Integer, y_Rational] := Sin[x] + Cos[y]
```

получаем значения ее вызовов в зависимости от типов ее фактических аргументов:

```
In[1352]:= G[68.42, 19/42]
Out[1352]= G[68.42, 19/42]
In[1353]:= G[68, 42]
Out[1353]= G[68, 42]
In[1354]:= G[68.42, 42.68]
Out[1354]= G[68.42, 42.68]
In[1355]:= G[68, 19/42]
Out[1355]= Cos[19/42] + Sin[68]
In[1356]:= {MM[42, 68], MM[19.42, 68], MM[68, 19.42], MM[19.42, 19.47]}
Out[1356]= {6388, MM[19.42, 68], MM[68, 19.42], MM[19.42, 19.47]}
```

Из приведенного примера следует, что в случае указания при вызове функции  $G[x, y]$  по меньшей мере одного фактического аргумента, чей общий тип отличен от типа в определении функции, вызов функции возвращается *невывчисленным*. Таким образом, описанный механизм шаблонов стандартным образом невозможно использовать для точного тестирования фактических аргументов, передаваемых функции/процедуре, чьи общие типы отличны от допустимых для алгоритмов, реализуемых процедурами или функциями. Между тем, искусственный прием, базирующийся на применении условия  $\{;/\}$  в форме конструкции « $x\_;/ Тест$ », обеспечивает не только тестирование фактического значения для формального аргумента  $x$ , передаваемого при вызове, но и позволяет в случае недопустимости значения не только возвращать вызов функции либо процедуры невывчисленным, но и через предопределенную для формального  $x$ -аргумента переменную, например, *faultX* возвращать недопустимое значение, легко программно обрабатываемое, что весьма важно для робастного программирования.

```
In[1068]:= Sv[x_;/ If[PrimeQ[x], Clear[faultX]; True, faultX = x; False]] := x^3 + 450
In[1069]:= {Sv[7], faultX}
Out[1069]= {793, faultX}
```

```
In[1070]:= {Sv[45], faultX}
Out[1070]= {Sv[45], 45}
In[1071]:= {Sv[11], faultX}
Out[1071]= {1781, faultX}
```

Предыдущий фрагмент достаточно наглядно иллюстрирует вышеописанный прием. В определении простой функции **Sv** для ее единственного формального аргумента  $x$  задается тест в форме булевой функции на основе стандартной **If**-функции, которая при вызове **Sv**[ $x$ ] процедуры возвращает результат  $x^3 + 450$  на допустимом значении  $x$  (простое число) и невычисленный вызов на недопустимом значении  $x$ ; при этом, через предопределенную для аргумента  $x$  переменную **faultX** возвращается недопустимое значение, полученное аргументом  $x$  при вызове процедуры. Несложно данный прием распространяется и на произвольное фиксированное число формальных аргументов, тогда как в случае использования процедурой неопределенного числа формальных аргументов, приходится тестирование получаемых процедурой при вызове значений для ее формальных аргументов проводить в теле процедуры. Примеры данного типа рассматриваются несколько ниже в целом ряде наших процедур и функций.

В принципе, *Mathematica* располагает всеми необходимыми средствами по созданию достаточно развитого механизма тестирования типов, используя имеющиеся типы и определяя новые типы, которые отсутствуют в пакете. Так, нижеследующий пример представляет достаточно простой модуль **Type**, который обеспечивает тестирование *Mathematica*-объектов на **ListList**-тип, т.е. на тип вложенного списка, чьи элементы, в свою очередь, также являются списками одинаковой длины, а именно:

```
In[2664]:= Type[x_, ListList] := Module[{a, b}, If[! ListQ[x], Return[False],
      a = DeleteDuplicates[Map[ListQ, x]]; b = If[a == {True} &&
      Length[DeleteDuplicates[Map[Length, x]]] == 1, True, False]]
In[2665]:= L := {a, b, c}; L1 := {{a, b}, {c, d}, {e, f}}; Type[L, ListList]
Out[2665]= False
In[2666]:= Type[L1, ListList]
Out[2666]= True
```

Более того, можно использовать достаточно простой подход для создания процедуры **Type**, аналогичной процедуре *type* пакета *Maple*. Вызов данной процедуры **Type**[ $x$ ,  $y$ ] возвращает **True**, если  $x$ -объект имеет  $y$ -тип, в противном случае возвращается **False**. Предлагаемый подход позволяет достаточно легко включать в тело процедуры **Type** определения новых типов, не изменяя принципиальной организации процедуры, с которой несложно ознакомиться по нижеследующему варианту процедуры **Type** для тестирования объектов на их принадлежность к двум типам **ListList** и *module*.

```
In[66]:= Type[x_, y_] := Module[{err, c}, err::Type = "The type <1`> has not been recognized";
      c = {ListList, module};
      If[MemberQ[c, y], Goto[y], Return[Message[err::Type, y]]];
      Label[ListList]; Module[{a, b}, If[! ListQ[x], b = False,
      a = DeleteDuplicates[Map[ListQ, x]]; b = If[a == {True} &&
```

```
Length[DeleteDuplicates[Map[Length, x]]] == 1, True, False]; Return[b]];
Label[module]; Module[{a, b, c}, a := ToString[Definition[x]];
    b := StringPosition[a, " := Module[{"];
    If[b == {}, Return[False], Null]; c := StringReplace[StringTake[a,
        {1, First[First[b]]}], {ToString[x] -> "", "_" -> "", "[" -> "{", "]" -> "}"}];
    Return[If[Head[ToExpression[c]] == List, True, False]]]
```

In[67]:= Type[{{a, b}, {c, d}, {e, h}}, ListList]

Out[67]= True

In[68]:= Type[{{a, b}, c\*d, {e, h}}, ListList]

Out[68]= False

In[69]:= Type[Type, module]

Out[69]= True

In[70]:= Type[Type, function]

err\$4287::type: The type <function> has not been recognized.

Как следует из исходного текста процедуры **Type**, вполне достаточно определить для типа *Имя*, запрограммировать и отладить определение *мина* и разместить его в теле процедуры после последнего определения, предварив его меткой **Label[Имя]**, и затем включить в список, определяемый локальной *\$a*-переменной, *имя* нового типа.

Достаточно несложно программировать и набор тестирующих процедур/функций, подобных стандартным средствам пакета и именуемым в форме «*Name\_TypeQ*». Так, располагая числовыми тестерами **IntegerQ**, **Head**, **PrimeQ**, **NumberQ**, **EvenQ**, **OddQ**, а также в определенной мере функцией **Head**, чей вызов **Head[Ex]** возвращает верхний уровень выражения *Ex*, *Mathematica*, между тем, не располагает средством проверки на *Complex*-тип произвольного выражения, как иллюстрирует простой фрагмент:

```
In[1949]:= ComplexQ[x_] := If[NumberQ[N[x]] && Im[x] != 0, True, False]
```

```
In[1950]:= ComplexQ := If[NumberQ[N[#]] && Im[#] != 0, True, False] &
```

```
In[1951]:= Mapp[ComplexQ, {Sqrt[-47], 2012, a + b*I, 69 + 42*I, 75*I}]
```

```
Out[1951]= {True, False, False, True, True}
```

```
In[1952]:= Map[Head, {69 + 42*I, Sqrt[-47], a + b*I, 75*I}]
```

```
Out[1952]= {Complex, Times, Plus, Complex}
```

Простая функция **ComplexQ**, реализованная двумя способами: в форме стандартной и чистой функции, решает задачу и полезна при работе с числовыми выражениями. Такой подход позволяет довольно просто организовать систему *типирования* пакета, однако он требует определенной *квалификации* пользователя; при этом, его довольно затруднительно *полностью* интегрировать в программную среду *Mathematica* (подобно случаю пакета *Maple*), когда в целом ряде версий *Maple* можно просто сохранять типы пользователя непосредственно в его *главной* библиотеке. В более старших версиях это весьма просто обеспечивается посредством логического сцепления пользовательских библиотек, содержащих определения типов, с *главной* библиотекой пакета. Тогда как в *Mathematica* возможно обеспечить автоматическую загрузку в *текущий* сеанс пакета

определения требуемых типов при каждой загрузке пакета. Между тем, *Mathematica* допускает программирование пользовательских типов с сохранением их в отдельном пакете, скажем, «*UserTypes`*», который при каждой новой загрузке *Mathematica* будет автоматически загружаться в его среду, обеспечивая вычисление содержащихся в нем определений типов в виде процедур/функций и делая доступными (*активными*) их в текущем сеансе наравне со стандартными средствами пакета. Именно данный подход использовался нами в период подготовки примеров для настоящей книги.

## 4.2. Дополнительные средства тестирования типов объектов в программной среде пакета *Mathematica*

Прежде всего следует отметить, набор распознаваемых *Maple* – наиболее серьезного конкурента *Mathematica* – типов существенно значительнее *аналогичного* набора для второго. Тут же вполне уместно подчеркнуть, средства тестирования типов, которые обеспечиваются, например, функцией `{typematch|type}` существенно более развиты, чем подобные им средства *Mathematica* [28-30,32,42,43,99]. В результате на основе этих средств предоставляется достаточно удобная возможность разработки эффективных средств программного тестирования типов выражений, данных и их структур наряду с организацией эффективной программной обработки как особых, так и ошибочных ситуаций, возникающих при рассогласовании типов.

Естественно, средства *Mathematica* вполне позволяют создавать механизмы, которые в значительной степени аналогичны средствам типирования и тестирования пакета *Maple*, однако это требует программирования, в целом ряде случаев нетривиального. В качестве одного из таких механизмов может выступить, в частности, приведенная в предыдущем разделе процедура `Type[x, y]`, чей вызов возвращает `True`, если *x*-объект имеет *y*-тип, в противном случае возвращается `False`. Организация процедуры `Type` обеспечивает ее достаточно простое расширение определениями новых типов. Таким образом, данная процедура является определенным аналогом процедуры `type` пакета *Maple* и вполне может быть расширена до функциональных возможностей второй.

Концепция *процедурно-функционального* программирования весьма существенный акцент делает на вопросе тестирования недопустимости *фактических* аргументов при вызовах процедур, который чрезвычайно важен с двух точек зрения, а именно:

- (1) использование процедур с достаточно развитой системой проверки передаваемых им фактических аргументов позволяет довольно существенно повышать робастность программных систем, содержащих подобные процедуры и располагая возможностью программной обработки ошибочных ситуаций, *обусловленных* передачей процедурам/функциям недопустимых фактических аргументов уже на этапе их вызова;
- (2) развитый механизм проверки передаваемых процедурам фактических аргументов дает возможность как на стадии отладки, так и в точках их вызова идентифицировать ошибки, обусловленные наличием среди фактических аргументов *недопустимых*, что позволяет существенно облегчить задачу отладки с концентрацией внимания только на ошибках, связанных с самой логикой запрограммированного алгоритма. И если в

случае функций, чей код, как правило, легко обозрим, отладка относительно проста, то для больших процедур задача их отладки требует, порой, значительных усилий.

Подобно механизму *типирования* формальных аргументов *Maple*, *Mathematica* также поддерживает не менее развитый механизм типирования *формальных* аргументов как функций, так и процедур; однако здесь имеется ряд *довольно* существенных отличий. Механизм *Maple* располагает большим количеством как простых, так и структурных встроенных типов наряду с достаточно простым приемом подключения к механизму типов пакета *Maple* пользовательских типов, что мы и делали в случае возникновения такой необходимости, создавая определения разных новых типов согласно принятым соглашениям пакета. В пакете *Maple*, в отличие от *Mathematica*, в значительной мере поддерживается механизм идентификации недопустимых фактических аргументов посредством возврата в точках вызова процедур на значениях подобных фактических аргументов соответствующих диагностических сообщений, тогда как *Mathematica* в аналогичных ситуациях возвращает вызов процедуры *невычисленным* (за исключением *встроенных функций и грубых ошибок*), довольно существенно усложняя важную задачу программной обработки недопустимого фактического аргумента.

Тогда как наиболее типичной реализацией механизма *проверки* аргументов процедур *Mathematica* на их допустимость является некоторый аналог рассмотренного метода типизации, который в качестве тестов, приписываемых формальным аргументам при определении процедуры, использует тестирующие конструкции формата «*x\_ /; Тест*»

**Proc[x\_ /; T<sub>x</sub>[x], y\_ /; T<sub>y</sub>[y], ...] := Module[{...}, <Тело процедуры>]**

где {T<sub>x</sub>[x], T<sub>y</sub>[y], ...} – некоторые *булевы функции*, тестирующие при вызове процедуры **Proc[x, y, ...]** допустимость фактических аргументов {x, y, ...}. Итак, на функции T<sub>p</sub>[p], возвращающей при вызове процедуры *True* на фактическом *p*-аргументе, *p*-аргумент полагается *допустимым*, и *недопустимым* при возврате значения *False*; *p* ∈ {x, y, ...}. Если вызов **Proc[x, y, ...]** процедуры будет содержать по крайней мере один недопустимый *фактический* аргумент, то на таком кортеже *фактических* аргументов он возвращается *невычисленным*. Ниже указанный метод тестирования допустимости аргументов при вызовах процедур в программной среде *Mathematica* вполне наглядно иллюстрирует следующий достаточно несложный фрагмент с процедурой **P**, а именно:

```
In[1464]:= P[x_ /; If[IntegerQ[x], Clear[$Args1]; True, $Args1 =
               "Args1::Error: type should be Integer, but was received <" <>
               ToString[Head[x]] <> ">"], y_ /; If[ListQ[y], Clear[$Args2];
               True, $Args2 = "Args2::Error: type should be List, but was
               received <" <> ToString[Head[y]] <> ">"]] := Module[{}, y[[x]]]

In[1465]:= P[72.42, {63, 68, 43, 14, 22, 48}]
Out[1465]= P[72.42, {63, 68, 43, 14, 22, 48}]
In[4166]:= $Args1
Out[1466]= "Args1::Error: type should be Integer, but was received <Real>"
In[1467]:= {P[3, {63, 68, 43, 14, 22, 48}], $Args1}
Out[1467]= {43, $Args1}
```

```
In[1468]:= {P[68, "Art22_Kr14"], $Args1, $Args2}
Out[1468]= {P[68, "Art22_Kr14"], $Args1, "Args2::Error: type should be List, but
            was received <String>"}
In[1469]:= Clear[$Args1, $Args2]
In[1470]:= {P[6.8, "Art22_Kr14"], $Args1, $Args2}
Out[1470]= {P[6.8, "Art22_Kr14"], "Args1::Error: type should be Integer, but
            was received <Real>", $Args2}
```

Выше представлен простой искусственный прием, базирующийся на использовании конструкции «*x\_ /; Test*», который обеспечивает не только тестирование *фактического* значения для формального аргумента *x*, передаваемого при вызове, но и позволяет в случае *недопустимого* значения не только возвращать вызов функции или процедуры *невывисленным*, но и через определенную для формального *x*-аргумента переменную, например, *faultX* возвращать недопустимое значение, программно обрабатываемое, что весьма важно для робастного программирования, как иллюстрирует пример:

```
In[1068]:= Sv[x_ /; If[PrimeQ[x], Clear[faultX]; True,
                    faultX = "Actual argument should have the type Prime" <>
                    ", but received " <> StrStr[x]; False]] := x^3 + 450
In[1069]:= {Sv[19], faultX}
Out[1069]= {7309, faultX}
In[1070]:= Sv[450]
Out[1070]= Sv[450]
In[1071]:= faultX
Out[1071]= "Actual argument should have the type Prime, but received 450"
In[1072]:= {Sv[127], faultX}
Out[1072]= {2 048 833, faultX}
```

В определении простой функции **Sv** для ее единственного формального аргумента *x* задается тест в форме булевой функции на основе стандартной **If**-функции, которая при вызове **Sv[x]** процедуры возвращает результат  $x^3 + 450$  на *допустимом* значении *x* (*простое число*) и невычисленный вызов на *недопустимом* значении *x*; при этом, через предопределенную для аргумента *x* переменную *faultX* возвращается соответствующее сообщение в связи с недопустимым значением, полученным аргументом *x* при вызове процедуры. Однако, на основе такого приема невозможен выход непосредственно из вызова процедуры на основе функции **Return** в случае обнаружения недопустимого фактического значения, а также прямая печать для данного случая соответствующей диагностики, как это наглядно иллюстрирует весьма простой пример, а именно:

```
In[1950]:= Art[x_ /; If[PrimeQ[x], True, Return["Error in x"]; Print["Error in x"]; False]] := x
In[1951]:= Art[13]
Out[1951]= 13
In[1952]:= Art[450]
Out[1952]= Art[450]
```

Кстати, в *Maple* также невозможна реализация такого механизма непосредственного выхода из процедуры (без выполнения предложений ее тела) по причине получения при ее вызове недопустимого значения на основе встроенной функции **RETURN**, однако выводить текстовую диагностику такой подход позволяет, как иллюстрирует пример:

```
> `type/PrimeQ` := x -> `if` (type(x, prime), true, [print("Argument x should be prime
                                     number, but received " | x), false][1]):
> PR := (x::PrimeQ) -> x^3 + 450: PR(19);  => 7309
> PR(121);
      "Argument x should be prime number, but received 121"
Error, invalid input: PR expects its 1st argument, x, to be of type PrimeQ, but received 121
> lasterror;
      "invalid input: %1 expects its %-2 argument, %3, to be of type %4, but received %5"
```

Определенный пользователем тип *PrimeQ* используется в простой функции *PR* для тестирования ее единственного формального аргумента; на допустимых значениях *x* вызов функции возвращает значение  $x^3+450$ , тогда как на недопустимых – на печать выводится соответствующее диагностическое сообщение с инициацией ошибки, чья программная обработка обеспечивается на основе значения *lasterror*-переменной. В данной ситуации предложенный подход к детализации тестирования фактических аргументов при вызове процедур/функций теряет свою актуальность и представляет определенный интерес лишь конкретно для программной среды пакета *Mathematica*.

Подобно случаю типирования формальных аргументов *Maple*-процедуры для случая процедуры *Sv* пакета *Mathematica* использован тот же самый подход, однако с учетом особенностей *второго* пакета. Между тем, и здесь дается довольно детальное описание ошибочных ситуаций, возникающих при *несоответствии* фактических аргументов, получаемых *Sv*-процедурой при ее вызове, типам соответствующих им формальных аргументов на основе значений глобальных **faultX**-переменных (*по числу формальных аргументов*). В принципе, все различие, сводится только к тому, что для пакета *Maple* тестирующую процедуру/функцию необходимо отдельно определить в виде *muna* в соответствии с принятыми соглашениями пакета, тогда как для пакета *Mathematica* подобный *тест* можно непосредственно определять и в самом заголовке процедуры, используя, в частности, механизм чистых функции для большей компактности.

Естественно, в случае достаточно простых проверок фактических аргументов вполне *резонно* кодировать их в самом определении процедуры, как наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[1510]:= VS[N_;/ IntegerQ[N], L_;/ ListQ[L] && MemberQ[{0, 1, {0, 1}}
                                     Sort[DeleteDuplicates[Flatten[L]]]] := Block[{}, L[[N]]]
In[1511]:= VS[2, {63, 68, 14, 22, 43, 48}]
Out[1511]= VS[2, {63, 68, 14, 22, 43, 48}]
In[1512]:= VS[-1, {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[1512]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
```

```
In[1523]:= VS[N_/, IntegerQ[N], L_/, BinaryListQ[L]] := Block[{}, L[[N]]]
In[1524]:= VS[2, {63, 68, 14, 22, 43, 48}]
Out[1524]= VS[2, {63, 68, 14, 22, 43, 48}]
In[1525]:= VS[-1, {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[1525]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
```

Между тем, если проверка даже достаточно сложная носит ярко выраженный *разовый* характер, то ее можно полагать вполне уместной для кодирования непосредственно в самом шаблоне формального аргумента, что в целом ряде случаев достаточно просто обеспечивается использованием средств *функционального* программирования. Однако, в случае довольно сложных тестирующих функций или функций, носящих довольно массовый характер, мы рекомендовали бы оформлять отдельными тестами, позволяя делать определения процедур более прозрачными, наряду с обеспечением доступа к этим тестам другим процедурам и функциям.

При этом, следует иметь в виду, что понятие *типа* в *Mathematica* трактуется намного шире, чем это имеет место в обычном понимании. Под определение «*типа*» подходит любое описание объекта, которое может быть представлено любой *булевой* функцией, возвращающей на некотором свойстве объекта значение **True**, и в отсутствие данного свойства возвращающей значение **False**. Следующий фрагмент иллюстрирует весьма простую процедуру **VSV[x, y]**, чьи оба *формальных* аргумента типированы простыми отношениями {**IntegerQ**, **<**, **>**}, определяя группы типов – целые, большие или равные **72**, и целые, меньшие или равные **420**. Естественно, что типизация может принимать достаточно сложный вид, определяемый реализуемым процедурой алгоритмом.

```
In[529]:= VSV[x_/, If[IntegerQ[x] && x >= 72, True, err::VSV = "First argument <`1`>
is less than 72"; Message[err::VSV, x]; False],
y_/, If[IntegerQ[y] && y <= 420, True, err::VSV = "Second argument <`1`>
is greater than 420"; Message[err::VSV, y]; False]] := Module[{}, x + y]
In[530]:= VSV[72, 420]
Out[530]= 492
In[531]:= VSV[63, 420]
err::VSV: First argument <63> is less than 72
Out[531]= VSV[63, 420]
In[532]:= VSV[72, 450]
err::VSV: Second argument <450> is greater than 420
Out[532]= VSV[72, 450]
In[533]:= VSV[63, 450]
err::VSV: First argument <63> is less than 72
Out[533]= VSV[63, 450]
In[534]:= VSV[x + y, 450]
err::VSV: First argument <x+y> is less than 72
Out[534]= VSV[x + y, 450]
In[535]:= Messages[err]
Out[535]= {HoldPattern[err::VSV] :> "First argument <`1`> is less than 72"}
```

Тогда как возможность включения в определение типа диагностических сообщений в значительной степени позволяет облегчить программную обработку возникающих в точке вызова процедуры **VSV[x, y]** особых или ошибочных ситуаций, обусловленных передачей процедуре недопустимых фактических аргументов. Алгоритмы подобной обработки могут базироваться, например, на информации, получаемой по функции **Messages[err]**, которая возвращает список всех сообщений, приписанных указанному символу, в частности, *err*, как это иллюстрирует последний пример фрагмента.

Кодирование определений типов непосредственно в заголовке процедур имеет место лишь для пакета *Mathematica*, предоставляя возможность в точке вызова процедуры, не выполняя ее и не обращаясь к внешним для нее средствам, проводить проверку на допустимость получаемых ею фактических аргументов. Такой подход как повышает эффективность процедуры, так и делает ее более мобильной. Данный подход весьма удобен, когда тип носит узкоспециальный характер или его определение описано не очень большим и достаточно прозрачным кодом.

Подобно *Maple*, пакет *Mathematica* не имеет возможности тестировать в точке вызова процедуры недопустимость всех ее фактических аргументов, прерывая вызов уже на первом встреченном недопустимом фактическом аргументе. Между тем, принимая во внимание важность установления *всех недопустимых* аргументов за один проход, была создана процедура **TestArgsTypes[P]**, решающая данную весьма важную задачу.

```
In[1847]:= TestArgsTypes[P_] := Module[{a, b, c, d, h, p, f = "$$Art23$$Kr15$$",
    t, s = "", Test}, a = ToString[InputForm[P]]; b = Flatten[StringPosition[a, "["]];
    If[b != {}, c = Symbol[StringTake[a, {1, b[[1]] - 1}]], Clear[$TestArgsTypes]; Return[P]];
    Test[x_List, y_List] := Module[{a = {}, b, c, d, h, k, t},
        If[Length[x] != Length[y], Return[$Failed], k = 1];
        For[k, k <= Length[x], k++, d = y[[k]];
            If[StringTake[d, {-1, -1}] == "_", a = Append[a, True],
                b = Flatten[StringPosition[d, "_/; "]];
                c = StringTake[d, {b[[2]] + 1, -1}];
                h = StringTake[d, {1, b[[1]] - 1}]; t = StringPosition[c, h];
                t = Quiet[Select[t, #[[1]] == 1 &&
                    ! LetterQ[StringTake[c, {#[[2]] + 1, #[[2]] + 1}]] | |
                    ! LetterQ[StringTake[c, {#[[1]] - 1, #[[1]] - 1}]] &&
                    ! LetterQ[StringTake[c, {#[[2]] + 1, #[[2]] + 1}]] | |
                    #[[2]] == 1 && ! LetterQ[StringTake[c, {#[[1]] - 1, #[[1]] - 1}]] &]];
                b = StringReplacePart[c, ToString[x[[k]]], t];
                a = Append[a, b]]; Map[ToExpression, a];
        If[! ProcQ[c], Return[P], t = Definition[c]; Save[f, t];
        b = SymbolName[c]; p = StringLength[b]; Label[h];
        d = Read[f, String]; If[StringLength[d] < p + 1, Goto[h],
            If[b <> "[" == StringTake[d, {1, StringLength[b] + 1}], s = s <> d, Goto[h]];
            Label[p]; d = Read[f, String];
            If[d === EndOfFile, Close[f]; DeleteFile[f], s = s <> d; Goto[p]];
```

```
s = StringTake[s, {1, Flatten[StringPosition[s, "] := "][[1]]];
a = ToExpression[StringReplacePart[StringReplace[a, b <> "[" -> "{", "}", {-1, -1}]];
s = Map[ToString, ToExpression[
StringReplacePart[StringReplace[s, b <> "[" -> "{", "}", {-1, -1}]]];
b = Test[Map[If[StringQ[#], StrStr[#], #] &, a], s]; Clear[$TestArgsTypes];
$TestArgsTypes = {}; For[k = 1, k <= Length[a], k++, If[b[[k]] == True,
Null, $TestArgsTypes = Append[$TestArgsTypes, {k, a[[k]]}]]]; b]
```

```
In[1848]:= 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[1849]:= VS[6, -4, "A", "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
```

```
Out[1849]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
```

```
In[1850]:= VS[6, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
```

```
Out[1850]= VS[6, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
```

```
In[1851]:= TestArgsTypes[VS[9, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
```

```
Out[1851]= {True, False, True, True, True}
```

```
In[1852]:= $TestArgsTypes
```

```
Out[1852]= {{2, 7.2}}
```

```
In[1853]:= TestArgsTypes[VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
```

```
Out[1853]= {True, False, True, False, False}
```

```
In[1854]:= $TestArgsTypes
```

```
Out[1854]= {{2, 7.2}, {4, vz}, {5, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}}}
```

```
In[1855]:= TestArgsTypes[VS[9, 0, "A", "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
```

```
Out[1855]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
```

```
In[1856]:= $TestArgsTypes
```

```
Out[1856]= $TestArgsTypes
```

Процедура **TestArgsTypes[P[...]]** обрабатывает вызов процедуры **P** таким образом, что возвращает результат *вызова* процедуры **P[...]** в отсутствие *недопустимых* фактических аргументов или *список*, состоящий из значений **{True, False}**, который порядком строго соответствует кортежу фактических аргументов при вызове процедуры **P**. Более того, через глобальную переменную **\$TestArgsTypes** возвращается *вложенный* список, чьи 2-элементные подписки описывают набор недопустимых фактических аргументов, а именно: первый элемент подписка определяет номер недопустимого фактического аргумента, а *второй* – его значение. Однако, для упрощения алгоритма тестирования, реализуемого процедурой, предполагается, что формальные аргументы процедуры **P** типированы или *шаблоном* «**\_**» (*подчеркивание*), или конструкцией «**Аргумент\_ /; Тест**» наряду с 1-1 соответствием при вызове процедуры **P** числа *фактических* и *формальных* аргументов определения процедуры. Более того, предполагается, вызов процедуры **P** невычисленным обусловлен только несоответствием типов фактических аргументов формальным аргументам процедуры. Впрочем, вопрос тестирования фактических на уровне заголовка процедуры/функции имеет смысл лишь тогда, когда их количество

фиксировано. Если же процедура/функция допускает и *необязательные* аргументы, то 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 следующий простой фрагмент, а именно:

```
In[1946]:= Ag[x_] := Plus[Sequences[{x}]]
In[1947]:= Ag[2, 3, 7, 11, 13, 17, 19, 23, 29, 31]
Out[1947]= 155
In[1948]:= Ag1[x_ /; PrimeQ[x]] := Plus[Sequences[{x}]]
In[1949]:= Ag1[2, 3, 7, 11, 13, 17, 19, 23, 29, 31]
PrimeQ::nonopt: Options expected (instead of 31) beyond position 1 in
PrimeQ[2,3,7,11,13,17,19,23,29,31]. An option must be a rule or a list of rules. >>
Out[1949]= Ag1[2, 3, 7, 11, 13, 17, 19, 23, 29, 31]
In[1950]:= Ag2[x_ /; DeleteDuplicates[Map[PrimeQ, {x}]] == {True}] :=
Plus[Sequences[{x}]]
In[1951]:= Ag2[2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 75, 450]
Out[1951]= Ag2[2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 75, 450]
In[1952]:= Ag2[2, 3, 7, 11, 13, 17, 19, 23, 29, 31]
Out[1952]= 155
In[1953]:= Av[x_] := Module[{a = {x}, b, d = {}, k = 1},
Plus[For[k, k <= Length[a], k++, b = a[[k]];
If[! PrimeQ[b], Print[ToString[k] <>
"th argument should be prime, but received " <> StrStr[b]];
Return[Defer[Av[x]], Continue[]]; Sequences[{x}]]]
In[1954]:= Av[2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 75, 450]
11th argument should be prime, but received 75
Out[1954]= Av[2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 75, 450]
In[1955]:= Av[2, 3, 7, 11, 13, 17, "Ian", 23, 29, 31, 75, 450]
7th argument should be prime, but received "Ian"
Out[1955]= Av[2, 3, 7, 11, 13, 17, "Ian", 23, 29, 31, 75, 450]
In[1956]:= Av[2, 3, 7, 11, 13, 17, 19, 23, 29, 31]
Out[1956]= 155
```

Вызов простой функции **Ag**[x,y,z,...] возвращает результат *суммирования* переданных ей фактических аргументов, количество которых произвольно; при этом, допустимо использование в качестве фактических аргументов произвольных выражений. Затем, предлагается модификация предыдущей функция **Ag1**[x] с попыткой наделить все ее формальные аргументы одинаковым типом, однако наиболее естественный способ в данном случае не работает. И только новая модификация в виде функции **Ag2**, когда в качестве тестера выступает булево выражение, охватывающее проверку на тип всех передаваемых на входе **Ag2**[x,...] фактических аргументов, обеспечивает корректное

типирование формальных аргументов, при обнаружении первого же недопустимого возвращая вызов функции *невывчисленным*. Наконец, небольшое усложнение функции **Ag2** до функции **Av** обеспечивает не только типирование всех поступающих на вход фактических аргументов, однако дополнительно к стандартному возврату вызова **Av** невывчисленным при обнаружении недопустимого аргумента на печать выводится по нему диагностическое сообщение. Здесь мы имеем дело с *типированием* формальных аргументов не в заголовке функции, а непосредственно в ее теле. Итак, при сложных алгоритмах проверки получаемых фактических аргументов на допустимость следует их программировать в теле процедур/функций, точнее, целесообразнее. Возвращаясь вновь к процедуре **TestArgsTypes[P[...]]**, обеспечивающей диагностику одновременно всех полученных на вход процедуры *P* фактических аргументов, отметим только, что дальнейшее *развитие* процедуры **TestArgsTypes** вполне может завершиться созданием во многих отношениях достаточно полезного средства манипулирования объектами процедурного типа, однако в задачу настоящей книги данный вопрос не входит.

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[2848]:= 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[57]:= TestArgsTypes[VSG[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[57]= {True, False, True, False, False}
In[58]:= $TestArgsTypes
Out[58]= $TestArgsTypes

In[252]:= LoadNameFromM[F_String, N_String] := Module[{a = "", b, c, h = "(*End[*]", g,
           d = "(*Begin[\"\" <> N <> \"\"]*)"},
           If[FileExistsQ[F] && FileExtension[F] == ".m" &&
           Read[F, String] == "(*::Package::*)", Null, $Failed];
           Label[b]; c = Read[F, String]; If[c === EndOfFile, Close[F];
           Return["Object <" <> N <> "> has not been found."],
           If[c != d, Goto[b], Label[g]; c = Read[F, String];
```

```

If[c === h, Close[F]; Return[ToExpression[a]],
a = a <> StringTake[c, {3, -3}]; Goto[g]]]]
In[253]:= LoadNameFromM["D:\Math_myLib\AVZ_Package.m", "Map3"]
In[254]:= Map3[LoadNameFromM, "D:\Math_myLib\AVZ_Package.m", {"StrStr",
"ProcQ", "TestArgsTypes"]}
Out[254]= {Null, Null, Null}
In[255]:= TestArgsTypes[VSG[9, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[255]= {True, False, True, True, True}
In[256]:= $TestArgsTypes
Out[256]= {{2, 7.2}}
In[257]:= TestArgsTypes[VSG[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[257]= {True, False, True, False, False}
In[258]:= $TestArgsTypes
Out[258]= {{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 *"D:/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*. Но как иллюстрирует первая часть предыдущего фрагмента, глобальная *\$TestArgsTypes*-переменная пакетом *Mathematica* таковой не рассматривается.

Для обеспечения механизма глобальных переменных (в том числе) создана достаточно простая процедура *LoadNameFromM[F, N]*, обеспечивающая загрузку и активизацию в текущем сеансе процедуры *N*, сохраненной в файле *F* *m*-формата с пакетом. Вторая часть фрагмента представляет исходный код процедуры и пример ее использования в загрузке процедуры *TestArgsTypes* с сопутствующими ей двумя процедурами наряду с последующей иллюстрацией корректных значений *\$TestArgsTypes*-переменной.

Для обеспечения корректной загрузки процедуры посредством *LoadNameFromM* при условии возможной необходимости дозагрузки в текущий сеанс также нестандартных процедур, чьи вызовы используются в такой процедуре, может быть довольно полезна процедура *CallsInProc[P]*, возвращающая список всех внешних процедур и функций, вызовы которых использует заданная процедура *P*.

```

In[388]:= CallsInProc[P_/, ProcQ[P]] := Module[{a = ToString[FullDefinition[P]],
b, c = {}, k = 1, TN},
TN[S_String, L_/, ListQ[L] && Length[Select[L, IntegerQ[#] &]] == Length[L]
&& L != {} := Module[{a = "", c, b = {}, k, p = 1},
For[p, p <= Length[L], p++, For[k = L[[p]] - 1, k != 0, k--, c = StringTake[S, {k,k}];
a = c <> a; If[c === " ", a = StringTake[a, {2, -1}];
If[Quiet[Check[Symbol[a], False]] === False, a = ""; Break[],
b = Append[b, a]; a = ""; Break[]]]]; b];
b = TN[a, b = DeleteDuplicates[Flatten[StringPosition[a, "["]]]][2 ;; -1];
b = DeleteDuplicates[Select[b, StringFreeQ[#, "`"] && !

```

```
MemberQ[{"Block", ToString[P], "Module"}, #] &&
ToString[Definition[#]] != "Null" &]; Map[ToExpression, b]]
```

```
In[389]:= CallsInProc[TestArgsTypes]
```

```
Out[389]= {Append, Clear, Close, Definition, DeleteFile, First, Flatten, For, Goto, If,
  InputForm, Label, Length, ProcQ, Quiet, Read, Return, Save, Select,
  StringLength, StringPosition, StringQ, StringReplace, StringReplacePart,
  StringTake, StrStr, Symbol, SymbolName, ToExpression, ToString}
```

```
In[390]:= CallsInProc[LoadNameFromM]
```

```
Out[390]= {Close, FileExistsQ, FileExtension, Goto, If, Label, Read, Return, ToExpression}
```

```
In[391]:= CallsInProc[CallsInProc]
```

```
Out[391]= {Append, Break, Check, Definition, DeleteDuplicates, First, Flatten, For,
  FullDefinition, If, IntegerQ, Length, ListQ, ProcQ, Quiet, Select, Sort,
  StringFreeQ, StringPosition, StringTake, Symbol, ToExpression, ToString}
```

Список, возвращаемый процедурой **CallsInProc[P]**, содержит как стандартные, так и внешние процедуры и функции пользователя, чьи вызовы использует процедура *P*. В любом случае все использованные в процедурах и функциях нестандартные средства представлены и в настоящей книге, и в нашем небольшом пакете *AVZ\_Package* [90]. В связи с вышеизложенным можно рассматривать механизмы *типирования Mathematica* достаточно развитыми; более того, с учетом функционального программирования и *функциональных* средств именно программная среда пакета *Mathematica* наилучшим образом отвечает задаче создания новых типов как в виде отдельных процедур, так и в составе шаблонов для формальных аргументов процедур/функций. Ниже приведен ряд процедур и функций, позволяющих более эффективно решать данную задачу. В первую очередь, данные средства носят достаточно общий характер, тогда как другие подобные средства *типирования* дополнительно к уже рассмотренным представлены в соответствующих главах настоящей книги и в целом ряде приведенных примеров.

Прежде всего, вызов простой функции **CharacterQ[x]** возвращает **True**, если аргумент *x* определяет символ с характеристическим кодом из диапазона 0 .. 255, в противном случае возвращается **False**. Ниже представлен код функции с примером применения.

```
In[953]:= CharacterQ[c_] := !NumberQ[c] && StringLength[ToString[c]] == 1
```

```
In[954]:= Map[CharacterQ, {Agn, "75", G, S, "V", 6}]
```

```
Out[954]= {False, False, True, True, True, False}
```

Эта булева функция имеет многочисленные приложения, включая представленные в настоящей книге примеры процедур и функций различного назначения.

Следующая простая функция обеспечивает тестирование строчной конструкции *W* на предмет наличия в ней корректного выражения; при этом, тестируется не часть *W* (*подстрока как выражение*), а строка полностью. Успешный вызов **ExprQ[W]** возвращает **True**, иначе возвращается **False**. Ниже даны исходный код функции и ее применение.

```
In[1815]:= ExprQ[W_] := If[StringQ[N], If[Quiet[ToExpression[W]] === $Failed, False,
  True], False]
```

```
In[1816]:= Map[ExprQ, {"(a+b)", "(a^^c+b)", "a\\b+c", "a**b+cos[x]/c", a + b}]
Out[1816]= {True, False, False, True, False}
```

Процедура **ExprQ** оказывается довольно полезной при программировании процедур/функций (*прежде всего, системного характера*) для обработки разного рода выражений, представленных в строчном формате.

Вызов процедуры **MixCaseQ[x]** возвращает *True*, если строка *x* содержит буквы (*также допустимы специальные символы*) на разных регистрах, и *False* в противном случае; если же *x* содержит лишь специальные символы, то возвращается "Special Characters"; ниже представлен исходный код процедуры наряду с примерами ее применения.

```
In[969]:= 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[970]:= Map[MixCaseQ, {Agn, "Asv", "RANS", "j", "∞"}]
Out[970]= {MixCaseQ[Agn], True, False, "Special Characters", "Special Characters"}
```

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, у которого среди элементов имеются списки. Следующий фрагмент представляет исходный код функции **NestListQ** наряду с типичными примерами ее применения.

```
In[901]:= NestListQ[x_ /; ListQ[x]] := If[x != {} && Select[x, ListQ[#] &] != {}, True, False]
In[902]:= Map[NestListQ, {{a, c, c, {m, n}}, {a, b, c}, {{a, b}, {a, b}}, {}]
Out[902]= {True, False, True, False}

In[903]:= NestQL[L_ /; ListQ[L]] := If[MemberQ[Map[ListQ[#] &, L], True], True, False]
In[904]:= {NestQL[{a, b, c}], NestQL[{a, {b, x}, c}], NestQL[{{a, b, x, c}}]}
Out[904]= {False, True, True}
```

Функция **NestQL[L]** из *второй* части предыдущего фрагмента эквивалентна функции **NestListQ[L]**, при вызове возвращая *True*, если список *L* является вложенным, и *False* в противном случае, но реализует несколько иной алгоритм; при этом, определения обеих функций используют чистые функции и весьма полезны в работе со списками.

Вызов простой функции **ListListQ[L]** возвращает *True*, если *L* является *списком списков* (*ListList*), иначе возвращается *False*; при этом, под списком *ListList*-типа понимается *вложенный* список, все элементы которого являются *списками*, имеющими одинаковую длину. Следующий фрагмент представляет код функции с примерами применения.

```
In[948]:= ListListQ[L_] := If[L != {} && ListQ[L] && Length[Select[L, ListQ[#] &]
Length[#] == Length[L[[1]]] &]] == Length[L], True, False]
In[949]:= Map[ListListQ, {{a, {b}, c}, {{a}, {b}, {c}}, {{a, b}, {c, d}}, {}]
Out[949]= {False, True, True, False}
```

Вызов простой, но достаточно полезной функции **SymbolQ[x]** возвращает **True**, если  $x$  является *символом*, и **False** в противном случае. Функция используется в целом ряде процедур и функций, представленных в настоящей книге. Фрагмент представляет и исходный код функции **SymbolQ**, и типичные примеры ее применения.

```
In[963]:= SymbolQ[x_] := If[NameQ[ToString[x]], True, False]
In[964]:= {x, y} = {Art, 75}; Map[SymbolQ, {Vsv, "Art", 2012, Sin[x], a + b, Ian, x, y}]
Out[964]= {True, False, False, False, False, True, True, False}
```

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 применения.

```
In[1989]:= SortQ[s_ /; StringQ[s]] := If[s === StringJoin[Sort[Characters[s]]], True, False]
In[1990]:= Map[SortQ, {"dsertyuhgfd", "123454321", "123456789"}]
Out[1990]= {False, False, True}
```

Наконец, 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.

```
In[2570]:= AtomicQ[x_] := Module[{F}, If[Map[F, x] === x, True, False]]
In[2571]:= Map[AtomicQ, {sin, a + b, 75, sqrt(c + d), a/b, a^x}]
Out[2571]= {True, False, True, False, False, False}
In[2671]:= StrExprQ[x_ /; StringQ[x]] := Module[{a}, Quiet[Check[ToExpression[x],
a = 450]]; If[a === 450, False, True]]
In[2672]:= Map[StrExprQ, {"/a/b", "a^b", "a (a + b)", "Avz=75=450", "(a + b/)" } ]
Out[2672]= {False, False, True, False, False}
```

Следующая процедура является полезным дополнением к процедуре **TestProcCalls**.

```
In[1593]:= CorrCall[P_ /; ProcQ[P] | | QFunction[ToString[P]], x__] :=
Module[{b = ArgsTypes[P], c = {}, d = {}, h = {}, k = 1},
b = If[NestListQ[b], b, {b}]; If[Length[{x}] != Length[b], Return[$Failed],
For[k, k <= Length[b], k++, c = Append[c, b[[k]][[1]]]];
b = MinusList[Flatten[b], {"Arbitrary"}]];
For[k = 1, k <= Length[{x}], k++, d = Append[d, c[[k]] -> ToString[{x}][[k]]]];
For[k = 1, k <= Length[b], k++, h = Append[h, StringReplace[b[[k]], d]]];
If[MemberQ[Map[ToExpression, h], False], $Failed, P[Sequences[{x}]]]
In[1594]:= M[x_, y_ /; ListQ[y], z_ /; PosIntQ[z]] := x + Length[y] + z
In[1595]:= {CorrCall[M, 75, {a, b}, 450], CorrCall[M, 75, 2012, 450], CorrCall[ProcQ, ProcQ]}
Out[1595]= {527, $Failed, True}
```

Вызов **CorrCall[P, x, y, ...]** возвращает **P[x, y, ...]** на допустимых аргументах, и **\$Failed** в противном случае. Фрагмент представляет исходный код с примерами применения.

## Глава 5. Обработка особых и ошибочных ситуаций в Mathematica

В процессе выполнения достаточно сложных алгоритмов возникают различного рода особые и ошибочные ситуации, которые для повышения робастности алгоритмов мы должны программно обрабатывать. От успешного решения данной задачи с учетом того, насколько мы сумели предусмотреть программную обработку всех возможных ситуаций, могущих возникать в процессе работы алгоритма, в значительной степени зависит его надежность, устойчивость, эффективность и конкурентноспособность. И в первую очередь это относится к средствам широкого (*массового*) использования, где робастность играет весьма существенную роль. В данном контексте мы и рассмотрим средства, предоставляемые пакетом *Mathematica* для решения данной задачи.

В процессе выполнения *Math*-алгоритма возможно появление особых и аварийных ситуаций, в большинстве случаев требующих специальной обработки во избежание серьезного нарушения вычислительного процесса. Пакет *Mathematica* располагает средствами для программной обработки *ошибочных* и *особых* ситуаций. Прежде всего, пакет располагает механизмом для обработки сообщений, генерируемых в процессе вычислений. Большинство функций *Mathematica* использует стандартный механизм для генерации ошибочных и ряда других сообщений. При этом, пользователь также вполне может использовать такой механизм для генерации сообщений, связанных с выполнением его функций. Основным принцип состоит в использовании сообщений в формате '*символ::tag*', где *символ* – некоторое имя, например, функции, тогда как *tag* – сообщение в формате строки. В целом, '*символ::tag*' определяет идентификатор сообщения, к которому именно по нему и обращаются. Сообщения можно задавать и прямым определением идентификатора, и по функции *MessageName*, например:

```
In[447]:= MessageName[Fn, "nargs"]
```

```
Out[447]= Fn::nargs
```

```
In[448]:= Fn::"nargs"
```

```
Out[448]= Fn::nargs
```

Строка *tag* может содержать любые символы, но для определения текста сообщения используется конструкция вида *Символ::tag* = «Текст», как иллюстрирует пример:

```
In[651]:= Fn::nargs = "incorrect number of actual arguments"
```

```
Out[651]= "incorrect number of actual arguments"
```

```
In[652]:= Message[Fn::nargs]
```

```
Fn::nargs: incorrect number of actual arguments
```

При этом, *MessageName[Символ, "tag", "language"]* или *Символ::tag::language* задает сообщение на конкретном *language*-языке. Стандартный набор сообщений для всех встроенных функций пакета написан на английском языке. Но в некоторых версиях пакета сообщения доступны также и на других языках, для чего следует определить переменную *\$Language*, имеющую по умолчанию значение "English". При этом, свои сообщения пользователь может задавать на других языках. При генерации некоторой пакетной функцией сообщения выборка, прежде всего, производится из сообщений группы *Z::tag::Language*, где язык определяется значением переменной *\$Language*, но

если соответствующего сообщения не было найдено, то поиск производится в группе сообщений **Z::tag** без указания языка. В случае невозможности *переопределения* языка **\$Language = Language** формальное переопределение производится, но реального нет, как иллюстрирует следующий достаточно наглядный пример, а именно:

```
In[764]:= $Language = "French"
Out[764]= "French"
In[765]:= $Language
Out[765]= "French"
In[766]:= 1/0
Power::infy: Infinite expression 1/0 encountered. >>
Out[766]= ComplexInfinity
```

Пакет имеет ряд зарезервированных групп сообщений, определяемых **tag**, например, идентификатор сообщения **Fn::usage** служит для определения справки по **Fn**-объекту, как это иллюстрирует следующий простой пример, а именно:

```
In[853]:= Fn::usage = "function Fn(n) serves for calculation of n-th prime number"
Out[853]= "function Fn(n) serves for calculation of n-th prime number"
In[854]:= ?Fn
function Fn(n) serves for calculation of n-th prime number
```

Назначения для **Z::tag** сохраняются в списке сообщений для **Z**-объекта, например:

```
In[855]:= Messages[Fn]
Out[855]= {HoldPattern[Fn::"nargs"] :> "incorrect number of actual arguments"
HoldPattern[Fn::"usage"] :> "function Fn(n) serves for calculation of n-th prime number"}
```

Управлять выводом сообщений для **Z**-объекта можно переключателями **Off[Z::tag]** и **On[Z::tag]** – выключать и включать вывод соответственно. Более того, выключать или включать можно не отдельные сообщения, а заданные либо группу сообщений. При этом, здесь следует отметить довольно существенное обстоятельство. Если в текущем документе все обстоит именно так, то использование переключателей **On/Off** внутри процедур требует специального приема, который состоит в том, что использование в процедуре переключателя **Off[Z::tag]** для подавления вывода соответствующих **Z::tag** сообщений требует включения в точки возврата результатов выполнения процедуры соответствующего переключателя **On[Z::tag]**, чтобы вызов процедуры не оказывал на внешнюю относительно процедуры программную среду нежелательного влияния из-за подавления вывода в ней **Z::tag** сообщений в результате вызова процедуры, как это весьма наглядно иллюстрирует следующий достаточно простой пример, а именно:

```
In[966]:= A[x_, y_] := Module[{}, Off[Power::infy];
If[PrimeQ[x], {x/y, On[Power::infy]}][[1]], {(x + 75)/y, On[Power::infy]}][[1]]]
In[967]:= A[75, 0]
Out[967]= ComplexInfinity
In[968]:= {x, y} = {75, 0}; a = x/y
Power::infy: Infinite expression 1/0 encountered. >>
Out[968]= ComplexInfinity
```

```

In[969]:= B[x_, y_] := Module[{}, Off[Power::infy];
                                If[PrimeQ[x], {x/y, On[Power::infy]}[[1]], (x + 75)/y]]
In[970]:= B[450, 0]
Out[970]= ComplexInfinity
In[971]:= {x, y} = {75, 0}; a = x/y
Out[971]= ComplexInfinity

```

Из приведенного фрагмента со всей очевидностью следует, обеспечение процедурой **A** отмены **Off**-переключателя во всех возможных точках выхода процедуры позволяет не распространять действие соответствующего ему активированного в процедуре **On**-переключателя на область вне процедуры. Тогда как в процедуре **B** игнорирование в одной из точек выхода из процедуры этого требования приводит к распространению действия **Off**-переключателя на область вне процедуры, что, вообще говоря, неверно.

Если *переключатели* **On/Off** позволяют управлять выводом сообщений на протяжении от и до точек вызова переключателей, то функция **Quiet[expr]** вычисляет выражение *expr* без вывода генерируемых в процессе вычисления выражений, например:

```

In[872]:= Sin[0]/0
Power::infy: Infinite expression 1/0 encountered. >>
∞::indet: Indeterminate expression 0 ComplexInfinity encountered. >>
Out[872]= Indeterminate
In[873]:= Quiet[Sin[0]/0]
Out[873]= Indeterminate

```

При этом формат функции **Quiet[expr, ...]** позволяет подавлять не только вывод всех сообщений, генерируемых выражением, но также заданные либо группу сообщений. В качестве еще одного полезного средства обработки сообщений выступает функция **Check**, которая не производит проверки на сообщения, выключенные по **Off**, а также сообщения, заблокированные по **Quiet**. При этом, функция **Check** действует также при использовании ее внутри **Quiet**. Вызов функции **Check** следующего формата

**Check[expr, expr1]**

вычисляет выражение *expr* с возвращением результата, если не были сгенерированы сообщения, в противном случае возвращается *expr1*; при этом, сами сообщения будут выводиться. Подавлять вывод сообщений можно либо на основе переключателя **Off**, либо функции **Quiet**, как это иллюстрирует следующий простой пример, а именно:

```

In[887]:= Check[Sin[0]/0, 1]
Power::infy: Infinite expression 1/0 encountered. >>
∞::indet: Indeterminate expression 0 ComplexInfinity encountered. >>
Out[887]= 1
In[888]:= Quiet[Check[Sin[0]/0, 1]]
Out[888]= 1

```

При этом, использование следующих двух форматов кодирования функции **Check**

**Check[expr, expr1, a1::tag1, ..., an::tagn]**    и    **Check[expr, expr1, "Имя"]**

позволяет обеспечивать вышеуказанную процедуру относительно только указанных сообщений или группы сообщений соответственно. Такие группы сообщений можно получить по предопределенной *переменной* **\$MessageGroups**, их состав определяется текущей версией пакета *Mathematica* и для 8-й версии предусмотрено 7 групп:

```
In[121]:= $MessageGroups[[All, 1]]
```

```
Out[121]= {"Compiler", "CompilerWarnings", "Graphics", "Obsolete", "Packing", "Spelling",  
          "Symbolics"}
```

При этом, явные имена сообщений формата **a::tag** могут сочетаться с именами групп. Большинство сообщений ассоциируется с конкретными функциями, однако имеется ряд сообщений, генерируемых и различными функциями. Например, при вызове **G-**функции с недопустимым числом фактических аргументов возвращается сообщение **G::argx**. В случае отсутствия текста для такого сообщения будет использоваться текст сообщения **General::argx**, как это иллюстрирует следующий простой пример:

```
In[941]:= Sin[42, 47]
```

```
Sin::argx: Sin called with 2 arguments; 1 argument is expected. >>
```

```
Out[941]= Sin[42, 47]
```

```
In[942]:= General::argx
```

```
Out[942]= "'1` called with `2` arguments; 1 argument is expected. "
```

При некоторых вычислениях могут генерироваться одинаковые сообщения, историю которых пакет сохраняет, однако выводит только 3 таких сообщения, завершая вывод информационным сообщением **General::stop**. При необходимости обеспечить вывод всех сообщений по **Off[General::stop]** подавляется действие данного механизма:

```
In[863]:= Do[1/0, {4}]
```

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

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

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

```
General::stop: Further output of Power::infy will be suppressed during  
this calculation. >>
```

```
In[864]:= Off[General::stop]
```

```
In[865]:= Do[1/0, {3}]
```

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

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

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

По вызову функции **MessageList[n]** возможно получать список имен всех сообщений, которые были сгенерированы в процессе вычисления *входа* с номером *n*; в то же время получать все сообщения, генерируемые в процессе вычисления конкретного входа *m*, можно по переменной **\$MessageList**, как это иллюстрирует следующий пример:

```
In[866]:= MessageList[68]
```

```
Out[866]= {Power::infy, Power::infy, Power::infy, General::stop}
```

```
In[867]:= In[42] := Tan[x, 2]; Cos[x, 4] + Sin[x, 2]; $MessageList
```

```
SetDelayed::write: Tag In in In[42] is Protected. >>
```

Cos::argx: Cos called with 2 arguments; 1 argument is expected. >>

Sin::argx: Sin called with 2 arguments; 1 argument is expected. >>

Out[867]= {SetDelayed::write, Cos::argx, Sin::argx}

В процессе создания и отладки программ довольно важным вопросом представляется предусмотреть обработку максимально возможного числа сообщений, которые могут возникать в процессе ее выполнения. Это напрямую связано с робастостью программ, в противном случае программа завершается аварийно или возвращает некорректный результат, что особенно нежелательно при использовании этого результата другими программами в сложных программных системах. Программная же обработка данных ситуаций позволяет создавать более надежное программное обеспечение. В качестве достаточно простого примера приведем фрагмент, в котором для тестирования типа актуального аргумента функции **Art[x]** используется механизм сообщений:

```
In[886]:= Art[x_] := Module[{Kr}, Kr::arg = "actual argument differs from rational";
  Kr[y_] := Module[{}, If[Element[y, Rationals], Numerator[y]*Denominator[y], Kr::arg]];
  If[Kr[x] === "actual argument differs from rational",
    ToExpression[StringReplace[ToString[x], "." -> "/"], Kr[x]]]
In[887]:= Art[89/96]
Out[887]= 8544
In[888]:= Art[89.96]
Out[888]= 89/96
```

В функции **Art[x]** определяется *сообщение* **Kr::arg**, на появление которого в результате вызова внутренней функции **Kr[y]** соответствующим образом реагирует **If**-функция, возвращая результат **a\*b**, если **x=a/b**, и **a/b**, если **x=a.b** (при наличии сообщения **Kr::arg**). Естественно, данную функцию можно закодировать несколько проще, не используя механизма сообщений, как это иллюстрирует следующий весьма простой фрагмент:

```
In[916]:= Art[x_] := Module[{}, If[Element[x, Rationals], Numerator[x]*Denominator[x],
  ToExpression[StringReplace[ToString[x], "." -> "/"]]]]
In[917]:= Art[89/96]
Out[917]= 8544
In[918]:= Art[89.96]
Out[918]= 89/96
```

Между тем, приведенный простой фрагмент довольно прозрачно иллюстрирует сам принцип использования механизма *сообщений* для обработки специальных ситуаций, программируемых пользователем. В случае обработки как ошибочных, так и особых ситуаций, генерируемых пакетом, можно использовать механизм, иллюстрируемый следующим достаточно простым фрагментом, а именно:

```
In[951]:= Kr[x_, y_] := Module[{}, Off[Power::"infy", Infinity::"indet"]; If[x/y ===
  ComplexInfinity | | x/y === Indeterminate, On[Power::"infy", Infinity::"indet"];
  Return[StringReplace["2nd or both arguments equal zero: 1", "1" -> ToString[{x, y}]]],
  On[Power::"infy", Infinity::"indet"]; x/y]]
```

```
In[952]:= Kr[89, 96]
Out[952]= 89/96
In[953]:= Kr[89, 0]
Out[953]= "2nd or both arguments equal zero: {89, 0}"
In[954]:= Kr[0, 0]
Out[954]= "2nd or both arguments equal zero: {0, 0}"
```

В теле определения функции **Kr**[*x*,*y*] выполняется блокировка двух типов сообщений, которые могут возникнуть при вычислении выражения *x/y* (ситуация вида *x/0* или *0/0*), и, если такая ситуация возникает, то возвращается соответствующая диагностика для последующей программной обработки; при этом, перед возвратом значения данная функция отменяет блокировку на вывод указанных типов сообщений. В качестве еще одного приема обеспечения функций пользователя диагностическими сообщениями приведем следующий простой пример, в котором блок проверки на тип получаемого функцией фактического аргумента кодируется сразу в заголовке процедуры. Однако перед вызовом процедуры следует вычислить возвращаемое сообщение, в противном случае при выводе сообщения появляется следующий текст «*Message text not found*»:

```
In[984]:= g::rational = "actual argument `1` is not rational"
Out[984]= "actual argument `1` is not rational"
In[985]:= g[x_] /; If[Head[x] === Rational, True, Message[g::rational, x]] := Module[{},
                                                    Numerator[x]^9 + Denominator[x]^9]
In[986]:= g[42.47]
           g::rational: actual argument 42.47 is not rational
Out[986]= g[42.47]
In[987]:= g[69/42]
Out[987]= 1821813708247
```

Пользователю при разработке проектов, содержащих программные средства, которые предполагают обработку различного типа сообщений *x::y*, возникающих в процессе их выполнения, в целом ряде случаев вполне целесообразно оформить такие сообщения отдельным файлом, например, *m-* или *nb-*формата с его предварительной загрузкой в текущий сеанс пакета, предваряя загрузку в него программной среды проекта.

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 [90].

```

In[1058]:= G::norational = "actual argument `1` is not rational"
Out[1058]= "actual argument `1` is not rational"
In[1059]:= G[x_] := If[Head[x] === Rational, Numerator[x]^9 + Denominator[x]^9,
                        Message[g::norational, x]; Defer[g[x]]]
In[1060]:= G[42.69]
G::norational: actual argument 42.69` is not rational
Out[1060]= G[42.69]
In[1061]:= Try[x_/, StringQ[x], y_/, ListListQ[y]] :=
Module[{a = Length[y], b = "If[Art22Kr15 === ", c = "", d, k = 1},
  For[k, k <= a, k++, c = c <> b <> ToString1[y[[k]][[1]]] <> ", " <>
    ToString1[y[[k]][[2]]] <> ", "; c = c <> x <> StringMultipleD["", a];
  ToExpression[x <> "; Art22Kr15 = Quiet[{MessageList[-1][[-1]]]; " <> c]]
In[1062]:= Try["G[42/69]", {{b, b1}, {{G::norational}, "Res"}, {c, c1}, {d, d1}}}
Out[1062]= 1821813708247
In[1063]:= Try["G[42.69]", {{b, b1}, {{G::norational}, "Res"}, {c, c1}, {d, d1}}}
G::norational: actual argument 42.69` is not rational
G::norational: actual argument 42.69` is not rational
Out[1063]= "Res"
In[1064]:= Quiet[Try["G[42.69]", {{b, b1}, {{G::norational}, "Res"}, {c, c1}, {d, d1}}]]
Out[1064]= "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 инициируемые таким вычислением *x* сообщения активизированы в текущем сеансе пакета. 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 пакета *Mathematica* коды иллюстрирующих примеров и процедур данного раздела довольно прозрачны и особых дополнительных пояснений не требуют.

## Глава 6. Организация механизмов процедур и функций в среде программирования пакета Mathematica

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 этому 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 [97-99,100,104,107,109-115,117], так и в справочной системе пакета *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* пакета *Mathematica* во многих отношениях отвечает данному решению.

Процедура в *Mathematica* определяется конструкциями следующего общего вида:

$$G[x_, y_, z_ \dots] := \text{Module}[\{a, b, c, \dots\}, \text{<Тело процедуры>}]$$

$$G[x_, y_, z_ \dots] := \text{Module}[\{a=a1, b=b1, c=c1, \dots\}, \text{<Тело процедуры>}]$$

где *G* – имя процедуры, *x\_, y\_, z\_ ...* – ее формальные аргументы любого допустимого пакетом типа, и *a, b, c, ...* – локальные переменные процедуры. Предложения пакета, составляющие *тело процедуры*, описывают алгоритм задачи, решаемой процедурой. В частности, второй формат определения процедуры обеспечивает возможность для *локальных* переменных присвоить начальные значения *a=a1, b=b1, c=c1, ...*. При этом, в случае отсутствия для модуля локальных переменных этот факт кодируется пустым списком, ибо по определению модуль определяется двумя параметрами; в противном случае при обращении к такому модулю (*процедуре*) будет инициирована ошибочная ситуация как иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[454]:= R[x_] := Module[h := Sin[x] + Cos[x]; h]
In[455]:= R[69]
Module::argr: Module called with 1 argument; 2 arguments are expected. >>
Out[455]= Module[h := Sin[69] + Cos[69]; h]
In[456]:= R[x_] := Module[{h}, h := Sin[x] + Cos[x]; h]
In[457]:= R[69]
Out[457]= Cos[69] + Sin[69]
```

Говоря формальным языком, процедура (*модуль*) выступает на уровне функции от 2-х обязательных аргументов. Этим определением процедура в *Mathematica* достаточно

существенно отличается, в частности, от определения процедуры в пакете *Maple*, где *произвол* при представлении составляющих ее компонент является существенно более гибким, хотя, пожалуй, это и не столь принципиально.

При определении достаточно сложных процедур/функций в ряде случаев возникает целесообразность определять некоторые их формальные аргументы *необязательными*. Более того, если определение процедуры содержит такие *необязательные* аргументы, для них должны указываться значения по умолчанию на тот случай, если при вызове процедуры такие аргументы будут опущены. Пакет *Mathematica* допускает 2 базовых метода определения *необязательных* аргументов, а именно: *позиционный* и *ключевой* на основе зарезервированных имен *необязательных* аргументов. Первый из этих двух методов наиболее распространен как при разработке *пользовательских* процедур либо функций, так и в реализациях стандартных процедур пакета. Суть его весьма проста и сводится к кодированию *формального* аргумента процедуры в форме  $X\_T:D$ , где  $X$  – имя формального аргумента, тогда как  $D$  и  $T$  определяют его значение по умолчанию и *допустимый тип* соответственно. Следующий простой пример достаточно наглядно иллюстрирует применение данного метода, а именно:

```
In[643]:= GS[x_Integer, y_ : 68, z_Integer] := Module[{a}, a = x + y^2 + z; a]
```

```
In[644]:= {GS[42, 2, 47], GS[42, 47]}
```

```
Out[644]= {93, 4713}
```

```
In[645]:= GS[x_, y_Integer: 68, z_] := Module[{a}, a = x + y^2 + z; a]
```

```
In[646]:= {GS[42, 2, 47], GS[42, 47], GS[42, 72, 420, 47]}
```

```
Out[646]= {93, 4713, GS[42, 72, 420, 47]}
```

*Позиционный* аргумент, в принципе, может занимать *произвольную* позицию в кортеже формальных аргументов, однако по ряду соображений их рекомендуется кодировать в конце кортежа *формальных* аргументов. Так, практически все стандартные функции пакета *Mathematica* поступают именно таким образом. При этом, пакет полагает, что аргументы *опускаются* с конца, как весьма наглядно иллюстрирует следующий очень простой пример, а именно:

```
In[672]:= G[x_ : 69, y_ : 64] := x^2 + y^3
```

```
In[673]:= {G[], G[15]}
```

```
Out[673]= {266905, 262369}
```

т.е., в результате кодирования единственного фактического аргумента при вызове *G*, опускается именно второй формальный аргумент функции. В дальнейшем, ту часть определения процедуры, которая предшествует «:=» будем называть ее *заголовком*.

Второй метод дает возможность иметь дело с *необязательными* аргументами (*опциями*), задавая *явные* имена таким аргументам и определяя их значения на основе некоторых правил преобразования. Данный метод особенно удобен для графических функций, которые, как правило, имеют большое количество *необязательных* аргументов, лишь некоторые из которых используются в конкретном случае. При определении данных *именованных* *необязательных* аргументов для процедуры/функции *G* довольно удобно сохранять их значения по умолчанию в виде списка правил преобразования, которые присваиваются *Options[G]*. При данном методе существует ряд подходов, с которыми

детальнее можно ознакомиться в справке по пакету. Типичный подход состоит в том, что значения для именованных необязательных аргументов определяются указанием соответствующих правил преобразования (*опций*) в конце фактических аргументов в точках вызова соответствующей процедуры/функции. Следующий простой пример иллюстрирует использование данного метода определения необязательных опций.

```
In[716]:= Options[G] = {Art -> 22, Kr -> 15}
Out[716]= {Art -> 22, Kr -> 15}
In[717]:= G[Z_, OptionsPattern[]] := H[Z, Z + OptionValue[Art] + OptionValue[Kr]]
In[718]:= {G[69], G[69, Art -> 75, Kr -> 420]}
Out[718]= {266905, H[69, 564]}
```

Подобно пакету *Maple* пакет *Mathematica* при определении процедуры обеспечивает развитую проверку фактических аргументов, получаемых при вызове процедуры, на их допустимость, что задается кодированием их в одном из трех форматов, а именно:

- 1) *X\_* – формальный аргумент *X* допускает *фактическое* значение произвольного типа
- 2) *X\_Type* – формальный аргумент *X* допускает *фактическое* значение типа *<Type>*
- 3) *X\_;/ Test[X]* – формальный аргумент *X* допускает *фактическое* значение, на котором *Test[X]* возвращает значение **True**.

При этом, второй формат предполагает, в качестве *Type* выступает значение **Head[X]**. В случае получения при вызове процедуры *фактического* значения типа, отличного от типа, определенного для соответствующего для него *формального* аргумента, то вызов процедуры возвращается *невычисленным*, как иллюстрирует следующий фрагмент:

```
In[513]:= Agn = 47; GS[x_, y_] := Module[{}, x + y]
In[514]:= GS[63, Agn]
Out[514]= 110
In[515]:= GS1[x_List, y_] := Module[{}, x + y]
In[516]:= GS1[63, Agn]
Out[516]= GS1[63, 47]
In[517]:= GS1[{63, 68, 43}, Agn]
Out[517]= {110, 115, 90}
In[518]:= GS2[x_;/ If[x < 100 && x > 50, True, False], y_;/ PrimeQ[y]] := Module[{}, x + y]
In[519]:= GS2[420, Agn]
Out[519]= GS2[420, 47]
In[520]:= GS2[72, Art]
Out[520]= GS2[72, Art]
In[521]:= GS2[72, Agn]
Out[521]= 119
In[522]:= GSv[x_Integer, y_] := Module[{}, x*y]
In[523]:= {GSv[42.47, 75], GSv[75, 450]}
Out[523]= {GSv[42.47, 75], 33750}
```

Таким образом, получаемые фактические аргументы процедура имеет возможность в

весьма широком диапазоне *тестировать*, определяя в качестве *теста* не только типы, стандартно определяемые пакетом, но и уже в своем *заголовке* кодировать достаточно сложные тесты, определяющие допустимость фактических аргументов, получаемых при вызове процедур. Если хоть один их фактических аргументов не проходит теста, вызов процедуры возвращается невычисленным, как это иллюстрирует предыдущий фрагмент. Отсутствующие в пакете типы можно его средствами определять как *типы* пользователя с непосредственным включением их в заголовок процедуры/функции или сохранять их определения в *{nb, m}*-файлах, которые после загрузки в текущий сеанс выступают на уровне стандартных средств пакета *Mathematica*. При этом, если недопустимый *фактический* аргумент для процедуры/функции инициирует возврат ее вызова невычисленным, то инициируется вывод ошибочной ситуации с возвратом соответствующего сообщения, давая возможность ее программно обрабатывать (*в том числе и посредством нашей процедуры Try*), не прерывая вычислительного процесса.

Механизм обеспечения возможности присвоения формальным аргументам значений по умолчанию базируется на следующем весьма простом шаблоне, а именно:

### Формальный аргумент\_ : Значение

Смысл значения по умолчанию состоит в том, что оно выступает в качестве значения в отсутствие данного аргумента при вызове процедуры/функции. Следующий весьма простой пример достаточно наглядно иллюстрирует вышесказанное, а именно:

```
In[633]:= Sv[x_ : 15, y_ : 22] := Module[{}, x + y]
```

```
In[634]:= {Sv[75, 450], Sv[75], Sv[]}
```

```
Out[634]= {525, 97, 37}
```

Между тем, наряду со стандартным механизмом для задания *формальным* аргументам значений по умолчанию может быть использован нестандартный прием, суть которого несложно усмотреть из нижеследующего достаточно прозрачного фрагмента:

```
In[635]:= Agn[x_ /; Quiet[If[SymbolQ[x], x = 75; True, x; True]],  
           y_ /; Quiet[If[SymbolQ[y], y = 2012; True, y; True]],  
           z_ /; Quiet[If[SymbolQ[z], z = 450; True, z; True]]] :=  
           Module[{}, {x + y + z, Clear["x", "y", "z"]}][[1]]
```

```
In[636]:= {Agn[22, 450, 15], Agn[15, 100, z], Agn[x, 110, z], Agn[x, 120, 300], Agn[x, y, z]}
```

```
Out[636]= {487, 565, 635, 495, 2537}
```

Таким образом, пакет *Mathematica* в достаточно 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. Однако одно наиболее полезное недокументированное средство все же имеет смысл отметить. Касается оно часто используемой процедуры *If*, а именно:

```
In[638]:= x = 15; If[x == 15, agn]
```

```
Out[638]= agn
```

```
In[639]:= x = 22; If[x == 15, avz]
In[640]:= x = 6; 100*If[x == 15, x^2]
Out[640]= 100 Null
```

Таким образом, наряду с форматами **If[a,b,c]** и **If[a,b,c,d]** функция **If** допускает также формат **If[a,b]**, по которому в случае истинности выражения *a* функция **If** возвращает значение *b*, в противном случае возвращается *Null*, т.е. ничего. Правда, использовать этот формат следует осторожно, как иллюстрирует последний пример фрагмента. В этой связи следует отметить, что у *Maple* значение *NULL* является более естественным в вычислениях, чем *Null* пакета *Mathematica*, как иллюстрируют простые примеры:

```
> {seq('if'(type(k, prime), k, NULL), k=1..40)};  => {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}
In[7]:= a = {}; For[k = 1, k <= 18, k++, a = Append[a, If[PrimeQ[k], k, Null]]]; a
Out[7]= {Null, 2, 3, Null, 5, Null, 7, Null, Null, Null, 11, Null, 13, Null, Null, Null, 17, Null}
In[8]:= a = {}; For[k = 1, k <= 18, k++, a = Append[a, If[PrimeQ[k], k, Null]]];
Select[a, # != "Null" &]
Out[8]= {2, 3, 5, 7, 11, 13, 17}
```

В частности, для устранения из получаемых результатов вычислений значения **Null** рекомендуется использовать дополнительные приемы, как иллюстрирует последний пример предыдущего фрагмента. Отдельно следует рассмотреть вопрос определения процедур/функций с произвольным числом формальных аргументов. С этой целью используются следующие три формата шаблонов, а именно:

- 1) *X\_* – одиночный обязательный формальный аргумент *X*;
- 2) *X\_\_* – произвольное конечное число формальных аргументов с обязательным *X*;
- 3) *X\_\_\_* – произвольное конечное число формальных аргументов или их отсутствие.

При этом, указанные шаблоны могут иметь *любые* из списка формальных аргументов процедуры/функции, как наглядно иллюстрирует следующий простой фрагмент:

```
In[975]:= P[x_] := Module[{}, {x}]
In[976]:= P1[x__] := Module[{}, {x}]
In[973]:= P2[x___] := Module[{}, {x}]
In[977]:= {P[], P[75], P[450, 75], P[69, 64, 44]}
Out[977]= {P[], {75}, P[450, 75], P[69, 64, 44]}
In[978]:= {P1[], P1[75], P1[450, 75], P1[69, 64, 44], P1[69, 64, 44, 75]}
Out[978]= {P1[], {75}, {450, 75}, {69, 64, 44}, {69, 64, 44, 75}}
In[979]:= {P2[], P2[75], P2[450, 75], P2[69, 64, 44], P2[69, 64, 44, 75]}
Out[979]= {{}, {75}, {450, 75}, {69, 64, 44}, {69, 64, 44, 75}}
In[980]:= G[x___] := Module[{args = {x}, nargs = Length[{x}], k = 1},
                                {args, nargs, Table[{x}[[k]], {k, nargs}]]}
In[981]:= G[6, 75, 450, 44, 69, 64, 15, 22]
Out[981]= {{6, 75, 450, 44, 69, 64, 15, 22}, 8, {6, 75, 450, 44, 69, 64, 15, 22}}
In[993]:= V[x_, y_] := Module[{}, {{x}, {y}}]
```

```
In[994]:= V[a, b, c, d, g, h]
Out[994]= {{a, b, c, d, g}, {h}}
In[995]:= V1[x_, y_] := Module[{}, {{x}, {y}}]
In[996]:= V1[g]
Out[996]= {{}, {g}}
```

Для работы с *неопределенным* числом формальных аргументов *Maple* данный вопрос достаточно просто решает на основе процедурных переменных *args*, *nargs* и других, в программной среде *Mathematica* ситуация обстоит несколько иначе. Подобными процедурными переменными пакет не располагает. Между тем, некоторые аналоги в данной связи все же вполне могут быть определены, как иллюстрирует предыдущий и следующий фрагменты, в частности, в качестве данных аналогов могут выступать следующие конструкции при условии, что процедура имеет заголовок **P[x\_, y\_, z\_, ...]**

**{x}** – список всех фактических аргументов, полученных процедурой **P** при вызове;  
**Length[{x}]** – число фактических аргументов, полученных процедурой **P** при вызове;  
**{x}[[k]]** – *k*-й фактический аргумент, полученный процедурой **P** при вызове.

Следующий фрагмент иллюстрирует использование указанных конструкций как для процедур, так и для функций, включая чистые функции.

```
In[607]:= Fargs[x_] := Module[{args = {x}, nargs = Length[{x}], k, {args, nargs,
(A)
Sum[args[[k]], {k, 1, nargs}]]]

In[608]:= Fargs[42, 47, 67]
Out[608]= {{42, 47, 67}, 3, 156}

In[609]:= S[x_, y_] := Module[{args = {x, y}, nargs = Length[{x, y}]}, {args, nargs}]
In[610]:= S[42, 47, 67, 15, 22]
Out[610]= {{42, 47, 67, 15, 22}, 5}

In[611]:= G[x_, y_, z_] := Module[{args = {x, y, z}, nargs = Length[{x, y, z}]}, {args, nargs}]
In[612]:= G[42, 47, 67, 15, 22]
Out[612]= {{42, 47, 67, 15, 22}, 5}

In[652]:= Z := Map[H, {##}] & (B)
In[653]:= {Z[42, 47, 67, 15, 22], Z[75], Z[]}
Out[653]= {{H[42], H[47], H[67], H[15], H[22]}, {H[75]}, {}}

In[678]:= PP := {args = {##}, nargs = Length[args], Sum[args[[k]], {k, 1, nargs}]}[[-1]] &
In[679]:= {PP[], PP[42, 47, 67, 15, 22], {args, nargs}}
Out[680]= {0, 193, {{42, 47, 67, 15, 22}, 5}}

In[781]:= GG := Block[{k = 1, args = {##}, nargs = Length[{##}]}, (C)
Sum[args[[k]], {k, 1, nargs}]] &
In[782]:= {GG[], GG[42, 47, 67, 15, 22], args, nargs}
Out[782]= {0, 193, args, nargs}
```

Действительно, указанные механизмы позволяют достаточно несложно организовать работу с модулями, чистыми функциями и блоками, определения которых содержат

неопределенное число формальных аргументов. На примерах предыдущего фрагмента проиллюстрированы механизмы определения переменных *args* и *nargs*, аналогичных одноименным процедурным переменным *Maple*. Для модульных объектов в качестве *формального* аргумента определяется идентификатор *x* с шаблоном «*\_\_*», позволяя при вызове процедуры кодировать любое количество ( $\geq 1$ ) фактических аргументов; в то время как шаблон «*x\_\_*» позволяет при вызове процедуры кодировать любое число ( $\geq 0$ ) фактических аргументов; более того, для указанных аргументов используются простые конструкции *args* = {*x*}, *nargs* = Length[{*x*}], {*x*}[[*k*]] (А). Между тем, наиболее простым является определение неопределенного количества *формальных* аргументов для *чистых* функций (В), тогда как для *блочных* объектов (С) *блочные* переменные *args*, *nargs*, {*x*}[[*k*]] определяются способом, аналогичным способу для процедур/ функций, представленному в самом начале предыдущего фрагмента (А).

Следует обратить внимание на использование механизма *типирования* в процедурах/ функциях с *неопределенным* числом фактических аргументов. В этом случае имеется возможность для всех аргументов определять единый тип, как иллюстрирует весьма простой пример, в котором функция **H** суммирует значения простых чисел, которые следуют за простыми из кортежа полученных при вызове простых чисел, возвращая вызов невычисленным при обнаружении хоть *одного* значения, отличного от простого числа. В то время как подобная ей функция **H1** отличается лишь тем, что формально допускает вызов также и на пустом кортеже фактических аргументов, но при таком вызове инициируется ошибочная ситуация, вызванная отсутствием у тестирующей функции **PrimeQ** обязательного аргумента. С целью устранения данной ситуации в предпоследнем примере фрагмента в качестве *аналога* для функции **H1** представлена несколько усложненная функция **H2[x]**, которая обеспечивает корректный вызов на произвольном числе фактических аргументов, включая их отсутствие. Предлагается ряд подходов к решению данной задачи на основе стандартных средств пакета.

```
In[1007]:= H[x__;/ PrimeQ[x]] := Sum[NextPrime[{x}][[k]], {k, Length[{x}]]]
In[1008]:= k = 420; {H[29], H[2, 3, 7, 75]}
PrimeQ::nonopt: Options expected (instead of 75) beyond position 1 in
PrimeQ[2,3,7,75]. An option must be a rule or a list of rules. >>
Out[1008]= {31, H[2, 3, 7, 75]}
In[1009]:= H1[x__;/ PrimeQ[x]] := Sum[NextPrime[{x}][[k]], {k, Length[{x}]]]
In[1010]:= {H1[], H1[29], H1[2, 3, 7, 75]}
PrimeQ::argx: PrimeQ called with 0 arguments; 1 argument is expected. >>
PrimeQ::nonopt: Options expected (instead of 75) beyond position 1 in
PrimeQ[2,3,7,75]. An option must be a rule or a list of rules. >>
Out[1010]= {H1[], 31, H1[2, 3, 7, 75]}
In[1011]:= H2[x_] := If[{x} == {}, Defer[H2[x]], Block[{a = Sum[If[PrimeQ[{x}][[k]],
NextPrime[{x}][[k]], Null], {k, Length[{x}]}], If[Head[a] === Plus, Defer[H2[x]], a]]]
In[1012]:= {H2[], H2[29], H2[2, 3, 7, 75]}
Out[1012]= {H2[], 31, H2[2, 3, 7, 75]}
```

```
In[1013]:= H3[x_ /; If[{x} == {}, False, If[DeleteDuplicates[Map[PrimeQ, {x}]] == {True},
      True, False]]] := Sum[NextPrime[{x}][[k]], {k, Length[{x}]}]
In[1014]:= k = 450; {H3[], H3[29], H3[2, 3, 7, 75]}
Out[1014]= {H3[], 31, H3[2, 3, 7, 75]}
```

Наконец, с целью устранения данной ситуации в последнем примере предыдущего фрагмента в качестве *аналога* для **H1** представлена функция **H3[x]**, которая в качестве общего теста для проверки всех получаемых при вызове **H3** фактических аргументов использует соответствующую булеву функцию, обеспечивающую корректный вызов функции **H3** на *произвольном* числе фактических аргументов, включая их отсутствие. Из представленных примеров следует, что переменная *суммирования* **k** в стандартной функции **Sum** является относительно нее локальной, что весьма удобно использовать в программировании. В связи со сказанным отметим, что даже в общем случае вполне реально создавать общий тест для проверки получаемых фактических аргументов в случае неопределенного их количества. Однако, в целом ряде случаев это приводит к довольно существенному усложнению тестирующей булевой функции со снижением обозримости заголовков процедур/функций. В этой связи становится целесообразно размещать алгоритмы тестирования в теле, прежде всего, процедур либо функций на основе блочных конструкций, что обеспечивает возможность использования локалов.

Между тем, используя ряд допустимых конструкций при определении формальных аргументов в *заголовках* процедур/функций, возможно вполне успешно нагружать их существенно более сложными функциями, отличными от их типизации, в частности, вычислять определения процедур/функций, необходимых для выполнения функций и процедур, чьи заголовки содержат подобные конструкции. Фрагмент представляет типичные примеры подобных конструкций и базовые обосновывающие выражения.

```
In[1439]:= {Null | | True, Null | | False, True | | Null, Null | | Null | | True}
Out[1439]= {True, Null, True, True}
In[1440]:= Prog[x_ /; ToExpression["d[y_] := Module[{}, y^3]" | | IntegerQ[x], y_] :=
      Module[{a = x, c = ToExpression["b[y_] := Module[{}, y^2]"]}, {a, x, b[2012], d[x], y}]
In[1441]:= x = 70; Prog[x, 450]
Out[1441]= {70, 70, 4048144, 343000, 450}
In[1442]:= Map[DefFunc3, {b, d}]
Out[1442]= {"b[y_] := Module[{}, y^2]", "d[y_] := Module[{}, y^3]"}
In[1443]:= Prog1[x_ /; ToExpression["d[y_] := Module[{}, y^3]" | | IntegerQ[x], y_] :=
      Module[{a = x, c = ToExpression["b[y_] := Module[{}, y^2]"]}, {a, x, b[2012],
      d[x], y, Clear[d, b]}][[1 ;; -2]]]
In[1444]:= x = 70; Prog1[x, 450]
Out[1444]= {70, 70, 4048144, 343000, 450}
In[1445]:= Map[DefFunc3, {b, d}]
Out[1445]= {}, {}
In[1449]:= Prog2[If[x == 70, y_ /; IntegerQ[y], y_ /; ListQ[y]], z_] := Module[{}, {x, {z}}]
```

```

In[1450]:= x = 70; Prog2[x, 70, 65, 45, 23, 16]
Out[1450]= {70, {70, 65, 45, 23, 16}}
In[1454]:= x = 75; Prog2[{70, 65, 45, 23, 16}, "rans", "ian", 2012]
Out[1454]= {{70, 65, 45, 23, 16}, {"rans", "ian", 2012}}
In[1455]:= DefFunc3[Prog2]
Out[1455]= {"Prog2[y_ /; IntegerQ[y], z_] := Module[{x, {z}}]",
            "Prog2[y_ /; ListQ[y], z_] := Module[{y, {z}}]}"}
In[1466]:= Pr[y_ /; ToExpression["h[x_] := Module[{x^2}"] || ToExpression["h[x_, y_] :=
            Module[{x^3 + y^3}"] || ToExpression["h[x_, y_, z_] := Module[{x + y + z}"] ||
            IntegerQ[y], z_ /; ListQ[z]] := Module[{y, Length[z], h[75], h[75, 450], h[70, 65, 45]}]]
In[1467]:= Pr[70, {42, 47, 67, 23, 16}]
Out[1467]= {70, 5, 5625, 91546875, 180}
In[1468]:= DefFunc3[h]
Out[1468]= {"h[x_] := Module[{x^2}", "h[x_, y_] := Module[{x^3 + y^3}",
            "h[x_, y_, z_] := Module[{x + y + z}]}"}

```

Из представленных примеров довольно легко усматриваются механизмы усложнения функциональной нагрузки формальных аргументов, полезные в программировании. Между тем, наряду с процедурами в *Mathematica* используются весьма подобные им *модульные* объекты, определяемые аналогичным способом, а именно:

**Module[{a, b, c, ...}, <Тело процедуры>]**  
**Module[{a = a1, b = b1, c = c1, ...}, <Тело процедуры>]**

Таким образом, по аналогии с *Maple* подобного рода объекты вполне ассоциируются с непоименованными процедурами, в качестве формальных аргументов для которых выступают *глобальные* переменные текущего сеанса *Mathematica*. В данном контексте модули подобно *непоименованным* процедурам вполне могут выступать в выражении в качестве его подвыражений, как это иллюстрирует следующий простой фрагмент:

```

In[768]:= V := 68; G := 63; S := 43; (Module[{V/2}, Module[{G/3}]/(Module[{S}, 12])
Out[768]= 1
In[769]:= Sin[(14*Module[{V}, V] + 21*Module[{G}, G])/(Module[{S}, S]^2 + 72)] // N
Out[769]= 0.926228
In[770]:= Module[{V}, V*Y^2 + (Log[Module[{G}, G]] // N)*Y + Module[{S}, S]^2 + 72
Out[770]= 1921 + 4.14313 Y + 68 Y^2

```

В контексте вышесказанного в дальнейшем мы будем ассоциировать понятие *модуля* с понятием «*процедуры*», как однотипных объектов с соответствующими различиями, определяемыми средой, в которой они определяются и функционируют. Процедура либо блок внутри себя могут содержать другие процедуры, называемые *вложенными*, уровень вложенности ограничивается только размером рабочей области пакета. Так, следующий фрагмент представляет простой пример процедуры *A* с подпроцедурами *двух* уровней вложенности, а именно *C* – *первого* уровня (*содержится в A*) и *D* – *второго* уровня (*содержится в C*), тогда как блок используется как начальное значение для *B*.

```
In[971]:= A[x_, y_] := Module[{B = Block[{a = 75}, x + y], C, h},
      C[z_] := Module[{b = 450, D, h}, D[h_] := Module[[], h^2]; D[z]]; h = 2012; C[B]]
In[972]:= Mapp[A, {23, 16, 45, 65, 70}, 450]
Out[972]= {223729, 217156, 245025, 265225, 270400}
```

Между тем, организация процедурных конструкций в пакетах *Mathematica* и *Maple* имеет и существенные отличия. Прежде всего, в *Maple* допускаются пересечения как формальных аргументов, так и параметров других групп (*locals*, *globals*, *options* и др.) главной процедуры и ее подпроцедур, тогда как в *Mathematica*, в частности, в группе формальных аргументов подобное пересечение недопустимо, даже если определения *формальных* аргументов и ассоциированы с разными *шаблонами*. В целом, недопустимо пересечение формальных аргументов и локальных переменных. Тогда как *множества локальных* переменных главной и вложенных процедур могут пересекаться; при этом, в тексте одноименные с главной процедурой локальные переменные ее подпроцедур оформляются *красным* цветом. Эта картина имеет место при оформлении *подпроцедур* как модульными, блочными, так и функциональными конструкциями. Эта ситуация обусловлена механизмом *шаблонов*, создавая определенные неудобства. В следующем фрагменте представлены простые примеры, иллюстрирующие вышесказанное.

```
> G := proc(x::anything) local a, b, S; S := proc(x::anything) local a, b; x^2 end proc; x^2 +
      S(x) end proc: G(75); => 11250
In[834]:= G[x_] := Module[{a, b, S}, S[x_] := Module[{a, b}, x^2]; x^2 + S[x]]
      RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule G[x_]:> ...
In[835]:= G[75]
      Pattern::patvar: First element in pattern Pattern[75, _] is not a valid pattern name. >>
Out[835]= 5625 + S$2700[75]
In[836]:= V[x_] := Module[{a, b, S}, S[x_;/ IntegerQ[x]] := Module[{a, b}, x^2]; x^2 + S[x]]
      RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule V[x_]:> ...
In[837]:= V[75]
      Pattern::patvar: First element in pattern Pattern[75, _] is not a valid pattern name. >> ...
Out[837]= 5625 + S$1859[75]
In[838]:= W[x_] := Module[{a, b, S}, S[x_;/ IntegerQ[x]] := Block[{a, b}, x^2]; x^2 + S[x]]
      RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule W[x_]:> ..
In[839]:= W[75]
      Pattern::patvar: First element in pattern Pattern[75, _] is not a valid pattern name. >> ...
Out[839]= 5625 + S$1901[75]
In[840]:= H[x_] := Module[{a, b}, S[x_;/ IntegerQ[x]] := Block[{a, b}, x^2]; x^2 + S[x]]
      RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule H[x_]:> ...
In[841]:= H[75]
      Pattern::patvar: First element in pattern Pattern[75, _] is not a valid pattern name. >> ...
Out[841]= 5625 + S[75]
In[842]:= F[x_] := Module[{a, b}, S[x_;/ IntegerQ[x]] := x^2; x^2 + S[x]]
      RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule F[x_]:> ...
In[843]:= F[75]
      Pattern::patvar: First element in pattern _Pattern[75, _] is not a valid pattern name. >>
```

```

Out[843]= 11250
In[844]:= H[x_] := Module[{a, b}, S[y_ /; IntegerQ[y]] := Module[{x, b}, x^2; x^2 + S[x]]; H[t]
Out[844]= t^2 + S[t]
In[845]:= Z[x_] := Module[{a, b, Y}, Y[y_] := Module[{a, b}, x^2; x^2 + Y[x]]; Z[75]
Out[845]= 11250
In[994]:= A[x_, y_] := Module[{a = 2012}, B[x_ /; PrimeQ[x], y_ /; NumberQ[y]] :=
Module[{b = 450}, x + y + b]; a + B[x, y]]
RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule A[x_, y_] :>
Module[{a=2012}, B[x_ /; PrimeQ[x], y_ /; NumberQ[y]] := Module[{b=450}, x+y+b];
a+B[x,y]]. >>
RuleDelayed::rhs: Pattern y_ appears on the right-hand side of rule A[x_, y_] :>
Module[{a=2012}, B[x_ /; PrimeQ[x], y_ /; NumberQ[y]] := Module[{b=450}, x+y+b];
a+B[x,y]]. >>
In[995]:= A1[x_, y_] := Module[{a = 2012}, ToExpression["B[x_ /; PrimeQ[x],
y_ /; NumberQ[y]] := Module[{b = 450}, x+y+b]"]; a + B[x, y]]
In[996]:= Map[A1, {7, 13, 17, 19, 23}, 450]
Out[996]= {2919, 2925, 2929, 2931, 2935}

```

Между тем, имеется по меньшей мере один нестандартный прием обойти указанное ограничение, проиллюстрируем который на одном простом примере. В предыдущем фрагменте предпоследняя процедура **A[x, y]** содержит в своем теле процедуру **B[x, y]** с теми же формальными аргументами, отличающимися только тем, что тестируются соответствующие им фактические аргументы (*в процедуре A допустимы произвольные фактические аргументы*). При вводе определения процедуры формальные аргументы ее подпроцедуры **B** окрашиваются в красный цвет, в то время как попытка вычислить ее определение завершается аварийно с выводом диагностики по обоим аргументам, как недопустимым (*пересекающимся с формальными аргументами главной процедуры*). С целью устранения данной ситуации вместо самого определения процедуры **B** в тело **A** помещается конструкция формата **ToExpression["B"]**, а именно:

**ToExpression["B[x\_ /; PrimeQ[x], y\_ /; NumberQ[y]] := Module[{b = 450}, x+y+b]"]**

В результате не только *ввод* процедуры **A**, но также ее *вызовы* выполняются корректно. Синтаксис *Math*-языка допускает использование в заголовках процедур и функций *условной* генерации формальных аргументов, как иллюстрируют простые примеры фрагмента. Данная возможность имеет целый ряд весьма полезных приложений при программировании самых различных задач. Отметим, *Maple* такой возможностью не располагает, как иллюстрируют примеры второй части следующего фрагмента.

```

In[1265]:= M[x_ /; StringQ[y]; z_ /; IntegerQ[z]] := Block[{a = 2012}, a + z]
In[1266]:= {M[450], M[70, "RansIan", 450]}
Out[1266]= {2462, M[70, "RansIan", 450]}
In[1267]:= Res = 70; G[x_, If[Res == 75, y_ /; StringQ[y], y_ /; ListQ[y]]] :=
Block[{a = 450}, a + x + Length[y]]

```

```

In[1268]:= DefFunc[G]
Out[1268]= G[x_, y_ /; ListQ[y]] := Block[{a = 450}, a + x + Length[y]]
In[1269]:= G[6, {45, 65, 70, 23, 16, 50}]
Out[1269]= 462
In[1270]:= n = 2; V[Part[{x_, x_ /; StringQ[x], x_ /; IntegerQ[x]}, n]] := x; DefFunc[V]
Out[1270]= V[x_ /; StringQ[x]] := x
In[1271]:= y = "Hs"; W[ToExpression["x_/" <> ToString[Head[y]] <> ToString[Q[x]]]] := x
In[1272]:= DefFunc[W]
Out[1272]= W[x_ /; StringQ[x]] := x
In[1273]:= W["RansIanAvz"]
Out[1273]= "RansIanAvz"
In[1274]:= y = 70; W1[ToExpression["x_/" <> ToString[Head[y]] <> ToString[Q[x]]]] := x
In[1275]:= DefFunc[W1]
Out[1275]= W1[x_ /; IntegerQ[x]] := x
In[1276]:= R = 70; T[x_, If[R == 70, y_, y___]] := {x, y}; DefFunc[T]
Out[1276]= T[x_, y_] := {x, y}
In[1277]:= Clear[T]; R = 75; T[x_, If[R == 70, y_, y___]] := {x, y}; DefFunc[T]
Out[1277]= T[x_, y_] := {x, y}
In[1278]:= t = 75; R[If[t != 75, Sequences[{x_, y_ /; StringQ[y], z_ /; IntegerQ[z]},
                                         x_ /; ListQ[x]]] := 75; DefFunc[R]
Out[1278]= R[x_ /; ListQ[x]] := 75
In[1279]:= Clear[R]; t = 70; R[If[t != 75, Sequences[{x_, y_ /; StringQ[y], z_ /; IntegerQ[z]},
                                         x_ /; ListQ[x]]] := 75; DefFunc[R]
Out[1279]= R[x_, y_ /; StringQ[y], z_ /; IntegerQ[z]] := 75
> m := 70: P := proc(x::integer, `if` (m = 70, y::list, y::string)) [x, y] end proc:
Error, `( unexpected
> m := 70: P := proc(x::integer, if m = 70 then y::list else y::string end if) [x, y] end proc:
Error, reserved word `if` unexpected
> m := 70: P := proc(x::integer, y::`if` (m = 70, list, string)) [x, y] end proc:
> P(2012, [70, 65, 45, 23, 16]);
Error, invalid input: P expects its 2nd argument, y, to be of type `if` (m = 70, list, string),
but received [70, 65, 45, 23, 16]

```

*Math*-язык допускает весьма сложные алгоритмы условной генерации заголовков.

В отличие от *Maple*, в качестве формальных аргументов процедур, функций и блоков могут выступать и *Math*-предложения, содержащие выражения, разделенные «;», как иллюстрирует следующий достаточно простой фрагмент, содержащий набор весьма прозрачных демонстрационных примеров, а именно:

```

In[1420]:= Pr[y = 75; z = Pr; x_ /; SymbolQ[x]] := Module[{}, ToExpression[x <> "[" <>
                                                         ToString[Pr] <> "]" ]
In[1421]:= Pr["ProcQ"]
Out[1421]= True
In[1429]:= Pr1[h = 75; y[] := Module[{a = 6}, a + h^2]; t_ /; EvenQ[t], x_ /; IntegerQ[x]] :=

```

```

Module[{}, x + y[] + t]
In[1430]:= Pr1[6, 2012]
Out[1430]= 7649
In[1449]:= Pr2[y = 75; t_ /; EvenQ[t], h_ /; OddQ[h], x_ /; IntegerQ[x]] :=
Module[{}, x + h + t]
In[1450]:= Pr2[6, 9, 2012]
Out[1450]= 2027
In[1474]:= Pr3[x_ /; If[x == 75, True, Print[x]]] := Module[{}, x^2]
In[1475]:= Map[Pr3, {75, 450}]
450
Out[1475]= {5625, Pr3[450]}
In[1483]:= Pr4[x_ /; If[x == 75, True, Abort[]]] := Module[{}, x^2]
In[1488]:= Pr4[75]
Out[1488]= 5625
In[1489]:= Pr4[2012]
Out[1489]= $Aborted
In[1544]:= Pr5[t_ /; EvenQ[t], x[p_] := Module[{a = 6}, a + p], y_ /; OddQ[y],
h[x_] := Module[{}, {x}]] :=
Module[{}, t + y + x[450] + Length[h[42, 47, 67, 23, 16]]]
In[1545]:= Pr5[6, Null, 2011, Null]
Out[1545]= 2478
In[1553]:= Pr6[x[a_ /; ListQ[a]] := Module[{b = 75}, Length[a] + b];
y[c_ /; IntegerQ[c]] := Block[{a = 450}, a + c + 2012];
z[d_] := d^2] := Module[{}, x[{42, 47, 67}] + y[6] + z[450]]
In[1554]:= Pr6[Null, Null, Null]
Out[1554]= 205046
> Pr := proc(h = 75; y := proc() local a; a := 6; a + h^2 end proc; t::even, x::integer)
x + y() + t end proc:
Error, use `:=` instead of `=` to specify default values for parameters

```

Представленный выше механизм определения формальных аргументов процедур и функций оказывается весьма полезным в процедурном программировании, позволяя в ряде случаев программировать, например, весьма компактные *процедурные* объекты.

## 6.1. Средства тестирования процедурных объектов в программной среде пакета Mathematica

В настоящем разделе рассматриваются нестандартные средства как для тестирования процедурных и функциональных объектов, так и манипулирования с их *заголовками* и *определениями*. Отметим, что *Mathematica* – достаточно закрытый пакет в отличие, в частности, от его основного конкурента – пакета *Maple*, в котором вполне допустима перлюстрация исходных кодов его программных средств, находящихся как в главной,

так и вспомогательных библиотеках. Тогда как *Mathematica* подобными средствами не располагает. В этой связи предлагаемые ниже программные средства относятся к пользовательским процедурам/функциям, загруженным в текущий сеанс из пакета (*m-файл*) или документа (*nb-файл*) и активизированным в нем.

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 с примерами ее использования.

```
In[999]:= DefFunc[x_;/ SymbolQ[x] || StringQ[x]] := Module[{a = ToString[WhatObj[x]],
                                                             b = ToString[x] <> ""},
  Quiet[ToExpression[Quiet[StringReplace[ToString1[Definition[x]], {a -> "", b -> ""}]]]];
  Definition[x]]

In[1000]:= DefFunc[ProcQ]
Out[1000]= ProcQ[x_] := Module[{a, b, c, d, h}, a := ToString[Definition[x]];
  b := StringTake[a, {1, First[First[StringPosition[a, " := Block[" - 1]]]};
  c := StringTake[b, {1, First[First[StringPosition[b, "[" - 1]]]};
  d = Quiet[If[ToExpression[c] === x, True, False]];
  b := StringTake[a, {1, First[First[StringPosition[a, " := Module[" - 1]]]};
  c := StringTake[b, {1, First[First[StringPosition[b, "[" - 1]]]};
  h = Quiet[If[ToExpression[c] === x, True, False]]; d || h]

In[1006]:= Definition[ListListQ]
Out[1006]= 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[1007]:= DefFunc[ListListQ]
Out[1007]= 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 как процедур, так и функций.

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  $x, y, \dots$  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 **DynamicModule** непосредственно не дает возможности получать его определение ни по стандартной функции **Definition**, ни по нашей процедуре **DefFunc**, представленной выше. Как это иллюстрирует следующий фрагмент, который содержит и исходные коды процедур **DefFunc1[x]** и **DefFunc2[x]**, которые возвращают компактное определение объекта  $x$ , заданного в строчном формате, реализуя для иллюстрации различные алгоритмы.

```
In[643]:= DefFunc1[x_ /; StringQ[x] && HowAct[x]] := Module[{a = "$ArtKr$.avz", k = 1,
c, d = ToString[x] <> "", g = $Packages},
  Save[a, x]; c = StringTrim[Read[a, String]];
  If[c == StringTrim[EndOfFile] || StringFreeQ[c, "["], Close[a];
  DeleteFile[a]; Return[Defer[DefFunc1[x]]], Null];
  While[Quiet[ToExpression[c]] == $Failed,
  c = c <> StringTrim[Read[a, String]]]; Close[a]; DeleteFile[a];
  For[k, k <= Length[g], k++, c = StringReplace[c, g[[k]] <> d -> ""]]; c]

In[644]:= Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 75, b = 420}, a + x + y]
In[645]:= Dm[42, 47]
Out[645]= 164
In[646]:= Definition[Dm]
Out[646]= Dm[x_, y_ /; PrimeQ[y]] := a$$ + x + y
In[647]:= DefFunc["Dm"]
Out[647]= Dm[x_, y_ /; PrimeQ[y]] := a$$ + x + y
In[648]:= DefFunc1["Dm"]
Out[648]= "Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 75, b = 420}, a + x + y]"
In[744]:= DefFunc2[x_ /; StringQ[x]] := Module[{a, b = $Packages, d = ToString[x] <> "",
k = 1}, If[! SymbolQ[x], Return[Defer[DefFunc2[x]]], a = ToString[Definition[x]];
  For[k, k <= Length[b], k++, a = StringReplace[a, b[[k]] <> d -> ""]]; a];
In[745]:= DefFunc2["Dm"]
Out[745]= "Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[{a = 75, b = 420}, a + x + y]"
In[746]:= Definition["2012"]
Definition::notfound: Symbol 2012 not found. >>
Out[746]=
In[747]:= {DefFunc["2012"], DefFunc1["2012"], DefFunc2["2012"]}
Definition::notfound: Symbol 2012 not found. >>
Out[747]= { , DefFunc1["2012"], DefFunc2["2012"]}
In[791]:= ModuleQ[x_ /; StringQ[x], y_ /; ! HowAct[y]] := Module[{a = x <> "[" <>
  StringTake[ToString[Args[x]], {2, -2}] <> "]" := Module[{",
```

```
b = ToString[DefFunc2[x]], If[! StringFreeQ[b, a], y = "Module"; True,
If[! StringFreeQ[b, StringReplace[a, "Module" -> "DynamicModule"]],
y = "DynamicModule"; True, False]]]
```

```
In[792]:= {ModuleQ["Dm"], ModuleQ["ProcQ"], ModuleQ["ModuleQ"], ModuleQ["75"]}
Out[792]= {True, True, True, False}
In[793]:= {ModuleQ["Dm", t], ModuleQ["ProcQ", g], ModuleQ["2012", w]}
Out[793]= {True, True, False}
In[794]:= {t, g, w}
Out[794]= {"DynamicModule", "Module", w}
```

В случае некорректного аргумента  $x$  вызовы процедур завершаются *невывисленными*, что обеспечивает более простую обработку особых ситуаций, возникающих в точках вызова данных процедур. Завершает данный фрагмент довольно полезная процедура **ModuleQ[x,y]**, базирующаяся на процедуре **DefFunc2** и возвращающая значение *True*, если объект  $x$ , заданный в строчном формате, является модулем, и *False* в противном случае. При этом, через второй аргумент  $y$ , возвращается тип модуля, если основным возвратом является *True*, в противном случае  $y$  возвращается *неопределенным*. Данный фрагмент представляет исходный код процедуры и примеры ее применения. На наш взгляд представленные фрагментом средства довольно полезны в программировании различного типа задач, прежде всего, задач системного характера.

В связи с представленными процедурами **DefFunc**, **DefFunc1**, **DefFunc2** следует иметь в виду следующее немаловажное обстоятельство, а именно. В отличие от большинства систем программирования, в которых очередное *переопределение* процедуры заменяет предшествующее без возможности доступа к нему, т.е. в текущем сеансе история всех определений процедуры не сохраняется, в программной среде пакета *Mathematica* в этом отношении ситуация совершенно иная. Следующий фрагмент весьма наглядно представляет возможность перлюстрации *всех* определений одноименной процедуры.

```
In[1044]:= 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}, x];
P[x_, y_] := Module[{x, y}, x + y]
In[1045]:= Definition[P]
Out[1045]= P[y_]; PrimeQ[y]] := Module[{a = "agn"}, y]
P[x_]; StringQ[x]] := Module[{x}, x]
P[x_]; ListQ[x]] := Module[{x}, 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**. Следующий фрагмент представляет исходный код процедуры **DefFunc3[x]** наряду с наиболее типичными примерами ее применения.

```
In[1103]:= DefFunc3[x_ /; ProcQ[x] || ! DefOpt[ToString[x]] == Null] := Module[{b = {},
    k = 1, a = StringSplit[StrStr[StrStr[InputForm[DefFunc[x]]], "\n\n"]],
    {a[[1]] = StringTake[a[[1]], {2, -1}], a[[-1]] = StringTake[a[[-1]], {1, -2}];
    For[k, k <= Length[a], k++, b = Append[b, SubsDel[a[[k]], "" <>
        ToString[x] <> "", {"[", ",", " ", -1}]]];
    b = Select[Mapp[StringReplace, b, "Global" -> ""], ! SuffPref[#, "Attributes[" 1] &];
    If[b == {"Null"}, {}, 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** пакета.

В качестве достаточно полезного средства при работе с процедурами, функциями и блоками в определенной мере отвечает процедура, чей вызов **HeadPF[F]** возвращает *заголовок* в строчном формате активизированного в текущем сеансе объекта с именем *F* указанного типа. При этом, для системного объекта возвращается его имя *F*, тогда как во всех остальных случаях вызов возвращается *невычисленным*. Ниже представлен фрагмент с исходным кодом процедуры наряду с примерами ее применения.

```
In[900]:= HeadPF[F_ /; SymbolQ[F]] := Module[{a = ToString[InputForm[DefFunc[F]]],
    b, c = " := "}, If[StringFreeQ[a, c], Return[HoldForm[F]],
    b = Flatten[StringPosition[a, c]]; b = StringTake[a, {1, b[[1]] - 1}];
    SubsDel[If[! StringFreeQ[b, ToString[F] <> "["], b, HoldForm[F]],
    "" <> ToString[F] <> "", {"[", ",", " ", -1}]]

In[901]:= G[x_, y_] := x*Sin[y] + y*Cos[x]; s[] := 75*x; g := 47
In[902]:= Map[HeadPF, {G, s, Sin, 2012, g}]
Out[902]= {"G[x_, y_]", "s[]", Sin, HeadPF[2012], HeadPF[47]}
In[903]:= Map[HeadPF, {If, Tan, Log, True, "Infinity", While, Do, InputForm}]
Out[903]= {If, Tan, Log, True, "Infinity", While, Do, InputForm}
In[904]:= Map[HeadPF, {G, s, g, 2012}]
Out[904]= {"G[x_, y_]", "s[]", HeadPF[47], HeadPF[2012]}
```

Непосредственно на основе нашей процедуры **HeadPF**, рассмотренной выше, можно определить и булеву функцию, вызов которой **SystemQ[W]** возвращает **True**, если *W*-объект является *системным*, т.е. определен пакетом *Mathematica*, и **False** в противном

случае. Фрагмент представляет исходный код функции с примерами ее применения. В ряде приложений данная функция представляет вполне определенный интерес и, прежде всего, предоставляя возможность эффективно дифференцировать средства.

```
In[975]:= SystemQ[S_] := If[Off[Definition::ssle]; ! ToString[Definition[S]] === Null &&
      SysFuncQ1[S], On[Definition::ssle]; True, On[Definition::ssle]; False]
In[976]:= Map[SystemQ, {75, G, Sin, Do, While, False, ProcQ, "Infinity"}]
Out[976]= {False, False, True, True, True, True, False, True}
In[1081]:= Head1[x_] := Block[{a}, Off[Definition::ssle]; a = If[StringQ[x], String,
      If[Quiet[QFunction[ToString[x]]] === True, Function, If[ProcQ[x], Procedure,
      If[SystemQ[x], System, If[HeadPF[x] === x, Symbol, Head[x]]]]]; On[Definition::ssle]; a]
In[1082]:= Map[Head, {ProcQ, Sin, 75, a + b, Function[{x, y}, x + y], G[x], S[6], Head1}]
Out[1082]= {Symbol, Symbol, Integer, Plus, Function, G, S, Symbol}
In[1083]:= Map[Head1, {ProcQ, Sin, 75, a + b, Function[{x, y}, x + y], G[x], S[6], Head1}]
Out[1083]= {Procedure, System, Integer, Plus, Function, G, S, Procedure}
```

Этим же целям служит и функция **Head1[x]**, приведенная во второй части фрагмента исходным кодом наряду с примерами применения, обобщая стандартную функцию **Head[x]** и возвращая тип выражения  $x$  в разрезах *{Procedure, Function, Symbol, System, Head[x]}*. В качестве сравнения приводятся примеры использования обоих функций на одном и том же списке тестируемых объектов, подтверждающие предпочтение в целом ряде случаев именно функции **Head1**. Функция существенно использует наши средства **SystemQ**, **ProcQ**, **HeadPF**, **ToString1** и **HeadingQ**, из которых первая функция рассмотрена выше, тогда как **ProcQ**, **HeadPF** и **HeadingQ** непосредственно относятся к задаче тестирования как процедурных, так и функциональных объектов. Довольно простая процедура **Tostring1** используется в наших средствах достаточно широко, но отнесена к средствам манипулирования строчными конструкциями. Рассмотрим две упомянутые процедуры **ProcQ** и **HeadingQ**, процедура **HeadPF** рассмотрена выше.

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**.

```
In[784]:= HeadingQ[x_ /; StringQ[x]] := Module[{a, b, c, k = 1, m = True, n = True},
      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]]; a = Map[ToString, ToExpression[a]];
```

```

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[785]:= {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[785]= {True, True, False}
In[786]:= {HeadingQ["D[x_, y_]; ListQ[y], z_:75, h]"], HeadingQ["[x_, y_]; ListQ[y], z:75]"]}
Out[786]= {False, False}
In[787]:= {HeadingQ["D[]"], HeadingQ["Z[x_]"], HeadingQ["Q[x_]"]}
Out[787]= {True, True, True}
In[788]:= {HeadingQ["D[_ , x_]"], HeadingQ["Z[x_]"], HeadingQ["Q[x_]"]}
Out[788]= {False, True, True}
In[789]:= {HeadingQ["D[x_, y_]; ListQ[y], z_:75, h]"], HeadingQ["V[x_, y_];ListQ[y], z_.]"]}
Out[879]= {False, True}

```

Данное средство достаточно полезно, в частности, при тестировании типов объектов, подобно использованию в определении весьма полезной функции **Head1**.

Отметим, в общем смысле под *процедурными* объектами в пакете *Mathematica* можно полагать созданные как на основе *модульной*, так и *блочной* организаций. Процедура *блочной* организации в *Mathematica* определяется конструкциями общего вида:

$$V[x_, y_, z_ \dots] := \text{Block}[\{a, b, c, \dots\}, \text{<Тело процедуры>}]$$

$$V[x_, y_, z_ \dots] := \text{Block}[\{a=a1, b=b1, c=c1, \dots\}, \text{<Тело процедуры>}]$$

где *V* – имя процедуры, *x\_, y\_, z\_ ...* – ее формальные аргументы любого допустимого пакетом типа, и *a, b, c, ...* – локальные переменные процедуры. Предложения пакета, составляющие *тело процедуры*, описывают алгоритм задачи, решаемой процедурой. В частности, второй формат определения процедуры обеспечивает возможность для *локальных* переменных присвоить начальные значения *a=a1, b=b1, c=c1, ...*. При этом, в случае отсутствия для блока локальных переменных такой факт кодируется пустым списком, т.е. {}, т.к. по определению *блок* определяется двумя параметрами; иначе при обращении к такому блоку (*процедуре*) будет инициирована ошибочная ситуация как наглядно иллюстрирует следующий весьма простой фрагмент, а именно:

```

In[1409]:= Art[x_, y_] := Block[{}]
In[1410]:= Art[70, 42]
Block::argr: Block called with 1 argument; 2 arguments are expected. >>
Out[1410]= Block[{}]

```

Ниже представлены примеры двух типов синтаксических ошибок при определениях процедур типов {**Module**, **Block**}, которые не распознаются пакетом 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 применения и представляет нижеследующий фрагмент.

```
In[1729]:= Art[x_, y_] := Module[{a, b}, ]; Art1[x_, y_] := Module[{a, b}]
In[1730]:= Kr[x_, y_] := Block[{a, b}, ]; Kr1[x_, y_] := Block[{a, b}]
In[1731]:= {Art[70, 450], Art1[70, 450]}
Module::argr: Module called with 1 argument; 2 arguments are expected. >>
Out[1731]= {Null, Module[{a, b}]}
In[1432]:= {Art[70, 450], Art1[70, 450]}
Out[1432]= {Null, Module[{a, b}]}
In[1733]:= {Kr[70, 450], Kr1[70, 450]}
Block::argr: Block called with 1 argument; 2 arguments are expected. >>
Out[1733]= {Null, Block[{a, b}]}
In[1734]:= SyntCorProcQ[x_;/; ProcQ[x]] := Module[{b = ToString1[DefOpt[ToString[x]]],
a = HeadPF[x], c = $ArtKr$, d, h},
Clear[$ArtKr$]; ProcQ1[ToString[x], $ArtKr$];
h = Quiet[Check[Locals2[x][[2]], Locals1[x][[2]]];
h = If[Quiet[h] == {}[[2]], "{}", h];
d = a <> " := " <> $ArtKr$ <> "[" <> h;
d = StringReplace[b, d -> "", 1]; $ArtKr$ = c;
!MemberQ[{"", "", Null}], d]]
In[1735]:= Map[SyntCorProcQ, {ProcQ, Kr, Kr1, Art, Art1}]
Out[1735]= {True, False, False, False, False}
In[1736]:= KrArt[x_, y_, z_] := Module[{}, 70]
In[1737]:= Map[SyntCorProcQ, {Locals, Mapp, BlockToMod, KrArt}]
Out[1737]= {True, True, True, True}
In[1438]:= Map[SyntCorProcQ, {Art, Do, If, 6}]
Out[1438]= {SyntCorProcQ[Art], SyntCorProcQ[Do], SyntCorProcQ[If], SyntCorProcQ[6]}
```

Вызов процедуры **SyntCorProcQ[x]** возвращает **True**, если определение процедуры *x*, активированное в текущем сеансе, синтаксически корректно в указанном контексте, в противном случае возвращается **False**. Если *x* – не процедура, то вызов возвращается невычисленным. Определение **SyntCorProcQ** использует и наши процедуры **DefOpt**, **HeadPF**, **Locals1**, **Locals2**, **ProcQ1** и **ToString1**, рассматриваемые в настоящей книге. В процедурном программировании эта процедура оказывается достаточно полезной.

Между тем, в общем случае между процедурами типов **Module** и **Block** существуют и принципиальные отличия, базирующиеся, в частности, на различных используемых механизмах *локальных* переменных, как иллюстрирует следующий довольно простой фрагмент, представляющий результаты вызовов процедур разных типов, а именно:

```
In[1427]:= Kr[x_] := Block[{a = x}, Head[a]]; Art[x_] := Module[{a = x}, Head[a]]
In[1428]:= Kr[a + b]
$RecursionLimit::reclim: Recursion depth of 256 exceeded. >>
Out[1428]= Plus
In[1429]:= Art[a + b]
Out[1429]= Plus
In[1430]:= Kr1[x_] := Block[{c = x}, Head[c]]; Kr1[a + b]
Out[1430]= Plus
```

Ниже приведены и более сложные примеры таких различий. Поэтому тип *процедуры* следует выбирать довольно осмотрительно, отдавая предпочтение процедурам типа **Module**. Вызов процедуры **BlockToMod[x]** возвращает *Null*, обеспечивая конвертацию процедуры **Block**-типа в процедуру **Module**-типа. Фрагмент содержит код и пример.

```
In[1450]:= Kr[x_] := Block[{a = x}, Head[a]]; Definition[Kr]
Out[1450]= Kr[x_] := Block[{a = x}, Head[a]]

In[1451]:= BlockToMod[x_/; ProcQ[x]] := Module[{a = $ArtKr$, b = ToString[x]},
    If[Clear[$ArtKr$]; ProcQ1[b, $ArtKr$]; $ArtKr$ == "Block", $ArtKr$ = a;
    ToExpression[StringReplace[ToString1[DefFunc[b]], "Block[{" ->
    "Module[{" , 1]], $ArtKr$ = a]]
Out[1451]= BlockToMod[x_/; ProcQ[x]] := Module[{a = $ArtKr$, b = ToString[x]},
    If[Clear[$ArtKr$]; ProcQ1[b, $ArtKr$]; $ArtKr$ == "Block", $ArtKr$ = a;
    ToExpression[StringReplace[ToString1[DefFunc[b]], "Block[{" ->
    "Module[{" , 1]], $ArtKr$ = a]]

In[1452]:= $ArtKr$ = 70; BlockToMod[Kr]; Definition[Kr]
Out[1452]= Kr[x_] := Module[{a = x}, Head[a]]
```

Следующий фрагмент иллюстрирует простые примеры процедур на основе обоих организаций и процедуру **ProcQ[x]**, которая обеспечивает тестирование  $x$ -объектов на предмет быть объектами процедурного типа; полезны также **ProcQ1** и **ProcQ2**.

```
In[344]:= Kr[x_] := Block[{y = a, h = b}, (y^2 + h^2)*x]; Art[x_] := Module[{a}, a = 68; x + a]
In[345]:= {Kr[(c + d)], Art[42]}
Out[345]= {(a^2 + b^2) (c + d), 110}

In[346]:= ProcQ[x_] := Module[{a = SubsDel[ToString[InputForm[Definition[x]]], "`" <>
ToString[x] <> "`", {"[" , " ", " " ], -1], b, c, d, h},
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]], True,
If[c === ToString[HeadPF[x]], True, If[d === ToString[HeadPF[x]], True, False]]]]
In[347]:= {ProcQ[Kr], ProcQ[Art], ProcQ[D], ProcQ[Sin], ProcQ[Block], ProcQ[Module]}
Out[347]= {True, True, False, False, False, False}

In[418]:= ProcQ1[x_ /; StringQ[x], t_ /; Definition1[t] == "Null"]:= Module[{b = ":=", p, x1 =
"Block[{", d, c = DefOpt[x], a = Quiet[ToString[Args[x]]], x2 = "Module[{",
x3 = "DynamicModule[{", If[! HowAct[ToExpression[x]] ||
NumberQ[ToExpression[x]] || SuffPref[a, "If[, 1] || a == "$Failed", t = "Others";
Return[False], d = x <> "[" <> StringTake[a, {2, -2}] <> "]" := ";"}]
```

```

If[d == StringTake[c, {1, StringLength[d]}], p = True, Return[False]];
If[! StringFreeQ[c, x1] && d <> x1 == StringTake[c, {1, StringLength[d <> x1]}], t = "Block",
If[! StringFreeQ[c, x2] && d <> x2 == StringTake[c, {1, StringLength[d <> x2]}], t =
"Module", If[! StringFreeQ[c, x3] && d <> x3 == StringTake[c, {1, StringLength[d <> x3]}],
t = "DynamicModule", t = "Others"]]; p]
In[419]:= {ProcQ1["G", y1], ProcQ1["ProcQ", y2], ProcQ1["Dm", y3], ProcQ1["BB", y4]}
Out[419]= {True, True, True, True}
In[420]:= {y1, y2, y3, y4}
Out[420]= {"Others", "Module", "DynamicModule", "Block"}
In[421]:= H = 2012; V := 69; {ProcQ1["75", x1], ProcQ1["H", x2], ProcQ1["V", x3]}
Out[421]= {False, False, False}
In[422]:= {x1, x2, x3}
Out[422]= {"Others", "Others", "Others"}

```

Процедура **ProcQ1** является достаточно полезной модификацией процедуры **ProcQ**, ее вызов **ProcQ1[w, t]** возвращает **True**, если *w* является объектом типа **Block**, **Module** или **DynamicModule**, и **"Others"** или **False** в противном случае; при этом, тип объекта *w* возвращается через фактический *t*-аргумент. Предыдущий фрагмент представляет исходный код процедуры **ProcQ1** и типичные примеры ее использования.

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** и примеры ее 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 = DefFunc3[x]; c = Length[b]];
Do[d = Append[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]]]]}]]]]]; d]
In[2221]:= Sv[x_, y_] := x + y; G[x_] := Block[{}, 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, 75, 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_ /; ProcQ[x] || Length[Args[x]] > 0]", Module}}
In[2225]:= F := Function[{x, y}, x + y]; {F[75, 450], ObjType[F]}
Out[2225]= {525, Function}
In[2226]:= F1 := #1 + #2 &; {F1[75, 450], ObjType[F1]}
Out[2226]= {525, Function}
In[2227]:= Map[ObjType, {HeadPF, StrStr}]
Out[2227]= {{{"HeadPF[F_ /; SymbolQ[F]]", Module}}, {"StrStr[x_]", {Function, String}}}}
In[2228]:= Agn := "4247679886"; Avz = 2012; 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]}}. В связи с изложенным процедура **ObjType** представляется достаточно полезной при тестировании *Math*-объектов. В следующем разделе рассматривается весьма важная компонента процедур и блоков – *локальные переменные*, которые играют достаточно существенную роль в процедурном программировании в целом.

## 6.2. Локальные переменные процедур в пакете Mathematica

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[572]:= G[x_, y_, z_] := Module[{a, b, c}, h := a*x + b*y + c*z; Return[{h, a, b,
Symbol[ToString[a] <> "$" <> ToString[$ModuleNumber - 1]]}]];
In[573]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[573]= {9088, {68 a$9088 + 63 b$9088 + 43 c$9088, a$9088, b$9088, a$9088$9088}, 9089}
In[574]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[574]= {9090, {68 a$9090 + 63 b$9090 + 43 c$9090, a$9090, b$9090, a$9090$9090}, 9091}
In[575]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[575]= {9092, {68 a$9092 + 63 b$9092 + 43 c$9092, a$9092, b$9092, a$9092$9092}, 9093}
In[576]:= n = 1; While[n < 5, Print[$ModuleNumber]; n++]
9277
9277
9277
9277

```

```
In[577]:= {$ModuleNumber, $ModuleNumber}
Out[577]= {9278, 9279}
```

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[630]:= S[x_, y_] := Module[{a = $ModuleNumber - 1, b = $ModuleNumber - 1,
                                c = $ModuleNumber - 1}, h := a*x + b*y + c;
                                Return[{h, Symbol["a$" <> ToString[$ModuleNumber - 1]],
                                Symbol["b$" <> ToString[$ModuleNumber - 1]],
                                Symbol["c$" <> ToString[$ModuleNumber - 1]], a b, c}]]

In[631]:= S[68, 63]
Out[631]= {126588, 959, 959, 959, 959, 959, 959}
In[632]:= g := {a$959, b$959, c$959}
In[633]:= S[68, 63]
Out[633]= {126984, 962, 962, 962, 962, 962, 962}
In[634]:= d := {g, {a$962, b$962, c$962}}
In[635]:= S[68, 63]
Out[635]= {127380, 965, 965, 965, 965, 965, 965}
In[636]:= {d, {a$965, b$965, c$965}}
Out[636]= {{{959, 959, 959}, {962, 962, 962}}, {965, 965, 965}}
```

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 различных задач, прежде всего, *системного* характера. Следующий простой фрагмент достаточно наглядно иллюстрирует сказанное, а именно:

```
In[644]:= Kr[x_, y_] := Module[{a, b}, h := a*x + b*y; {{a, b, h}, h}]
In[645]:= Kr[14, 21]
Out[645]= {{a$846, b$846, 14 a$846 + 21 b$846}, 14 a$846 + 21 b$846}
In[646]:= First[First[%]]^2 + Take[First[%], {2, 2}]^2
Out[646]= {a$846^2 + b$846^2}
In[647]:= Kr[x_, y_] := Module[{a, b}, a = 96; b = 89; h := a*x + b*y;
                                Print[{"a$" <> ToString[$ModuleNumber - 1],
                                "b$" <> ToString[$ModuleNumber - 1]};
                                {Symbol["a$" <> ToString[$ModuleNumber - 1]],
                                Symbol["b$" <> ToString[$ModuleNumber - 1]]}]
```

```
In[648]:= Kr[14, 21]
           {a$836, b$836}
Out[648]= {96, 89}
In[649]:= First[%]^2 + Last[%]^2
Out[649]= 17137
```

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. Однако, именно такой подход делает переменные процедур действительно *локальными* с областью действия, ограниченной самой процедурой. В таком случае локальные переменные недоступны извне процедуры, что в некотором отношении можно рассматривать как определенную предпосылку для определения «*черного ящика*» и естественного перехода к парадигме модульной организации.

В ряде случаев возникает необходимость генерации уникальных для текущего сеанса имен объектов. Для этой цели предназначена функция **Unique**, имеющая 4 формата:

**Unique[]** – генерация нового уникального символа с именем формата *\$nnnn*;

**Unique[G]** – генерация нового уникального символа с именем формата *G\$nnnn*;

**Unique[{x,y,z, ...}]** – генерация списка новых уникальных символов с именами формы *{x\$nnnn, y\$nnnn, z\$nnnn, ...}*;

**Unique["S"]** – генерация нового уникального символа с именем формата *Snnnn*.

Следующий весьма простой пример хорошо иллюстрирует сказанное, а именно:

```
In[500]:= G:= 63; {Unique[], Unique[S], Unique[{x, y, z}], Unique["G"]}
Out[500]= {$18, S$1422, {x$1423, y$1423, z$1423}, G19}
In[501]:= n = 1; S = {}; While[n < 6, S = Append[S, Unique[G]]; n++]; S
Out[501]= {G$1503, G$1504, G$1505, G$1506, G$1507}
```

При этом, для обеспечения символам уникальности каждый вызов функции **Unique** обеспечивает приращение для значения переменной *\$ModuleNumber*, как это хорошо иллюстрирует последний пример предыдущего фрагмента, тогда как в случае вызова функции **Unique** в списочной структуре для значения переменной *\$ModuleNumber* не производится приращения. Механизм функционирования функции **Unique** подобен механизму генерации имен для локальных переменных модуля (*процедуры*). Простой пример иллюстрирует один из вариантов программной реализации функции **Unique** посредством процедуры **Un**, чей исходный код с примерами приводятся ниже:

```
In[1055]:= Un[x__] := Module[{a}, If[{x} == {}, Return[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[1056]:= {Un[], Un[S], Un["G"], Un[{x, y, z}], Un[V]}
Out[1056]= {$3390, S$3391, G3392, {x$3393, y$3393, z$3393}, V$3394}
```

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[714]:= n = 1; While[n < 5, Unique[{x, y, z}]; n++]
In[715]:= ?x*
```

▼Global`

```
x      x$1754      x$1755      x$1756      x$1757
```

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, как показывает пример:

```
In[716]:= Clear[a]; M[x_] := Module[{a = 14}, h = x + a^2]
In[717]:= {M[21], a, h}
Out[717]= {217, a, 217}
```

Между тем, следует иметь ввиду, что использование имен формы *name\$nnnn* является соглашением *Mathematica* для генерируемых модулями (*процедурами*) их локальных переменных. Посему во избежание *конфликтных* ситуаций с генерируемыми пакетом именами указанной формы пользователю не рекомендуется использовать подобной формы имена в своих программах. При этом, необходимо помнить, что переменные, сгенерированные процедурами (*модулями*), являются уникальными только в течение текущего сеанса и определяющая их *нумерацию* переменная *\$ModuleNumber* в начале каждого сеанса с пакетом устанавливается в свое начальное значение.

В частности, при сохранении пользовательского выражения, именнованного формой *name\$nnnn* в файле с последующим его чтением, даже в другом сеансе с пакетом могут вполне возникать конфликтные ситуации упомянутого типа. Естественно, из данной ситуации при крайней необходимости имеется выход посредством переопределения пакетной переменной *\$ModuleNumber* в текущем сеансе, в частности, приведенным в нижеследующем фрагменте способом, а именно:

```
In[720]:= SerNum := ToExpression[ToString[$SessionID] <> ToString[$ModuleNumber]]
In[721]:= $ModuleNumber = SerNum
Out[721]= 231802037889173586783411
In[722]:= $ModuleNumber
Out[722]= 231802037889173586783412
```

Действительно, так как переменная `$SessionID` пакета определяет уникальное число, характеризующее конкретный сеанс с пакетом на конкретном компьютере, тогда как переменная `$ModuleNumber` пакета получает приращение при каждом новом вызове функции **Unique** или обращении к процедуре (*модулю*), то определенное имя формы `name$nnn` будет уникальным в каждой текущей сессии пакета. Между тем, обычному пользователю пакета описанные возможности ни к чему и ему, во избежание разного рода усложнений, рекомендуется избегать имен формы `name$nnn`, исключая случаи, когда имеется настоятельная необходимость в генерации новых имен объектов. Тем более крайне осторожно следует использовать переопределения переменной пакета `$ModuleNumber` во избежание возможных конфликтных ситуаций в текущем сеансе, о чем и предупреждает пакет при такой попытке, как иллюстрирует пример:

```
In[750]:= $ModuleNumber := 2012
```

`$ModuleNumber::modnc:` Newly created symbol names may conflict with existing ones.

Механизм использования локальных переменных при вызове модуля в целом состоит в следующем. После генерации символов формы `name$nnn`, представляющих данные локальные переменные, посредством функции **With** в самом теле модуля (*процедуры*) производится замена локальных переменных представляющими их *сгенерированными* символами после чего тело модуля (*процедуры*) вычисляется, как достаточно наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[624]:= HSR[x_, y_] := Module[{a, b}, h := With[{a = ToExpression["a$"] <>
ToString[$ModuleNumber - 1]], b = ToExpression["b$"] <>
ToString[$ModuleNumber - 1]]], a*x + b*y]; z = a*x + b*y; {h, z}]
```

```
In[625]:= HSR[69, 64]
```

```
Out[625]= {69 a$2018 + 64 b$2018, 69 a$2018 + 64 b$2018}
```

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**, как достаточно наглядно иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[327]:= Body := Sin[x] + Cos[x]; Module[{x = Pi/2}, Body]
```

```
Out[327]= Cos[x] + Sin[x]
```

```
In[328]:= Body := Sin[x] + Cos[x]; Module[{x = Pi/2}, Evaluate[Body]]
```

```
Out[328]= 1
```

```
In[329]:= Body := Sin[x] + Cos[x]; Res[] := Module[{x = Pi/2}, Body]; Res[]
```

```
Out[329]= Cos[x] + Sin[x]
```

```
In[330]:= Body := Sin[x] + Cos[x]; Res[] := Module[{x = Pi/2}, Evaluate[Body]]; Res[]
```

```
Out[330]= 1
```

Поскольку функции **Module** имеет атрибут **HoldAll**, то тело модуля (*процедуры*), как правило, сохраняется невычисленным до тех пор, пока модуль не будет выполнен. 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 пакета.

Между тем, следует иметь ввиду, что при создании *вложенных* процедур *Mathematica* имеет и достаточно существенные ограничения, например, вложенная процедура не может использовать *шаблоны* для формальных аргументов, аналогичные содержащей ее процедуры, вызывая ошибочные ситуации уже на стадии компиляции процедур:

```
In[45]:= A[x_, y_] := Module[{a, b, c, B}, B[x_, y_] := Module[{a}, a = x^2 + y^2; Sqrt[a]];
          a = x*y; b = x + y; c = B[a, b]; c]
      RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule A[x_, y_]:>
Module[{a,b,c,B},B[x_,y_]:=Module[{a},a=Plus[<<2>>];Sqrt[a]];a=x y;b=x+y;c=B[a,b];c]. >>
      RuleDelayed::rhs: Pattern y_ appears on the right-hand side of rule A[x_, y_]:>
Module[{a,b,c,B},B[x_,y_]:=Module[{a},a=Plus[<<2>>];Sqrt[a]];a=x y;b=x+y;c=B[a,b];c]. >>
In[46]:= A[63, 68]
      Pattern::patvar: First element in pattern Pattern[63,_] is not a valid pattern name. >>
      Pattern::patvar: First element in pattern Pattern[68,_] is not a valid pattern name. >>
Out[46]= B$955[4284, 131]
In[47]:= A[x_, y_] := Module[{a, b, c, B}, B[z_, t_] := Module[{a}, a = z^2 + t^2; Sqrt[a]];
          a = x*y; b = x + y; c = B[a, b]; c]
In[48]:= A[63, 68]
Out[48]= Sqrt[18369817]
In[49]:= A[x_, y_] := Module[{a, b, c, B, C}, B[h_, p_] := Module[{a}, a = h^2 + p^2; Sqrt[a]];
          C[h_, p_] := Module[{a}, h^p]; a = x*y; b = x + y; c = B[a,b] + C[x,y]; c]
In[50]:= A[14, 22]
Out[50]= 16398978063355821105872896 + 4 Sqrt[6010]
```

Из представленного фрагмента со всей очевидностью следует, что пакет *Mathematica* не допускает использования идентичных шаблонов для формальных аргументов как во *вложенных*, так и в *обрамляющих* их процедурах, т.е. *главная* и *вложенные* процедуры имеют единую область определения шаблонов формальных аргументов, не позволяя им *пересекаться*, что в целом ряде случаев представляется нам достаточно неудобным, требуя для каждой вложенной процедуры. С другой стороны, вложенные процедуры относительно главной могут иметь *идентичные шаблоны* для формальных аргументов, как это иллюстрирует один из примеров фрагмента. Тогда как определения *процедур* в пакете *Maple* допускают использование *идентичных* как *формальных* аргументов, так и *локальных* переменных для главной и вложенных процедур любой вложенности, как это довольно наглядно иллюстрирует один пример [99] фрагмента с *Maple*-аналогом весьма простой *Mathematica*-процедуры *A[x, y]*. Однако, данное ограничение далеко не единственное в организации процедурного программирования в *Mathematica*, но в рамках настоящей книги данный *вопрос* не рассматривается. По этому важному для программиста показателю *Mathematica* достаточно существенно уступает *Maple* [99].

С другой стороны, *локальные* переменные *Math*-процедур допускают присвоение им *начальных* значений в виде произвольных выражений, включая выражения, значения которых могут зависеть от фактических аргументов, получаемых при вызове:

```
In[1066]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], NextPrime[x]]}, a*(x + y)]
In[1067]:= G[13, 75]
Out[1067]= 6952
```

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[1106]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], Return[x]]}, a*(x + y)]
In[1107]:= G[75, 450]
Out[1107]= 39375
In[1108]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], Defer[G[x]]]}, a*(x + y)]
In[1109]:= G[75, 450]
Out[1109]= 525 G[75]
In[1110]:= G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], Abort[]]}, a*(x + y)]
In[1111]:= G[75, 450]
Out[1111]= $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 достаточно существенно разнятся. Нижеприведенный фрагмент достаточно наглядно иллюстрирует вышесказанное, а именно:

```
In[1324]:= a = 1; b = 2; c = 3; {67*Block[{a = 42, b = 47, c = 67}, (a*x + x^b)/c], {a, b, c}}
Out[1324]= {42 x + x^47, {1, 2, 3}}
In[1325]:= a = 1; b = 2; c = 3; {72*Block[{a, b, c}, a = 14; b = 21; c = 72; (a*x + x^b)/c], {a, b, c}}
Out[1325]= {14 x + x^21, {1, 2, 3}}
In[1326]:= Clear[a, b, c]; a = 1; b = 2; c = 3; {3*Block[{a, b, c}, (a*x + x^b)/c], {a, b, c}}
Out[1326]= {x + x^2, {1, 2, 3}}
In[1327]:= Clear[a, b, c]; {c*Block[{a, b, c}, (a*x + x^b)/c], {a, b, c}}
Out[1327]= {a x + x^b, {a, b, c}}
```

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.

Между тем, следует отметить, что и в рамках *модульной* структуры вполне может быть реализован механизм *локализации* глобальных переменных, подобно используемому в блочной структуре. Проиллюстрируем это утверждение на весьма простом примере двух возможных вариантов такого типа реализации, а именно:

```
In[91]:= Clear[a]; a = 69; Module[{b, c}, b = a; a = Sin[x] + Cos[x]; c = a; If[a === b, a, a = b; c]]
Out[91]= Cos[x] + Sin[x]
In[92]:= a
Out[92]= 69
In[93]:= Clear[a]; a = 69; Module[{b}, b = a; a = Sin[x] + Cos[x]; First[{a, a = b}]]
Out[93]= Cos[x] + Sin[x]
In[94]:= a
Out[94]= 69
```

При этом, возможны некоторые другие варианты реализации указанного механизма *локализации* глобальных переменных, используемого блочными структурами. Тут же уместно отметить, существует довольно простой и универсальный механизм работы с глобальными переменными в теле процедур, сохраняющий их значения на момент входа в процедуру и после выхода из нее. Формально суть его можно 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[1130]:= Q = 75; P[x_] := Module[{a = Q}, Q = 450; Res = Q + x; {Q = a, Res}][[-1]]
In[1131]:= {Q, Proc[70]}
Out[1131]= {75, 520}
```

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 используется достаточно эффективно.

Приведем некоторые примеры применения упомянутой возможности по *локализации* глобальных переменных текущего сеанса пакета *Mathematica*, а именно:

```
In[710]:= 14*x + 21*y
Out[710]= 14 x + 21 y
In[711]:= {Block[{x = a + 68, y = b + 63}, Expand[%]], %}
Out[711]= {2275 + 14 a + 21 b, 14 x + 21 y}
In[712]:= Avg = 63
Out[712]= 63
In[713]:= {Module[{Avg}, Avg + 14], Avg}
Out[713]= {14 + Avg$832, 63}
In[714]:= {Block[{Avg}, Avg + 14], Avg}
Out[714]= {77, 63}
In[715]:= Clear[a, b, h]; h := a^2 + b^3; a = 21
Out[715]= 21
In[716]:= {Block[{a = 72}, h + 14], h, a}
Out[716]= {5198 + b^3, 441 + b^3, 21}
In[717]:= Clear[Avz]; {Block[{Avz = 42}, Avz + 68], Avz}
Out[717]= {110, Avz}
```

Приведенные примеры достаточно просты и прозрачны, и с учетом вышесказанного не требуют какого-либо специального комментирования.

Блочная структура неявно используется в реализациях целого ряда функций пакета *Mathematica* таких как **Do**, **Table**, **Product**, **Sum** и др., прежде всего, *итеративного* типа для локализации переменных индексирования, что иллюстрирует простой пример:

```
In[959]:= n := 69; {{Sum[n^2, {n, 15}], n}, {Product[n, {n, 22}], n}}
Out[959]= {{1240, 69}, {1124000727777607680000, 69}}
```

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 (для *процедур* данный вопрос рассматривался в настоящем разделе выше), например:

```
In[1142]:= h = 75; G[x_, y_] := Sin[x] + h*Log[y]
In[1143]:= 1/G[Pi/4, 1]^2
Out[1143]= 2
In[1144]:= S[x_] := Block[{y = a, h = b}, G[Pi/2, y]]
In[1145]:= {S[10], h}
Out[1145]= {1 + b Log[a], 75}
```

```
In[1146]:= S1[x_] := Block[{y = a, h}, G[Pi/2, y]]
```

```
In[1147]:= {S1[10], h}
```

```
Out[1147]= {1 + 75 Log[a], 75}
```

В приведенном фрагменте определяется функция  $G[x, y]$ , зависящая и от глобальной переменной  $h$  в том числе. Посредством *блочной* структуры на базе данной функции, локализовав глобальные переменные  $h$  и  $y$ , создается  $S$ -функция одной переменной. В то же время следует иметь ввиду, что локализованные в блоке переменные только тогда являются таковыми, если им в блоке присваиваются значения, в противном случае их значения в блоке совпадают со значениями одноименных с ними переменных, внешних относительно блока, как наглядно иллюстрирует следующий простой фрагмент:

```
In[1846]:= a := 14; {Block[{a = 21}, a], a}
```

```
Out[1846]= {21, 14}
```

```
In[1847]:= a := 14; {Block[{a}, a := 21; a], a}
```

```
Out[1847]= {21, 14}
```

```
In[1848]:= a := 14; {Block[{a}, a], a}
```

```
Out[1848]= {14, 14}
```

```
In[1849]:= a := 14; {Block[{a}, a^2], a}
```

```
Out[1849]= {196, 14}
```

```
In[1850]:= Clear[a]; {Block[{a}, a^2], a}
```

```
Out[1850]= {a^2, a}
```

```
In[1851]:= Clear[a]; {Block[{a = 72}, a^2], a}
```

```
Out[1851]= {5184, a}
```

```
In[1852]:= Clear[a]; {Block[{a = 420}, a], a}
```

```
Out[1852]= {420, a}
```

Таким образом, в случае, если локализованной переменной в блоке не присваивалось в нем значений, то реальной локализации для такой переменной не производится. Есть и некоторые другие особенности локализации переменных в *блочной* структуре, здесь нами не рассматриваемые. Теперь вновь имеет смысл обратиться к сравнению блоков и модулей, имея в виду, что в целом ряде случаев имеет смысл создавать *процедурные* структуры также и на основе *блочной* организации. Таким образом, в общем смысле под *процедурными* объектами в *Mathematica* можно полагать созданные как на основе *модульной*, так и *блочной* организаций. Представим несколько процедур, позволяющих тестировать *локальные* переменные процедурного объекта, играющие существенную роль в задачах продвинутого процедурного программирования.

Прежде всего, следующая процедура **Locals[x]** возвращает значение **False**, если объект  $x$  не является *Mathematica*-процедурой, иначе вызов процедуры возвращает список в строчном формате *локальных* переменных процедуры  $x$ . Фрагмент ниже представляет исходный код процедуры **Locals** с результатами применения к ряду наших процедур из пакета [90] и рассматриваемых в настоящей книге.

```
In[625]:= Locals[x_] := Module[{a, b, c, d}, a = UprocQ[x];
```

```
    If[a === False, False, b = ToString[Definition[x]];
```

```
    If[Last[a] === Module, c = StringPosition[b, ":=" Module["],
```

```

c = StringPosition[b, ":=" Block[{""];
d = StringPosition[b, ","]; StringTake[b, {Last[First[c]], First[First[d]]}]
In[626]:= Map[Locals, {UprocQ, StringEnd, Uprocs, Kr, Gs, Art}]
Out[626]= {"{a, b, c, d, h}", "{a, b}", "{a, b, c, d, h, g, k, t1, t2}", "{y = a, h = b}", "{a, b, c}", "{a}"}

```

Представленная выше процедура **Locals[P]** обеспечивает возврат в строчном формате блока *локальных* переменных процедуры **P**. Однако в целом ряде случаев применение процедуры требует дополнительных операций. Вопрос упрощает процедура **Locals1**.

```

In[966]:= Locals1[P_ /; ProcQ[P]] := Module[{a = ToString1[DefFunc[P]], b, c = {}, d, h = 75,
t, k = 1}, If[Locals[P] === {}, Return[{}], b = ExprOfStr[a, Flatten[StringPosition[a,
":=" Module[{""]][2]], 1, {"", ""}]]; While[h > 1, c = Append[c, d = ExprOfStr[b, 2, 1, {"", ""}]];
b = StringTake[b, {StringLength[d] + 2, -1}]; h = StringLength[b];
c = Map[StringTrim, c]; t = c;
For[k, k <= Length[c], k++, d = c[[k]]; c[[k]] = If[StringFreeQ[d, "="], d,
StringTake[d, {1, Flatten[StringPosition[d, "="][[1]] - 1}]]; If[c == t, c, {c, t}]]
In[967]:= Map[Locals1, {ProcQ, Map}]
Out[967]= {"{a", "b", "c", "d", "h"}, {"{a", "b", "c", "d", "h", "t", "k"}, {"a = Level[Expr, 1]",
"b = Head[Expr]", "c = {x}", "d = Quiet[F /@ Expr]", "h", "t = {}", "k = 1"}]}

```

Вызов процедуры **Locals1[P]** возвращает список *локальных* переменных процедуры **P** в строчном формате, если *локальным* переменным не назначалось *начальных* значений, в противном случае вызов возвращает 2-элементный список *ListList*-типа, чей *первый* подсписок содержит *локальные* переменные процедуры **P**, тогда как *второй* подсписок содержит локальные переменные, наряду с их *начальными* значениями. В случае, если процедура **P** не имеет *локальных* переменных, вызов процедуры **Locals1[P]** возвращает пустой список, т.е. {}. Предыдущий фрагмент представляет исходный код процедуры **Locals1** с некоторыми наиболее типичными примерами ее применения.

Между тем, отождествление *процедурных* объектов на основе структур *Module* и *Block* следует проводить весьма осмотрительно, поскольку в общем случае они не являются эквивалентными, о чем свидетельствует следующий довольно прозрачный фрагмент:

```

In[881]:= Subs[x_, y_, z_] := Module[{a, b, c, h, t}, If[! HowAct[y], x /. y -> z, {a, b, c, h} =
First[{Map[ToString, Map[InputForm, {x, y, z, 1/y}]}]];
t = Simplify[ToExpression[StringReplace[StringReplace[a, b -> c,
h -> "1/" <> c]]]; If[t === x, x /. y -> z, t]]
In[882]:= Subs1[x_, y_, z_] := Block[{a, b, c, h, t}, If[! HowAct[y], x /. y -> z, {a, b, c, h} =
First[{Map[ToString, Map[InputForm, {x, y, z, 1/y}]}]];
t = Simplify[ToExpression[StringReplace[StringReplace[a, b -> c,
h -> "1/" <> c]]]; If[t === x, x /. y -> z, t]]
In[883]:= Subs[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[883]= (b^2 + c)/Sqrt[z]
In[884]:= Subs1[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[884]= ("Sqrt[z]" + ("x^2")^2)/Sqrt[z]

```

Одно из принципиальных различий между *процедурными* объектами на базе *Module* и *Block* состоит во *взаимосвязи* между значениями *фактических* аргументов и *локальными* переменными обоих типов объектов. Если в процедурном объекте, созданном на базе *Module*, вполне допустимо использование *фактических* аргументов, значения которых идентичны именам *локальных* переменных, то в случае с процедурными объектами на основе *Block*-конструкции это в общем случае недопустимо, вызывая ошибки.

```
In[93]:= Map4[F_, L_List, x_] := Block[{a = ToString[F] <> "[", b = ToString[x], c = {}, k = 1},
    For[k, k <= Length[L], k++, c = Append[c, a <> ToString[L[[k]]] <> "," <> b <> "]]; c]
In[94]:= Map4[F, {a, b, c, d, h}, x]
Out[94]= {"F[F[,x]", "F[x,x]", "F[{F[F[,x], F[x,x]],x]", "F[d,x]", "F[h,x]"}
```

```
In[95]:= ToExpression[%]
ToExpression::sntxi: Incomplete expression; more input is needed.
ToExpression::sntx: Invalid syntax in or before "F[{F[F[,x], F[x,x]],x]".
Out[95]= {$Failed, F[x, x], $Failed, F[d, x], F[h, x]}
```

```
In[96]:= Map4[F_, L_List, x_] := Block[{$Art22 = ToString[F] <> "[", $Kr14 = ToString[x],
    $gsv = {}, k = 1}, For[k, k <= Length[L], k++, $gsv =
    Append[$gsv, $Art22 <> ToString[L[[k]]] <> "," <> $Kr14 <> "]];
    ToExpression[$gsv]]
In[97]:= Map4[F, {a, b, c, d, h}, x]
Out[97]= {F[a, x], F[b, x], F[c, x], F[d, x], F[h, x]}
```

```
In[98]:= Map4[F_, L_List, x_] := Module[{a = ToString[F] <> "[", b = ToString[x], c = {},
    k = 1}, For[k, k <= Length[L], k++,
    c = Append[c, a <> ToString[L[[k]]] <> "," <> b <> "]]; ToExpression[c]]
In[99]:= Map4[F, {a, b, c, d, h}, x]
Out[99]= {F[a, x], F[b, x], F[c, x], F[d, x], F[h, x]}
```

Так, в предыдущем фрагменте на основе *Block*-конструкции реализована процедура **Map4**[*F*, *L*, *x*], возвращающая по логике реализуемого ею алгоритма результат в форме {*F*[*a*<sub>1</sub>, *x*], *F*[*a*<sub>2</sub>, *x*], *F*[*a*<sub>3</sub>, *x*], ...}, где *L*={*a*<sub>1</sub>, *a*<sub>2</sub>, *a*<sub>3</sub>, ...}. Между тем, если хоть один элемент из списка *L* совпадет с именем локальной переменной {*a*, *b*, *c*}, то результат выполнения процедуры будет инициировать ошибочную ситуацию, что наглядно иллюстрирует первая часть фрагмента. Во второй части фрагмента представлена реализация **Map4**, реализующая аналогичный алгоритм, но с обеспечением непересечения (до некоторой степени) множества *L* и локальных переменных, обеспечивая довольно существенную корректность процедуры **Map4**. И только реализованный в *третьей* части фрагмента аналогичный алгоритм на основе *Module*-конструкции обеспечивает надежность вне зависимости от пересечений множеств *L* и локальных переменных процедуры **Map4**.

Еще на одном довольно существенном моменте, относящемся к механизму *локальных переменных*, имеет смысл остановиться отдельно. Как известно, в большинстве систем программирования область локальных, впрочем, как и глобальных переменных при определении процедуры не допускает дублирования таких переменных, в частности, сказанное в полной мере имеет место для среды программирования пакета *Maple*, как

это весьма наглядно иллюстрирует простой пример. Так, уже на стадии вычисления определения процедуры *Proc* инициируется соответствующая ошибочная ситуация:

```
> Proc := proc(x::anything, y::anything) local a, b, c, a; x + y end proc:
```

Error, local `a` is declared more than once in procedure Proc

```
> Proc := proc(x::anything, y::anything) local a, b, c; global b, c; x + y end proc:
```

Error, local and global `b` have the same name in procedure Proc

Между тем, в среде программирования пакета *Mathematica* вычисление определений процедур, содержащих дублирование локальных переменных, производится вполне корректно без инициирования каких-либо ошибочных ситуаций, которые возникают только в момент вызова процедуры, инициируя ошибочную ситуацию **Module::dup** с возвратом вызова невычисленным. Правда, механизм идентификации дублируемой локальной переменной неясен, ибо первыми до нее в списке *локальных* переменных в определении процедуры **Proc** выступают *a* и *d*, как иллюстрирует следующий весьма простой пример. С целью определения факта дублирования локальных переменных в определениях активированных в текущем сеансе процедур создана процедура, чей вызов **DuplicateLocalsQ[P]** возвращает **True** при наличии в определении процедуры *P* дублирования *локальных* переменных, в противном случае возвращается **False**. При этом, при возврате **True** через второй необязательный аргумент возвращается список, простой либо *ListList*-типа, элементы которого определяют имена дублирующихся *локальных* переменных с их кратностями вхождений в список локальных переменных. Следующий фрагмент представляет исходный код с примерами ее применения.

```
In[1056]:= Proc[x_] := Module[{a, y, d = {x}, h, c, h, d, a = 75, h = 450, c = 2012, t, t, a}, a + d]
In[1057]:= Proc[75, 450]
Module::dup: Duplicate local variable h found in local variable specification
{a,y,d={75,450},h,c,h,d,a=75,h=450,c=2012,t,t,a}. >>
Out[1057]= Module[{a, y, d = {75, 450}, h, c, h, d, a = 75, h = 450, c = 2012, t, t, a}, a + d]
In[1058]:= DuplicateLocalsQ[P_] := Module[{a, b = {P}, c = Length[{P}]},
If[ProcQ[b[[1]]], a = Locals1[b[[1]]];
a = StringReplace[If[NestListQ[a, a[[1]], a], "\\\:0441" -> "c"],
Return[Defer[DuplicateLocalsQ[P]]];
a = Select[Gather[Flatten[a], Length[#] > 1 &];
If[a == {}, False, If[c > 1 && ! HowAct[b[[2]]],
ToExpression[ToString1[b[[2]]] <> "=" <> ToString1[Gather2[a]], Null]; True]]
In[1059]:= {DuplicateLocalsQ[ProcQ, G], G}
Out[1059]= {True, G}
In[1060]:= Clear[H]; {DuplicateLocalsQ[Proc, H], H}
Out[1060]= {True, {{a", 3}, {"d", 2}, {"h", 3}, {"c", 2}, {"t", 2}}}
```

В определении процедуры **DuplicateLocalsQ[P]** наряду со стандартными средствами используются и наши средства **ProcQ**, **Locals1**, **NestListQ**, **HowAct**, **ToString1**, которые рассматриваются в настоящей книге и представлены в пакете *AVZ\_Package* [90].

В целом ряде случаев появляется необходимость динамического расширения списка локальных переменных у процедуры, которая определена модульной конструкцией

и активирована в текущем сеансе пакета, без изменения самой исходной процедуры. Эту задачу решает процедура **ExpLocals**, представленная следующим фрагментом. Успешный вызов **ExpLocals[N,L]** возвращает список *локальных* переменных в *строчном* формате, на которые расширены локальные переменные процедуры *N*. При этом, в общем случае этот список может быть меньше заданного при вызове списка *L* (либо и вовсе пустым), так как из него исключаются переменные, имеющиеся в процедуре *N* в качестве формальных аргументов либо локальных переменных. Завершает фрагмент применение процедуры **ExpLocals** к весьма простым процедурам **Agn** и **Avz** с целью расширения их списка локальных переменных на 5 переменных; при этом, в первой процедуре список локальных переменных пуст, тогда как во второй имеется непустое пересечение *объединенного* списка формальных аргументов и локальных переменных со списком, на которые следует расширить список *локальных* переменных процедуры. Если объединенный список совпадает со списком *L*, то вызов процедуры возвращает пустой список {}, не изменяя исходной процедуры *N*. Отметим, процедуру **ExpLocals** можно довольно несложно обобщить на возможность расширения списков *локальных* переменных процедур на локальные переменные и с начальными условиями.

```
In[1165]:= ExpLocals[P_;/ ProcQ[P], L_;/ ListQ[L] &&
           If[DeleteDuplicates[Map[SymbolQ, L]] == {True}, True, False] :=
           Module[{a = ToString1[DefFunc[P]], t = Locals1[P], h, b,
                  d = Map[ToString, L], m, c = Map[ToString, Args1[P]]},
           t = If[NestListQ[t], t[[1]], t]; h = Join[t, c];
           h = Select[MinusList[d, h], SymbolQ[#] &];
           If[h == {}, Return[{}], b = Flatten[StringPosition[a, HeadPF[P] <> " := Module[{"];
           m = ListStrToStr[h];
           ToExpression[StringInsert[a, If[t == {}, m, m <> ", "], b[[2]] + 1]; h]

In[1166]:= Avz[x_] := Module[{a = 75, b, c}, a + x^2]; Agn[x_] := Module[{}, {x}];
In[1167]:= ExpLocals[Agn, {x, a, b, Art, Sv, Kr}]
Out[1167]= {"a", "b", "Art", "Sv", "Kr"}
In[1168]:= DefFunc[Agn]
Out[1168]= Agn[x_] := Module[{a, b, Art, Sv, Kr}, {x}]
In[1169]:= ExpLocals[Avz, {x, a, b, c, z, v, Art, Sv, Kr}]
Out[1169]= {"z", "v", "Art", "Sv", "Kr"}
In[1170]:= DefFunc[Avz]
Out[1170]= Avz[x_] := Module[{z, v, Art, Sv, Kr, a = 75, b, c}, a + x^2]
In[1171]:= ExpLocals[Agn, {b, Art, Sv, Kr}]
Out[1171]= {}
```

Представленные здесь и далее по книге средства работы с локальными переменными процедур оказываются достаточно полезными в процедурном программировании в среде пакета *Mathematica*. Перейдем теперь к рассмотрению механизмов *глобальных* переменных и возврата результатов вызовов процедурами (*модулями*) в программной среде пакета *Mathematica*.

### 6.3. Глобальные переменные процедур и механизмы возврата результатов процедурами в среде пакета 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*, то в *Mathematica*-процедуре *глобальными* полагаются все переменные, которые явно не определены в качестве локальных. Следующий пример весьма наглядно иллюстрирует сказанное.

```
In[1521]:= Sv[x_] := Module[{}, y := 69; z = 64; {y, z}]
In[1522]:= {y, z} = {42, 47}; {Sv[2012], y, z}
Out[1522]= {{69, 64}, 69, 64}
```

Следовательно, любое переопределение в процедуре (*модуле*) *Mathematica* *глобальной* переменной автоматически переопределяет *одноименную* с ней переменную вне такой процедуры, что требует *большой* внимательности на предмет недопущения *возможных* нежелательных ситуаций, чем в аналогичной ситуации с *Maple*-процедурами. Таким образом, *уровень* обеспечения *робастности* программных средств, которые используют процедуры *Mathematica*, представляется нам несколько более низким относительно упомянутого математического пакета *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[1651]:= Art[x_ /; If[! IntegerQ[x], h = 75; True, h = 420; True], y_] :=
Module[{a = 100}, x + y + h + a]
In[1652]:= {Art[75, 420], Art[15.22, 420]}
Out[1652]= {1015, 610.22}
In[1711]:= Kr[x_, y_] := Module[{a = If[IntegerQ[x], 75, 420]}, x + y + a]
In[1712]:= {Kr[7.5, 420], Kr[100, 200]}
Out[1712]= {847.5, 375}
In[1713]:= Sv[x_, y_] := Module[{a = If[IntegerQ[x] && PrimeQ[y], 75, 420]}, x + y + a]
In[1714]:= {Sv[75, 420], Kr[15, 127]}
Out[1714]= {915, 217}
In[1888]:= H[x_: 75, y_, z_] := Module[{}, x + y + z]
In[1889]:= {H[100, 200, 300], H[100, 200], H[100]}
Out[1889]= {600, 375, H[100]}
```

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** [90]. Следующий фрагмент представляет *исходный* код процедуры **Globals** наряду с наиболее типичными примерами ее применения.

```
In[944]:= Globals[P_ /; ProcQ[P]] := Module[{a = If[P === ExprOfStr, {}, Sort[Locals1[P]],
b = ToString1[DefFunc[P]], c, d = {}, 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 = 75, c = 420}, a = (x^2 + y^2)/(b + c); {z, h} = {a, b};
t = z + h; t]; GS[x_] := Module[{a, b = 75, c = 42}, Kr[y_] := Module[{}, y^2 + Sin[y]];
a = x^2; {z, h, p} = {a, b, 15}; t = z + h*Kr[6] - Cos[x + Kr[75]]; t]
In[946]:= Map[Globals, {Locals1, Locals, Globals, ProcQ, ExprOfStr, GS, DefFunc, Sv}]
Out[946]= {{}, {}, {}, {}, {}, {"h", "Kr", "p", "t", "z"}, {}, {"h", "t", "z"}}
```

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. В таком случае требуется дополнительное исследование либо процедура **Globals** может быть расширена и на этот 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 достаточно простой процедурой **Globals1**, вызов которой **Globals1[P]** возвращает список глобальных переменных в строчном формате для процедуры *P*; при этом, в качестве аргумента *P* может выступать и процедура, не содержащая в своем теле подпроцедур различного уровня *вложенности*, и процедура, содержащая такие подпроцедуры. Следующий фрагмент представляет исходный код процедуры **Globals1** с наиболее типичными примерами ее применения.

```
In[898]:= 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[899]:= 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]; Globals1[P]
Out[899]= {"Q", "T", "V"}
In[900]:= Sv[x_, y_] := Module[{a, b = 75, c = 420}, a = (x^2 + y^2)/(b + c); {z, h} = {a, b};
t = z + h; gs = t^2]; Globals1[Sv]
Out[900]= {"gs", "h", "t", "z"}
```

В программной среде пакета *Mathematica* процедура может одним или несколькими способами одновременно возвращать результаты своего выполнения, располагая 4–я следующими возможностями, а именно:

- (1) *через результат выполнения последнего предложения процедуры;*
- (2) *через глобальные переменные процедуры;*
- (3) *через Return-функцию пакета;*
- (4) *через неопределенные фактические аргументы-переменные.*

Следующий простой фрагмент иллюстрирует первые 3 перечисленные возможности возврата результатов одной *Mathematica*-процедурой, а именно:

```
In[11]:= GS[x_, y_] := Module[{a}, a := 68; b := 63; If[x + y < a, Return[x^2 + y^2], c = x*y]; a]
In[12]:= {{GS[14, 21], b, c}, {GS[42, 47], b, c}}
Out[12]= {{637, 63, c}, {68, 63, 1974}}
In[13]:= Clear[a, b, c]; AK[x_] := Block[{a}, a := 68; b := 63; If[x < 68, b = 47; Return[x^2],
c = 42*x]; b + c]
In[14]:= {{AK[47], b, c}, {AK[72], b, c}}
Out[14]= {{2209, 47, c}, {3087, 63, 3024}}
```

Здесь уместно также упомянуть весьма полезную возможность возврата результатов и через *фактический* аргумент, в качестве которого выступает символ, не получивший в текущем сеансе значения до вызова процедуры или функции. Следующий фрагмент иллюстрирует данный подход к организации *возврата* результатов как основного, так и сопутствующих из весьма простой процедуры **GS**, в определении которой второй и третий формальные аргументы определяются невычисленными и именно через них производится возврат некоторых результатов выполнения данной процедуры.

```
In[425]:= GS[x_, Res1_ /; ! HowAct[Res1], Res2_ /; ! HowAct[Res2]] := Module[{a},
Res1 = 68; Res2 = 63; a = Res1^2 + Res2^2 + x^2; a]
In[426]:= Z = 420; {GS[14, Z, Q], Z, Q}
Out[426]= {GS[14, 420, Q], 420, Q}
In[427]:= {GS[22, W, Q], W, Q}
Out[427]= {9077, 68, 63}
In[428]:= G[x_, y_] := Module[{}, x + y; %]
Out[428]= G[72, 420]
> G := proc(x, y) x + y; % end proc: G(72, 420); => 492
```

В определенном смысле данный подход к механизмам возврата вызова процедур или функций аналогичен использованию для этих целей *evaln*-аргументов пакета *Maple*. Наконец, последние *два* примера фрагмента иллюстрируют тот немаловажный факт, что в отличие от пакета *Maple*, пакет *Mathematica* не допускает использования в теле процедур *%*-оператора, возвращающего результат последнего вычисления.

Довольно важной представляется возможность возврата результатов вызова процедур через фактические аргументы при неопределенном *числе* формальных аргументов. Но здесь возникают некоторые затруднения. В программной среде пакета *Maple* для этих целей можно воспользоваться процедурной переменной *args[k]*, принимающей в теле процедуры в точке ее вызова значение переданного ей *k*-го фактического аргумента. Достаточно детально данный вопрос рассмотрен нами в книгах [34,35,42-44,99]. Тогда как в программной среде *Mathematica* для обеспечения возврата результата через *k*-й необязательный фактический аргумент, которому не присваивалось значений перед входом в процедуру, вполне достаточно использовать конструкцию следующего вида

**ToExpression[ToString[{x}][[k]]] <> "=" <> ToString[Result]]**

где {x} – список фактических аргументов процедуры, как наглядно иллюстрируют *два* достаточно простых примера нижеследующего фрагмента, а именно:

```

In[1755]:= Agn[x_] := Module[{a = {x}}, If[Length[a] == 2, If[! HowAct[a[[2]]], a[[2]] =
a[[1]]^4, Null], Null]; a[[1]]^2]
In[1756]:= {Agn[75], {Agn[75, t], t}}
Out[1756]= {5625, {5625, t}}
In[1757]:= Agn[x_] := Module[{a = {x}}, If[Length[a] == 2, If[! HowAct[a[[2]]],
ToExpression[ToString[a[[2]]] <> "=" <>
ToString[a[[1]]^4]], Null], Null]; a[[1]]^2]
In[1758]:= {Agn[75], {Agn[75, t], t}}
Out[1758]= {5625, {5625, 31640625}}

```

В заключение данного раздела еще раз акцентируем внимание на принципиальной значимости заголовков процедур, т.е. на той конструкции в определении процедуры, которая в нем является *префиксом* до оператора присвоения «:=». Если в большинстве систем программирования объект определяется его уникальным *именем* допустимого системой формата, то в *Math*-языке в качестве *идентификатора* процедуры выступает не ее имя, а именно ее заголовок. При этом, одноименные процедуры, но с разными заголовками являются доступными в текущем сеансе пакета, т.е. сохраняется история активизации их определений. В качестве достаточно наглядного примера приведем определения трех идентичных по смыслу процедур, каждая из которых располагает своим индивидуальным *заголовком* – главным идентификатором процедур/функций.

```

In[968]:= P[x_] := Module[{}, x]; P[y_] := Module[{}, y]; P[y_ /; PrimeQ[y]] := Module[{}, y]
In[969]:= InputForm[DefFunc[P]]
Out[969]// InputForm=
P[y_ /; PrimeQ[y]] := Module[{}, y]
P[y_] := Module[{}, y]

```

Из приведенного примера довольно наглядно следует, что различными полагаются лишь те заголовки, которые различны семантически, но не синтаксически (*например, разные имена формальных аргументов*). Так, пакет полагает две процедуры различными из-за наличия у них семантически разных заголовков, оставляя активной процедуру с одинаковыми заголовками, определение которой было активизировано последним в текущем сеансе. Выше приведен ряд полезных процедур для работы с заголовками процедур, довольно широко используемых в программировании системных средств.

Вполне допускается использование процедур и блоков без *заголовков* (*непоименованные объекты*). В таком случае оба данных объекта, по сути дела, становятся идентичными, допуская использование в качестве *подвыражений* в составе более сложных выражений, как это весьма наглядно иллюстрирует нижеследующий простой пример, а именно:

```

In[989]:= {x, y} = {2012, 69}; (5*Module[{a = 75}, x + y + a] + Block[{b = 22.15}, b + x/y])/
(Block[{a = 100}, x + y + a] + Module[{}, x*y])
Out[989]= 0.0768129

```

Между тем, результат использования для выходов внутри непоименованных блоков и процедур функции **Return** в стандартном формате не поддерживается, как довольно наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```

In[997]:= {x, y} = {2012, 69}; (5*Module[{a = 75}, If[x > y, Return["agn"], a = 450]; x + y + a] +
      Block[{b = 22.15}, b + x/y])/(Block[{a = 100}, x + y + a] + Module[{}, x*y])
Out[997]= (51.3094 + 5 Return["agn"])/141009
In[998]:= {x, y} = {"64", 450}; Module[{a = 69}, If[StringQ[x], "agn", x + y]]
Out[998]= "agn"
In[999]:= Module[{a = 69}, If[StringQ[x], Return["agn"], x + y]]
Out[999]= Return["agn"]
In[1000]:= Block[{a = 69}, If[StringQ[x], "agn", x + y]]
Out[1000]= "agn"
In[1001]:= Block[{a = 69}, If[StringQ[x], Return["agn"], x + y]]
Out[1001]= Return["agn"]
In[1002]:= Module[{a = 69}, If[StringQ[x], ToExpression["Return[" <> ToString1["agn"] <>
      "]"], x + y]]
Out[1002]= Return["agn"]
In[1003]:= Block[{a = 69}, If[StringQ[x], ToExpression["Return[" <> ToString1["agn"] <>
      "]"], x + y]]
Out[1003]= Return["agn"]
In[1004]:= Map[ObjType, {Module[{a = 69}, Return["agn"]; x + y], Block[{h = 64}, h^t]}]
Out[1004]= {Return, Power}

```

Из данного фрагмента следует, что не дает ожидаемого результата и искусственный прием. Более того, *непоименованные* процедуры и блоки тестируются представленной выше процедурой **ObjType** лишь по типу возвращаемого ими результата, как весьма наглядно иллюстрирует последний пример предыдущего фрагмента. В данной связи использование в *непоименованных* процедурах и блоках функции **Return** для возврата результатов не представляется целесообразным, требуя дополнительной обработки.

## 6.4. Некоторые полезные средства для работы с процедурами в программной среде Mathematica

Выше был представлен целый ряд довольно полезных средств работы с процедурами и в значительной мере также с функциями, здесь будут дополнительно представлены *общие* средства для работы с объектами данного типа. Прежде всего, в качестве весьма полезного средства можно привести процедуру, чей вызов **Args[x]** возвращает список типированных *формальных* аргументов процедуры/функции *x*. Следующий фрагмент представляет исходный код и примеры применения процедуры.

```

In[1339]:= Args[x_] := Module[{a, b, c, ArtKr},
      ArtKr[P_] := Module[{a, b, c, d = {}, Test, k, h},
            Test[S_String, N_] := Module[{a = "" <> ToString[N] <> "", b = "", c = {},
                  d, k = 1, p, h}, h = StringPosition[S, a];
                  For[k, k <= Length[h], k++, For[p = h[[k]][[1]] - 1,
                        p >= 1, p--, d = StringTake[S, {p, p}]]];

```

```

If[p == 1 || MemberQ[{" ", "["], d], c = Append[c, If[p == 1,
d <> b <> a, b <> a]]; b = ""; Break[], b = d <> b]];
DeleteDuplicates[c]; a = ToString[Definition[P]];
a = StringTake[a, {StringLength[ToString[P]] + 1,
StringPosition[a, " := "][[1]][1] - 1}];
h = Map[ToString, ToExpression["{" <> StringTake[a, {2, -2}] <> "}"];
For[k = 1, k <= Length[h], k++, d = Append[d, Test[h[[k]], P]];
d = DeleteDuplicates[Flatten[d]]; c = {};
For[k = 1, k <= Length[d], k++, c = Append[c, d[[k]] -> "]];
ToExpression["{" <> StringTake[StringReplace[a, c], {2, -2}] <> "}"];
a = Quiet[Check[ToString[Definition[x]], {}];
If[a === {} || a === "Null" || If[StringTake[a,
{12, 11 + StringLength[ToString[x]]}] === ToString[x],
Information[x]; True, False], Return[Undefined],
a = StringTake[a, {1, Flatten[StringPosition[a, " := "][[1]] - 1}];
If[! StringFreeQ[a, "{" <> ToString[x] <> "}, ArtKr[x],
a = StringTake[a, {Flatten[StringPosition[a, ToString[x] <> "["][[2]] + 1, -2}];
ToExpression["{" <> a <> "}"]]]

```

In[1340]:= Map[Args, {StrDelEnds, Args, 75, Length, (a + b)/(c + d), Sin, 2012}]

Length[expr] gives the number of elements in expr. >>

Attributes[Length] = {Protected}

Sin[z] gives the sine of z. >>

Attributes[Sin] = {Listable, NumericFunction, Protected}

Out[1340]= {{S\_String, h\_;/; CharacterQ[h], p\_;/; MemberQ[{1, 2, 3}, p]}, {x\_}, \$Failed,  
Undefined, \$Failed, Undefined, \$Failed}

При этом, в случае невозможности вычислить аргументы  $x$  вызов возвращает **\$Failed**, тогда как на стандартных функциях *Mathematica* возвращается значение **Undefined** с выводом по ним справочной информации. Более того, при вызове процедуры **Args[x]** в качестве фактического аргумента может выступать произвольное выражение.

В качестве довольно полезной модификации предыдущей процедуры **Args** является процедура **Args0**, вызов которой **Args0[x]** возвращает список формальных аргументов процедуры/функции  $x$ , тогда как при невозможности определения аргументов вызов **Args0[x]** возвращается *невывчисленным*. Следующий фрагмент представляет исходный код процедуры **Args0** наряду с типичными примерами ее применения.

```

In[1084]:= Args0[P_;/; ProcQ[P]] := Module[{a, b, c, d = {}, Test, k, h}, Test[S_String, N_] :=
Module[{a = "" <> ToString[N] <> "", b = "", c = {}, d, k = 1, p, h},
h = StringPosition[S, a];
For[k, k <= Length[h], k++,
For[p = h[[k]][[1]] - 1, p >= 1, p--, d = StringTake[S, {p, p}];
If[p == 1 || MemberQ[{" ", "["], d], c = Append[c, If[p == 1, d <> b <> a, b <> a]];
b = ""; Break[], b = d <> b]]; DeleteDuplicates[c]; a = ToString[Definition[P]];

```

```

a = StringTake[a, {StringLength[ToString[P]] + 1, StringPosition[a, " := "][[1]][[1]] - 1}];
h = Map[ToString, ToExpression["{" <> StringTake[a, {2, -2}] <> "}"];
For[k = 1, k <= Length[h], k++, d = Append[d, Test[h[[k]], P]]];
d = DeleteDuplicates[Flatten[d]]; c = {};
For[k = 1, k <= Length[d], k++, c = Append[c, d[[k]] -> ""];
ToExpression["{" <> StringTake[StringReplace[a, c], {2, -2}] <> "}"]
In[1085]:= Map[Args0, {H, Sin, ProcQ, DefFunc3, 75}]
Out[1085]= {Args0[H], Args0[Sin], {x_}, {x_/; ProcQ[x] | | Length[Args[x]] > 0}, Args0[75]}

```

Процедура **Args1** – еще одна полезная модификация вышерассмотренной процедуры **Args**, вызов которой **Args1[P]** возвращает список формальных аргументов процедуры **P** в строчном формате, активизированной в текущем сеансе тем либо иным способом. Следующий фрагмент представляет исходный текст функции и пример применения, тогда как последний пример представляет применение процедуры для проверки на многозначность одноименных процедур **Rans** с различными заголовками. В отличие от представленных процедур **Args** и **Args0** процедура **Args1** обрабатывает ситуацию «одноименные процедуры с различными заголовками».

```

In[7]:= Args1[P_/; ProcQ[P]] := Module[{a = Length[ProcCalls[P]]}, If[a == 1,
Map[StringTake[#, {1, StringPosition[#, " _"][[1]][[1]] - 1]} &,
Map[ToString, Args[P]]], ToString[P] <> " has " <> ToString[a] <>
" copies with different headings"]]
In[8]:= A[m_, n_, p_/; IntegerQ[p], h_/; PrimeQ[h]] := Module[{a=42.75}, h*(m + n + p)/a]
In[9]:= Rans[x_] := Module[{}, x]; Rans[x_, y_/; PrimeQ[y]] := Module[{}, x+y]; Args1[A]
Out[9]= {"m", "n", "p", "h"}
In[10]:= Rans[x_, y_] := Module[{}, x+y]; Args1[Rans]
Out[10]= "Rans has 3 copies with different headings"

```

Вызов простой процедуры **Args2[P]** возвращает список формальных аргументов для невычисленного вызова **P** формата **F[x, y, z, ...]** процедуры/блока/функции/объекта **F** в формате **{F, x, y, z, ...}**, как иллюстрирует следующий прозрачный фрагмент:

```

In[934]:= Args2[x_] := Module[{a = ToString1[x], b = StringLength[ToString[Head[x]]]},
ToExpression["{" <> StrStr[Head[x]] <> "," <> StringTake[a, {b + 2, -2}] <> "}"]]
In[935]:= Args2[InputStream["C:/Temp\\GrSu.doc", 156]]
Out[935]= {InputStream, "C:/Temp\\GrSu.doc", 156}
In[936]:= Quiet[Map[Args2, {F[], Sin[x], Sin[19.42], ProcQ[x]}]]
Out[936]= {{F, Null}, {Sin, x}, {Real, 5865601997}, $Failed}

```

Использовать процедуру **Args2** следует внимательно, однако в целом ряде случаев ее применение оказывается достаточно эффективным, например, в работе с файлами, как весьма наглядно иллюстрирует первый пример предыдущего фрагмента. Более того, в ряде случаев целесообразно использовать **Args2** в композиции с **Quiet**.

Вызов процедуры **ArgsProc[P]** возвращает список *формальных* аргументов процедуры **P** с приписанными им *условиями*, т.е. в форме «**x\_/; Test**», определяющей допустимым

для аргумента  $x$  то значение, на котором *Test* принимает значение **True**, в противном случае возвращая вызов процедуры *P* *невычисленным*. Если аргумент *P* не определяет активизированную в текущем сеансе процедуру, то вызов **ArgsProc[P]** возвращается невычисленным; данный результат имеет место и для системной функции в качестве фактического аргумента *P*. При невозможности определить формальные аргументы вызов процедуры **ArgsProc[P]** возвращает значение *Undefined*. Следующий фрагмент представляет исходный код процедуры **ArgsProc** с примерами ее применения.

```
In[986]:= ArgsProc[x_ /; ProcQ[x]] := Module[{a = HeadPF[x], b, c = ToString[x] <> "["},
                                             b = Flatten[StringPosition[a, {c}]];
                                             a = ToExpression["{" <> StringTake[a, {b[[2]] + 1, -2}] <> "}"];
                                             If[a == {}, Undefined, a]]

In[987]:= F[] := Module[{}, 75]; Map[ArgsProc, {Mapp, ProcQ, Sin, HeadPF, ExpFunc}, F]
Out[987]= {{f_, g_ /; ListQ[g], h_}, {x_}, ArgsProc[Sin], {F_ /; SymbolQ[F]},
           {f_, x_}, Undefined}

In[988]:= Map[ArgsProc, {Sin, Agn, Vsv + Art + Kr}]
Out[988]= {ArgsProc[Sin], ArgsProc[Agn], ArgsProc[Art + Kr + Vsv]}
```

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 либо локальные переменные в строчном формате внешней по отношению к **Avg** процедуры, или 2-х элементные списки, первые элементы которых определяют локальные переменные в строчном формате внешней процедуры, тогда как вторые – их *начальные* значения. При отсутствии у *внешней* процедуры локальных переменных вызов процедуры **Avg[]** возвращает пустой список, т.е. {}. Вызов **Avg[]** вне другой процедуры вызывает ошибочную ситуацию, как иллюстрирует фрагмент.

```
In[6]:= Avg[] := Module[{a = ToString[ToExpression[ToString[InputForm[Stack[_][[1]]]]],
                        b, c, d, k = 1], b = Map[ToString, ToExpression[ExprExpr[a, 8, StringLength[a]]];
                        If[b == {}, Return[{}], c = b[[-1]] <> ""]; c = StringPosition[a, b[[-1]] <> ""];
                        c = ToExpression[StringReplace[StringTake[a, {8, c[[1]][[2]]], {"=" -> ";", "$" -> ""}]];
                        d = {}; For[k, k <= Length[c] - 1, k++, a = c[[k]];
                        If[SymbolQ[a] && ! SymbolQ[c[[k + 1]]], d = Append[d, {ToString[a], c[[k + 1]]}];
                        k = k + 1, d = Append[d, ToString[a]]]; a = c[[Length[c]];
                        d = Append[d, If[! HowAct[a], ToString[a], Null]]; If[! FreeQ[d, Null], d[[1 ;; -2]], d]]

In[7]:= Z[m_, n_, p_ /; IntegerQ[p]] := Module[{h}, m + n + p; h = Avg[]; h]
In[8]:= Z[69, 64, 44]
Out[8]= {h}

In[9]:= G[m_, n_, p_ /; IntegerQ[p]] := Module[{a, b = 6, c, d = 75}, d = Avg[]; m + n + p; d]
In[10]:= G[t, p, 69]
Out[10]= {"a", {"b", 6}, "c", {"d", 75}}

In[11]:= A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] := Module[{a = 420.75, b, c, t, q,
                                                                d = 75, z = 45}, b = Avg[]; m + n + p + h; m*n; b]
```

```

In[12]:= A[x, y, 42, 47]
Out[12]= {{a, 420.75}, {b, "c", "t", "q", {"d", 75}, {"z", 45}}
In[13]:= B[m_, n_, p_, h_ /; PrimeQ[h]] := Module[{a = 420.75, b, c = {h, p}, t, q, d = 75,
                                         z = p*t, s}, b = Avg[]; m + n + p + h; m*n; b]
In[14]:= B[x, y, 42, 47]
Out[14]= {{a, 420.75}, {b, {"c", {47, 42}}}, {t, "q", {"d", 75}, {"z", 42*t}, {"s"}
In[15]:= Avg[]
ToExpression::sntx: Invalid syntax in or before ....
Out[15]= {"$Failed", "$Failed", "AladjevProceduresAndFunctions`Avg`b",
"AladjevProceduresAndFunctions`Avg`c", {"AladjevProceduresAndFunctions`Avg`d", 1}}

```

Предыдущий фрагмент представляет исходный текст процедуры **Avg** и примеры ее применения для получения в теле процедуры списка ее локальных переменных. В то время как следующая процедура **Nproc** является определенным дополнением к **Avg** процедуры; вызов процедуры **Nproc[]**, закодированной в теле процедуры, при вызове последней обеспечивает получение списка  $\{W, m\}$ , где  $W$  – имя главной процедуры в строчном формате и  $m$  – число ее формальных, но не фактических аргументов. Ниже представлен исходный код процедуры наряду с примерами ее применения.

```

In[1828]:= Nproc[] := Module[{a, b, c, g, f = "$$Art23$Kr15$.txt", k = 1, h = ""},
    a = Uprocs[]; b := Stack[Module][[1]]; Write[f, b]; Close[f];
    b = StringReplace[StringTake[SubDelStr[Read[f, String], {{1, 9}}], {1, -2}],
    "$" -> ""]; Close[f]; DeleteFile[f]; c = OpenWrite[f];
    For[k, k <= Length[a], k++, Write[c, ToExpression[StringReplace[
    ToString[InputForm[Definition[x]]], "x" -> ToString[a[[k]][[1]]]]]; Close[c];
    Label[g]; h = Read[f, Expression]; If[h == EndOfFile, Close[f]; DeleteFile[f];
    Return[$Failed], If[StringEnd[h, b], Null, Goto[g]]]; Close[f]; DeleteFile[f];
    a = StringPosition[h, {"[", ""]];
    {StringTake[h, {1, a[[1]][[1]] - 1}], Length[DeleteDuplicates[Flatten[
    StringPosition[ StringTake[h, {a[[1]][[1]], a[[2]][[1]]], ", "]]] + 1]}
In[1829]:= Avz[x_Integer, y_Integer, z_Real] := Module[{a, b, c}, b = c*(x + y);
                                         b = Sqrt[GS[x] + y + z*b]; Nproc[]
In[1830]:= Avz[15, 23, 69.64]
Out[1830]= {"Avz", 3}
In[1831]:= Agn[x_] := Module[{a, b, c}, b = c*(x + y); b = Sqrt[GS[x] + y + z*b]; Nproc[]
In[1832]:= Agn[15, 23, 69, 64, 44]
Out[1832]= {"Agn", 1}

```

Здесь вполне уместно еще раз отметить то немаловажное обстоятельство, процедуры различаются их *заголовками*, как это иллюстрирует следующий фрагмент. Более того, при вызове процедур в качестве выбранной полагается первая из списка *одноименных* процедур, определяемого стандартной функцией **Definition**, на которой допустимым является кортеж фактических аргументов. Это обстоятельство следует учитывать при программировании и оно было учтено нами в программировании ряда процедур из

нашего пакета *AVZ\_Package* [90]. Для определения одноименных процедур текущего сеанса пакета предназначена довольно простая процедура, вызов которой **ProcCalls[g]** возвращает список *заголовков* в строчном формате всех процедур с именем **g**, которые активированы в текущем сеансе. Следующий фрагмент представляет исходный текст процедуры **ProcCalls** наряду с типичными примерами ее применения.

```
In[601]:= A[x_] := Module[{a = 450}, x + a]; A[x_, y_] := Module[{a = 75}, x + y + a];
        A[x_]; IntegerQ[x]] := Module[{a = 450}, x + a];
In[602]:= Definition[A]
Out[602]= A[x_]; IntegerQ[x]] := Module[{a = 450}, x + a]
        A[x_] := Module[{a = 450}, x + a]
        A[x_, y_] := Module[{a = 75}, x + y + a]
In[603]:= {A[100], A[100, 200], A[450.75]}
Out[603]= {550, 375, 900.75}
In[699]:= ProcCalls[P_]; ProcQ[P]] := Module[{a = Select[StringSplit[ToString[
        InputForm[DefFunc[P]]], "\n"], ! StringFreeQ[#, " := "] &], b},
        b[x_]; StringQ[x]] := Module[{a = StringPosition[x, " := "]},
        StringTake[x, {1, a[[1]][[1]] - 1}]]; Map[b, a]]
In[700]:= 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];
In[701]:= ProcCalls[G]
Out[701]= {"G[x_]; PrimeQ[x]", "G[x_]", "G[x_, y_]", "G[x_, y_]; ListQ[y], z_"]
In[909]:= ScanLikeProcs[x_ : {}] := Module[{b = {}, c = {}, d, h, k = 1, a = Select[Names["`*"],
        ProcQ[ToExpression[#]] && StringFreeQ[#, "$"] &], If[a == {}, Return[{}],
        For[k, k <= Length[a], k++, d = ToExpression[a[[k]]]; h = ProcCalls[d];
        If[Length[h] > 1, b = Append[b, d]; c = Append[c, h]]];
        If[! HowAct[x], x = c, Null]; b]
In[910]:= 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];
In[911]:= V[x_] := Module[{x^2}; V[x_]; ListQ[x]] := Module[{x, Length[x]}
In[912]:= {ScanLikeProcs[], ScanLikeProcs[S], S}
Out[912]= {{G, V}, {G, V}, {"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 процедур текущего сеанса пакета предназначена довольно простая процедура, чей вызов **ScanLikeProcs[]** возвращает список *всех* одноименных процедур, активизированных в текущем сеансе, тогда как в результате вызова **ScanLikeProcs[W]** дополнительно через переменную **W** возвращается список *заголовков* в строчном формате *подобных* процедур. Предыдущий фрагмент представляет исходный код процедуры и примеры ее применения. В ряде приложений данные средства представляются нам достаточно полезными, в первую очередь, при разработке системных средств манипулирования с процедурами.

Достаточно полезной при определении для процедуры как формальных аргументов, так и локальных переменных является процедура **ArgsLocals**, чей вызов **ArgsLocals[x]** возвращает вложенный 2-элементный список, первый элемент-подсписок которого определяет *формальные* аргументы процедуры *x*, тогда как второй элемент-подсписок определяет ее *локальные* переменные с начальными значениями в строчном формате. При отсутствии локальных переменных возвращается *пустой* подсписок, тогда как на *стандартных* функциях вызов процедуры **ArgsLocals** возвращает **System**, на *остальных* фактических аргументах *x* вызов процедуры **ArgsLocals[x]** возвращает выражение *x* в приведенном виде. Следующий фрагмент представляет исходный текст процедуры с некоторыми типичными примерами ее использования.

```
In[1021]:= ArgsLocals[x_] := Module[{a = "" <> ToString[x] <> "", b, c, d, h},
    If[MemberQ[{And, Complex, Integer, List, Not, Or, Plus,
    Power, Rational, Real, String, Times}, Head[x]], Return[x],
    If[SystemQ[x], Return[ToExpression[SubsDel[ToString[System], a, {" ", ",", "["], -1]],
    b = SubsDel[HeadPF[x], a, {"[", " ", "-1"}]; c = SubsDel[Locals[x], a, {"[", " ", " ", "-1"}];
    b = ToExpression["{" <> StringTake[b, {StringLength[ToString[x]] + 2, -2}] <> "}"];
    c = Flatten[StringSplit[c, " "]; c[[1]] = StringTake[c[[1]], {2, -1}];
    c[[-1]] = StringTake[c[[-1]], {1, -2}]; {b, If[c == {}, {}, c]}]

In[1022]:= {ArgsLocals[Subs], ArgsLocals[Sin]}
Out[1022]= {{{x_, y_, z_}, {"d", "k = 2", "subs"}}, System}
In[1023]:= {ArgsLocals[ProcQ], ArgsLocals[Args]}
Out[1023]= {{{x_}, {"a", "b", "c", "d", "h"}}, {{x_}, {"c", "ArtKr"}}}
In[1024]:= ArgsLocals[ArgsLocals]
Out[1024]= {{x_}, {"a = `<>ToString[x]<>`", "b", "c", "d", "h"}}
In[1025]:= Map[ArgsLocals, {75, "2012", Return, a + b, a/b, Sqrt}]
Out[1025]= {75, "2012", System, a + b, a/b, System}
```

Задачу арности процедуры/функции решает довольно простая процедура **Arity**, чей вызов **Arity[W]** возвращает арность процедуры/функции *W*. Фрагмент представляет исходный код процедуры **Arity** наряду с типичными примерами ее применения.

```
In[1104]:= Arity[x_] := Module[{a = If[SystemQ[x], System, ArgsProc[x]]},
    If[a === System, System, If["InputForm[ArgsProc[" <> ToString1[x] <> "]" ===
    ToString1[InputForm[a]], Return[$Failed], If[ListQ[a],
    If[Length[Select[Map[ToString, a], ! StringFreeQ[#, "___"] &]] != 0,
    Undefined, Length[a]], Undefined]]]

In[1105]:= Map[Arity, {ProcQ, Subs, Sin, 2012, SubsDel, x + y, Log}]
Out[1105]= {1, 3, System, $Failed, 4, $Failed, System}
```

При этом, для процедур/функций с неопределенным числом аргументов вызов **Arity** возвращает значение **Undefined**, на системных функциях возвращается **System**, тогда как в остальных случаях вызов **Arity[x]** возвращает значение **\$Failed**. Процедура **Arity** достаточно полезна при работе с функциональными и процедурными объектами.

В задачах формальной обработки *функциональных* выражений вполне определенный интерес представляет процедура **ExpFunc**, чей вызов **ExpFunc[G, x, y, ...]** обеспечивает расширение списка формальных аргументов функции, процедуры либо блока *G* на аргументы {*x, y, z, ...*} вправо относительно кортежа формальных аргументов объекта *G* с возвратом *обновленного* определения объекта *G* в строчном формате. Распирение кортежа формальных аргументов объекта *G* производится только на переменные из списка {*x, y, z, ...*}, не являющиеся его ни формальными аргументами, ни локальными переменными; в противном случае расширения не производится и вызов процедуры эквивалентен вызову **ExpFunc[G]**, возвращающему определение объекта *G* в *строчном* формате. Отметим, в целях недопущения инициации возможных *особых* и *ошибочных* ситуаций результат вызова процедуры **ExpFunc[G, x, y, z, ...]** возвращается в строчном формате, который пригоден для дальнейшей обработки. При этом, вызов процедуры **ExpFunc[G]** на недопустимом объекте *G*, например, пакетной функции, возвращается невычисленным. Следующий фрагмент представляет как исходный код процедуры **ExpFunc**, так и некоторые типичные примеры ее использования.

```
In[9]:= ExpFunc[f_ /; SymbolQ[f], x___ /; {x} == {} | |
      DeleteDuplicates[Map[SymbolQ, {x}]] == {True}] :=
Module[{b = ToString[f], c, d, a = ArgsLocals[f], h = {}, k = 1, p, V, H},
  If[ListQ[a], {c, d} = a, Return[Defer[ExpFunc[f, x]]]; c = Map[ToString, c];
  If[d == {}, Null, For[k, k <= Length[d], k++, p = d[[k]];
    h = Append[h, If[StringFreeQ[p, "="], p <> "_",
StringTake[p, {1, Flatten[StringPosition[p, "="][[1]] - 1]} <> "_"]]]; d = h; h = {};
  For[k = 1, k <= Length[c], k++, p = c[[k]];
    h = Append[h, StringTake[p, {1, Flatten[StringPosition[p, "_"]][[1]]}]]; c = h;
  h = Select[h = {}; For[k = 1, k <= Length[{x}], k++,
    h = Append[h, ToString[{x}][[k]]] <> "_"]; h, ! MemberQ[Join[c, d], #] &];
  H = f[Sequences[Flatten[{a[[1]]}, Sequences[Map[ToExpression, h]]]];
  h = DefFunc1[ToString[f]];
  h = ToString[H] <> StringTake[h, {Flatten[StringPosition[h, ":="][[1]], -1}]
In[10]:= Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
In[11]:= ExpFunc[Art, m, n, p, a, x, b, c, y]
Out[11]= "Art[x_, y_ /; PrimeQ[y], m_, n_, p_] := Module[{a = 2, b = 6, c = 5},
      Length[Join[x, y]]*a*b*c]"
In[12]:= Definition[Art]
Out[12]= Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
In[13]:= Map[ExpFunc, {Sin, 75, G, x + y}]
Out[13]= {ExpFunc[Sin], ExpFunc[75], ExpFunc[G], ExpFunc[x + y]}
In[14]:= ExpFunc1[f_ /; SymbolQ[f], x___ /; {x} == {} | |
      DeleteDuplicates[Map[SymbolQ, {x}]] == {True}] :=
Module[{b = ToString[f], c, d, a = ArgsLocals[f], h = {}, k = 1, p, V, H},
  If[ListQ[a], {c, d} = a, Return[Defer[ExpFunc1[f, x]]]; c = Map[ToString, c];
```

```

If[d == {}, Null, For[k, k <= Length[d], k++, p = d[[k]];
h = Append[h, If[StringFreeQ[p, " = ", p <> "_",
StringTake[p, {1, Flatten[StringPosition[p, " = "][[1]] - 1]} <> "_"]]];
d = h; h = {}; For[k = 1, k <= Length[c], k++, p = c[[k]];
h = Append[h, StringTake[p, {1, Flatten[StringPosition[p, "_"]][[1]]]];
c = h; h = Select[h = {}; For[k = 1, k <= Length[{x}], k++,
h = Append[h, ToString[{x}][[k]] <> "_"]; h, ! MemberQ[Join[c, d], #] &];
h = DefFunc1[ToString[f]];
H = f[Sequences[Flatten[{a[[1]], Sequences[Map[ToExpression, h]]}]]];
ToExpression[ToString[H] <>
StringTake[h, {Flatten[StringPosition[h, " := "][[1]], -1}]]

```

```
In[15]:= ExpFunc1[Art, m, n, p, a, x, b, c, y]
```

```
In[16]:= Definition[Art]
```

```
Out[16]= Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
```

```
Art[x_, y_ /; PrimeQ[y], m_, n_, p_] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
```

Функция **ExpFunc1**[*G*, *x*, *y*, *z*, ...] является весьма полезным расширением предыдущей процедуры **ExpFunc**, обеспечивая не только расширение списка аргументов объекта *G* на аргументы {*x*, *y*, *z*, ...} вправо относительно кортежа формальных аргументов *G*, но и вычисляет обновленное определение объекта *G*, делая его доступным в текущем сеансе. Завершает предыдущий фрагмент исходный код процедуры **ExpFunc1** наряду с примером ее применения. Из представленного фрагмента видно, что применение к процедуре **Art** рассмотренной процедуры **ExpFunc**[*Art*, *m*, *n*, *p*, *a*, *x*, *b*, *c*, *y*] обеспечивает расширение списка формальных аргументов процедуры **Art** на аргументы {*m*, *n*, *p*}, т. е. остальные переменные из кортежа {*m*, *n*, *p*, *a*, *x*, *b*, *c*, *y*} совпадают как с формальными аргументами {*x*, *y*} **Art**, так и с ее локальными переменными {*a*, *b*, *c*}. Между тем, вызов **ExpFunc**[*Art*, *m*, *n*, *p*, *a*, *x*, *b*, *c*, *y*] возвращает лишь обновленное определение процедуры **Art** в строчном формате, не влияя на список активизированных в текущем сеансе **Art**-процедур. Тогда как использование для решения такой же задачи вызова процедуры **ExpFunc1**[*Art*, *mn*, *p*, *a*, *x*, *b*, *c*, *y*] возвращает *Null*, т.е. ничего, активируя в текущем сеансе обновленное определение процедуры **Art**, что подтверждает применение **Definition**.

Между тем, вызовы процедур **ExpFunc**[*G*, *x*, *y*, *z*, ...] и **ExpFunc1**[*G*, *x*, *y*, *z*, ...] имеют смысл только на символах *G*, для которых имеют место *одновременно* соотношения **Args**[*G*] != **Undefined**, **!SysFuncQ**[*G*] и **HeadPF**[*G*] != *G*, т.е. символ *G* не определяет стандартную функцию пакета, а также переменную, для которой не определяются формальные аргументы, в данных случаях возвращая вызов неопределенным, как иллюстрируют примеры следующего фрагмента. Для расширения сферы применимости процедуры **ExpFunc1** определена процедура **ExpFunc2**, успешный вызов которой **ExpFunc2**[*F*, *Ex*] возвращает *Null*, т.е. ничего, вычисляя определение процедуры/функции *F*, которое расширено на кортеж *Ex* формальных аргументов. На стандартной функции *F* вызов процедуры возвращает ее имя, точно такой же результат имеет место в случае вызова **ExpFunc2**[*F*]. В остальных случаях вызов процедуры возвращается невычисленным.

В следующем фрагменте представлены исходный код процедуры **ExpFunc2** наряду с примерами ее использования для расширения кортежа формальных аргументов.

```
In[1205]:= {ExpFunc[Sin, x, y], ExpFunc1[Sin, x, y]}
Out[1205]= {ExpFunc[Sin, x, y], ExpFunc1[Sin, x, y]}
In[1206]:= Q = 75; {ExpFunc[Q, x, y], ExpFunc1[Q, x, y]}
Out[1206]= {ExpFunc[75, x, y], ExpFunc1[75, x, y]}
In[1207]:= Q = x1 + y1; {ExpFunc[Q, x, y], ExpFunc1[Q, x, y]}
Out[1207]= {ExpFunc[x1 + y1, x, y], ExpFunc1[x1 + y1, x, y]}
In[1269]:= ExpFunc2[F_, Expr_] := Module[{a}, If[{Expr} == {} || SysFuncQ[F],
Return[F], a = Quiet[Attributes[F]]];
If[a == {}, Null, Quiet[ClearAttributes[F, a]]];
If[Args[F] === $Failed, Quiet[SetAttributes[F, a]]; Return[Defer[ExpFunc2[F, Expr]]],
If[ToString[HeadPF[F]] == ToString[F] || Args[F] === Undefined,
ToExpression[ToString[F] <> "[" <> ListStrToStr[Map[ToString1, {Expr}]] <> ""]],
ExpFunc1[Sequences[{F, Expr}]]; Quiet[SetAttributes[F, a]]]]
In[1270]:= G := 75; {ExpFunc2[G, x1, x2, x3], ExpFunc2[G]}
Out[1270]= {ExpFunc2[75, x1, x2, x3], 75}
In[1271]:= {ExpFunc2[Q, x1, x2, x3], ExpFunc2[Q]}
Out[1271]= {Q[x1, x2, x3], Q}
In[1272]:= Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6, c = 5},
Length[Join[x, y]]*a*b*c]; SetAttributes[Art, Protected]
In[1273]:= {ExpFunc2[Art, m, n, p], ExpFunc2[Art], ExpFunc2[Sin]}
Out[1273]= {Null, Art, Sin}
In[1274]:= Definition[Art]
Out[1274]= Attributes[Art] = {Protected}
Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
Art[x_, y_ /; PrimeQ[y], m_, n_, p_] := Module[{a = 2, b = 6, c = 5}, Length[Join[x, y]]*a*b*c]
In[1275]:= Map[ExpFunc2, {Cos, Svetla, Tan, H, If, 450}]
Out[1275]= {Cos, Svetla, Tan, H, If, 450}
```

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[1437]:= 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[1438]:= 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[1440]:= Definition[G]
Out[1440]= 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[1441]:= Definition[V]
Out[1441]= V[x_ /; ListQ[x]] := Module[{}, Length[x]]
          V[x_] := Module[{}, x^2]
In[1442]:= {RemProcOnHead["G[x_, y_]"], RemProcOnHead["V[x_]"]}
Out[1442]= {"Done", "Done"}
In[1443]:= Definition[G]
Out[1443]= 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[1444]:= Definition[V]
Out[1444]= 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[1620]:= 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[1621]:= Select[Names["*"], ProcQ[ToExpression[#]] &]
Out[1621]= {"G", "V"}
In[1622]:= Definition[G]
Out[1622]= G[x_ /; PrimeQ[x]] := Module[{a = 75}, x + a]
          G[x_] := Module[{a = 75}, x^2 + a]
          G[x_, y_] := Module[{}, x + y]
          G[x_, y_ /; ListQ[y], z_] := Module[{}, x + Length[y] + z]
In[1623]:= Definition[V]
Out[1623]= V[x_ /; ListQ[x]] := Module[{}, Length[x]]

```

```

V[x_] := Module[{}, x2]
In[1624]:= MdP[] := Module[{a = Select[Select[Names["`*"], ProcQ[ToExpression[#]] &],
    Length[ProcCalls[ToExpression[#]]] > 1 &], b = {}, c, k = 1},
    For[k, k <= Length[a], k++, c = a[[k]];
    b = Append[b, {c, Length[ProcCalls[ToExpression[c]]}]]; b]
In[1625]:= MdP[]
Out[1625]= {"G", 4}, {"V", 2}

```

В частности, завершает настоящий фрагмент процедура, чей вызов **MdP[]** возвращает вложенный список, чьи 2-элементные списки имеют следующую структуру, а именно: первые элементы представляют имена в строчном формате одноименных процедур и функций с различными заголовками, тогда как вторые элементы представляют число данных объектов. Некоторые примеры подобных процедур приведены выше.

```

In[2540]:= $HeadProc := Block[{}, ToString[$ProcName] <> "[" <>
    StringTake[ToString[Args[$ProcName]], {2, -2}] <> "]"
In[2541]:= Agn[x_, y_] := Module[{a = $HeadProc},
    ToExpression[StringTake[a, {1, -2}] <> ", z_" <> "]:=Module[{}, x+2*y+z];
    ToExpression[StringTake[a, {1, -2}] <> ", z_/_; PrimeQ[z]] <> "]:=Module[{}, x+3*y+z]]
In[2542]:= Agn[42, 47]
In[2543]:= DefFunc[Agn]
Out[2543]= Agn[x_, y_] := Module[{a = $HeadProc}, ToExpression[StringJoin[StringTake[a,
    {1, -2}], ", z_", "]:=Module[{},x+2*y+z]"]; ToExpression[StringJoin[
    StringTake[a, {1, -2}], ",z_/_; PrimeQ[z]", "]:=Module[{},x+3*y+z]]]
    Agn[x_, y_, z_/_; PrimeQ[z]] := Module[{}, x + 3*y + z]
    Agn[x_, y_, z_] := Module[{}, x + 2*y + z]
In[2544]:= {Agn[42, 47, 23], Agn[42, 47, 75]}
Out[2544]= {206, 211}

```

Тогда как *вторая* часть фрагмента представляет процедурную переменную **\$HeadProc** блочного типа, которая имеет смысл только в теле процедуры, возвращая ее заголовок. Приводятся исходный код определения переменной наряду с ее применением в *теле* процедуры с целью генерации на ее основе двух различных одноименных процедур с их активацией в текущем сеансе пакета. Данная переменная имеет целый ряд весьма интересных приложений как системного, так и прикладного характера.

Тут же вполне уместно отметить следующее немаловажное обстоятельство, а именно: если в текущем сеансе активирован ряд одноименных процедур или функций, то при вызове процедуры/функции с таким именем выбирается лишь та из списка, у которой не только соответствует *заголовок*, но и кортеж фактических аргументов допустим для такого заголовка. В противном случае вызов возвращается невычисленным.

```

In[1657]:= ProcCall[] := Block[{}, ToExpression[StringReplace[ToString[DefFunc[
    $ProcName]], "ProcCall[]" -> ToString[Map[ToExpression, Args1[$ProcName]]]]]]
In[1658]:= ProcCall[]

```

```
SetDelayed::shape: Lists {} and Block[{}], ToExpression[StringReplace[ToString[DefFunc[
$ProcName]], {} -> ToString[ToExpression/@Args1[$ProcName]]]] are not the same shape.
Out[1658]= $Failed

In[1659]:= Mn[x_, y_ /; PrimeQ[y], z_, h_, g_ /; PrimeQ[g]] := Module[{args = ProcCall[]},
{args, Length[args], Quiet[Take[args, {1, Length[args]}]]}]

In[1660]:= Mn[42, 2, 47, 67, 13]
Out[1660]= {Null, 0, Take[Null, {1, 0}]}
In[1661]:= Mn[42, 2, 47, 67, 13]
Out[1661]= {{42, 2, 47, 67, 13}, 5, {42, 2, 47, 67, 13}}
In[1662]:= DefFunc[Mn]
Out[1662]= Mn[x_, y_ /; PrimeQ[y], z_, h_, g_ /; PrimeQ[g]] := Module[{args = {x, y, z, h, g}},
{args, Length[args], Quiet[Take[args, {1, Length[args]}]]}]

In[1663]:= Mn[75, 7, 420, 15, 41]
Out[1663]= {{75, 7, 420, 15, 41}, 5, {75, 7, 420, 15, 41}}
In[1664]:= Agn[] := Module[{args = ProcCall[]}, {args, Length[args]}]
In[1665]:= Agn[]
Out[1665]= {Null, 0}
In[1666]:= Agn[]
Out[1666]= {}, 0}
In[1667]:= Avz[x_] := Module[{args = ProcCall[]}, {args, Length[args]}]
In[1668]:= Avz[75]
Out[1668]= {Null, 0}
In[1669]:= Avz[69, 64, 44, 15, 22, 6]
Out[1669]= {{69, 64, 44, 15, 22, 6}, 6}
```

Как хорошо известно [99], пакет *Maple* располагает рядом процедурных переменных, обеспечивающих, в частности, возможность получать список переданных процедуре при вызове фактических аргументов. В *Mathematica* подобные средства отсутствуют, хотя в целом ряде случаев представляют вполне определенный интерес. Некоторые средства подобного типа для *Mathematica* представлены выше. В качестве еще одного довольно полезного средства имеет смысл представить также блочную конструкцию **ProcCall[]**, которая имеет смысл лишь в теле процедуры, обеспечивая возврат списка получаемых ею *фактических* аргументов. При этом, *первый* вызов процедуры, которая содержит в определении вызов **ProcCall[]**, носит в определенном контексте характер *инициализационного*, в то время как все последующие в текущем сеансе обеспечивают корректное выполнение основной процедуры. Предыдущий фрагмент представляет *исходный* код процедуры **ProcCall** наряду с примерами ее применения. При этом, для получения в процедуре списка фактических аргументов в точке ее вызова возможно обойтись и без **ProcCall**, для чего в теле процедуры можно использовать конструкцию формата {последовательность формальных аргументов без приписанных им шаблонов}, как было показано выше, или использовать стандартную функцию **With**, как следующий достаточно простой фрагмент весьма наглядно иллюстрирует, а именно:

```

In[664]:= Mn[x_, y_ /; PrimeQ[y], z_, h_, g_ /; PrimeQ[g]] := With[{Fargs = {x, y, z, h, g}},
Module[{args = Fargs}, {args, Length[args]}]]
In[665]:= Mn[42, 2, 47, 67, 13]
Out[665]= {{42, 2, 47, 67, 13}, 5}
> Agn := subs(Res = [x, y, z, h], proc(x, y, z, h) local Args; Args := Res; Args end proc):
> Agn(42, 47, 67, 75);    => [42, 47, 67, 75]
> Agn := proc(x, y, z, h) local Args, t; Args := convert('procname(args)', string);
search(Args, ("", t), t; Args := parse(cat("[", Args[t + 1..-2], "]"))); Args end proc:
> Agn(42, 47, 67, 75);    => [42, 47, 67, 75]

```

Между тем, в обоих указанных случаях мы априори должны знать список *формальных* аргументов до конкретного вызова процедуры, тогда как вызов **ProcCall** обеспечивает вычисление такого списка уже при вызове процедуры, что в ряде случаев оказывается более предпочтительным. Таким образом, пакет *Mathematica* располагает достаточно развитыми встроенными средствами для решения задач пользователя, расширяющих встроенный *Math*-язык программирования пакета. Тогда как вторая часть фрагмента представляет как аналог функции **With** в пакете *Maple*, так и получение фактических аргументов в теле процедуры без стандартных процедурных переменных. Несложно заметить, что средства *Maple* более развиты, чем подобные средства *Mathematica*, что в ряде случаев довольно существенно упрощает процедурное программирование.

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 *Module*-типа. Такую задачу успешно решает процедура **SubProcs[P]**, вызов которой возвращает *вложенный* 2-элементный список *ListList*-типа, чей *первый* элемент представляет *список заголовков* подпроцедур, составляющих *главную* процедуру *P* типа *Module*, в то время как второй элемент определяет *список* сгенерированных имен *всех* подпроцедур, включая *главную* процедуру *P*, и активизированных в текущем сеансе пакета процедур.

```

In[76]:= SubProcs[P_ /; ProcQ[P]] := Module[{a = DefFunc1[ToString[P]], b, c = {}, d, t, h,
k = 1, p = {}, g = {}}, b = StringPosition[a, "]" := 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 = 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[77]:= P[x_, y_] := Module[{a, b, P1, P2}, P1[z_, h_] := Module[{m, n}, z^2 + h^2];
      P2[z_] := Module[{P3}, P3[h_] := Module[{}, h^4]; P3[z]]; x*P2[x] + P1[x, y] + P2[y]]
In[78]:= P[75, 450]
Out[78]= 43 379 505 000
In[79]:= SubProcs[P]
Out[79]= {{P[x_, y_]", "P1[z_, h_]", "P2[z_]", "P3[h_]", {P$3788, P1$3793, P2$3799, P3$3804}}
In[80]:= DefFunc[P2$3799]
Out[80]= P2$3799[z_] := Module[{P3}, P3[h_] := Module[{}, h^4]; P3[z]]
```

Итак, между элементами подписков возвращаемого вложенного списка имеет место взаимно-однозначное соответствие. В представленном выше фрагменте представлен исходный код процедуры **SubProcs** и достаточно типичные примеры ее применения. Процедура допускает целый ряд достаточно интересных расширений.

Здесь же следует отметить одно немаловажное обстоятельство, а именно. Если в *Maple* подпроцедура в теле главной процедуры идентифицируется как *процедурный* объект, то в *Mathematica* подпроцедура идентифицируется, например, процедурой **ProcQ** в качестве такого объекта лишь при условии, что по отношению к *главной* процедуре она является *глобальной*, как весьма наглядно иллюстрирует следующий фрагмент.

```
In[1864]:= P[x_, y_] := Module[{a, b, c, F}, F[z_, h_] := Module[{a, b, c}, z^2 + h^2];
      If[PrimeQ[x], x*y + F[x, y], ProcQ[F]]]
In[1865]:= {P[75, 450], P[7, 450]}
Out[1865]= {False, 205 699}
In[1866]:= Map[ProcQ, {P, F}]
Out[1866]= {True, False}
In[1867]:= P[x_, y_] := Module[{a, b, c}, F[z_, h_] := Module[{a, b, c}, z^2 + h^2];
      If[PrimeQ[x], x*y + F[x, y], ProcQ[F]]]
In[1868]:= {P[75, 450], P[7, 450]}
Out[1868]= {True, 205 699}
In[1869]:= Map[ProcQ, {P, F}]
Out[1869]= {True, True}
```

Но в данном случае ее определение не имеет особого смысла включать в тело главной процедуры. Именно это обстоятельство в определенной мере усложнило реализацию процедуры **SubProcs** и может рассматриваться в качестве определенного ограничения при процедурном программировании в программной среде пакета *Mathematica*.

В ряде приложений возникает необходимость определения у процедуры *подпроцедур*. Вызов процедуры **SubsProcQ[x, y]** возвращает **True**, если *y* является подпроцедурой *x* процедуры, и **False** в противном случае. Но так как *Math*-процедуры различаются не по именам, как в *Maple*, а заголовками, то через 3-й необязательный аргумент вызов процедуры возвращает *вложенный* список, чьи подписки *первым* элементом содержат *заголовок* процедуры с именем *x*, но разными заголовками, *вторым* – соответствующие им *заголовки* подпроцедур с именем *y*. Следующий фрагмент представляет исходный текст с примером применения. В принципе, на основе представленных выше средств программируется целый ряд достаточно полезных средств работы с процедурами.

```
In[1929]:= SubsProcQ[x_] := Module[{a = {x}, b = Length[{x}], c, d, k = 1, j = 1, Res = {}},
  If[b >= 2 && ProcQ[a[[1]]] && ProcQ[a[[2]]],
    {c, d} = {StringSplit[ToString[InputForm[DefFunc[a[[1]]]]], "\n\n"],
    StringSplit[ToString[InputForm[DefFunc[a[[2]]]]], "\n\n"]},
    Defer[SubsProcQ[x]]; For[k, k <= Length[d], k++,
  For[j, j <= Length[c], j++, If[! StringFreeQ[c[[j]], d[[k]]],
    Res = Append[Res, {StringTake[c[[j]],
    {1, Flatten[StringPosition[c[[j]], " := "][[1]] - 1}], StringTake[d[[k]],
    {1, Flatten[StringPosition[d[[k]], " := "][[1]] - 1}]], Continue[]];
  If[b > 2 && ! HowAct[a[[3]]], Quiet[ToExpression[ToString[a[[3]]] <>
    " = " <> ToString[Res]], Null]; If[Res == {}, False, True]]

In[1930]:= {SubsProcQ[P, P3, x], x}
Out[1930]= {True, {"P[x_, y_]", "P3[h_]"}]}
```

Данная и ряд ранее представленных процедур в целом ряде случаев могут оказаться достаточно эффективными средствами в программировании процедур, в некоторой степени расширяя элементы процедурного программирования. В данном контексте следует отметить, что функциональные средства пакета *Mathematica* предоставляют возможность расширять элементы *процедурного программирования* до уровня, не ниже, чем обеспечивает *процедурное* программирование пакета *Maple*. Однако, обеспечение подобного *паритета* требует создания дополнительных средств общего назначения, ряд из которых проиллюстрирован в этой книге. Итак, из более детального *сравнения* процедурных объектов в *Maple* и *Mathematica* определенное предпочтение отдается нами первому пакету, сам *Maple*-язык которого ориентирован именно на *процедурное* программирование. Стандартные средства организации и работы с процедурными объектами в пакете *Maple* существенно более развиты относительно второго пакета и предоставляют возможность создавать достаточно качественные процедуры не только прикладного, но и системного характера, что является весьма важным показателем. На этом уже вполне можно завершить примеры организации достаточно прозрачных и небольших по размеру процедур в среде пакета *Mathematica*, которые дают вполне адекватное, на наш взгляд, представление по организации объектов подобного типа. Между тем, следует иметь в виду, что многие из представленных здесь средств, в свою очередь, содержат созданные нами средства, большая часть из которых описана здесь же, тогда как в полном объеме средства представлены в небольшом пакете [90]. Тогда как список всех процедур и функций данного пакета, загруженного в текущий сеанс, наряду с другими активными объектами указанных типов можно получать по вызову **ProcsAct[]** довольно простой процедуры с исходным кодом, представленным ниже:

```
In[995]:= ProcsAct[] := Module[{a = Names["*"], b = Names["System`*"], c, d = {}, k = 1, j, h,
  x = t, g = {"Module", "Block", "DynamicModule", "Others"}},
  c = Select[Select[a, ! MemberQ[b, #] &], Quiet[ProcQ[ToExpression[#]]] &];
  For[k, k <= Length[c], k++, h = c[[k]]; Clear[t];
  Quiet[ProcQ1[ToString[h], t]];
  d = Append[d, {ToExpression[h], t}]; t = x;
```

```

For[k = 1, k <= Length[d], k++, For[j = 1, j <= Length[g], j++,
  If[d[[k]][[2]] == g[[j]][[1]], g[[j]] = Append[g[[j]], d[[k]][[1]], Continue[]]; g]
In[996]:= ProcsAct[]
Out[996]= {"Module", ActiveProcess, ActRemObj, Adrive, Affiliate, AllMatrices, Aobj,
  Args, Args0, Args1, Args2, ArgsLocals, ArgsProc, Arity, ArrayInd, AssignToList,
  AtomicQ, Attrib, AutoLoadPack, Avg, BitGet1, BlockQ, CALL, CallsInProc,
  CatN, CDFOpen, Cdir, Contexts1, Decomp, Defaults, DefFunc, DefFunc1,
  DefFunc2, DefFunc3, DelEl, DelSubStr, Df, Df1, Df2, DO, DuplicateLocalsQ,
  Email, Email1, ExpFunc, ExpFunc1, ExpLocals, ExprOfStr, ExtrCall, ExtrExpr,
  ExtrPackName, FileFormat1, FileOpenQ, FileQ, FreeSpaceVol, FunCompose,
  FunctionQ, FunctionQ1, Gather1, Gather2, Globals, Globals1, GotoLabel,
  HeadingQ, HeadPF, Help, Iff, Ifk, Ifk1, Ind, Index, IndexedQ, InsertN, Int, Int1,
  IsFileOpen, LeftFold, ListAssign, ListListGroup, ListOp, ListStrToStr,
  ListToString, LoadFile, LoadNameFromM, Locals, Locals1,
  LongestCommonSubSequence, LongestCommonSubString, Map1, Map2, Map3,
  Map4, Map5, Map6, MapInSitu, MapInSitu1, MapInSitu2, Mapp,
  MaximalPalindromicSubstring, MaxNestLevel, MdP, MemberLN, MemberQ1,
  MemberQ2, MinusList, MixCaseQ, ModLibraryPath, ModuleQ, NamesCS,
  NamesNbPackage, NamesNbPackage1, NamesProc, NbName, Need, NestCycles,
  Nobj, Nproc, Nvalue, ObjType, Op, OP, OpenFiles, OverLap, PackNames,
  PackPaclet, PacletInformation, PartialSums, PredecessorsL, PredecessorsR,
  ProcCalls, ProcQ, ProcQ1, ProcsAct, ProtectedQ1, Qfunction, Range1, Range2,
  Range3, RemProcOnHead, Rename, ReplaceAll1, RhsLhs, RightFold,
  ScanLikeProcs, SearchDir, SearchFile, Seq, SeqDel, SeqIns, SequenceQ, SetDir,
  SortNL, Spos, StrDelEnds, StrExprQ, StringEnd, StringMultiple, StringMultipleD,
  StringPosition1, StringReplace1, StringTake1, StringTake2, StrOfSymbIQ,
  SubDelStr, SubProcs, SubsDel, SubsProcQ, SubStr, SuffPref, Tbl, TestArgsTypes,
  TestProcCalls, ToList, ToServiceRequest, ToString1, Try, Tuples1, TwoHandQ,
  Type, TypeActObj, Un, UpdateContextPaths, UpdatePackages, UpdatePath,
  UprocQ, Uprocs, UserLib, VarExch, VarExch1, WhatObj, WhatType, WhichN},
{"Block", IsMonotonic, IsPermutation, Prev, ProcCall, SeqQ, Ver},
{"DynamicModule"},
{"Others", Bits, DeCod, InstallServiceOperation, PacletCheckUpdate,
  PacletDirectoryAdd, PacletDirectoryRemove, PacletEnable, PacletFind,
  PacletFindRemote, PacletInstall, PacletSetLoading, Subs}}
In[1012]:= Agn[x_] := Module[{a = 75, b}, Clear[a]; Map[HowAct, {a, b, y, x}]]
In[1013]:= Agn[64]
Out[1013]= {True, True, False, True}

```

Вызов процедуры **ProcsAct[]** возвращает вложенный 4-элементный список, подписки которого своим первым элементом определяют *типы* объектов в разрезах **"Module"**, **"Block"**, **"DynamicModule"** и **"Others"**, активизированных в текущей сессии, тогда как остальные элементы определяют *имена* объектов соответствующего *первому* элементу

типа. Еще раз следует отметить, для *Mathematica*-процедуры локальные переменные изначально не рассматриваются как неопределенные; более того, придать им статус *неопределенных* в теле процедуры не представляется возможным, что и иллюстрирует второй пример фрагмента. Именно поэтому в *Mathematica*-процедуре невозможно использовать *локальные* переменные в качестве *формальных* аргументов подпроцедур, через которые возвращаются дополнительные результаты. Для данной цели следует использовать *глобальные* переменные, обеспечив в основной процедуре сохранение их значений, очистку от значений и восстановление их значений перед (*либо в его точке*) каждым возможным выходом из основной процедуры, как показано, в частности, и в процедуре **ProcsAct**, и в ряде других, рассмотренных в настоящей книге. Процедура **ProcsAct[]** представляет вполне определенный интерес в целом ряде приложений и, прежде всего, в задачах системного характера.

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  $\{Module | Block | DynamicModule\}$  процедуры *x*. Следующий фрагмент представляет исходный код процедуры **UprocQ** с результатами ее применения к процедурам разных типов.

```
In[9]:= UprocQ[x_ /; ProcQ[x]] := Module[{a = ToString1[DefFunc1[ToString[x]]], b, d, m, r},
      {m, b, d} = Map[Flatten, {StringPosition[a, {" := Module[" := Module["],
      StringPosition[a, {" := Block[" := Block["],
      StringPosition[a, {" := DynamicModule[" := DynamicModule["]"];
      r = Quiet[Sort[{m, b, d}, # != {} && #1[[1]] < #2[[1]] &]];
      If[m == r[[1]], {True, Module}, If[b == r[[1]], {True, Block},
      If[d == r[[1]], {True, DynamicModule}, $Failed]]]]
In[10]:= 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[11]:= Kr[x_] := Block[{y = a, h = b}, (y^2 + h^2)*x]; Art[x_] := Module[{a}, a = 69; x + a]
In[12]:= Sv[y_] := DynamicModule[{x = y}, {Slider[Dynamic[x]], Dynamic[x]}]
In[13]:= Map[UprocQ, {Agn, Kr, Art, Sv, ProcQ, Subs, Sin, 2012, ProcsAct}]
Out[13]= {{True, Module}, {True, Block}, {True, Module}, {True, DynamicModule},
      {True, Module}, {True, Module}, False, False, {True, Module}}
```

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 (*процедурный* | *блочный*), тогда как во втором случае элементами списка, возвращаемого процедурой **Uprocs**, являются 3-элементные *подписки*, первые элементы которых определяют *имена* пользовательских процедур, активизированных в текущем сеансе, вторые определяют формат вызова процедур в строчном формате, третьи – *тип* процедуры  $\{Block | Module\}$ . Следующий фрагмент иллюстрирует текст

самой процедуры **Uprocs** и примеры ее применения, при этом приводятся 2 варианта реализации процедуры, определяющие форму возвращаемого списка, а именно:

```
In[501]:= Gs[x_] := Block[{a, b, c}, Evaluate[(a*x + x^b)/c]]
In[502]:= S[x_] := Block[{y = a, h = b}, G[Pi/2, y*x]]
In[503]:= S[x_] := Module[{y = a, h = b}, G[Pi/2, y*x]]
In[504]:= S[x_, y_] := Block[{z = a, h = b}, G[Pi/2, (y*x)/z]]
In[505]:= Bl[y_] := Block[{h = z}, G[Pi/2, y]]
In[506]:= MM[x_, y_] := Module[{}, x + y]
In[507]:= ProcQ[x_] := Module[{a = DefFunc1[ToString[x]], b, c, d, h},
    b = Quiet[StringTake[a, {1, First[First[StringPosition[a, {" := Block[{"",
        " :=Block[{"}] - 1]]}]]];
    c = Quiet[StringTake[a, {1, First[First[StringPosition[a, {" := Module[{"",
        " :=Module[{"}] - 1]]}]]];
    d = Quiet[StringTake[a, {1, First[First[StringPosition[a, {" := DynamicModule[{"",
        " := DynamicModule[{"}] - 1]]}]]];
    If[b === ToString[HeadPF[x]], True, If[c === ToString[HeadPF[x]], True,
    If[d === ToString[HeadPF[x]], True, False]]]]
In[508]:= Map[ProcQ, {Gs, S, Bl, ProcQ, BlockQ, Sv, Agn, 2012, "450", V70, Sin, Subs}]
Out[508]= {True, True, True, True, True, True, True, False, False, False, False, True}
In[509]:= Uprocs[] := Module[{a, b, c, d, h, g, k}, a := "_$Art23_Kr15$.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]]];
    If[ProcQ[h], g = Append[g, h], Null]; k = k + 1]; DeleteFile[a]; Return[Flatten[g]]
In[510]:= Uprocs[]
Out[510]= {Agn, Args, S, Art, Bl, Gs, Kr, MM, Sv, Avz, Vsv}
In[511]:= ProcName[] := Module[{}, Names["``*" <> ToString[$ModuleNumber]]]
In[512]:= Uprocs[] := Module[{a, b, c, d, h, g, k, t1, t2}, a := "_$Art23_Kr15$.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[ProcQ[h], g = Append[g, {h, t2, t1}], Null]; k = k + 1]; DeleteFile[a]; Return[g]]
```

```
In[513]:= Uprocs[]
```

```
Out[513]= {{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 пакета процедура **Uprocs** весьма существенно использует и нашу процедуру **ProcQ**, уже рассматриваемую в книге, которая служит для тестирования объектов пакета на предмет их процедурности в ранее отмеченном смысле. Тогда как в общем понимании модули и блоки пакета 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. Более того, представленная здесь процедура **ProcQ** является обобщенной модификацией ранее представленной процедуры **ProcQ**, чей вызов **ProcQ[x]** возвращает **True**, если *x* является процедурой типа *Block, Module, DynamicModule*, и **False** в противном случае.

Следующий фрагмент представляет достаточно полезную процедуру, вызов которой **NamesProc[]** возвращает отсортированный список *имен* пользовательских процедур, активированных в текущем сеансе. Идентична ей и функция, чей вызов **NamesProc1[]** тоже возвращает отсортированный список *имен* пользовательских процедур, которые активизированы в текущем сеансе. В целом ряде случаев процедура/функция может оказаться довольно полезным средством. Фрагмент представляет исходные коды для обоих рассмотренных средств наряду с типичными примерами их использования.

```
In[1407]:= NamesProc[] := Module[{a, b, c, d, k, h}, {a, b, d} = {Names["*"], {}, {}};
For[k = 0, k <= Length[a], k++,
h = Quiet[Check[ToExpression[First[Take[a, {k, k}]]], Null]];
If[h === Null, Null, b = Append[b, h]]];
b = DeleteDuplicates[Flatten[b]]; For[k = 0, k <= Length[b], k++,
If[Quiet[ProcQ[First[Take[b, {k, k}]]],
d = Append[d, First[Take[b, {k, k}]]], Null]]; Sort[d]]
In[1408]:= NamesProc1[] := Select[Map[ToExpression, Sort[Names["*"]]], ProcQ[#] &]
In[1409]:= NamesProc[]
Out[1409]= {Df, Dff, NamesProc, ProcQ, SortNL, Spos, Subs, Uprocs}
In[1410]:= NamesProc1[]
Out[1410]= {Df, Dff, NamesProc, ProcQ, SortNL, Spos, Subs, Uprocs}
```

При реализации **&**-функции **NamesProc1[]** существенно использовалась и процедура **ProcQ**, некоторые варианты реализации которой рассматривались несколько выше. В определенной степени представляет интерес и 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 блочной конструкции, с примерами ее использования.

```
In[488]:= $ProcName := Block[{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[489]:= Avz[x_, y_, z_] := Module[{$$NameProc$$ = "Avz", b}, b = $ProcName; x+y+z; b]
In[490]:= Agn[x_, y_, z_] := Module[{b, $$NameProc$$ = "Agn"}, x+y+z; b = $ProcName; b]
In[491]:= Ian[x_, y_, z_] := Module[{b, c, h}, x+y+z; b = $ProcName; b]
In[492]:= Agn[44, 64, 69]
Out[492]= {Agn, "Agn[x_, y_, z_]"}
In[493]:= Avz[44, 64, 69]
Out[493]= {Avz, "Avz[x_, y_, z_]"}
In[494]:= Ian[44, 64, 69]
Out[494]= "UndefinedName"
```

Вызов **ExtrCall[N, P]** простой процедуры возвращает **True**, если процедура **P** содержит вызовы процедуры/функции **N**, и **False** в противном случае. Если же вызов в качестве аргумента **N** определяет список процедур/функций, то возвращается подсписок **N** из входящих в процедуру **P** вызовов процедур/функций из списка **N**. Процедура имеет целый ряд полезных приложений, прежде всего, системного характера [99].

```
In[844]:= ExtrCall[N_, P_] := Module[{a = Join[CharacterRange["A", "Z"],
      CharacterRange["a", "z"]], b, x},
      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[845]:= Map3[ExtrCall, Run, {ActiveProcess, Attrib, SearchDir, SearchFile, Df, Uprocs}]
Out[845]= {True, True, True, True, False, False}
In[846]:= ExtrCall[{Run, Write, Read, If, Return}, Attrib]
Out[846]= {Run, Read, If, Return}
```

Дополнительно к ранее представленной процедуре **TestArgsTypes**, обеспечивающей *вызов* указанной процедуры таким образом, что возвращает результат данного вызова процедуры в отсутствие *недопустимых* фактических аргументов или список, который состоит из значений **{True, False}**, порядком соответствующих порядку фактических аргументов при вызове тестируемой процедуры, интерес представляет и процедура, чей вызов **TestProcCalls[P, L]** возвращает **True**, если кортеж фактических аргументов, определенных списком **L**, является допустимым для вызова процедуры **P**, и **\$Failed** в противном случае. При этом полагается, что процедура **P** определяет фиксированное число фактических аргументов. В случае возврата **\$Failed** через 3-й необязательный аргумент возвращается вложенный список, чьи 2-элементные подсписки определяют *первым* элементом формальный аргумент без условия в строчном формате, тогда как *вторым* элементом – значение **{True, False}** в зависимости от допустимости аргумента, полученного им при вызове анализируемой процедуры **P**. В следующем фрагменте представлен исходный код процедуры наряду с примерами ее применения.

```
In[2092]:= TestProcCalls[P_;/; ProcQ[P], L_;/; ListQ[L], x___] :=
Module[{a, b = {}, c = {}, d, p, k = 1}, a = Map[ToString1, Args[P]];
If[Length[L] != Length[a], Return[$Failed],
a = ToExpression[StringReplace[ToString1[a], {"_" -> ", ", "/" -> ""}]]];
For[k, k <= Length[a], k++, p = a[[k]]; If[StringFreeQ[p, ","],
c = Append[c, True], d = Flatten[StringPosition[p, ","]][[1]];
c = Append[c, {StringTake[p, {1, d - 1}], StringTake[p, {d + 2, -1}]}]];
If[DeleteDuplicates[c] === {True}, Return[True], d = {}];
For[k = 1, k <= Length[c], k++, If[c[[k]] === True, d = Append[d, True],
d = Append[d, ToExpression[c[[k]][[1]] <> "=" <> ToString1[L[[k]]] <> "; " <>
c[[k]][[2]]]]]; p = {}; For[k = 1, k <= Length[c], k++, p = Append[p, c[[k]][[1]]];
Print["The procedure call updated the global variables " <> ToString[p]];
{b, p} = {{P, L, x}, {}}; d = Map[ToExpression, d];
If[DeleteDuplicates[d] != {True}, For[k = 1, k <= Length[c], k++,
p = Append[p, {c[[k]][[1]], d[[k]]}]; If[Length[b] > 2 && ! HowAct[b[[3]]],
ToExpression[ToString[b[[3]]] <> "=" <> ToString1[p], Null]; $Failed, True]]

In[2093]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_75_450", "75_450", 2}]
The procedure call updated the global variables {S, s, n}.
Out[2093]= True
In[2094]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_75_450", "75_450", 6}]
The procedure call updated the global variables {S, s, n}.
Out[2094]= $Failed
In[2095]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_75_450", "75_450", 6}, Art23]
The procedure call updated the global variables {S, s, n}.
Out[2095]= $Failed
In[2096]:= Art23
Out[2096]= {"S", True}, {"s", True}, {"n", False}
```

Относительно реализованного процедурой **TestProcCalls** алгоритма следует сделать довольно существенное замечание. Пакет *Mathematica* не располагает стандартными средствами обеспечения корректной реализации конструкций следующего вида:

```
In[649]:= ToExpression[StringReplace[ToString[Test[T]], ToString[T] -> ToString[Expr]]]
Out[649]= Expresst[Expr]
In[650]:= ToExpression["Replace[" <> ToString[Test[T]] <> "," <> ToString[T] <> "->" <>
                                                    ToString[Expr] <> "]" ]
Out[650]= Test[T]
```

которые наиболее естественны при реализации алгоритма тестирования *фактических* аргументов, получаемых процедурой при вызове, на предмет допустимости условию **Test[T]**. И если в первом случае в общем случае имеется явное искажение ожидаемого результата подстановки, то во втором случае подстановка производится только после вычисления **Test[T]**, как правило, возвращающего **False**. Поэтому для устранения этой ситуации был использован простой прием (*несложно усматриваемый из представленного кода процедуры*), который, однако, переопределяет глобальные переменные текущего сеанса, одноименные с формальными аргументами тестируемой процедуры **P**, о чем и печатается соответствующее сообщение. Однако, для робастного программирования данный подход не представляется, вообще говоря, приемлемым. Поэтому для замены вышеупомянутых двух конструкций использовалась наша процедура **StringReplace2**, рассматриваемая в рамке средств работы со строками. С учетом использования этой процедуры код модифицированной процедуры **TestProcCalls** принимает вид:

```
In[2126]:= TestProcCalls[P_/, ProcQ[P], L_/, ListQ[L]] :=
Module[{a, b = {}, c = {}, d, p, k = 1}, a = Map[ToString1, Args[P]];
If[Length[L] != Length[a], Return[$Failed],
a = ToExpression[StringReplace[ToString1[a], {"_ " -> ", ", "/" -> ""}]]];
For[k, k <= Length[a], k++, p = a[[k]]; If[StringFreeQ[p, ","],
c = Append[c, True], d = Flatten[StringPosition[p, ", "]]][1];
c = Append[c, {StringTake[p, {1, d - 1}], StringTake[p, {d + 2, -1}]}];
If[DeleteDuplicates[c] == {True}, Return[True], d = {}];
For[k = 1, k <= Length[c], k++, If[c[[k]] == True, d = Append[d, True],
d = Append[d, StringReplace2[c[[k]]][2], c[[k]][1], ToString1[L[[k]]]]];
{p, d} = {}, Map[ToExpression, d];
If[DeleteDuplicates[d] != {True}, For[k = 1, k <= Length[c], k++,
p = Append[p, {c[[k]][1], d[[k]]}]; {$Failed, p}, True]]

In[2127]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_75_450", "75_450", 2}]
Out[2127]= True
In[2128]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_75_450", "75_450", 6}]
Out[2128]= {$Failed, {"S", True}, {"s", True}, {"n", False}}
In[2129]:= Asv[x_/, PrimeQ[x]] := Module[{}, x]
In[2130]:= TestProcCalls[Asv, {71}]
Out[2130]= True
```

```
In[2131]:= TestProcCalls[Asv, {75}]
Out[2131]= {$Failed, {"x", False}}
```

Вызов модифицированной процедуры **TestProcCalls**[*P*, *L*] возвращает значение **True**, если кортеж *фактических* аргументов, определенных списком *L*, является *допустимым* при вызове процедуры *P*, и вложенный список в противном случае. При этом, *первым* элементом в данном списке выступает значение **\$Failed**, тогда как в качестве второго элемента в списке выступает вложенный список, чьи 2-элементные подсписки *первым* элементом определяют *формальный* аргумент без условия в строчном формате, тогда как *вторым* элементом – значение {**True**, **False**} в зависимости от того, какое *выражение* является допустимым и полученным в качестве фактического аргумента при вызове анализируемой процедуры *P*. В предыдущем фрагменте представлен исходный код процедуры наряду с примерами ее применения. Процедура допускает ряд довольно полезных для процедурного программирования модификаций.

Во многих случаях актуально знание о доступности в текущем сеансе пакета того или иного объекта (*переменной, процедуры и др.*). В этом плане функция **HowAct**[*Q*] может оказаться намного полезней и эффективнее стандартной функции **ValueQ**[*x*] пакета. Вызов **HowAct**[*Q*] возвращает **True** на активизированном в текущем сеансе объекте *Q*, включая конкретные значения любого допустимого типа, и **False** в противном случае. Следующий фрагмент представляет исходный код функции **HowAct** с примерами ее применения, из которых с очевидностью следуют важные отличия обеих функций.

```
In[1325]:= HowAct[Q_] := If[Quiet[Check[ToString[Definition[Q]], True]] ==
                                     "Null", False, True]
In[1326]:= x = 6; {Map[ValueQ, {ProcQ, Sin, 75, x}], Map[HowAct, {ProcQ, Sin, 75, x}]}
Out[1326]= {{False, False, False, False}, {True, True, True, True}}
```

Данная функция используется нами достаточно широко для *проверки* переменных на *неопределенность* и, прежде всего, при обеспечении возврата процедурой результатов через ее формальные аргументы как обязательные, так и необязательные.

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 значений у них. С другой стороны, на *неопределенных* локальных переменных блоков **HowAct** возвращает **False**, как это также весьма наглядно иллюстрирует следующий простой фрагмент:

```
In[587]:= A[x_] := Module[{a, b = 75}, {HowAct[a], HowAct[b], SymbolQ[a], SymbolQ[b]}]
In[588]:= A[75420]
Out[588]= {True, True, True, False}
In[589]:= Map[HowAct, {69, IAN, "RANS", Cos, Args, TestArgsTypes, Label, HowAct}]
Out[589]= {True, True, True, True, True, True, True, True}
In[590]:= Map[SymbolQ, {69, IAN, "RANS", Cos, Args, TestArgsTypes, Label, HowAct}]
Out[590]= {False, True, False, True, True, True, True, True}
```

```

In[591]:= Avz[x_] := Module[{a, b = 75}, {SymbolQ[a], SymbolQ[b]}]
In[592]:= Avz[75420]
Out[592]= {True, False}
In[1097]:= {SymbolQ[Sin], HowAct[Sin], ValueQ[Sin]}
Out[1097]= {True, True, False}
In[1182]:= Agn[x_;/; StringQ[x]] := Module[{a}, FileExistsQ1[x, a]; a]
In[1183]:= Agn["BirthDay.html"]
Out[1183]= a$551852
In[1184]:= Av := Block[{a}, HowAct[a]]
In[1185]:= Av
Out[1185]= False
In[1186]:= Ag := Block[{a = 450}, HowAct[a]]
In[1187]:= Ag
Out[1187]= True
In[1188]:= HowAct[x_] := If[Quiet[Check[ToString[Definition[x]], True]] === Null, False,
    If[Quiet[ToString[Definition[P]]] === "Attributes[" <>
        ToString[x] <> "] = {Temporary}", False, True]]
In[1189]:= Agn["BirthDay.html"]
Out[1189]= {"C:\\Temp\\Birthday.html", "G:\\Rasnoe\\Birthday.html"}

```

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 последнюю модификацию [90]. В ряде представленных здесь процедур использование первоначальной версии **HowAct** обусловило применение в их телах вызовов процедур и функций, чьи определения в качестве формальных используют типированные **HowAct** аргументы, использовать в качестве их *глобальные* переменные, тогда как вторая версия **HowAct** вполне позволяет использовать в качестве таких аргументов локальные переменные главных процедур. Заинтересованный читатель в качестве весьма полезного практического упражнения может провести соответствующие модификации такого типа процедур подобно тому, как это проиллюстрировано примером процедуры **Agn** предыдущего фрагмента, и в процедурах и функциях из нашего небольшого пакета **AVZ\_Package** [90].

Стандартной реакцией на получение процедурой/функцией недопустимого кортежа при ее вызове является возврат вызова невычисленным, за исключением стандартных *упрощений* 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 значение **"ErrorInNumArgs"**.

```
In[1046]:= 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[1047]:= {UnevaluatedQ[F, x, y, z], UnevaluatedQ[Sin, x, y, z]}
Out[1047]= {True, "ErrorInArgs"}
In[1048]:= {UnevaluatedQ[Sin, 75], UnevaluatedQ[Sin, 450.75], UnevaluatedQ[Sin]}
Out[1048]= {True, False, "ErrorInNumArgs"}
```

Предыдущий фрагмент представляет *исходный* код процедуры **UnevaluatedQ** наряду с примерами ее использования. Процедура представляет определенный интерес при программировании обработки результатов вызовов процедур и функций.

В целом ряде задач возникает необходимость тестирования *объектов* на предмет быть стандартными функциями пакета *Mathematica*. Процедура **SysFuncQ** решает задачу; вызов процедуры **SysFuncQ[F]** возвращает **True**, если *F*-объект является стандартной функцией пакета *Mathematica*, и **False** в противном случае. В следующем фрагменте представлены исходный код процедуры наряду с примерами ее использования.

```
In[819]:= SysFuncQ[F_;/; SymbolQ[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[820]:= Map[SysFuncQ, {Sin, Tan, While, If, Do, ProcQ, 75, Length, a + b, Art}]
Out[820]= {True, True, True, True, True, False, False, True, False, False}
In[1037]:= SysFuncQ1[x_] := MemberQ[Names["System`*"], ToString[x]]
In[1038]:= Map[SysFuncQ, {Sin, Tan, While, If, Do, ProcQ, 75, Length, a + b, Art}]
Out[1038]= {True, True, True, True, True, False, False, True, False, False}
```

Тогда как достаточно простая функция **SysFuncQ1** – функционально эквивалентная модификация предыдущей процедуры **SysFuncQ**.

Довольно простой прием, проиллюстрированный ниже, дает возможность работать с различными фактическими аргументами при вызове процедуры *F* с неопределенным количеством формальных аргументов, довольно естественно имитируя процедурные переменные *args*, *args[n]*, *args[n..m]* и *nargs* программной среды пакета *Maple*.

```
In[1429]:= F[x_] := Module[{args, nargs}, args = {x}; nargs = Length[args];
StringLength[args[[1]]] + args[[2]]*args[[3]] + (args[[4]] + args[[5]])/N[args[[6]]]
In[1430]:= F["Agn", Sqrt[x], (a + b), 68, 420, Log[72]]
Out[1430]= 117.108 + (a + b) Sqrt[x]
In[1430]:= Avz[x_] := Module[{}, {Nproc[[1]], {x}}]
In[1430]:= Avz[42, 69, 47, 66, 67, 44]
Out[1430]= {"Avz", {42, 69, 47, 66, 67, 44}}
```

Тогда как использование процедуры **Nproc**, рассмотренной выше, дает возможность реализовать довольно полезный механизм, аналогичный *procname*-механизму пакета *Maple*. Один из подобных подходов довольно наглядно иллюстрирует процедура *Avz* последнего примера предыдущего фрагмента.

В ряде случаев дополнительно к вышерассмотренным средствам тестирования *Math*-объектов достаточно полезной может оказаться и довольно простая процедура, вызов **BlockQ[x]** которой возвращает **True**, объект *x* является блочной конструкцией, и **False** в противном случае. Фрагмент представляет исходный код с примером применения.

```
In[1177]:= BlockQ[x_] := Module[{a = If[SymbolQ[x], (x // DefFunc3)[[1]], $Failed],
    b = " := Block[{", If[a === $Failed, False, If[StringFreeQ[a, b], False,
    If[StringTake[a, {1, First[Flatten[StringPosition[a, b]]] - 1]}] ===
    HeadPF[x], True, False]]]

In[1178]:= Sv[x_] := Module[{y := 69; z := 64; {y, z}}; Agn[x_] := Block[{a = 75}, a*x]
In[1179]:= Kr[x_] := Block[{y = a, h = b}, (y^2 + h^2)*x]; Art[x_] := Module[{a = 69}, x*a]
In[1180]:= Map[BlockQ, {Sv, Kr, Agn, Art}]
Out[1180]= {False, True, True, False}
```

Наконец, 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* возвращается тип *x*-объекта {"DynamicModule", "Module", "Function", "Pure function", "Block"}, иначе второй аргумент остается неопределенным. Фрагмент представляет исходный код процедуры **ProcFuncBlQ** с примерами ее применения.

```
In[1178]:= ProcFuncBlQ[x_, y_ /; ! HowAct[y]] := Module[{a = ToString[HeadPF[x]],
    b = ToString[y] <> " = ", c = ToString1[DefFunc[x]], k},
    If[SuffPref[a, "HeadPF", 1], If[SuffPref[a, " & "], 2],
    ToExpression[b <> "\"Pure function\""]; True, False],
    If[HeadingQ[a],
    If[SuffPref[c, a <> " := Module[{", 1], ToExpression[b <> "\"Module\""]; True,
    If[SuffPref[c, a <> " := Block[{", 1], ToExpression[b <> "\"Block\""]; True,
    If[SuffPref[c, a <> " := DynamicModule[{", 1],
    ToExpression[b <> "\"DynamicModule\""]; True,
    ToExpression[b <> "\"Function\""]; True]], False]]

In[1179]:= Dm[] := DynamicModule[{x}, {Slider[Dynamic[x]], Dynamic[x]}]
In[1180]:= DPOb[] := Module[{a = 75, b = 64, c = 15, d = 23}, Plus[a, b, c, d]]
In[1181]:= B[x_] := Block[{a}, a = x]; G := Function[450 + 75*# &];
In[1182]:= Clear[g, g1, g2, g3, g4, g5]; {ProcFuncBlQ[Dm, g], ProcFuncBlQ[DPOb, g1],
    ProcFuncBlQ[B, g2], ProcFuncBlQ[G, g3], ProcFuncBlQ[450 + 75*# &, g4],
    ProcFuncBlQ[450, g5]}
Out[1182]= {True, True, True, True, True, False}
```

```

In[1183]:= {g, g1, g2, g3, g4, g5}
Out[1183]= {"DynamicModule", "Module", "Block", "Pure function", "Pure function", g5}
In[1184]:= Clear[t]; F[x_] := 450 + 75*x; {ProcFuncBlQ[F, t], t}
Out[1184]= {True, "Function"}

```

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 применения.

```

In[1170]:= WhatValue[x_] := Module[{a = ToString1[DefFunc[x]], c = "HeadPF[" ,
                                     b = ToString[HeadPF[x]]},
                                     If[a == "Attributes[" <> b <> "]" = {Temporary}, {"Local", b},
                                     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[1171]:= Ag[x_] := Module[{}, x^2]; Sv[x_] := Block[{a}, a + x]; F[x_] := x^2
In[1172]:= Map[WhatValue, {450 + 75*# &, hg, Sin, Ag, Sv, 70, a + b, F, Gs}]
Out[1172]= {"450 + 75 #1 & ", "Undefined", {"System", "Sin"}, "Module[{}, x^2]",
            "Block[{a}, a + x]", "70", "a + b", "x^2", "Undefined"}
In[1173]:= M = Module[{avz}, avz]; WhatValue[M]
Out[1173]= {"Local", "avz$2012"}

```

Довольно полезной при работе с процедурами представляется процедура **ExtrNames**, чей исходный код с примерами применения представлен следующим фрагментом.

```

In[1186]:= ExtrNames[x_ /; ProcQ[x]] := Module[{a = Locals2[x][[1]], f = Locals2[x][[2]],
                                     b = HeadPF[x], c = DefFunc3[x][[1]], d, p = {}, g, k = 1},
                                     g = StringReplace[c, {b <> " := Module[" -> "", b <> " := Block[" -> "", 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[1187]:= GS[x_] := Block[{a = 75, b, c}, b = 450; c = 6; x = a + b + c; x]
In[1188]:= ExtrNames[GS]
Out[1188]= {"a", "b", "c"}, {"a"}, {"x"}]
In[1189]:= ExtrNames[ProcQ]
Out[1189]= {"a", "b", "c", "d", "h"}, {"a", "h"}, {}]
In[1190]:= ExtrNames[ExtrNames]
Out[1190]= {"a", "f", "b", "c", "d", "p", "g", "k"}, {"a", "b", "c", "f", "k"}, {}]

```

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* будут производиться присваивания значений по операторам {":=", "="}.

Достаточно простая функция **Globals2** расширяет действие рассмотренных процедур **Globals** и **Globals1** на процедуры произвольного типа; вызов **Globals2[x]** возвращает список имен в строчном формате глобальных переменных процедуры *x*. Следующий фрагмент представляет исходный код функции с примерами ее применения.

```
In[6]:= Globals2[x_ /; ProcQ[x]] := ExtrNames[x][[3]]
In[7]:= GS[h_] := Module[{a = 75, b, c}, b = 450; c = 6; x = a + b + c; x + h]
In[8]:= VG[h_] := Block[{a = 75, b, c}, b = 450; c = 6; x = a + b + c; y = h^2]
In[9]:= Map[Globals2, {GS, VG, ProcQ, Tuples1, TestArgsTypes, LoadFile, WhatObj}]
Out[9]= {"x"}, {"x", "y"}, {"Res"}, {"k", "$TestArgsTypes"}, {"$Load$Files$"}, {"$Art22Kr14$"}]
```

На первый взгляд представленные в книге исходные коды процедур не всегда можно рассматривать в качестве *оптимальных*, между тем, во многих случаях это обусловлено рядом особенностей *Math*-языка, рассмотрение которых не входит в цели настоящей книги. В качестве примера рассмотрим весьма простые процедуры **A**, **A1**, **A2**, **A3** и **A4**, из которых следует, что применение функции **Attributes** требует особого подхода.

```
In[873]:= A[x_] := Module[{a = x}, {Attributes[a], Attributes[x]}]
In[874]:= A[Sin]
Out[874]= {{Temporary}, {Listable, NumericFunction, Protected}}
In[875]:= A1[x_] := Module[{a = x}, {ToExpression["Attributes[" <> ToString[a] <> ""]],
Attributes[x]}]
In[876]:= A1[Sin]
Out[876]= {{Listable, NumericFunction, Protected}, {Listable, NumericFunction, Protected}}
In[877]:= A2[x_] := Module[{a = {x}}, {Attributes[a[[2]]], Attributes[Cos]}]
In[878]:= A2[Sin, Cos]
Attributes::ssle: Symbol, string, or HoldPattern[symbol] expected at position 1 in
Attributes[a$2712[[2]]]. >>
Out[878]= {Attributes[a$2712[[2]]], {Listable, NumericFunction, Protected}}
In[879]:= A3[x_] := Module[{x}, {Attributes[{x}[[2]]], Attributes[Cos]}]
In[880]:= A3[Sin, Cos]
Attributes::ssle: Symbol, string, or HoldPattern[symbol] expected at position 1 in
Attributes[{Sin,Cos}[[2]]]. >>
Out[880]= {Attributes[{Sin, Cos}[[2]]], {Listable, NumericFunction, Protected}}
In[881]:= A4[x_] := Module[{x}, {ToExpression["Attributes[" <> ToString[{x}[[2]]] <> ""]],
Attributes[Cos]}]
In[882]:= A4[Sin, Cos]
Out[882]= {{Listable, NumericFunction, Protected}, {Listable, NumericFunction, Protected}}]
```

Вопрос работы с *локальными* переменными процедур довольно важен в процедурном программировании. Для обеспечения этой работы предложены такие процедуры, как **ExpLocals**, **Locals**, **Locals1** и **Locals2**, используемые достаточно эффективно. Процедура **ProcLocals** использует несколько другой подход для вычисления списков локальных переменных процедур. Вызов **ProcLocals[x]** с одним аргументом в строчном формате возвращает список *локальных переменных* процедуры *x*, тогда как вызов **ProcLocals[x, y]** дополнительно через неопределенную переменную *y* возвращает номер позиции в строчном представлении определения процедуры *x*, с которой начинается ее тело. Во фрагменте представлен исходный код процедуры с примерами ее применения.

```
In[995]:= ProcLocals[x_ /; ProcQ[x], z___] := Module[{a = DefOpt[ToString[x]], b = ArtKr,
    c, m = 1, n = 0, k, p, h = ""}, Clear[ArtKr]; ProcQ1[ToString[x], ArtKr];
    c = StringPosition[a, ArtKr <> "{"]; ArtKr = b; k = c[[1]][[2]] + 1;
    While[m != n, p = StringTake[a, {k, k}]; If[p == "{", m++; h = h <> p,
        If[p == "}", n++; h = h <> p, h = h <> p]; k++;
    If[{z} != {} && Definition1[{z}][[1]] === "Null", z = k + 2]; "{" <> h]

In[996]:= ProcLocals[ProcLocals]
Out[996]= "{a = ToString1[DefOpt[ToString[x]]], b = $ArtKr$, c, m = 1, n = 0, k, p, h = \"\"}"
In[997]:= {ProcLocals[ProcQ, t], t}
Out[997]= {"{a = SubsDel[ToString[InputForm[Definition[x]]],
    StringJoin[\"\"\", ToString[x], \"\"\", {\"[\", \",\", \" \", -1], b, c, d, h]\", 140}
In[1586]:= ProcBody[x_ /; ProcQ[x]] := Module[{a = DefOpt[ToString[x]], b = Art$Kr},
    Clear["Art$Kr"]; Quiet[ProcLocals[x, Art$Kr]];
    {StringTake[a, {Art$Kr, -2}], Art$Kr = b}][[1]]

In[1587]:= Art[x_, y_, z_] := Module[{a = x + y + z, b = 70}, a^2 + a + b]
In[1588]:= ProcBody[Art]
Out[1588]= "a^2 + a + b"
```

На основе процедуры **ProcLocals** реализована и весьма полезная процедура **ProcBody**, чей вызов **ProcBody[x]** возвращает собственно тело процедуры *x* в строчном формате. В конце предыдущего фрагмента представлен исходный код процедуры с примером. Обе процедуры играют весьма существенную роль в целом ряде задач процедурного программировании, имеющих дело с различными манипуляциями с определениями процедур различного типа и их составляющими компонентами.

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 процедура **PartProc** может оказаться довольно полезной. Вызов процедуры **PartProc[P]** 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 *P*. 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 применения.

```
In[1224]:= PartProc[P_;/; ProcQ[P]] := Module[{a=Locals1[P][[2]], d=ToString[DefFunc[P]],
      b = HeadPF[P], c = ArtKr, h}, Clear[ArtKr];
      If[! ProcQ1[ToString[P], ArtKr], Return[{ArtKr = c, $Failed}][[2]]],
      h = b <> " := " <> ArtKr <> "[" <> a <> ", "; ArtKr = c;
      {h <> "Procedure Body" <> "}], StringTake[DelSuffPref[d, h, 1], {1, -2}]]]

In[1225]:= ArtKr = {23, 15}; PartProc[CatN]
Out[1225]= {"CatN[s_;/; StringQ[s], n_;/; IntegerQ[n] && n >= 1] := Module[{a = "\"\",
      k = 1}, Procedure Body]", "For[k, k <= n, k++, a = StringJoin[a, s]]; a"}

In[1226]:= ArtKr
Out[1226]= {23, 15}

In[2855]:= Kr[x_, y_, z_] := Module[{a = x + y + z, b = 70}, b*a + a^2 + b]

In[2856]:= ReplaceProcBody[x_;/; ProcQ[x], y_;/; StringQ[y]] :=
      ToExpression[StringReplace[PartProc[x][[1]], "Procedure Body" -> y]]

In[2857]:= ReplaceProcBody[Kr, "b*(x + y + z)"]; Definition[Kr]
Out[2857]= Kr[x_, y_, z_] := Module[{a = x + y + z, b = 70}, b (x + y + z)]
```

Так, вызов функции **ReplaceProcBody[x, y]**, основывающейся на **PartProc**, возвращает **Null**, т.е. ничего, и обеспечивает замену тела процедуры **x** на новое тело **y**, заданное в строчном формате. Код функции с примером применения завершает фрагмент.

Процедура **ProcQ2** расширяет процедуру **ProcQ** на случай *непоименованных* процедур; ее вызов **ProcQ2[x]** возвращает **True**, если **x** – в строчном формате имя процедуры или *блок* непоименованной процедуры, и **False** в противном случае. Ниже представляется исходный код процедуры **ProcQ2** с наиболее типичными примерами ее применения. Процедура имеет наибольший смысл в выражениях, использующих такие объекты.

```
In[1376]:= ProcQ2[x_;/; StringQ[x]] := Module[{a = Art$Kr, b = If[HowAct[Art$Kr], 1, 2]},
      Clear[Art$Kr]; ToExpression["Art$Kr[x_] := " <> x];
      If[Quiet[ProcQ[x]] || ProcQ[Art$Kr],
      If[b == 1, Art$Kr = a, Clear[Art$Kr]; ToExpression["Art$Kr = . Art$Kr"]; True,
      If[b == 1, Art$Kr = a, Clear[Art$Kr]; ToExpression["Art$Kr = . Art$Kr"]; False]]

In[1377]:= Art$Kr = 75; Map[ProcQ2, {"Block[{a = 450}, x = a]", "ProcQ", "Locals2", "Ian"}]
Out[1377]= {True, True, True, False}

In[1378]:= Art$Kr
Out[1378]= 75

In[1379]:= Clear[Art$Kr]; Map[ProcQ2, {"Block[{a = 6}, x = a]", "ProcQ", "Locals2", "Ian"}]
Out[1379]= {True, True, True, False}

In[1380]:= HowAct[Art$Kr]
Out[1380]= False
```

В ряде случаев определенный интерес представляет вопрос определения *всех* средств, используемых процедурой/функцией. Процедура **ProcContent** обеспечивает анализ активизированного в текущем сеансе объекта *x* с корректным заголовком на предмет использования его определением пользовательских средств, снабженных справочной информацией. Вызов процедуры **ProcContent[x]** возвращает исчерпывающий список *имен* средств, используемых определением объекта *x* и прямых, и опосредствованных; при этом, первым элементом списка выступает собственно *x*. В случае недоступности в текущем сеансе средств, используемых объектом *x*, вызов **ProcContent[x]** возвращается невычисленным с выводом соответствующих сообщений. Нижеследующий фрагмент представляет исходный код процедуры **ProcContent** с примерами ее применения.

```
In[941]:= ProcContent[x_;/ ProcQ[x] || QFunction[ToString[x]]] := Module[{b = "", c = {x},
    a = "$ArtKr$"}, Save[a, x]; While[! SameQ[b, EndOfFile], b = Read[a, String];
    If[! StringFreeQ[ToString[b], "::usage = \""],
    c = Append[c, StringTake[b, {1, Flatten[StringPosition[b, "/" ][[1]] - 1}]], Null]];
    a = Map[ToExpression, {DeleteFile[Close[a]], c}][[2]]; DeleteDuplicates[a]

In[942]:= ProcContent[ProcQ]
Out[942]= {ProcQ, SubsDel, Sequences, ToString1, StrDelEnds, CharacterQ, HeadPF,
    SymbolQ, DefFunc, WhatObj, SubStr, HowAct}
In[943]:= ProcContent[ProcContent]
Out[943]= {ProcContent, HeadingQ, Map3, HeadPF, SymbolQ, DefFunc, WhatObj, SubStr,
    CharacterQ, HowAct, ToString1, StrDelEnds, SubsDel, Sequences}
In[944]:= ProcContent[Attrib2]
Out[944]= {Attrib2, Map3, LoadExtProg, FileExistsQ1, SearchFile, Adrive, StringEnd,
    HowAct, ToString1, StrDelEnds, CharacterQ, DirQ, StandPath, PathToFileQ, StrStr}
In[945]:= Map[ProcContent, {StrStr, ToString1, DirName}]
Out[945]= {{StrStr}, {ToString1, StrDelEnds, CharacterQ}, {DirName, DirQ, FileExistsQ1,
    SearchFile, Adrive, StringEnd, HowAct, ToString1, StrDelEnds, CharacterQ}}
In[946]:= Clear[ToString1]; ProcContent[DirName]
    StringTake::strse: String or list of strings expected at position 1 in ... >>
    StringJoin::string: String expected at position 2 in ... >>
    ToExpression::notstrbox: Sequence[<>StringTake[ToString1[{1,1,1}],{2,-2}] ... >>
Out[946]= ProcContent[DirName]

In[1288]:= ProcContent1[x_;/ ProcQ[x] || QFunction[ToString[x]]] := Module[{b = "", d,
    a = "$ArtKr$", c = {x}, h = "", p}, Save[a, x];
    While[! SameQ[b, EndOfFile], b = Read[a, String];
    If[! MemberQ[{" ", "EndOfFile"}, ToString[b]], h = h <> ToString[b]; 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}]]; h = "", Null]];
    a = Map[ToExpression, {DeleteFile[Close[a]], c}][[2]]; DeleteDuplicates[a]]
```

```

In[1289]:= ProcContent1[ProcQ]
Out[1289]= {ProcQ, SubsDel, Sequences, ToString1, StrDelEnds, CharacterQ, HeadPF,
           SymbolQ, DefFunc, WhatObj, SubStr, HowAct}
In[1290]:= ProcContent1[ProcContent]
Out[1290]= {ProcContent, HeadingQ, Map3, HeadPF, SymbolQ, DefFunc, WhatObj, SubStr,
           CharacterQ, HowAct, ToString1, StrDelEnds, SubsDel, Sequences}
In[1291]:= ProcContent1[ProcContent1]
Out[1291]= {ProcContent1, HeadingQ, Map3, HeadPF, SymbolQ, DefFunc, WhatObj,
           SubStr, CharacterQ, HowAct, ToString1, StrDelEnds, SubsDel, Sequences}
In[1292]:= ProcContent1[Attrib2]
Out[1292]= {Attrib2, Map3, LoadExtProg, FileExistsQ1, SearchFile, ADrive, StringEnd,
           HowAct, ToString1, StrDelEnds, CharacterQ, DirQ, StandPath, PathToFileQ, StrStr}
In[1293]:= Map[ProcContent1, {StrStr, ToString1, DirName}]
Out[1293]= {{StrStr}, {ToString1, StrDelEnds, CharacterQ}, {DirName, DirQ, FileExistsQ1,
           SearchFile, ADrive, StringEnd, HowAct, ToString1, StrDelEnds, CharacterQ}}
In[1400]:= Kr[x_, y_] := Plus[x, y]; Art[x_] := Module[{a = 75, b = 450, c}, c = Kr[a, b]; c]
In[1401]:= Map8[ProcContent, ProcContent1, {Art}]
Out[1401]= {{Art}, {Art, Kr}}

```

Процедура **ProcContent1**, представленная в конце предыдущего фрагмента, подобно процедуре **ProcContent** обеспечивает анализ процедуры/функции  $x$ , активированной в текущем сеансе, на предмет использования ею пользовательских средств, которые не обязательно снабжены справочной информацией. Вызов процедуры **ProcContent1**[ $x$ ] возвращает исчерпывающий список имен средств, используемых процедурой  $x$  либо функцией как прямых, так и опосредствованных; при этом, первым элементом этого списка выступает собственно имя  $x$ . В случае недоступности в текущем сеансе средств, которые используются процедурой/функцией  $x$ , вызов **ProcContent1**[ $x$ ] возвращается невычисленным с выводом соответствующих сообщений. Последние примеры этого фрагмента довольно наглядно иллюстрируют более расширенные функциональные возможности именно второй процедуры, обеспечивающей корректное тестирование объектов, использующих пользовательские средства, не снабженные справками. При этом, процедуры **ProcContent**[ $x$ ] и **ProcContent1**[ $x$ ] обеспечивают тестирование факта использования процедурой/функцией  $x$  лишь внешних по отношению к ней средств, игнорируя, в частности, процедуры, определенные в теле процедуры  $x$ . Между тем, в целях проверки используемых процедурой/функцией  $x$  внутренних процедур может использоваться процедура **SubProcs**, как иллюстрирует следующий фрагмент. В свою очередь, весьма простая процедура на основе **ProcContent1** и **SubProcs** решает задачу:

```

In[1253]:= Kr[x_, y_] := Plus[x, y]; Art[x_] := Module[{a = 75, b = 450, c, d}, c = Kr[a, b];
           d[z_] := Module[{}, 75]; c + d[450]]
In[1254]:= Map8[ProcContent, ProcContent1, {Art}]
Out[1254]= {{Art}, {Art, Kr}}
In[1255]:= SubProcs[Art][[1]]
Out[1255]= {"Art[x_]", "d[z_]"}

```

```

In[1278]:= HeadName[x_;/ HeadingQ1[x]] :=
              StringTake[x, {1, StringPosition[x, "[", 1][[1]][[1]] - 1}]
In[1279]:= ProcContent2[x_;/ ProcQ[x] | | QFunction[ToString[x]]] :=
              Block[{a = ProcContent1[x], b = SubProcs[x][[1]]}, {x, If[Length[a] > 1, a[[2 ;; -1]], {}],
              If[Length[b] > 1, Map[ToExpression, Map[HeadName, b[[2 ;; -1]]], {}]}]
In[1280]:= ProcContent2[Art]
Out[1280]= {Art, {Kr}, {d}}
In[1281]:= ProcContent2[Attrib2]
Out[1281]= {Attrib2, {Map3, LoadExtProg, FileExistsQ1, SearchFile, Adrive, StringEnd,
              HowAct, ToString1, StrDelEnds, CharacterQ, DirQ, StandPath, PathToFileQ, StrStr}, {}}

```

В отличие от процедур **ProcContent** и **ProcContent1**, вызов процедуры **ProcContent2[x]** возвращает вложенный список из 3 элементов, первый элемент которого определяет имя процедуры/функции *x*, второй – список имен всех *внешних* процедур/функций, используемых объектом *x*, и третий – список имен *внутренних* процедур/функций, определенных в теле *x*. При этом, процедура **ProcContent2** существенно использует весьма простую функцию, вызов которой **HeadName[x]** возвращает имя в строчном формате заголовка *x*. Предыдущий фрагмент представляет исходные коды данных программных средств с достаточно типичными примерами их применения.

Следующая процедура **HeadingQ1** представляет собой весьма полезное расширение рассмотренной процедуры **HeadingQ** относительно ее возможностей тестирования заголовков на их корректность, где как отмечалось выше, под *заголовком* понимается левая часть определения процедуры либо функции. Вызов процедуры **HeadingQ1[x]** возвращает **True**, если *фактический* аргумент *x*, заданный в строчном формате, может быть рассмотрен как синтаксически корректный *заголовок*; иначе возвращается **False**. Фрагмент представляет исходный код процедуры с примерами ее применения.

```

In[2512]:= HeadingQ1[x_;/ StringQ[x]] := Module[{b, c = {}, d, h = "F", k = 1,
              a = StringTake[x, {Flatten[StringPosition[x, "[", 1][[1]] + 1, -2]}],
              b = StringSplit1[a, ","]; For[k, k <= Length[b], k++, d = b[[k]];
              c = Append[c, If[StringFreeQ[d, "_"], False, If[MemberQ[
              Map[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] == 75]", "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]= {{False, False, False}, {True, True, True}}

```

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 время данные процедуры оказываются во многих случаях довольно полезными. В представленной ниже процедуре дан пример дальнейшего расширения **HeadingQ1**.

Следующая процедура **ArgsTypes** служит для тестирования формальных аргументов функции/процедуры, активизированной в текущем сеансе пакета. Вызов процедуры **ArgsTypes[x]** возвращает вложенный список, чьи 2-элементные подспiski в *строчном* формате определяют имена формальных аргументов и допустимые их типы (*а в более широком смысле тесты на их допустимость и начальные по умолчанию*) соответственно. В случае отсутствия для аргумента типа он определяется как **"Arbitrary"**, тогда как один фактический аргумент *x* иницирует возврат простого списка отмеченного формата; на *недопустимом* формальном аргументе возвращается **"FAIL"**. Следующий фрагмент представляет исходный код процедуры **ArgsTypes** с примерами ее применения.

```
In[1307]:= ArgsTypes[x_;/ ProcQ[x] || QFunction[ToString[x]]] :=
Module[{a = Flatten[StringPosition[HeadPF[x], "["], b, c = {}, d, k = 1],
b = StringSplit1[StringTake[HeadPF[x], {a[[1]] + 1, -2}], ","],
For[k, k <= Length[b], k++, d = b[[k]];
c = Append[c, If[StringFreeQ[d, "_"], {"FAIL"}, a = StringSplit1[d, "_/"];
If[a === {d}, StringSplit1[d, "_"], a]]];
c = Map[If[Length[#] == 1 && #[[1]] != "FAIL", Append[#, "Arbitrary"], #] &,
Mapp[Select, Map[StringTrim, c], # != "" &]; If[Length[c] == 1, c[[1]], c]]

In[1308]:= Map[ArgsTypes, {StringSplit1, StrStr, ArgsTypes, Mapp, Map8}]
Out[1308]= {{{"x", "StringQ[x]"}, {"y", "StringQ[y]"}}, {"x", "Arbitrary"}, {"x", "ProcQ[x] ||
QFunction[ToString[x]]"}, {"F", "ProcQ[F] || SysFuncQ[F] || SymbolQ[F]"},
{"Expr", "Arbitrary"}, {"x", "Arbitrary"}}, {"x", "Arbitrary"}, {"y", "ListQ[y]"}]}

In[1309]:= W[x_, y_, z_;/ StringQ[z] && StringLength[z] > 3] := {x, y, z}; ArgsTypes[W]
Out[1309]= {{{"x", "Arbitrary"}, {"y", "Arbitrary"}, {"z", "StringQ[z] && StringLength[z] > 3"}}}

In[1310]:= G[x_Integer, y_ : 68, z_Integer] := Module[{a}, a = x + y^2 + z; a]; ArgsTypes[G]
Out[1310]= {{{"x", "Integer"}, {"y", "Arbitrary"}, {"z", "Integer"}}}

In[1311]:= V[x_, y_Integer: 70, z_] := Module[{a}, a = x + y^2 + z; a]; ArgsTypes[V]
Out[1311]= {{{"x", "Arbitrary"}, {"y", "Integer:70"}, {"z", "Arbitrary"}}}

In[1312]:= K[x_, y_Integer: 70, Art : 80, z_] := Module[{a}, a = x + y^2 + z; a]; ArgsTypes[K]
Out[1312]= {{{"x", "Arbitrary"}, {"y", "Integer:70"}, {"FAIL"}, {"z", "Arbitrary"}}
```

На основе предыдущей процедуры **ArgsTypes** появляется возможность расширения функциональных свойств процедуры **HeadingQ1**, позволяя достаточно существенно расширить типы корректно тестируемых *заголовков* процедур/функций. Следующий фрагмент представляет исходный код процедуры с примерами ее применения.

```
In[1300]:= 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[AvzRansIan]; AvzRansIan[" <> a <> "] := 75"]];
If[Quiet[Select[ToExpression["ArgsTypes[AvzRansIan]"], ! 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[1301]:= Map8[HeadingQ1, HeadingQ2, {"V[x_/_String]"}]
Out[1301]= {True, False}
In[1302]:= Map8[HeadingQ1, HeadingQ2, {"V[x_/_; StringQ[x]]"}]
Out[1302]= {True, True}
In[1303]:= F[x_/_; StringQ[x]; y_/_; IntegerQ[y]; z_/_; ListQ[z]] := 75 + Length[z]
In[1304]:= F[{70, 65, 45, 23, 16, 49}]
Out[1304]= 81
In[1305]:= Map[HeadingQ2, {"F[x_/_; StringQ[x]]", "F[x/_; StringQ[x]]", "F[x; StringQ[x]]",
"F[x_/_ StringQ[x]]", "F[x_/_; StringQ[x]]", "F[x_/_; y_/_; z_]"}]
Out[1305]= {True, False, False, False, False, True}
In[1306]:= HeadingQ3[x_/_; StringQ[x]] := Block[{a = "AvzRansIn", b}, Clear[AvzRansIn];
b = Quiet[ToExpression[a <> StringTake[x,
{Flatten[StringPosition[x, "["]][[1]], -1}] <> " := 70"]];
If[SameQ[b, Null], Clear[a]; HeadingQ2[x], Clear[a]; False]]
In[1307]:= Map[HeadingQ3, {"F[x_/_; StringQ[x]]", "F[x/_; StringQ[x]]", "F[x; StringQ[x]]",
"F[x_/_ StringQ[x]]", "F[x_/_; StringQ[x]]", "F[x_/_; y_/_; z_]"}]
Out[1307]= {True, False, False, False, 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 значение **False**.

Вызов **ProcFuncCS[]** возвращает вложенный список, чьи подсписки определяют *имена* в строчном формате процедур и функций соответственно, *определения* которых были вычислены в текущем сеансе. Фрагмент представляет исходный код функции.

```

In[99]:= ProcFuncCS[] := Quiet[Map3[Select, Names["*"], {ProcQ[#] &, QFunction[#] &}]]
In[100]:= ProcFuncCS[]
Out[100]= {"Av", "Ga", "Manur", "ProcFuncCS"}, {"Rtb", "Tab", "V", "W", "Whs"}

```

Процедурная переменная **\$TypeProc** используется лишь в теле процедур *любого* типа и получает значение типа в строчном формате данной процедуры в разрезах "**Block**", "**Module**" и "**DynamicModule**"; *вне* процедуры переменная получает значение **\$Failed**, тогда как использование переменной *вне* процедур некорректно, вызывая аварийную ситуацию с возвратом **\$Failed**, как хорошо иллюстрирует фрагмент, представляющий исходный код переменной с примерами ее наиболее типичного использования.

```
In[955]:= $TypeProc := CheckAbort[If[$$$Art23$Kr15$$$ = Select[{Stack[Module],
                                Stack[Block], Stack[DynamicModule]}, # != {} &];
                                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[956]:= M[x_] := Module[{a = 75, b = 450, c = $TypeProc}, c]; M[70]
Out[956]= "Module"
In[957]:= G[x_] := Module[{a = 70, b = 450, c}, c = a + b + x; c^2; $TypeProc]; G[70]
Out[957]= "Module"
In[958]:= B[x_] := Block[{a = 75, b = 450, c = $TypeProc}, c]; B[70]
Out[958]= "Block"
In[959]:= DM[x_] := DynamicModule[{a, c = $TypeProc}, x; c]; DM[70]
Out[959]= "DynamicModule"
In[960]:= F[x_ /; ListQ[x]] := Append[Select[x, OddQ[#] &], $TypeProc]; F[{65, 70, 45, 16}]
Out[960]= $Failed
In[961]:= $TypeProc
Out[961]= $Failed
```

В ряде случаев процедурного программирования данная переменная весьма полезна. К **\$TypeProc** непосредственно примыкает процедурная переменная **\$CallProc**, вызов которой в строчном формате возвращает содержимое тела процедуры, содержащей ее, на момент ее вызова. Остальное, сказанное по **\$TypeProc**, относится и к **\$CallProc**.

```
In[2584]:= $CallProc := InputForm[Quiet[Check[StringTake[ToString1[Flatten[
                                {Stack[Block], Stack[Module], Stack[DynamicModule]}][[1]], {10, -2}], $Failed]]]
In[2585]:= M[x_, y_ /; StringQ[y]] := Module[{a = $CallProc, b, c}, x + StringLength[y]; a]
In[2586]:= M[70, "AvzAgnVsArtKr"]
Out[2586]= "Module[{a$ = $CallProc, b$, c$}, 70 + StringLength["AvzAgnVsArtKr"]; a$]"
In[2587]:= B[x_, y_ /; StringQ[y]] := Block[{a = $CallProc, b, c}, x + StringLength[y]; a]
In[2588]:= B[70, "AvzAgnVsvArtKr"]
Out[2588]= "Block[{a = $CallProc, b, c}, 70 + StringLength["AvzAgnVsvArtKr"]; a] "
In[2589]:= $CallProc
Out[2589]= $Failed
```

Вполне определенный интерес представляет следующая процедура **DefOpt**, в целом ряде случаев более приемлемая, чем функция **Definition** и наши процедуры **DefFunc**, **DefFunc1**, **DefFunc2** и **DefFunc3**, рассмотренные выше; примеры иллюстрируют это.

```
In[1341]:= DefOpt[x_ /; StringQ[x]] := Module[{a = Definition1[x], b = StringJoin["", x, ""],
      c, d = t}, c = Flatten[StringPosition[a, b]]; If[c == {}, Definition1[x], Clear[t];
      ToExpression[StringReplace[a, StringJoin[StringReplace[StringJoin[SubStr[a,
      c[[1]], {"[", " ", "\\", t, "\\", b -> ""], b -> ""]]; Clear[t]; t = d; Definition1[x]]]
In[1342]:= Definition[SystemQ]
Out[1342]= SystemQ[AladjevProceduresAndFunctions`SystemQ`S_] :=
      If[! SymbolQ[AladjevProceduresAndFunctions`SystemQ`S], False,
      If[ToString[HeadPF[AladjevProceduresAndFunctions`SystemQ`S]] ==
      ToString[AladjevProceduresAndFunctions`SystemQ`S], True, False]]
In[1343]:= DefOpt["SystemQ"]
Out[1343]= SystemQ[S_] := If[! SymbolQ[S], False, If[ToString[HeadPF[S]] == ToString[S],
      True, False]]
In[1344]:= DefFunc[$TypeProc]
Out[1344]= Attributes[$Failed] = {HoldAll, Protected}
In[1345]:= DefOpt["$TypeProc"]
Out[1345]= $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[1346]:= DefOpt["Sin"]
Out[1346]= Attributes[Sin] = {Listable, NumericFunction, Protected}
```

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 иллюстрирует первый пример фрагмента. С другой стороны, и наши процедуры оказываются непригодными при необходимости получения определений некоторых процедурных переменных, в частности, **\$TypeProc**, как иллюстрирует 3-й пример фрагмента. И только вызов процедуры **DefOpt[x]** возвращает определение *x*-объекта в оптимальном формате независимо от типа пользовательского объекта. При этом, вызов **DefOpt[x]** не только возвращает *оптимальный* формат *определения* объекта *x*, но и вычисляет его в текущем сеансе, что в ряде случаев является весьма полезным; имя объекта *x* кодируется в строчном формате. Предыдущий фрагмент представляет исходный код процедуры **DefOpt** наряду с типичными примерами ее применения.

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 применения.

```
In[1282]:= RemoveNames[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[1283]:= {Length[Names["*"]], RemoveNames[], Names["*"]}
Out[1283]= {656, {"Art", "Kr", "Rans"}, {"Rac", "Rea"}, {"Art", "Kr", "Rans", "Rac", "Rea"}}
In[1284]:= RemoveNames[]
Out[1284]= {"Art", "Kr", "Rans"}, {"Rac", "Rea"}
```

Процедура **RemoveNames** достаточно полезна в целом ряде приложений, связанных с освобождением рабочей области *Mathematica* от неиспользуемых символов.

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[1309]:= Names1[] := Block[{a = Names["*"], b = {}, {}, {}, c = 1, d},
  While[c <= Length[a], d = a[[c]]; If[ProcQ[d], AppendTo[b[[1]], d],
    If[Quiet[Check[QFunction[d], False]], AppendTo[b[[2]], d],
    If[! SameQ[ToString[Quiet[DefFunc[d]]], "Null"],
    AppendTo[b[[3]], d], AppendTo[b[[4]], d]]; c++; b]

In[1310]:= Names1[]
Out[1310]= {"Bt", "Mas", "Names1", "W"}, {"F", "G"}, {"Art23$", "Kr", "$NameProcCS"}, {}]
```

Процедура **Names1** достаточно полезна в ряде приложений, в частности, в некоторых вопросах процедурного программирования, в определенных отношениях расширяя стандартную функцию **Names** пакета.

Выше представлена процедура, чей вызов **HeadPF[F]** возвращает в строчном формате *заголовок* активизированного в текущем сеансе объекта с именем *F* типа процедуры и функции. При этом, для объекта *F*, имеющего несколько *различных* заголовков, вызов возвращает заголовок только того объекта, чье определение возвращается по вызову функции **Definition[F]** первым. Тогда как вызов процедуры **HeadPF1[F]** возвращает в

строчном формате *заголовок* всех активизированных в текущем сеансе объектов  $F$  типа процедур и функций; на объектах  $F$  других типов вызов возвращается *невычисленным*. В ошибочных ситуациях вызов **HeadPF1**[ $F$ ] возвращает **\$Failed**. Следующий фрагмент представляет исходный код процедуры наряду с примерами ее применения.

```
In[1300]:= HeadPF1[F_ /; ProcQ[F] || QFunction[ToString[F]]] := Module[{c = " := ", d = {},
    a = StringReplace[Quiet[ToString["InputForm"[ToString1[DefFunc[F]]]]],
    "InputForm[" -> ""], b = ToExpression[ToString1[DefOpt["StrDelEnds"]]]],
    Delim, k = 1, t},
    Delim[s_String] := Module[{a = StringPosition[s, "<>"], b = {}, c, k = 1},
    If[MemberQ[{0, 1}, Length[a]], Return[$Failed], While[k <= Length[a] - 1,
    c = StringTake[s, {a[[k]][[1]], a[[k + 1]][[2]]}]; If[StringTake[c, {3, 4}] == "a$" &&
    IntegerQ[ToExpression[StringTake[c, {5, -3}]]], b = AppendTo[b, c], Null]; k++]; b];
    b = Map[StringTrim, StringSplit[a, Flatten[{"\n" \n", Delim[a]}]]];
    While[k <= Length[b], t = Flatten[StringPosition[b[[k]], c]];
    d = AppendTo[d, StringTake[b[[k]], {1, t[[1]] - 1}]; k++];
    If[Length[d] == 1, d[[1]], d]]

In[1301]:= M[x_ /; x == "avzagn"] := Module[{a, b, c, x}; 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_, y_, z_] := x + y + z; M[x_List, y_] := Block[{a, b, c}, "abc"; Length[x] + y]

In[1302]:= HeadPF1[M]
Out[1302]= {"M[x_ /; x == \"avzagn\"]", "M[x_ /; IntegerQ[x], y_String]", "M[x_List, y_]",
    "M[x_, y_]", "M[x_String]", "M[x_, y_, z_]"}

In[1303]:= HeadPF1[HeadPF1]
Out[1303]= "HeadPF1[F_ /; ProcQ[F] || QFunction[ToString[F]]]"

In[1304]:= HeadPF[M]
Out[1304]= "M[x_ /; x == \"avzagn\"]"
```

Процедура **HeadPF1** выступает в качестве достаточно полезного средства при работе с объектами типа процедур, функций и блоков; ее применение, порой, существенно упрощает решение целого ряда задач процедурного программирования. Между тем, на объектах  $F$ , имеющих единственные определения, результаты вызовов **HeadPF1**[ $F$ ] и **HeadPF**[ $F$ ] идентичны, как иллюстрируют примеры предыдущего фрагмента.

Обработка *одноименных* объектов с разными *заголовками* представляет особый интерес.

```
In[1536]:= RemovePF[x_ /; ProcQ[x] || FunctionQ[ToString[x]], y_ /; HeadingQ1[y] ||
    ListQ[y] && DeleteDuplicates[Map[HeadingQ1, y]] == {True}] :=
    Module[{b = {}, c = If[ListQ[y], y, {y}], d, k = 1, j},
    a = StringSplit[ToString[InputForm[DefOpt[ToString[x]]]], "\n \n"],
    d = Mapp[StringJoin, c, " := ";
    While[k <= Length[a], j = 1; While[j <= Length[d],
    If[PrefixQ[d[[j]], a[[k]]], b = Append[b, a[[k]]]; j++]; k++];
    Remove[x]; Map[ToExpression, MinusList[a, b]] ;]
```

```

In[1537]:= 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]; M[x_String] := x;
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x];
M[x_, y_] := Module[{a, b, c}, "agn"; x + y]
M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "ArtKr"; Length[x] + y]
In[1538]:= DefOpt["M"]
Out[1538]= 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}, "ArtKr"; Length[x] + y]
M[x_, y_] := Module[{a, b, c}, "agn"; x + y]
M[x_String] := x
M[x_, y_, z_] := x + y + z
In[1539]:= RemovePF[M, {"M[x_ /; x == \"avz\"]", "M[x_, y_, z_]"}]
In[1540]:= DefOpt["M"]
Out[1540]= M[x_ /; x === "avz", y_] := Module[{a, b, c}, y]
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]
M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "ArtKr"; Length[x] + y]
M[x_, y_] := Module[{a, b, c}, "agn"; x + y]
M[x_String] := x
In[1541]:= F[x_, y_] := x + y; RemovePF[F, {"F[x_, y_]", "F[x_, y_, z_]"}]; Definition["F"]
Definition::notfound: Symbol F not found. >>
Out[1541]=
In[1542]:= F[x_, y_] := x + y; RemovePF[F, "F[x_, y_, z_]"]; DefOpt["F"]
Out[1542]= F[x_, y_] := x + y

```

Как уже отмечалось, пакет допускает наличие одноименных объектов с различными *заголовками*, именно которые идентифицируют объекты, а не их имена. Стандартная функция **Definition**, а также наши процедуры **DefFunc**, **DefFunc2**, **Deffunc3** и **DefOpt** 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**, чей вызов **RemovePF[x, y]** возвращает **Null**, т.е. ничего, обеспечивая удаление из текущего сеанса объектов с именем *x* и заголовками, определенными вторым аргументом *y* (*заголовок в строчном формате либо их список*). В предыдущем фрагменте представлены исходный код процедуры **RemovePF** наряду с примерами ее применения. Следует отметить, что, используя в качестве начального значения для *локальной* переменной *a* вызов **DefFunc3[x]**, можно несколько упростить исходный код процедуры. В процедурном программировании оказывается довольно полезной процедура **RemovePF1**, являющаяся модификацией процедуры **RemovePF**. Фрагмент представляет исходный код процедуры с примерами ее применения.

```

In[1378]:= RemovePF1[x_ /; HeadingQ1[x] || ListQ[x] &&
DeleteDuplicates[Map[HeadingQ1, x]] == {True}] :=
Module[{a = DeleteDuplicates[Map[HeadName, Flatten[{x}]]], b, c = {}, d, k = 1, j},
b = Flatten[Map[DefFunc3, a]];
b = Quiet[Check[Flatten[Map[DefFunc3, a]], Return[Defer[RemovePF1[x]]]];
While[k <= Length[b], j = 1; While[j <= Length[d],
If[PrefixQ[d[[j]], b[[k]]], c = Append[c, b[[k]]]; j++]; k++];
Map[Remove, a]; Map[ToExpression, MinusList[b, c]] ;]

In[1379]:= 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

In[1380]:= Flatten[Map[DefFunc3, {M, F}]]
Out[1380]= {"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",
"M[x_, y_, z_] := x + y + z", "F[x_, y_Integer] := x + y", "F[x_, y_] := x + y",
"F[x_, y_, z_] := x + y + z"}

In[1381]:= RemovePF1[{"M[x_, y_]", "F[x_, y_, z_]", "M[x_String]", "F[x_, y_Integer]"}]
In[1382]:= Flatten[Map[DefFunc3, {M, F}]]
Out[1382]= {"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_, z_] := x + y + z", "F[x_, y_] := x + y"}

```

Вызов процедуры **RemovePF1[x]** возвращает **Null**, т.е. ничего, обеспечивая удаление из текущего сеанса объектов с заголовками, определенными аргументом *x* (заголовок в строчном формате либо их список). В случае некорректных заголовков в *x* возвращается невычисленный вызов **RemovePF1[x]**. Аналогичная ситуация имеет место при вызове **RemovePF[x, y]**, если *x* не определяет процедуру/функцию. В предыдущем фрагменте представлены исходный код процедуры **RemovePF1** с примерами ее применения.

Процедура **Headings** - полезная модификация процедур **HeadPF**, **HeadPF1** на случай одноименных процедур/функций с различными заголовками. **Headings[x]** возвращает заголовок в строчном формате или их список для процедур/функций *x*. Следующий фрагмент представляет исходный код процедуры с примерами ее применения.

```

In[1347]:= Headings[x_ /; ProcQ[x] || QFunction[ToString[x]]] := Module[{b = {}, c, k = 1,
a, d = Attributes[x]}, ClearAttributes[x, d];

```

```

a = Quiet[Check[DefFunc3[x], DefOpt[ToString[x]]];
While[k <= Length[a], c = If[! ListQ[a], {ToString[a]}, a][[k]];
b = AppendTo[b, StringTake[c, {1, Flatten[StringPosition[c, " := "][[1]] - 1}]; k++];
In[1348]:= 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[1349]:= Headings[M]
Out[1349]= {"M[x_ /; x === \"avz\", y_]", "M[x_ /; x == \"avz\"]", "M[x_ /; IntegerQ[x],
y_String]", "M[x_ /; ListQ[x], y_]", "M[x_, y_]", "M[x_String]", "M[x_, y_, z_]"}
In[1350]:= Map[Headings, {L, L1}]
Out[1350]= {"L[x_]", "L1[x_, y_]"}

```

Достаточно полезным оказывается расширение **Headings1** предыдущей процедуры. В общем случае вызов процедуры **Headings1[x]** возвращает вложенный список, чьими элементами являются подписки, определяющие соответственно заголовки процедур и функций с именем *x*; первый элемент такого подписка определяет тип *x*-объекта, тогда как остальные определяют соответствующие ему заголовки. Вызов возвращает простой список, если какой-либо из подписков не содержит заголовков. Следующий фрагмент представляет исходный код процедуры с примерами ее применения.

```

In[1445]:= Headings1[x_ /; ProcQ[x] || QFunction[ToString[x]]] := Module[{c = {}, k = 1,
h = {"Procedure"}, {"Function"}}, d = $ArtKr$, p, a = Flatten[{Headings[x]}],
b = Quiet[Check[DefFunc3[x], DefOpt[ToString[x]]]], Clear[$ArtKr$];
While[k <= Length[b], p = b[[k]]; ToExpression[StringReplacePart[p, "$ArtKr$",
{1, Flatten[StringPosition[p, HeadName[a[[k]]][[2]]]}];
If[ProcQ[$ArtKr$], h[[1]] = Append[h[[1]], a[[k]],
h[[2]] = Append[h[[2]], a[[k]]]; Clear[$ArtKr$]; k++;
ToExpression["$ArtKr$ = " <> ToString[d]]; k = 1;
While[k <= Length[c], If[c[[k]][[2]] == "Procedure",
h[[1]] = Append[h[[1]], c[[k]][[1]], h[[2]] = Append[h[[2]], c[[k]][[1]]]; k++;
h = Select[h, Length[#] > 1 &]; If[Length[h] == 1, h[[1]], h]]
In[1446]:= Headings1[M]
Out[1446]= {"Procedure", "M[x_ /; x === \"avz\", y_]", "M[x_ /; x == \"avz\"]",
"M[x_ /; IntegerQ[x], y_String]", "M[x_ /; ListQ[x], y_]", "M[x_, y_]"},
{"Function", "M[x_String]", "M[x_, y_, z_]"}
In[1447]:= $ArtKr$ = 450; {Map[Headings1, {L, L1}], $ArtKr$}
Out[1447]= {"Function", "L[x_]", "Procedure", "L1[x_, y_]"}, 450}

```

Достаточно полезным оказывается расширение **HeadingsPF** предыдущей процедуры. В общем случае вызов процедуры **HeadingsPF[]** возвращает вложенный список, чьими элементами являются подписки, определяющие соответственно заголовки процедур

и функций, определения которых были вычислены в текущем сеансе; *первый* элемент такого подписка определяет тип объекта, тогда как остальные определяют *заголовки*, соответствующие ему. Вызов процедуры возвращает простой список, если какой-либо из подписков не содержит *заголовков*. Следующий фрагмент представляет исходный код процедуры **HeadingsPF** с примерами ее наиболее типичного применения.

```
In[1913]:= HeadingsPF[] := Module[{b, c = {"Procedure"}, {"Function"}}, p, k = 1,
    a = MinusList[Flatten[Names1[[[1 ;; 2]]], {"$ArtKr$", "HeadingsPF"}]],
    b = Map[Headings1, a]; b = Map[If[NestListQ[#, Sequences[#, #] &, b];
        While[k <= Length[b], p = b[[k]]; If[p[[1]] == "Procedure",
            c[[1]] = Append[c[[1]], p[[2 ;; -1]]], c[[2]] = Append[c[[2]], p[[2 ;; -1]]]; k++];
    c = Select[Map[Flatten, c], Length[#] > 1 &]; If[Length[c] == 1, c[[1]], c]]

In[1914]:= 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[1915]:= HeadingsPF[]
Out[1915]= {"Procedure", "M[x_ /; x === \"avz\\\", y_]", "M[x_ /; x == \"avz\\\"]",
    "M[x_ /; IntegerQ[x], y_String]", "M[x_ /; ListQ[x], y_]", "M[x_, y_]\"",
    {"Function", "M[x_String]", "M[x_, y_, z_]", "F[x_ /; x === \"avz\\\", y_]",
    "F[x_ /; x == \"avz\\\"]", "L[x_, y_]", "L[x_]"}]

In[1947]:= HeadingsPF[]
Out[1947]= {"Function", "V[x_ /; x == \"avz\\\", y_]", "F[x_ /; x == \"avz\\\"]", "G[x_]", "L[x_]"}]
```

Для временного удаления из текущего сеанса процедур/ функций служит процедура **DelRestPF**, исходный код которой с примерами применения представляет фрагмент.

```
In[1620]:= DelRestPF[r_ /; MemberQ[{"d", "r"}, r], x___] := Module[{f = "$Art23Kr16$",
    a = Quiet[Select[{x}, ProcQ[#] || QFunction[ToString[#]] &]],
    If[r == "d", Save[f, a]; Map[Remove, a]; ,
    ToExpression[ToString[Get[ToString[f]]]; DeleteFile[f]]]

In[1621]:= M[x_] := Module[{a, b}, {x}]; M1[x_] := Module[{b, c}, {x}]; F[x___] := {x}
In[1622]:= G := 75; DelRestPF["d", M, M1, F, 75, GS, G]
In[1623]:= DelRest["r"]
In[1624]:= Definition[F]
Out[1624]= F[x___] := {x}
In[1625]:= Definition[M1]
Out[1625]= M1[x___] := Module[{b, c}, {x}]
```

Вызов **DelRestPF["d", x, y, ...]** возвращает **Null**, удаляя из текущего сеанса процедуры, функции {x, y, ...}, тогда как вызов **DelRestPF["r"]** восстанавливает их доступность в текущем сеансе с возвратом **Null**, т.е. ничего. Процедура полезна в ряде приложений.

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.

Прежде всего, в работе с атрибутами символов весьма полезна простая функция, чей вызов **ClearAllAttributes[x]** отменяет все атрибуты символа *x*.

```
In[1473]:= ClearAllAttributes[x_ /; SymbolQ[x]] := ClearAttributes[x, Attributes[x]]
In[1474]:= SetAttributes[M, {Flat, Protected}]; Attributes[M]
Out[1474]= {Flat, Protected}
In[1475]:= ClearAllAttributes[M]; Attributes[M]
Out[1475]= {}
```

Тогда как, например, вызов процедуры **RemProcOnHead[x]** вполне корректно удаляет из текущего сеанса процедуру/функцию с заголовком *x* при условии отсутствия для имени процедуры *Protected*-атрибута, в противном случае вызывая **\$Failed**. Тогда как вызов модификации **RemProcOnHead1[x]** корректно удаляет из текущего сеанса процедуру/функцию с заголовком *x* безотносительно наличия у нее атрибутов. Для этого предварительно необходимым оказалось сохранение атрибутов с последующей их отменой и восстановлением после выполнения удаления процедуры/функции с заголовком *x*. Протокол работы достаточно наглядно иллюстрирует сказанное.

```
In[1738]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[1739]:= RemProcOnHead["M[x_, y_, z_]"]
Out[1739]= "Done"
In[1740]:= Definition[M]
Out[1740]= M[x_, y_] := Module[{}, {x}]
           M[x_] := Module[{}, {x}]
In[1741]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[1742]:= SetAttributes[M, {Flat, Protected}]
In[1743]:= RemProcOnHead["M[x_, y_, z_]"]
Out[1743]= $Failed
In[1744]:= Definition[M]
Out[1744]= Attributes[M] = {Flat, Protected}
           M[x_, y_] := Module[{}, {x}]
           M[x_, y_, z_] := x + y + z
           M[x_] := Module[{}, {x}]
```

```
In[1745]:= RemProcOnHead1[x_;/ HeadingQ[x]] := Module[{a = HeadName[x],
               b = "Attributes[" <> HeadName[x] <> "]",
               a = ToExpression["Attributes[" <> HeadName[x] <> ""]];
  ToExpression["ClearAllAttributes[" <> HeadName[x] <> ""]]; RemProcOnHead[x];
  ToExpression["SetAttributes[" <> HeadName[x] <> "," <> ToString[a] <> ""]]; ]

In[1746]:= RemProcOnHead1["M[x___, y_, z_]"]
In[1747]:= Definition[M]
Out[1747]= Attributes[M] = {Flat, Protected}
          M[x_, y_] := Module[{}, {x}]
          M[x_] := Module[{}, {x}]
```

Подобным же способом процедура **Rename1** расширяет представленную процедуру **Rename** на случай одноименных процедур/функций с различными заголовками и с приписанными им произвольными атрибутами. Вызов **Rename1**[*x*,*y*] возвращает *Null*, т.е. ничего, переименовывая объект с именем *x* на *y* с сохранением всех атрибутов *x*-объекта. Фрагмент представляет исходный код **Rename1** с примерами ее применения.

```
In[2849]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x___, y_, z_] := x+y+z
In[2850]:= SetAttributes[M, {Flat, Protected}]

In[2851]:= Rename1[x_;/ HowAct[x], y_;/ ToString[Definition[y]] == "Null"] :=
               Module[{a = Attributes[x], b}, ClearAllAttributes[x];
               b = Map10[StringJoin, ToString[y], Mapp[DelSuffPref, DefFunc3[x], ToString[x], 1]];
               ToExpression[b]; SetAttributes[y, a]; Remove[x]]

In[2852]:= Rename1[M, G]; {DefFunc3[G], Definition[M]}
Out[2852]= {"G[x_, y_] := Module[{}, {x}]", "G[x___, y_, z_] := x + y + z",
            "G[x_] := Module[{}, {x}]", Null}

In[2853]:= Rename1[G, Art]; {DefFunc3[Art], Definition[G]}
Out[2853]= {"Art[x_, y_] := Module[{}, {x}]", "Art[x___, y_, z_] := x + y + z",
            "Art[x_] := Module[{}, {x}]", Null}
```

В отличие от предыдущей процедуры **Rename1**, процедура **RenameH** обеспечивает в определенной мере выборочное переименование одноименных процедур/функций 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 пакета.

```
In[1402]:= RenameH[x_;/ HeadingQ[x], y_;/ SymbolQ[y] | ToString[Definition[y], z_]
           == "Null"] := Module[{b = ToExpression["Attributes[" <> HeadName[x] <> ""]],
               a = HeadName[x], c, d}, c = DefFunc3[a];
  If[c == {}, Return[$Failed], ToExpression["ClearAllAttributes[" <> HeadName[x] <> ""]];
  ToExpression[ToString[y] <> DelSuffPref[Select[c, SuffPref[#, x <> " := ", 1] &][[1]], a, 1];
               If[{z} == {}, RemProcOnHead[x];
               ToExpression["SetAttributes[" <> ToString[y] <> "," <> ToString[b] <> ""]]; ]
```

```
In[1403]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[1404]:= SetAttributes[M, {Flat, Protected}]
In[1405]:= RenameH["M[x_, y_, z_]", G]
In[1406]:= DefFunc3[G]
Out[1406]= {"G[x_, y_, z_] := x + y + z"}
In[1407]:= DefFunc3[M]
Out[1407]= {"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 пакета. Предыдущий фрагмент представляет исходный код **RenameH** с наиболее типичными примерами ее применения.

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 работу с атрибутами объектов на основе *заголовков*.

```
In[1627]:= DefAttributesH[x_;/; HeadingQ[x], y_;/; MemberQ[{"Set", "Clear"}, y], z_] :=
Module[{a, b, c = ArtKr, h = Attributes[ArtKr]},
  ClearAllAttributes[ArtKr]; Clear[ArtKr]; RenameH[x, ArtKr, 75];
  If[y == "Set", SetAttributes[ArtKr, {z}], ClearAllAttributes[ArtKr]];
  b = StringSplit[ToString[HeadPF[ArtKr]], "\n\n"]][[-1]];
  RenameH[b, HeadName[x]]; ClearAllAttributes[ArtKr]; Clear[ArtKr];
  ArtKr = c; SetAttributes[ArtKr, h]]
In[1628]:= M[x_] := Module[{}, {x}]; M[x_, y_] := Module[{}, {x}]; M[x_, y_, z_] := x+y+z
In[1629]:= DefAttributesH["M[x_, y_, z_]", "Set", Flat, Protected]
In[1630]:= AttributesH[x_;/; HeadingQ[x]] := Module[{b, c = ArtKr, h = Attributes[ArtKr]},
  ClearAllAttributes[ArtKr]; Clear[ArtKr]; RenameH[x, ArtKr, 75];
  b = Attributes[ArtKr]; ClearAllAttributes[ArtKr]; ArtKr = c; SetAttributes[ArtKr, h]; b]
In[1631]:= ArtKr = 75; {AttributesH["M[x_, y_, z_]", ArtKr]}
Out[1631]= {{Flat, Protected}, 75}

```

Предыдущий фрагмент представляет исходные коды *двух* процедур **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*, ...}. В то время как вызов **AttributesH**[*x*] возвращает список атрибутов, приписанных объекту с заголовком *x*. Представленные средства работы с объектами на уровне *заголовков* в ряде случаев достаточно полезны.

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, например, функция **Definition**, как наглядно иллюстрирует 1-й пример следующего фрагмента. В целях расширения стандартной функции на типы, отличные от упомянутых, может быть использована процедура **Definition1**, чей вызов **Definition1[x]** в строчном формате возвращает определение объекта  $x$ , "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**.

```
In[2786]:= g = 70; L = {x, y, z, t, h, g, w, d, n, m}
Out[2786]= {x, y, z, t, h, 70, w, d, n, m}
In[2787]:= Definition[L[[6]]]
          Definition::ssle: Symbol, string, or HoldPattern[symbol] expected
          at position 1 in Definition[L[[6]]]. >>
Out[2787]=
In[2951]:= Definition1[x_] := Block[{a}, Off[Definition::ssle];
          a = ToString1[ToExpression["Definition[" <> ToString1[x] <> ""]]];
          On[Definition::ssle]; If[a == "", $Failed, a]]
In[2953]:= Map[Definition1, {L[[6]], L[[8]], 70, "g", Definition1}]
Out[2953]= {$Failed, "Null", $Failed, "g = 70",
          "Definition1[x_] := Block[{a}, Off[MessageName[Definition, \"ssle\"]];
          a = ToString1[ToExpression[StringJoin[\"Definition[\", ToString1[x], \"]\"];
          On[MessageName[Definition, \"ssle\"]]; If[a == \"\", $Failed, a]]"]
In[2961]:= Select[L, Definition[#] == Null &]
Out[2961]= {}
In[2962]:= Select[L, Definition1[#] == "Null" &]
Out[2962]= {y, z, t, h, w, d, n, m}
In[3403]:= SetAttributes[h, {Flat, Protected}]; Attributes[L[[5]]]
          Attributes::ssle: Symbol, string, or HoldPattern[symbol] expected
          at position 1 in Attributes[L[[5]]]. >>
Out[3403]= Attributes[L[[5]]]
In[3404]:= Attributes1[x_] := Block[{a}, Off[Attributes::ssle, Attributes::notfound];
          a = ToString1[ToExpression["Attributes[" <> ToString1[x] <> ""]]];
          On[Attributes::ssle, Attributes::notfound];
          If[a == "Attributes[" <> ToString1[x] <> "]", $Failed, ToExpression[a]]]
In[3405]:= SetAttributes[h, {Flat, Protected}]; Attributes1[L[[5]]]
Out[3405]= {Flat, Protected}
```

Подобно случаю с **Definition**, процедура **Attributes1** использует тот же механизм для расширения функции **Attributes** с учетом очевидных особенностей последней. Такой

прием может вполне успешно применяться и в ряде других случаев данного типа. С другой стороны, данные процедуры можно довольно эффективно использовать при программировании и других средств различного назначения. В качестве достаточно простого примера представим функцию **UnDef**, чей вызов **UnDef[x]** возвращает **True**, если символ  $x$  не определен, и **False** в противном случае. Фрагмент представляет код функции наряду с достаточно типичными примерами ее использования.

```
In[1443]:= UnDef[x_] := If[Off[Definition::notfound]; (ToString[x] // Definition1) ===
           "Null", On[Definition::notfound]; True, On[Definition::notfound]; False]
In[1444]:= x = 70; y = {a, b}; z = a + b; Map[UnDef, {t, h, x, y, z, 70}]
Out[1444]= {True, True, False, False, False, False}
In[1445]:= A[x_]; UnDef[x]] := Block[{}, x = 70]
In[1446]:= y := 2012; {A[y], A[g]}
Out[1446]= {A[2012], 70}
In[1447]:= L = {a, b, c, d, h, g, p, v, w}; Select[L, UnDef[#] &]
Out[1447]= {a, b, c, d, p, v, w}
In[1528]:= UnDef1[x_, y___] := Module[{a = {y}}, If[{y} != {} && UnDef[{y}][[1]],
           ToExpression[ToString[a[[1]]] <> " = Head1[" <> ToString1[x] <> "]"]; UnDef[x]]
In[1529]:= {UnDef1[a + b, n], UnDef1[a + b], n}
Out[1529]= {False, False, Plus}
In[1530]:= {UnDef1[70, m], UnDef1[70], m}
Out[1530]= {False, False, Integer}
In[1447]:= UnDef2[x_, y___] := Block[{a = {y}}, If[{y} != {} && UnDef[{y}][[1]],
           ToExpression[ToString[a[[1]]] <> " = Head1[" <> ToString1[x] <> "]"]; UnDef[x]]
In[1448]:= {UnDef2[a + b, n], UnDef2[a + b], n}
Out[1448]= {False, False, List}
In[1449]:= UnDef3[x_, y___] := Block[{a = {y}}, If[{y} != {} && UnDef[{y}][[1]],
           ToExpression[ToString[a[[1]]] <> " = Head1[" <> ToString1[x] <> "]"]; Head[x]]
In[1450]:= {UnDef3[a+b, n], UnDef3[a+b], n}
Out[1450]= {List, List, List}
```

Процедура **UnDef1** расширяет функцию **UnDef**, обеспечивая при вызове **UnDef1[x, y]** возврат через *второй* необязательный аргумент  $y$  значение **Head1[x]**; между тем, вызов **UnDef1[x]** эквивалентен вызову **UnDef[x]**. Отметим, что в ряде случаев процедурного программирования процедуры **UnDef**, **UnDef1** оказываются достаточно полезными.

Тут же вполне уместно еще раз проиллюстрировать принципиальное *различие* между процедурами типов **Module** и **Block** на примере процедур **UnDef1**, с одной стороны, **UnDef2** и **UnDef3**, с другой стороны. Так, если вызов процедуры **UnDef1** **Module**-типа выполняется корректно, то вызовы процедур **Block**-типа выполняются некорректно; причины этого здесь не рассматриваются. Поэтому тип процедуры следует выбирать достаточно осмотрительно, отдавая приоритет процедурам **Module**-типа. При этом,

в качестве *вложенных* процедур (*подпроцедур*) используются, как правило, процедуры **Module**-типа. В книге представлен ряд средств по работе с подпроцедурами, здесь мы представим довольно полезную процедуру, анализирующую процедуры на предмет присутствия в них *подпроцедур* типов **{Block, Module}**. Вызов процедуры **SubsProcs[x]** в общем случае возвращает вложенный список определений в строчном формате *всех* подпроцедур типов **{Block, Module}**, чьи определения находятся в теле процедуры *x*. Первый подсписок определяет подпроцедуры **Module**-типа, второй **Block**-типа. При наличии лишь **1** подписка возвращается *простой* список. При отсутствии такого типа подпроцедур вызов **SubsProcs[x]** возвращает пустой список – {}, тогда как на объекте *x*, отличном от процедуры, вызов процедуры **SubsProcs[x]** возвращается *невычисленным*. Следующий фрагмент представляет исходный код процедуры **SubsProcs** совместно с наиболее типичными примерами ее применения.

```
In[1606]:= SubsProcs[x_;/ ProcQ[x]] := Module[{a = DefOpt[ToString[x]], j, m = 1, n = 0,
                                             b = {" := Module[{" := Block[{" := 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[1607]:= P1[x_, y_] := Module[{Art, Kr, Gs, Vg, a}, Art[c_, d_] := Module[{b}, c + d];
    Vg[h_] := Block[{p = 70}, 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[1608]:= P1[42, 70]
Out[1608]= 665 799 220 982
In[1609]:= SubsProcs[P1]
Out[1609]= {"Art[c_, d_] := Module[{b}, c + d]", "Kr[n_] := Module[{}, n^2]",
    "Gs[z_] := Module[{}, x^3]", {"Vg[h_] := Block[{p = 70}, h^3 + p]"}
```

```
In[1610]:= Map[SubsProcs, {Locals, Attrib, Sin, BlockToMod}]
Out[1610]= {{}, {}, SubsProcs[Sin], {}}
In[1611]:= SubsProcs[Sin]
Out[1611]= SubsProcs[Sin]
```

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 наши процедуры **ProcBody**, **DefOpt**, **Map9** и **Map10**, рассмотренные несколько ранее. Более того, в связи с проблемой вложенности процедур выявляется весьма существенное *различие* между определениями *вложенных* процедур в среде пакетов *Mathematica* и *Maple*. Так, в среде пакета *Maple* определения *подпроцедур* допускают использование списков *формальных* аргументов, идентичных с главной процедурой, содержащей их; между тем, в среде пакета *Mathematica* такое совмещение недопустимо, вызывая в процессе вычисления определения главной процедуры ошибочные ситуации, как это достаточно наглядно иллюстрирует вторая часть нижеследующего фрагмента. Вообще говоря, именно это обстоятельство вызывает вполне определенные неудобства, ибо требует повышенной внимательности в процессе программировании *вложенных* процедур. В определенной степени аналогичная ситуация возникает и при пересечениях списков формальных аргументов главной процедуры и локальных переменных подпроцедур, тогда как это допустимо в программной среде пакета *Maple*, что довольно наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```
> A := proc(x::posint, y::integer) local Kr; Kr := proc(x,y) x+y end proc; Kr(x,y)^2 end proc;
> A(42, 70), Kr(75, 450);
12544, Kr(75, 450)
> A1:=proc(x::posint, y::integer) local a; Kr := proc(x, y) x+y end proc; Kr(x, y)^2 end proc;
Warning, `Kr` is implicitly declared local to procedure `A1`
> A1(42, 70), Kr(75, 450);
12544, Kr(75, 450)
> A2:=proc(x::posint,y::integer) global Kr; Kr:=proc(x,y) x+y end proc; Kr(x,y)^2 end proc;
> A2(42, 70), Kr(75, 450);
12544, 525
> A3 := proc(x,y) local Gs; Gs:=proc(m,n) local x; `(m,n,x) end proc; Gs(x,y)^2 end proc;
> A3(42, 70), Gs(75, 450);
(112 + x)^2, Gs(75, 450)
In[1436]:= A[x_Integer, y_] := Module[{Kr}, Kr[x_, y_] := Module[{}, x + y]; Kr[x, y]^2]
RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule
A[x_Integer, y_]>Module[{Kr}, Kr[x_, y_] := Module[{}, x + y]; Kr[x, y]^2]. >>
RuleDelayed::rhs: Pattern y_ appears on the right-hand side of rule
A[x_Integer, y_]>Module[{Kr}, Kr[x_, y_] := Module[{}, x + y]; Kr[x, y]^2]. >>
In[1437]:= {A[42, 70], Kr[75, 450]}
Pattern::patvar: First element in pattern Pattern[42, _] is not a valid pattern name. >>
```

```

Pattern::patvar: First element in pattern Pattern[70,_] is not a valid pattern name. >>
Out[1437]= {Kr$2734[42, 70]^2, Kr[75, 450]}
In[1438]:= A1[x_Integer, y_] := Module[{}, Kr[x_, y_] := Module[{}, x + y]; Kr[x, y]^2]
RuleDelayed::rhs: Pattern x_ appears on the right-hand side of rule
A1[x_Integer, y_] := Module[{}, Kr[x_, y_] := Module[{}, x + y]; Kr[x, y]^2]. >>
RuleDelayed::rhs: Pattern y_ appears on the right-hand side of rule
A1[x_Integer, y_] := Module[{}, Kr[x_, y_] := Module[{}, x + y]; Kr[x, y]^2]. >>
In[1439]:= {A1[42, 70], Kr[75, 450]}
Pattern::patvar: First element in pattern Pattern[42,_] is not a valid pattern name. >>
Pattern::patvar: First element in pattern Pattern[70,_] is not a valid pattern name. >>
Out[1439]= {Kr[42, 70]^2, Kr[75, 450]}
In[1440]:= A2[x_Integer, y_] := Module[{y}, Kr[m_, n_] := Module[{x}, x + y]; Kr[x, y] + y]
In[1441]:= A2[42, 70]
Module::lvsym: Local variable specification {70} contains 70, which is not a
symbol or an assignment to a symbol. >>
Out[1441]= Module[{70}, Kr[m$_, n$_] := Module[{42}, 42 + 70]; Kr[42, 70] + 70]
In[1442]:= A2[x_Integer, y_] := Module[{y}, Kr[m_, n_] := Module[{x}, x + y]; Kr[x, y] + y]
In[1443]:= SubsProcs[A2]
Out[1443]= {"Kr[m_, n_] := Module[{x}, x + y]}
In[1444]:= A2[42, 70]
Module::lvset: Local variable specification {70=75} contains 70=75, which is an
assignment to 70; only assignments to symbols are allowed. >>
Out[1444]= Module[{70 = 75}, Kr[m$_, n$_] := Module[{42}, 42 + 70]; Kr[42, 70]^2 + 70]
In[1445]:= A3[x_Integer, y_] := Module[{a}, Kr[m_, n_] := Module[{x}, x + y]; Kr[x, y] + y]
In[1446]:= A3[42, 70]
Module::lvsym: Local variable specification {42} contains 42, which is not a
symbol or an assignment to a symbol. >>
Out[1446]= 70 + Module[{42}, 42 + 70]

```

Как следует из последних 2 примеров предыдущего фрагмента рассмотренная выше процедура **SubsProcs** может вполне успешно применяться и к процедурам, которые содержат подпроцедуры типов **{Block, Module}**, при условии непустого пересечения списка формальных аргументов главной процедуры и списка *локальных* переменных как главной процедуры, так и ее подпроцедур. Тогда как вызовы таких процедур на фактических аргументах вызывают *ошибочные* ситуации, обусловленные указанными выше причинами, как иллюстрируют последние примеры предыдущего фрагмента.

Итак, в программной среде *Mathematica* механизм подпроцедур и процедур в целом представляется нам несколько менее развитым по отношению к среде пакета *Maple*, располагая при этом рядом весьма положительных черт. Прежде всего, не взирая на меньшую *визуальную* структурированность определений процедур/функций, которая обуславливает их более низкую читабельность, в целом, такие средства относительно пакета *Maple* более пролонгированы по отношению к релизам *Mathematica*, а также более эффективны по использованию основных ресурсов компьютера.

## 6.5. Функциональные конструкции в среде пакета 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 детальнее 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 именующихся 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 преемников;
- *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 перегрузки 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 надёжных, 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 функциональному программированию;

– *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, 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 их результата;

- 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 можно 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, 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 л-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. 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. Выше обсуждались свойства такого типа функций. В *Mathematica* чистые функции определяются следующим образом:

**Function**[*x*, *Тело*] – чистая функция от одного формального аргумента *x*

**Function**[{*x1*, *x2*, ...}, *Тело*] – чистая функция от формальных аргументов *x1*, *x2*, ...

*Тело*& – чистая функция, чьи формальные аргументы определяются как {# | #1, #2, ...}

Итак, пользователь, знакомый с формальной логикой или языком программирования *Lisp* обработки списков, легко распознает в чистых функциях пакета *Mathematica* *λ*-выражения или *непоименованные* функции. При этом, понятие *чистых функций* весьма близко к понятию математических *операторов*. Итак, в общем случае для определения формальных аргументов *чистой* функции допускается использование *#-заместителей* аргументов следующего формата, имеющих нижеследующий смысл, а именно:

# первый формальный аргумент чистой функции;

#*n* *n*-й формальный аргумент чистой функции;

## последовательность всех формальных аргументов чистой функции;

##*n* последовательность формальных аргументов чистой функции, начиная с *n*-го.

Индикатором *чистой* функции в *Mathematica* служит амперсанд (&), именно наличие его в выражении определяет его в качестве чистой функции. Следующий достаточно простой фрагмент наглядно иллюстрирует применение данных заместителей:

```
In[2116]:= {Plus[#2 &][a, b, c], Plus[## &][a, b, c, d, f], Plus[##4 &][a, b, c, d, g, h, p]}
Out[2116]= {b, a + b + c + d + f, d + g + h + p}
In[2117]:= {Map[##^2 &, {a, b, c, d}], Map[#^2 &, {a + b + c + d}]}
Out[2117]= {{a^2, b^2, c^2, d^2}, {(a + b + c + d)^2}}
In[2118]:= {Nest[##^2 &, {a, b, c, d}, 3], Nest[#^2 &, {a + b + c + d}, 3]}
Out[2118]= {{a^8, b^8, c^8, d^8}, {(a + b + c + d)^8}}
In[2119]:= G[##2, ##4] &[x, y, z, h, d, t]
Out[2119]= G[y, z, h, d, t, h, d, t]
In[2119]:= #2 &[a, b, c] + ##2 &[x, y, z, h, d, w, v]
Out[2119]= b + d + h + v + w + y + z
```

В общем случае для вызова функции требуется определение в точке вызова ее имени и фактических аргументов, тогда как механизм чистых функций позволяют задавать функции, которые возможно применять к аргументам без явного указания имен для функций, как это иллюстрирует следующий достаточно прозрачный фрагмент:

```
In[1840]:= Art := Function[{x, y}, x*Sin[y]]
In[1841]:= Kr := (#1^2 + #2^4) &
In[1842]:= {Art[69, 42.69], Kr[16, 23]}
Out[1842]= {-66.3414, 280097}

In[1843]:= {Function[{x, y}, x*Sin[y]][69, 42.69], (#1^2 + #2^4) &[16, 23]}
Out[1843]= {-66.3414, 280097}
In[1844]:= Map[H[#]*G[#] + w, {x, y, z}]
Out[1844]= {(w + G[#1] H[#1])[x], (w + G[#1] H[#1])[y], (w + G[#1] H[#1])[z]}
```

Во втором случае мы имеем дело с т.н. *непоименованными функциями* в общепринятой терминологии. Однако, пакет *Mathematica* не располагает стандартными средствами для тестирования поименованных функций и эту достаточно важную задачу решает достаточно простая процедура **FunctionQ[x]**, исходный код которой наряду с рядом примеров применения представляет следующий достаточно прозрачный фрагмент.

```
In[142]:= F := Function[{x, y, z, h}, Sqrt[x^2 + y^2 + z^2 + h^2]]
In[143]:= F[42, 69, 75, 450]^2
Out[143]= 214650

In[144]:= Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &
In[145]:= {Art[42.69, 450.75], Kr[75, 450]}
Out[145]= {-42.5897, 41 006 255 625}

In[146]:= RANS := 2012; IAN[x_] := Module[{}, x^2]; Sv := Block[{}, 45]
In[147]:= FunctionQ[x_] := Module[{a, b}, a = ToString[InputForm[x]];
    If[StringFreeQ[a, "#1"] == False && StringTake[a, {-3, -1}] == " & ", True,
        a = Quiet[Check[SubsDel[ToString[InputForm[Definition[x]]],
            "" <> ToString[x] <> "", {"[", ", ", "-1", Return[False]]];
        b = ToString[x] <> " := Function[{";
        Quiet[Check[If[b == StringTake[a, StringLength[b]] ||
            StringTake[a, -2] == "& " &&
            x <> " := " == StringTake[a, StringLength[x] + 4], True, False], False]]];
In[148]:= Map[FunctionQ, {"F", "Kr", "Art", "RANS", "IAN", Sv, IAN, #1^2 + #2^2 &}]
Out[148]= {True, True, True, False, False, False, False, True}
```

Из представленного фрагмента следует, в качестве *фактического* аргумента функции **FunctionQ** выступает или имя тестируемого объекта в строчном формате (*иначе вызов функции возвращается невычисленным с упрощением аргумента x*), или чистая функция в кратком формате, т.е. в форме *Тело&*. Если тестируемый *x*-объект является *функцией*, то вызов **FunctionQ[x]** возвращает значение **True**, в противном случае – **False**.

Ниже представлена тестирующая функция *PureFuncQ[F]*, возвращающая **True**, если *F* является чистой функцией в форме *Тело&*, и **False** в противном случае. Тогда как две процедуры **Map5** и **Map6** расширяют действие стандартной функции **Map** на случаи произвольной и чистой функции *Тело&* с произвольным числом аргументов. Затем иллюстрируется применение обоих процедур на функции *S[x, y]* и ее *&*-аналоге.

```
In[1954]:= PureFuncQ[F_] := StringTake[ToString[F], {-3, -1}] == " &" &&
! StringFreeQ[ToString[F], "#"]
In[1955]:= Map5[F_, L_ /; ListListQ[L]] := Module[{a, b = Length[L], c, d = {},
h = ToString[F], k = 1, a = Map[ToString, Map[F, L]];
For[k, k <= b, k++, c = StringTake[a[[k]], {1, -3}] <> ""];
d = Append[d, StringReplace[c, h <> "{" -> h <> "["]]]; ToExpression[d]]
In[1956]:= Map6[F_ /; PureFuncQ[F], L_ /; ListListQ[L]] := Module[{a, b = Length[L],
c = Length[L[[1]]], d = {}, h, k = 1, p}, h = StringTake[ToString[F], {1, -4}];
For[k, k <= b, k++, a = {}]; d = Append[d, StringReplace[h,
Flatten[{For[p = 1, p <= c, p++, a = Append[a, "#" <> ToString[p] ->
ToString[L[[k]]][p]]], a}][[2 ;; -1]]]]; ToExpression[d]]
In[1957]:= S[x_, y_] := a[x]*b[y] - c[x, y]
In[1958]:= Map5[S, {{x1, y1}, {x2, y2}, {x3, y3}}]
Out[1958]= {a[x1] b[y1] - c[x1, y1], a[x2] b[y2] - c[x2, y2], a[x3] b[y3] - c[x3, y3]}
In[1959]:= Map6[a[#1]*b[#2] - c[#1, #2] &, {{x1, y1}, {x2, y2}, {x3, y3}}]
Out[1959]= {a[x1] b[y1] - c[x1, y1], a[x2] b[y2] - c[x2, y2], a[x3] b[y3] - c[x3, y3]}
```

Вызов процедуры **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**[ ... **V**[*a*]]] ... ], **F**[**G**[**H**[ ... **V**[*b*]]] ... ], **F**[**G**[**H**[ ... **V**[*c*]]] ... ], ..., **F**[**G**[**H**[ ... **V**[*v*]]] ... ]}

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Следующий фрагмент представляет исходный код процедуры **Map7** с примерами ее применения.

```
In[1143]:= Map7[x_, L_ /; ListQ[L]] := Module[{a = {x}, b = "", c = {}, d = Length[L], k = 1,
h = Length[{x}]}, If[DeleteDuplicates[Map[SymbolQ, a]] != {True},
Return[Defer[Map7[x, L]], For[k, k <= h, k++, b = b <> ToString[a[[k]]] <> "["]];
For[k = 1, k <= d, k++, c = Append[c, b <> ToString1[L[[k]]] <>
StringMultiple["", h]]]; ToExpression[c]]
In[1144]:= Map7[F, G, H, {a, b, c, d}]
Out[1144]= {F[G[H[a]]], F[G[H[b]]], F[G[H[c]]], F[G[H[d]]]}
In[1145]:= Map7[Sin, Sqrt, N, {15, 23, 44, 64, 69}]
Out[1145]= {-0.667905, -0.996521, 0.342958, 0.989358, 0.899295}
In[1146]:= Map7[Sin, Sqrt, 75, {15, 23, 44, 64, 69}]
Out[1146]= Map7[Sin, Sqrt, 75, {15, 23, 44, 64, 69}]
```

Таким образом, в целом ряде случаев короткий формат чистых функций оказывается весьма эффективным приемом программирования и определения функций, прежде всего в тех случаях, где целесообразно использовать непоименованные функции. Так, короткий формат чистых функций позволяет упрощать многие *определения* объектов.

Как отдельный объект можно рассматривать функцию **Compile**, которая служит для компиляции функций с числовыми фактическими аргументами. Функция **Compile** с 4 форматами кодирования, из которых наиболее употребительный имеет вид:

**Compile[{x1, t1}, {x2, t2}, ..., {xn, tn}], Выражение**

создает скомпилированную функцию, вычисляющую *Выражение* в предположении, что числовые аргументы  $x_j$  имеют тип  $t_j \in \{\text{Integer}, \text{Real}, \text{Complex}, \text{True}, \text{False}\}$ ; более того, скомпилированная функция работает с машинными числами, а именно: *Real* – приближенные действительные числа с машинной точностью (по умолчанию), *Integer* – целые машинного размера, *Complex* – приближенные комплексные числа с машинной точностью, *{True, False}* – логическая переменная. Более детально с функцией **Compile** можно познакомиться в соответствующей литературе либо в справке по пакету. Здесь же представлены средства тестирования данного типа объектов, в качестве первого из которых выступает процедура **FunctionQ1[x]**, обобщающая процедуру **FunctionQ[x]** на объекты  $x$ , генерируемые функцией **Compile**, и возвращая на них значение *True*, и *False* в противном случае. Ниже представлены исходный код процедуры и примеры.

```
In[2877]:= Clear[V, Art, Kr, GS, Sv, S]; V := Compile[{x, _Real}, {y, _Real}, (x + y)^2];
          Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &;
          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[2878]:= FunctionQ1[x_;/ SymbolQ[x]] := Module[{a = StringLength[x],
          b = ToString[InputForm[DefFunc[x]]]},
  If[FunctionQ[x] || ToExpression[StringReplace[StringTake[ToString[
  DeleteDuplicates[Map4[PrefixQ, Map[ToString1, {"CompiledFunction",
  "Compile"}], ToString1[StringTake[b, {a + If[PrefixQ[x <> "=", b], 4, 5], -1]]]]],
  {2, -2}], ", " -> " || "]], True, False]]

In[2879]:= Map[FunctionQ1, {"V", "S", "Art", "Kr", "Pi", "75", "GS", "Sv"}]
Out[2879]= {True, True, True, True, False, FunctionQ1["75"], False, False}

In[2880]:= Qfunction[x_;/ SymbolQ[x]] := Module[{a = StringLength[x],
          b = ToString[InputForm[DefFunc[x]]], c},
  If[Quiet[ProcQ[ToExpression[x]]], False,
  If[StringSplit[ToString[InputForm[DefFunc[x]]], "="][[1]] ==
  "Attributes[" <> x <> "]", x, c = StringSplit[b, {"=", ":="}];
  If[c[[1]] == x, FunctionQ1[x],
  c = {StringSplit[StringTake[c[[1]], {a + 2, -2}], ", ", c[[2]]];
  c = DeleteDuplicates[Map4[StringFreeQ, Map[ToString1, c[[1]], ToString1["_"]]];
  If[FunctionQ[x] || FunctionQ1[x] || c == {False}, True, False]]]]

In[2881]:= Map[Qfunction, {"V", "S", "Art", "Kr", "Pi", "42.69", "GS", "Sv"}]
Out[2881]= {True, True, True, True, "Pi", Qfunction["42.69"], True, True}
```

Тогда как процедура **Qfunction[x]** является наиболее общим средством тестирования объектов  $x$  функционального типа, возвращая на функции  $x$  значение *True*, и *False* в противном случае. В качестве фактического  $x$ -аргумента выступает *символ* в строчном

формате, в противном случае вызов процедуры возвращается невычисленным. Выше представлен исходный код процедуры **Qfunction** с примерами ее использования.

*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  $G[x\_y\_ \dots] := \langle \text{Тело} \rangle$ , 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 формате  $\langle \text{Тело} \rangle \&$ . Следующий весьма простой фрагмент иллюстрирует сказанное:

```
In[680]:= G[x_] := x^2; a := {}; For[k = 0, k < 21, k++, a = Append[a, G[k]]]; a
Out[680]= {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
In[681]:= #1^2 &[14]
Out[681]= 196
In[682]:= a := {}; For[k = 0, k < 21, k++, a = Append[a, #1^2 &[k]]]; a
Out[682]= {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

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  $\text{Map}[F, E] \equiv \text{Mapp}[F, E]$ .

В качестве формальных аргументов стандартной функции  $\text{Map}[F, g]$  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 **Mapp**, обобщающей стандартную функцию *Map* пакета *Mathematica* на большее число аргументов, наряду с примерами ее использования.

```
In[1034]:= 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 == {}, Return[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[1035]:= Mapp[F, {a, b, c}, x, y, z]
Out[1035]= {F[a, x, y, z], F[b, x, y, z], F[c, x, y, z]}
In[1036]:= Mapp[F, a + b + c, x, y, z]
Out[1036]= F[a, x, y, z] + F[b, x, y, z] + F[c, x, y, z]
In[1037]:= Mapp[F, (m + n)/(g + h) + Sin[x], a, b, c]
Out[1037]= F[(m + n)/(g + h), a, b, c] + F[Sin[x], a, b, c]
In[1038]:= Mapp[StringPosition, {"11123", "33234"}, {"2", "3", "23"}]
```

```

Out[1038]= {{{4, 4}, {4, 5}, {5, 5}}, {{1, 1}, {2, 2}, {3, 3}, {3, 4}, {4, 4}}}
In[1039]:= Mapp[StringReplace, {"812345265", "72345957"}, {"2" -> "V", "5" -> "G"}]
Out[1039]= {"81V34GV6G", "7V34G9G7"}
In[1040]:= Map[F, {{a, b}, {c, d, e}}]
Out[1040]= {F[{a, b}], F[{c, d, e}]}
In[1041]:= Mapp[F, {{a, b}, {c, d, e}}, x, y, z]
Out[1041]= {F[{a, b}, x, y, z], F[{c, d, e}, x, y, z]}
In[1042]:= Mapp[ProcQ, {Sin, ProcQ, Mapp, ExpFunc2, SysFuncQ}]
Out[1042]= {False, True, True, True, True}

```

Отметим, что реализация алгоритма процедуры **Mapp** 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 **Mapp** в ряде случаев достаточно существенно упрощает программирование различных задач.

Несколько подробнее остановимся на определении пользовательских функций (**ПФ**), применение которых почти полностью идентично случаю встроенных функций, но с областью действия лишь в рамках текущего сеанса работы с пакетом, в котором было вычислено их определение. Перед своим *первым* использованием в сеансе **ПФ** должна быть предварительно определена и вычислена (*активирована*). Определяющая такую **ПФ** конструкция имеет один из *двух* наиболее используемых форматов кодирования:

1) *Id-функции*[<Шаблон аргументов>] := <Тело функции>

2) *Id-функции*[<Шаблон аргументов>] := {<Список аргументов{, |}><Тело функции>}

В качестве *Id-функции* используется любой *буквозависимый* идентификатор. При этом, шаблон аргументов (**ША**) во многом определяет тип **ПФ** и возвращаемого результата. Отметим здесь наиболее используемые **ША** и типы определяемых ими функций:

G[x_]	-	с одним <i>x</i> -аргументом, допускающим значения любого типа
G[x_, y_]	-	с двумя аргументами, допускающими значения любого типа
G[x_, y_, ...]	-	с любым конечным числом формальных аргументов любого типа
G[x_List]	-	используется <i>список</i> в качестве значения аргумента G-функции
G[x_String]	-	строка
G[x_Integer]	-	целое
G[x_Complex]	-	комплексное
G[x_Number]	-	число любого типа

---

$G[x\_Real]$	- $--$	действительное с плавающей точкой	$--$
$G[x\_;/\text{ЛУ}]$	-	для $x$ -аргумента допускается значение, для которого некоторое логическое условие ( <i>ЛУ</i> ) принимает значение <b>True</b> ( <i>Истина</i> )	
$G[x\_ША;/\text{ЛУ}]$	-	для $x$ -аргумента с допустимым типом, соответствующим <i>ША</i> , выбирается такое значение, для которого логическое условие ( <i>ЛУ</i> ) принимает значение <b>True</b> .	

---

Как следует из приведенного перечня типов *ША* (с учетом возможности очень широкого их сочетания) пакет допускает определение различных типов фактических значений для формальных аргументов *ПФ*, а также логических условий для них, позволяющих организовывать некие *логические* фильтры для передаваемых фактических значений аргументам вызываемых *ПФ*. Аппарат *ША* является довольно мощным средством для расширения возможностей пользователя по определению его собственных функций в среде пакета и подробнее с ним можно ознакомиться в справке. Подобно встроенной функции *ПФ* возвращает результат вычисления *Тела* функции на переданных при ее вызове фактических значениях *формальных* аргументов. Будем говорить, фактические значения соответствуют *ША* функции, если они соответствуют типам его *формальных* аргументов и удовлетворяют *логическим* условиям (*при их наличии*) их фильтров. При наличии такого соответствия *ПФ* возвращает искомый результат своего вычисления, в противном случае возвращается невычисленная конструкция формата  $Id[\#ША]$ , где  $\#ША$  – список фактических значений аргументов в *Output*-формате пакета.

При *возвращении* результата вычисления *ПФ* пакет определяет его тип в соответствии с типами переданных ей фактических значений для формальных аргументов и *самой* сущностью вычисления, определяемой телом функции. Если же тип для *формального* аргумента в определении *ПФ* не задан, пакет допускает для него фактические *данные* *любого* допустимого типа. Сказанное иллюстрирует весьма простой пример, который определяет *GS*-функцию и производится два ее вызова с различными фактическими значениями для ее формальных аргументов, а именно:

```
In[1745]:= GS[x_Integer, y_;/ MatrixQ[y], z_String] := y^(x + StringLength[z])
In[1746]:= GS[6.8, {1, 2, 3}, "RANS_IAN"]
Out[1746]= GS[6.8, {1, 2, 3}, "RANS_IAN"]
In[1747]:= GS[2, {{68, 63}, {14, 21}}, "IAN_72"]
Out[1747]= {{457163239653376, 248155780267521}, {1475789056, 37822859361}}
```

В первом случае вызов функции *GS* возвращается невычисленным в *Output*-формате (из-за нарушения указанного выше соответствия для формального  $x$ -аргумента функции), а во втором случае возвращается вложенный список –  $(2 \times 2)$ -матрица, которая является результатом вычисления 8-й степени  $y$ -матрицы.

*Первый* тип формата определения *ПФ* довольно прост, однако *тело* функции должно быть представлено единой вычисляемой конструкцией, что в целом ряде случаев не совсем удобно либо просто невозможно. Если *тело* функции содержит *неопределенные* переменные либо константы, то результат возвращается в символьном относительно их виде. *Второй* формат определения *ПФ* отличается от *первого* лишь правой частью, которая представляет собой списочную структуру, содержащую список формальных

аргументов (*необязателен*) и само тело функции. Вторым формат определения **ПФ** хоть и несколько сложнее первого, но предоставляет значительно большие возможности в определении функций. Прежде всего, в качестве тела функции может быть не только одно выражение, но и их последовательность (*разделенная точкой с запятой*), результат вычисления последнего из которой и возвращает **ПФ** (*составной оператор*). Более того, варьируя состав списка аргументов (*включая изменение формата разделителей*), вполне можно получать в нужном виде возвращаемый функцией результат, включая в него в нужной комбинации также фактические значения аргументов на момент вызова **ПФ**. В целях упрощения и лучшего понимания такого механизма в качестве иллюстрации сказанного приведем простой фрагмент *многократного* определения **RANS**-функции согласно *второго* формата и результаты ее вызова, из которого *вся* картина становится весьма прозрачной и особых дополнительных пояснений не требует:

```
In[1511]:= RANS[x_, n_Integer, y_List] := {{x, n, y}, a*x^(n + Length[y])}
In[1512]:= RANS[2, 3, {68, 63, 43}]
Out[1512]= {{2, 3, {68, 63, 43}}, 64 a}
In[1513]:= RANS[x_, n_Integer, y_List] := {x, n, y, a*x^(n + Length[y])}
In[1514]:= RANS[2, 3, {68, 63, 43}]
Out[1514]= {2, 3, {68, 63, 43}, 64 a}
In[1515]:= RANS[x_, n_Integer, y_List] := {x, n, y; a*x^(n + Length[y])}
In[1516]:= RANS[2, 3, {68, 63, 43}]
Out[1516]= {2, 3, 64 a}
In[1517]:= RANS[x_, n_Integer, y_List] := {x, n; y; a*x^(n + Length[y])}
In[1518]:= RANS[2, 3, {68, 63, 43}]
Out[1518]= {2, 64 a}
In[1519]:= RANS[x_, n_Integer, y_List] := {x; n; y; a*x^(n + Length[y])}
In[1520]:= RANS[2, 3, {68, 63, 43}]
Out[1520]= {64 a}
In[1521]:= RANS[x_, n_Integer, y_List] := {a*x^(n + Length[y])}
In[1522]:= RANS[2, 3, {68, 63, 43}]
Out[1522]= {64 a}
In[1523]:= RANS[x_, n_Integer, y_List] := {F = a*x^(n + Length[y]); d = Sqrt[f]; R = d*F}
In[1524]:= RANS[2, 3, {68, 63, 43}]
Out[1524]= {64 a Sqrt[f]}
In[1525]:= {F, d, R}
Out[1525]= {64 a, Sqrt[f], 64 a Sqrt[f]}
```

Наряду с рассмотренными при определении пользовательской функции в качестве *ее* *тела* могут выступать и другие конструкции языка **Mathematica** (*блоки, модули и др.*), о *первых* двух из которых речь шла выше. То же относится и к типам используемых **ША** при определении **ПФ**. Для определения **ША** и фильтров для **ПФ** можно использовать достаточно общего вида конструкции, например, следующего вида, а именно:

```
In[2]:= H[x_/; Floor[x/43] >= 68, y_/; Module[{A=14, S=43, z}, z=y/S; N[z]] <= 800] := x*y
```

```
In[3]:= H1[x_ /; Floor[x/43]>=68, y_ /; Module[{A=14, S=43, z}, z=x/A+y/S; N[z]]<=800]:=x*y
In[4]:= {{H[3000, 280], H1[72, 420]}, {H[300, 2011], H1[4268, 10]}}
Out[4]= {{840000, H1[72, 420]}, {H[300, 2011], 42680}}
```

Однако, следует иметь в виду, что *логическое условие (ЛУ)* для формального аргумента пользовательской функции может включать только ранее определенные значения и сам такой аргумент. Таким образом, до сих пор сказанное по функциям пользователя можно рассматривать только как вводную часть, детализация по которой может быть найдена в справке по пакету и в приведенной литературе по пакету *Mathematica*.

После создания определения *ПФ* его следует вычислить одним из принятых в пакете способов, чтобы получить к данной функции доступ. Однако, вновь созданная *ПФ* не имеет ни опций, ни атрибутов, что не только отличает ее от встроенных функций, но может приводить к *непредсказуемым* последствиям при вызове ее в текущем сеансе, т.к. идентификатор *ПФ*, в частности, может быть незащищенным от непосредственного участия в вычислениях и переопределения. Поэтому, после определения функции ей рекомендуется присвоить наиболее употребительные для встроенных функций ядра атрибуты *{Protected, Listable}*; при этом, специфика *ПФ* может и не допускать *второго* атрибута. Тогда как *первый* атрибут обеспечивает защиту *ПФ* от ее переопределения. Сказанное иллюстрирует следующий достаточно простой фрагмент, а именно:

```
In[483]:= GS[x_List, y_ /; IntegerQ[y]] := First[{z=Length[x]; R=x+IntegerDigits[y]; R^z}]
In[484]:= GS[{69, 64, 44}, 450]
Out[484]= {389017, 328509, 85184}
In[485]:= z = 3; R = {2, 4, 6}; {Options[GS], Attributes[GS]}
Out[485]= {{}}, {}
In[486]:= SetAttributes[GS, {Protected, Listable}]; Attributes[GS]
Out[486]= {Listable, Protected}
In[487]:= GS = 2012; N[Sqrt[GS]]
Set::wrsym: Symbol GS is Protected. >>
Out[487]= Sqrt[GS]
```

Из данного фрагмента следует, у вновь определенной *ПФ GS* отсутствуют как опции, так и атрибуты. По *SetAttributes*-функции ядра ей присваиваются атрибуты *{Listable, Protected}*, что, в частности, гарантирует невозможность как ее переопределения, так и использования *GS* в качестве переменной в вычислениях. В противном случае вполне допустимой могла бы быть конструкция вида *GS = 2012*, что полностью аннулировало бы определение *GS*-функции на весь оставшийся период работы с пакетом. Данный прием можно использовать для дезактивации любой как *встроенной*, так и *ПФ* (прежде отменив ее *Protected*-атрибут при его наличии) в текущем сеансе работы с пакетом, что позволяет обходиться без его *перезагрузки*. Из представленного фрагмента следует, что все *промежуточные* переменные, используемые в определении *ПФ*, будут *глобальными* (т.е. актуальны на весь период работы с пакетом), однако их атрибуты не изменяются с переопределением атрибутов самой *ПФ*. Между тем, рассмотренные выше средства (блоки, модули) *Math*-языка пакета позволяют локализовать внутренние переменные *тела* пользовательской функции.

Между тем, следует отметить, что механизм атрибутов *Math*-языка пакета имеет ряд весьма существенных особенностей при его использовании в процедурном режиме, а именно. Попытка присвоения в процедуре атрибута переменной *x* через локальную переменную не дает результата, как наглядно иллюстрирует простая процедура **Pr1** следующего фрагмента. Для устранения этой ситуации используется искусственный прием, приведенный в процедуре **Pr3**, суть которого довольно легко усматривается из ее исходного кода. Тогда как механизм присвоения атрибутов пакета *Maple* работает корректно на процедурном уровне и для глобальных, и для локальных переменных, как иллюстрирует пример простой процедуры **Pr3** в программной среде *Maple*.

```
In[1612]:= ClearAllAttributes["x"]
In[1613]:= Pr[x_ /; SymbolQ[x]] := Module[{a = x}, SetAttributes[a, Protected]]
In[1614]:= Pr["x"]; x = 70
Out[1614]= 70
In[1615]:= Clear[x]; x = 70;
In[1616]:= Pr1[x_ /; SymbolQ[x]] := Module[{a = x}, ToExpression["SetAttributes[" <>
ToExpression["ToString[a] <> ", Protected]]]]
In[1617]:= Pr1["x"]; x = 75
Set::wrsym: Symbol x is Protected. >>
Out[1617]= 75
In[1618]:= x
Out[1618]= 70
In[1619]:= ClearAllAttributes["x"]; Clear[x]
In[1620]:= Pr2[x_ /; SymbolQ[x]] := Module[{}, SetAttributes[x, Protected]]
In[1621]:= Pr2[x]; x = 70
Set::wrsym: Symbol x is Protected. >>
Out[1621]= 70
In[1622]:= x
Out[1622]= x
> Pr3 := proc(x) local a; a := x; setattribute(a, protected) end proc:
> Pr3(x): x := 450:
Error, attempting to assign to `x` which is protected
```

Отметим, что подобный подход требуется также и в ряде других случаев применения вызовов стандартных функций на локальных переменных в теле процедуры.

Определенная в текущем сеансе *ПФ* остается доступной *любому* документу в течение текущего сеанса с пакетом; при этом, обращение к *ПФ* корректно в любом доступном месте документа сразу же после вычисления определения функции, т.е. определение *ПФ* носит глобальный характер. Если в текущем сеансе работы с пакетом определено несколько *ПФ* с одинаковыми идентификаторами *Id*, но различными *ШАj* (т.е. есть несколько вычисленных определений  $\{Id[ША1], Id[ША2], Id[ША3], \dots\}$ ), то пакет реагирует следующим образом при вызове *Id*-функции, если передаваемые для ее формальных

аргументов фактические значения таковы, что:

(1) не согласуются ни с одним *ШАj*, то вызов функции возвращается невычисленным в *Output*-формате пакета; в частности, для функции предыдущего фрагмента имеем  $GS["AVZ"] \Rightarrow GS["AVZ"]$ ;

(2) согласуются лишь с одним из указанных *ШАj*, возвращается результат вычисления *Id*-функции, соответствующей данному шаблону;

(3) согласуется с несколькими *ШАj*, возвращается результат вычисления *Id*-функции, определение которой в текущем сеансе с пакетом было вычислено последним.

Данный аспект реализации механизма вызова *ПФ* является одной из отличительных черт ядра пакета. Следующий простой фрагмент иллюстрирует вышесказанное:

```
In[96]:= AR[x_]:=x^2; AR[x_List]:=x; AR[x_String]:=StringLength[x]; AR[x_Integer]:=x^3
In[97]:= {AR[19.42], AR[{66, 64, 44, 15, 23}], AR["RANS - IAN - 2012"], AR[69]}
Out[97]= {377.136, {66, 64, 44, 15, 23}, 17, 328 509}
In[98]:= AR[x_] := 2*x + 69; AR[x_String] := N[Sqrt[StringLength[x]]]
In[99]:= {AR[19.42], AR["RANS - IAN - 2012"]}
Out[99]= {107.84, 4.12311}
```

Во фрагменте приведены определения *AR*-функции пользователя с различными *ША* и результаты ее вызова на различных типах фактических значений аргументов. Затем произведено *переопределение* *AR*-функции. Следовательно, пакет рассматривает *ПФ* с одинаковыми идентификаторами, но с различными *ША* в качестве различных.

В отличие от ряда известных средств пакет допускает определение рекурсивных *ПФ*, как это наглядно иллюстрирует следующий весьма простой фрагмент, а именно:

```
In[1200]:= Sum6[0] := 0; Sum6[n_Integer] := Sum6[n - 1] + n
In[1201]:= Sum6[69]; Sum6[2012]
$RecursionLimit::reclim: Recursion depth of 256 exceeded. >>
Out[1201]= 2012 + (2011 + (2010 + (2009 + (2008 + (2007 + (2006 + (2005 + (2004 .... ))))))))
In[1202]:= $RecursionLimit = 3000; Sum6[2012]
Out[1202]= 2 025 078
```

В данном фрагменте определяется рекурсивная *Sum6*-функция (с начальным условием рекурсии *Sum6[0]=0*) для вычисления суммы *n* целых чисел. Так как пакетом для числа рекурсий «по умолчанию» установлено значение предопределенной переменной ядра пакета *\$RecursionLimit=256*, то эту переменную следует переопределить, чтобы иметь возможность производить рекурсии требуемой глубины, иначе возникает ошибочная ситуация с возвратом соответствующей диагностики, как иллюстрирует фрагмент.

Бывший в *первых* релизах пакета оператор *HeadCompose[A, B, C, ...]* (*в настоящее время не документированный*) возвращает композицию идентификаторов в форме *A[B][C] ...*, которая, например, может быть весьма полезной в различного рода функциональных преобразованиях как иллюстрирует очень простой пример *HeadCompose[F,x] => 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. С другой стороны, можно представить некоторый *аналог* данного средства, имеющий существенно больший прикладной интерес, а именно процедуру **FunCompose**[*L*, *x*], с чьей помощью можно создавать вложенные функции из заданного списка функций *L* от переменной (*выражения*), заданной вторым ее *x*-аргументом. Следующий весьма простой фрагмент достаточно наглядно иллюстрирует вышесказанное, а именно:

```
In[785]:= FunCompose[L_ /; ListQ[L], x_] := Module[{a, b, c, k}, a = L[[1]]@b;
      For[k = 2, k <= Length[L], k++, c = L[[k]]@b; a = a /. B -> c]; a /. B -> x]
In[786]:= FunCompose[{F, G, H, T, W, Q, V, U}, Sin[z]]
Out[786]= F[G[H[T[W[Q[V[U[Sin[z]]]]]]]]]
In[787]:= {FunCompose[{Sin, Cos, Log}, 19.42], FunCompose[{Sin, Cos, Log, Sqrt}, 2012.75]}
Out[787]= {-0.833093, -0.709467}
```

Один из возможных вариантов *Maple*-аналога для функции *FunCompose* представлен последним примером этого фрагмента. Для организации *передачи* идентификаторов функций в качестве фактических значений можно использовать также конструкции, например, следующих весьма простых видов, а именно:

```
In[88]:= SV[z_] := F@z + z^3; VSV[Id_] := Module[{}, Id@z]; {VSV[F, h], SV[68]}
Out[88]= {F[h], 314432 + F[68]}
```

а также целый ряд подобных им конструкций.

Принимая во внимание важность аппарата *шаблонов* для создания пользовательских функций, рассмотрим данный вопрос несколько детальнее. Прежде всего, в качестве альтернативного представления *шаблонов* проверки используются конструкции вида {*Ш1* | *Ш2* | ...}, где *Шk* – шаблоны проверки; например, в конструкции *X\_String* | *\_List* для *X*-выражения допустимыми являются только значения с заголовками {*String*, *List*}. Однако, имеется ряд весьма существенных особенностей, здесь не рассматриваемых. Следующий фрагмент иллюстрирует не только применение поименованных *ША*, но и эквивалентный им программный *эквивалент*, механизм которого во многих случаях намного более эффективен в качестве «*фильтра*» проверки фактических аргументов на допустимость. В частности, наш опыт работы с пакетом *Maple* вполне определенно подтверждает это утверждение, даже при условии намного более развитого аппарата стандартного типирования объектов, чем подобный аппарат пакета *Mathematica*.

```
In[897]:= Gs[x_List | x_String] := x
In[898]:= {Gs[avzransian], Gs["avzransian"], Gs[{a, v, z, r, a, n, s, i, a, n, 7, 2}]}
Out[898]= {Gs[avzransian], "avzransian", {a, v, z, r, a, n, s, i, a, n, 7, 2}}
In[899]:= Sv[x_] := Module[{}, If[ListQ[x], Length[x], If[StringQ[x], StringLength[x],
      HoldForm[Sv[x]]]]]
In[900]:= {Sv[avzransian], Sv["avzransian"], Sv[{a, v, z, r, a, n, s, i, a, n, 7, 2}]}
Out[900]= {Sv[avzransian], 10, 12}
```

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[1102]:= G[x_, y___] := a*x + b + y
In[1103]:= {G[x, y], G[x, y, z, h, g, v, d, s, w, r, t]}
Out[1103]= {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[56]:= {G[43], G[42.68], G[14/21], G[], F[h]}
Out[56]= {1849, G[42.68], G[2/3], 4624, F[h]}
In[57]:= SV[x_Integer: 43, y_List: {68, 63, 43}] := (x + First[y])^2
In[58]:= {SV[4, 6], SV[7, {3}], SV[], SV[{14, 21, 43, 63, 68}], SV[Null, {14, 21, 43, 63, 68}]}
Out[58]= {SV[4, 6], 100, 12321, 3249, SV[Null, {14, 21, 43, 63, 68}]}
In[61]:= Art[x_String: "rans", y_Integer: 72] := (420 + StringLength[x])*y^2
In[62]:= {Art[42, 68], Art["IAN_2011", 14], Art["AVZ"], Art[420]}
Out[62]= {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 иллюстрирует следующий достаточно простой фрагмент:

```
In[740]:= Default[FG] = 68; FG[x_, y_] := x^2 + y^2;
In[741]:= {FG[], FG[43], FG[14, 21]}
Out[741]= {9248, 6473, 637}
```

```
In[742]:= Default[GG, 1] = 68; Default[GG, 2] = 63; Default[GG, 3] = 43;
                                         GG[x_, y_, z_] := x + y + z
In[743]:= {GG[14], GG[14, 21], GG[]}
Out[743]= {120, 78, 174}
In[744]:= DefaultValues[GG]
Out[744]= {HoldPattern[Default[GG, 1]] :> 68, HoldPattern[Default[GG, 2]] :> 63,
           HoldPattern[Default[GG, 3]] :> 43}
```

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.

Наряду с механизмом типирования *формальных* аргументов *Mathematica* располагает механизмами определения для них значений по умолчанию, т.е. тех значений, которые получают при вызове соответствующие формальные аргументы при их отсутствии. В случае *Maple* кодирование значений по умолчанию производится непосредственно в заголовке процедуры, однако совместное кодирование типа и значения по умолчанию, не только не влияет на совместимость типа с типом такого значения, но и игнорирует сам тип при вызовах процедуры, как хорошо иллюстрирует ряд примеров [99].

```
In[1081]:= Default[G, 1] = 64; Default[G, 2] = 69; G[x_, y_] := Module[{}, x^2 + y^2]
In[1082]:= {G[47, 42], G[], G[67]}
Out[1082]= {3973, 8857, 9250}
In[1083]:= Defaults[x_/, ProcQ[x], y_/, ListQ[y] &&
           DeleteDuplicates[Map[IntegerQ, y]] == {True}, z_/, ListQ[z]] :=
           Module[{a = Arity[x], k = 1}, For[k, k <= a, k++,
           ToExpression["Default[" <> ToString[x] <> ", " <>
           ToString[y[[k]]] <> "]" <> "=" <> ToString1[z[[k]]]]]]
In[1084]:= Defaults[GS, {1, 2}, {64 + Sin[x], 69 + Cos[y]}]
In[1085]:= GS[x_, y_] := Module[{}, x^2 + y^2 + 2012]
In[1086]:= {GS[47, 42], GS[], GS[67]}
Out[1086]= {5985, 2012 + (69 + Cos[y])^2 + (64 + Sin[x])^2, 6501 + (69 + Cos[y])^2}
In[1087]:= Defaults[GS2, {1, 2}, {64 + Sin[x], 69 + Cos[y]}]
In[1088]:= {GS2[47, 42], GS2[], GS2[67]}
Out[1088]= {GS2[47, 42], GS2[], GS2[67]}
```

С другой стороны, механизм *Mathematica* значений по умолчанию несколько отличен и предполагает определение данного типа значений перед вычислением *определений* процедур на основе стандартной функции **Default** пакета, чей формат поддерживает установку различных значений по умолчанию поочередно для отдельных *формальных* аргументов или единого значения для всех аргументов. В предыдущем фрагменте мы представили довольно простую процедуру **Defaults[F, y, z]**, которая обеспечивает для

любого подкортежа кортежа формальных аргументов процедуры  $F$ , определяемого их списком позиций  $y$ , установку значений по умолчанию из списка  $z$ .

Детальнее с механизмом *шаблонов* пакета можно ознакомиться как в справочной базе по пакету, так и в книгах [57–60]; это средство достаточно эффективно при создании различного рода функциональных конструкций, ориентированных и на *символьные*, и на численные вычисления. В качестве иллюстрационных примеров использования рассмотренных средств приведем *две* использующие их пользовательские функции. В *первом* примере для создания макетных файлов требуемых размеров, резервирующих явно дисковое пространство, служит *TRG*-функция пользователя, а именно:

```
In[461]:= TRG[P_String: "C:\\Art_Kr$$", V_Integer: 72420] := Module[{W, m = 0, F},
    F = OpenWrite[P]; W = "V.Z. Aladjev. Classical Cellular Automata: Homogeneous \
Structures.– CA: Palo Alto, Fultus, 2010, 478 p. "; Open[F];
    While[++m <= Floor[V/225.], Write[F, W<>W<>W]]; Close[F]; "File "<>P<>" is filled"]
In[462]:= TRG[]
Out[462]= "File C:\\Art_Kr$$ is filled"
In[463]:= TRG["D:\\Math_myLib\\AGN.63", 68000]
Out[463]= "File D:\\Math_myLib\\AGN.63 is filled"
```

Так, организация *TRG*-функции (*модуля*) использует *Optional*-шаблоны для задания значений «по умолчанию» для ее *формальных* аргументов. Во втором же фрагменте для создания макетных строк *указанных* размеров, которые резервируют рабочую область пакета статически, определяется пользовательская *VASCO*-функция (*модуль*):

```
In[566]:= VASCO[L_Integer: 68000] := Module[{W, m = 0, F = ""},
    W = "Mathematical Theory of Classical HS/CA and its Applications";
    MemoryConstrained[While[PrimeQ[59], F = F <> W], L];
    "String of " <> ToString[StringLength[F]] <> "-length is created! "
In[567]:= VASCO[]
Out[567]= "String of 33748-length is created!"
In[568]:= VASCO[1200000]
Out[568]= "String of 599912-length is created!"
In[569]:= Args[x_: Args] := Module[{a, b}, a := ToString[FullDefinition[x]];
    b := StringPosition[a, {"[", "]"} := ""];
    Quiet["{" <> StringTake[a, {b[[1]][[1]] + 1, b[[2]][[1]] - 1}] <> "}"]
In[570]:= Map[Args, {Args, Sv, SV, FunCompose, Art, TRG, VASCO, FG, GG, Art_Kr}]
Out[570]= {"{}", "{x_:Args}", "{z_}", "{z_}", "{L_List, x_}", "{x_String:rans, y_Integer:72}",
    "{P_String:C:\\Art_Kr$$, V_Integer:72420}", "{L_Integer:68000}", "{x_, y_.}", "{x_, y_, z_.}"}
In[571]:= Args[]
Out[571]= "{x_:Args}"
```

Организация *VASCO*-функции использует *Optional*-шаблоны для задания значений «по умолчанию» для ее *формального* аргумента и *MemoryConstrained*-функцию, тогда

как функция *Args[x]*, обеспечивающая возврат формальных аргументов {процедуры | модуля | функции}, определенных ее фактическим аргументом, в строчном формате, с помощью *Optional*-шаблона определяет для него «по умолчанию» значение *Args*.

При определении *ПФ* пакет не вычисляет ее до момента *вызова* в текущем документе. По этой причине ошибки использования *ПФ* ассоциируются с точкой ее вызова, хотя причина их может находиться в самом определении функции. Для переопределения встроенной либо *ПФ* достаточно отменить ее *Protected*-атрибут (если он у нее имеется) и под ее именем ввести новое определение, которое после его вычисления становится глобальным, т.е. действует на период всего текущего сеанса работы с пакетом. Между тем, ввиду идентификации процедур/функций в среде *Mathematica* не по их именам, а по *заголовкам*, для замены определения *x*-объекта следует закодировать и вычислить его новое определение, но с тем же заголовком, иначе в текущем сеансе будем иметь дополнительно еще один одноименный объект с тем же самым телом.

Между тем, наиболее часто используемые *ПФ* можно сохранять в специальном файле *mx*-формата по функции *Save*, например, в "D:\\Math\_myLib\\UserLib72.mx", а 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 самим пакетом 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 достаточно широко используется и пакетом, однако нами предлагаются подходы на основе нестандартных организаций библиотек как довольно простых, так и весьма удобных для использования и обеспечения программной поддержки. Между тем, пакет *Maple* в данном отношении представляется нам более предпочтительным, предоставляя пользователю механизм ведения библиотек, практически, аналогичный механизму ведения стандартной библиотеки пакета. Более того, наряду с собственно созданием библиотеки пакет предоставляет также весьма простой механизм ведения соответствующей ей справочной базы, очень легко интегрируемой в стандартную базу пакета, что в целом помещает библиотеку пользователя на уровень стандартной.

Между тем, средства пакета *Mathematica* также позволяют создавать подобные *Maple* механизмы организации пользовательских библиотек с обеспечением *уровня* средств, чьи определения находятся в них, на логическом уровне стандартных средств пакета. Однако данная тема не входит в круг интересов настоящей книги.

## Глава 7. Дополнительные средства для работы с базовыми структурами данных и с выражениями в среде *Mathematica*

Строчная и списочная структура данных являются одними из базовых, с которыми в среде пакета работают как стандартные средства пакета, так и программные средства пользователя. Строчная структура представляет собой последовательность простых либо специальных символов, ограниченная двойными кавычками {«»}. Таким образом организованная структура называется просто «*строкой*», для представления которой может быть использован типированный шаблон *\_String*. В качестве простого примера строки можно привести "abc2012xyz". Списочные структуры или просто *списки* имеют формат {*x, y, z, ...*}, где в качестве элементов, разделенных запятой, выступают любые допустимые пакетом выражения, т.е. представляют собой наборы выражений. *Список*, в частности, может представлять такие объекты, как вектора, матрицы, тензоры и др. Для работы с обоими типами структур *Math*-язык располагает достаточно широким набором развитых стандартных средств, со многими из которых читатель встречается и в настоящей книге. Между тем, никакой даже широкий набор средств не покрывает всех потребностей, возникающих при программировании задач как прикладного, так и системного характера. В процессе апробации и эксплуатации пакета *Mathematica* в решении различных задач и, прежде всего, системного характера нами создан целый ряд средств, довольно существенно упрощающих программирование как *прикладных*, так и *системных* задач. Настоящая глава и представляет некоторые из этих средств.

### 7.1. Дополнительные средства работы со строчными структурами в программной среде пакета *Mathematica*

Не взирая на то, что *Mathematica* располагает достаточно большим набором средств для работы со строчными структурами, зачастую имеется потребность в средствах, в составе пакета отсутствующих. Некоторые из таких средств представлены настоящим разделом; среди них имеются как простые, так и более сложные, которые появились в процессе программирования задач различного назначения в качестве сопутствующих процедур и функций, облегчающих либо упрощающих процесс программирования.

В настоящем разделе на ряде примеров иллюстрируется *оформление* процедур пакета *Mathematica*, которое отражает основные его 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 разделах книги. Здесь рассматриваются лишь 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, начинающему программирование в среде *Mathematica*.

Следует отметить, что строчные структуры представляют особый интерес не только в качестве базовых структур данных, с которыми работает как пакет, так и пользователь, но и в качестве основы, в частности, *динамической* генерации объектов в *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.

Вызов достаточно простой процедуры **StringEnd**[*x*, *y*] возвращает значение **True**, если и только если подстрока *y* завершает строку *x*, и значение **False** в противном случае. В случае вызова процедуры **StringEnd** на недопустимых фактических аргументах вызов процедуры возвращается *невычисленным*. Следующий пример представляет исходный код процедуры **StringEnd** наряду с типичными примерами ее применения.

```
In[56]:= StringEnd[x_String, y_String] := Module[{a, b}, {a, b} = {StringLength[x],
    StringLength[y]}; If[a < b, False, If[StringTake[x, -b] == y, True, False]]
In[57]:= {StringEnd["123456", "3456"], StringEnd["123456", "3476"], StringEnd["123456", ""]}
Out[57]= {True, False, 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 с примерами ее применения.

```
In[847]:= Spos[x_;/; StringQ[x], y_;/; StringQ[y], p_;/; IntegerQ[p], dir_;/; IntegerQ[dir]] :=
    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[848]:= Q:= "AVZ72RANS420IAN2010"; {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[848]= {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 простая процедура, вызов которой **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. Если строка *S* не содержит вхождений *s*, вызов возвращает исходную строку *S*, тогда как на пустой строке *S* возвращается пустая строка. В определенном смысле **StringReplace2** совмещает возможности отмеченных стандартных функций. Следующий фрагмент представляет код процедуры **StringReplace2** наряду с примерами ее использования.

```
In[1166]:= StringReplace2[S_;/; StringQ[S], s_;/; StringQ[s], Exp_] :=
Module[{a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]], b, c, d, k = 1},
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[1167]:= StringReplace2["Length[\"abSin[x]\"] + Sin[x] + ab - Sin[x]*6", "Sin[x]", "a^b"]
Out[1167]= "Length[\"abSin[x]\"] + a^b + ab - a^b*6"
In[1168]:= StringReplace2["Length[\"abSin[x]\"] + Cos[x] + ab - Cos[x]*6", "Sin[x]", "a^b"]
Out[1168]= "Length[\"abSin[x]\"] + Cos[x] + ab - Cos[x]*6"
In[1169]:= StringReplace2["", "Sin[x]", "(a+b)"]
Out[1169]= ""
```

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 удаляется оставшаяся часть в строке *S*.

```
In[1021]:= 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[1022]:= S = "Subs[AladjevProceduresAndFunctions`Subs`x_,
  AladjevProceduresAndFunctions`Subs`y_,AladjevProceduresAndFunctions`Subs`z_]"
Out[1022]= "Subs[AladjevProceduresAndFunctions`Subs`x_,
  AladjevProceduresAndFunctions`Subs`y_,AladjevProceduresAndFunctions`Subs`z_]"
In[1023]:= SubsDel[S, "Subs`", {"[", ",", "}"}, -1]
Out[1023]= "Subs[x_,y_,z_]"
In[1024]:= SubsDel["12345avz6789", "avz", {"8"}, 1]
Out[1024]= "1234589"
In[1025]:= SubsDel["12345avz6789", "avz", {"8", 9}, 1]
Out[1025]= SubsDel["12345avz6789", "avz", {"8", 9}, 1]
In[1026]:= SubsDel["12345avz6789", "avz", {"5"}, 1]
Out[1026]= "12345"
```

Предыдущий фрагмент представляет исходный код процедуры **SubsDel**[*S*, *x*, *y*, *p*] с наиболее типичными примерами ее использования. Данная процедура используется целым рядом процедур из нашего пакета *AVZ\_Package* [90].

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 пакета *Mathematica* в качестве полезного аналога трех одноименных процедур из нашей Библиотеки процедур [43-45] для пакета *Maple*. Тогда как нижепредставленная процедура **SubDelStr**[*x*, *L*] обеспечивает удаление из строки *x* всех подстрок, которые ограничены номерами позиций, заданных списком списков *L* (список *ListList-muna*) 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[3568]:= StrOfSymb1Q[x_;/ StringQ[x], A_;/ ListQ[A]] := If[DeleteDuplicates[Map3[
  MemberQ, Map[ToString, A], Characters[x]]] == {True}, True, False]
In[3569]:= StrOfSymb1Q["134325678", {1, 2, 4, 5}]
Out[3569]= False
In[3570]:= StrOfSymb1Q["4134382556728", {1, 3, 2, 4, 5, 6, 7, 8}]
Out[3570]= True
In[3826]:= 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[3827]:= SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}}]
Out[3827]= "1269dffd"
In[3828]:= SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}, {40, 42}}]
Out[3828]= 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 **StringTake{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 и некоторые примеры их использования.

```
In[1747]:= 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[1748]:= StringTake1["ransianavzagnvsvartkr", {ian, agn, art}]
Out[1748]= {"rans", "avz", "vsv", "kr"}

In[1749]:= StringTake1["ransianavzagnvsvartkr", {ian, 420, art}]
Out[1749]= {"rans", "avzagnvsv", "kr"}

In[1750]:= StringTake1["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]
Out[1750]= {"s", "avz", "vsv"}

In[1751]:= 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[1752]:= StringTake2["ransianavzagnvsvartkr", {ian, agn, art}]
Out[1752]= {"rans", "avz", "vsv", "kr"}

In[1753]:= StringTake2["ransianavzagnvsvartkr", {ian, 420, art}]
Out[1753]= {"rans", "avzagnvsv", "kr"}

In[1754]:= StringTake2["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]
Out[1754]= {"s", "avz", "vsv"}
```

В дополнение к стандартной функции **StringReplace** в ряде случаев весьма полезной оказывается процедура **StringReplace1[S, L, P]**, возвращающая результат подстановки в строку *S* строк из списка *P* вместо ее подстрок, определяемых позициями *вложенного* списка *L* *ListList*-типа. В ряде случаев процедура дополняет стандартную функцию. Нижеследующий фрагмент представляет исходный код процедуры с примерами.

```
In[2479]:= 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[2480]:= StringReplace1["avz123456789agn", {{4, 7}, {8, 10}, {11, 12}}, {" RANS ", Tampere,
Sqrt[(a + b)*(c + d)]]]
```

```
Out[2480]= "avz RANS TampereSqrt[(a + b) (c + d)]agn"
```

С другой стороны, вызов процедуры **StrDelEnds**[*W*, *h*, *p*] возвращает усеченную слева строку *W* символами *h* при *p* = 1, усеченную справа строку *W* элементами *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[1359]:= StrDelEnds[S_/, StringQ[S], h_/, CharacterQ[h], p_/, MemberQ[{1, 2, 3}, p]] :=
Module[{a, b, c, k = 1, Fin, s}, s = S; Goto[p];
Label[1]; Label[3]; For[k, k <= StringLength[S], k++,
If[StringTake[S, {k, k}] != h, a = StringTake[S, {k, -1}]; Break[], Continue[]];
If[p == 3, s = a, Goto[Fin]];
Label[2]; For[k = StringLength[s], k >= 1, k--, If[StringTake[s, {k, k}] != h,
a = StringTake[s, {1, k}]; Break[], Continue[]]; Label[Fin]; a]
```

```
In[1360]:= StrDelEnds["xxxxxxx123456789xxxxxxxxxxx", "x", 1]
```

```
Out[1360]= "123456789xxxxxxxxxxx"
```

```
In[1361]:= StrDelEnds["xxxxxxx123456789xxxxxxxxxxx", "x", 2]
```

```
Out[1361]= "xxxxxxx123456789"
```

```
In[1362]:= StrDelEnds["xxxxxxx123456789xxxxxxxxxxx", "x", 3]
```

```
Out[1362]= "123456789"
```

Следующий фрагмент представляет примеры трех достаточно полезных процедур, а именно. Вызов простой процедуры **SuffPref**[*S*, *s*, *n*] обеспечивает тестирование строки *S* на предмет *начинаться* подстрокой *s* (*n*=1), *завершаться* подстрокой *s* (*n*=2) или быть с *обоих* концов *ограниченной* подстрокой *s*. При установлении данного факта **SuffPref** возвращает значение *True*, в противном случае возвращается *False*. Между тем, весьма простая функция **StrStr**[*x*] обеспечивает возврат выражения *x*, отличного от строки, в строчном формате и *двойной* строки в противном случае. Функция **StrStr** полезна при работе со строчными конструкциями, в частности, с функцией **StringReplace**.

```
In[2676]:= StrStr[x_] := If[StringQ[x], "\"" <> x <> "\" ", ToString[x]]
```

```
In[6277]:= Map[StrStr, {"RANS", a + b, IAN, {68, 63, 43}, F[x,y]]]
```

```
Out[2677]= {"\"RANS\"", "a + b", "IAN", "{68, 63, 43}", "F[x, y]"}
```

```
In[2678]:= SuffPref[S_/, StringQ[S], s_/, StringQ[s], 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[2679]:= SuffPref["IAN_RANS_RAC_REA_75_450", "75_450", 2]
Out[2679]= True
In[2680]:= SuffPref["IAN_RANS_RAC_REA_75_450", "IAN_RANS", 1]
Out[2680]= True

```

Для работы со строчными структурами достаточно полезной оказывается процедура **InsertN**[*S*, *L*, *n*] (аналог одноименной процедуры для *Maple*, рассмотренной в [90]), которая обеспечивает вставку в строку *S* подстроки из списка *L* после ее позиций из списка *n*. В случае  $n = \{< 1 \mid \text{StringLength}(S)\}$  подстрока помещается перед началом строки *S* либо 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_;/; StringQ[S], L_;/; ListQ[L], n_;/; ListQ[n] && Length[n] ==
Length[Select[n, IntegerQ[#] &]]] :=
Module[{a = Map[ToString, L], c = S, d, Ins, k = 1, m = Sort[n]},
  Ins[a_, b_, c_] := If[c < 1, b <> a, If[c >= StringLength[a], a <> b, StringInsert[a, b, c + 1]]];
  For[k, k <= Length[a], k++, d = a[[k]]; c = Ins[c, d, Quiet[Check[m[[k]], Return[c]]]];
  m = m + StringLength[d]]; c]

In[2584]:= InsertN["123456789Rans_Ian", {Agn, Avz, Vsv, Art, Kr}, {6, 9, 3, 0, 3, 15}]
Out[2584]= "Agn123AvzVsv456Art789KrRans_Ian"
In[2585]:= InsertN["123456789", {a, b, c, d, e, f, g, h, n, m}, {4, 2, 3, 0, 1, 5, 6, 7, 8, 9, 15}]
Out[2585]= "a1b2c3d4e5f6g7h8n9m"

```

В противоположность предыдущей процедуре следующая процедура **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_;/; StringQ[S], 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},
  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["123Rans4IAN575450", {{4, 8}, {9, 11}, {12}}]
Out[2740]= "12375450"

```

```
In[2741]:= DelSubStr["123456789abcdgh", Range[1, 15]]
Out[2741]= ""
In[2742]:= DelSubStr["123Rans4IAN575450", {4, 8}]
Out[2742]= "123ansIAN575450"
In[2743]:= DelSubStr["123Rans4IAN575450", {{4, 8}}]
Out[2743]= "123IAN575450"
In[2744]:= DelSubStr["123456789abcdgh", {6}]
Out[2744]= "12345789abcdgh"
```

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[1013]:= SortString[x_;/; StringQ[x], y_;/; MemberQ[{Greater, Less}, y]] := StringJoin[
    Sort[Characters[x], y[ToCharacterCode[#1]][[1]], ToCharacterCode[#2]][[1]]] &]]
In[1014]:= SortString["AVZ321AGN456Art789Kr", Less]
Out[1014]= "123456789AAAGKNVZrrt"
In[1015]:= SortString["AVZ321AGN456Art789Kr", Greater]
Out[1015]= "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 ниже 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.

```
In[2420]:= ToString1[x_] := Module[{a = "$Art23Kr15$.txt", b = "", c, d},
    Write[a, x]; Close[a];
    Label[d]; c = Read[a, String]; If[c === EndOfFile,
    Close[a]; DeleteFile[a]; Return[b],
    b = b <> StrDelEnds[c, " ", 1]; Goto[d]]]
```

```
In[2421]:= Kr[x_] := Module[{a = "Art_Kr", b = "="}, a <> b <> ToString[x]]
In[2422]:= ToString[Definition[Kr]]
Out[2422]= "Kr[x_] := Module[{a = Art_Kr, b = = }, a<>b<>ToString[x]]"
In[2423]:= ToExpression[%]
ToExpression::syntax: Invalid syntax in or before "Kr[x_] := Module[{a = Art_Kr, b = = } ... ".
Out[2423]= $Failed
In[2424]:= ToString1[Definition[Kr]]
Out[2424]= "Kr[x_] := Module[{a = \"Art_Kr\", b = \" = \"}, StringJoin[a, b, ToString[x]]]"
In[2425]:= ToExpression[%]; Kr[2]
Out[2425]= "Art_Kr = 2"
```

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.

Выше отмечалась возможность использования стандартных *шаблонов аргументов (ША)* для тестирования допустимости фактических аргументов при вызовах процедур или функций. В качестве наиболее часто используемых *ША* отметим такие, как: *x\_Integer*, *x\_String*, *x\_List*, *x\_Complex*, *x\_Number* и *x\_Real*, смысл которых достаточно прозрачен. Примеры (1) использования некоторых представлены следующим фрагментом.

```
In[1025]:= V[L_List] := Length[L] (1)
In[1026]:= Map[V, {{a, b, c, d, e, f, g, h}, a + b + c}]
Out[1026]= {8, V[a + b + c]}
In[1027]:= G[S_String] := StringLength[S]
In[1028]:= Map[G, {{a, b, c, d, e, f, g, h}, "AVZRANSIAN"}]
Out[1028]= {G[{a, b, c, d, e, f, g, h}], 10}
In[1029]:= S[P_Integer] := NextPrime[P]
In[1030]:= Map[S, {450, "AVZRANSIAN"}]
Out[1030]= {457, S["AVZRANSIAN"]}

In[1047]:= Sv[P_Integer && PrimeQ[P]] := NextPrime[P] (2)
In[1048]:= Map[Sv, {45, 450, 23}]
Out[1048]= {Sv[45], Sv[450], Sv[23]}

In[1049]:= Sv1[P_ /; IntegerQ[P] && PrimeQ[P]] := NextPrime[P] (3)
In[1050]:= Map[Sv1, {45, 450, 23}]
Out[1050]= {Sv1[45], Sv1[450], 29}

In[1051]:= Sv[P_Integer /; PrimeQ[P]] := NextPrime[P] (4)
In[1052]:= Map[Sv, {45, 450, 23}]
Out[1052]= {Sv[45], Sv[450], 29}

In[1060]:= IntegerList[x_] := If[ListQ[x] && DeleteDuplicates[Map[IntegerQ, x]] ==
{True}, True, False]
```

```
In[1061]:= Gs[x_IntegerList] := Plus[Sequences[x]] (5)
```

```
In[1062]:= Gs[{44, 23, 16, 64, 69}]
```

```
Out[1062]= Gs[{44, 23, 16, 64, 69}]
```

```
In[1063]:= Gs1[x_;/ IntegerList[x]] := Plus[Sequences[x]] (6)
```

```
In[1064]:= Gs1[{44, 23, 16, 64, 69}]
```

```
Out[1064]= 216
```

Между тем, вполне естественное сочетание *ША* с логическими условиями (*тестами*) в общем случае не работает, оставаясь с точки зрения синтаксиса языка корректным, как показывает пример (2) фрагмента. Для устранения данного недостатка возможно вместо чистого *ША* использовать смешанную конструкцию формата «*x\_ ША ; Тест*», как иллюстрирует пример (4) фрагмента, либо стандартный подход на основе *теста*, как иллюстрирует пример (3). Более того, пользовательское определение *ША* в общем случае не работает, как иллюстрирует пример (5), требуя его оформления в формате теста для соответствующего формального аргумента, как иллюстрирует пример (6). В этой связи рекомендуется использовать именно стандартный механизм типирования, позволяющий достаточно легко определять пользовательские типы объектов. Именно этот подход использовался при программировании средств пакета *AVZ\_Package* [90].

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 подписками их как начальных, так и конечных позиций вхождения в строку *x* в качестве подстроки.

```
In[1063]:= 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]];
```

```
    b = Append[b, {d, StringPosition[x, d]}];
```

```
    Sort[Select[b, #[[2]] != {} &]]]
```

```
In[1064]:= StringPosition1["abcdefghpstw", {a, c, h, w}]
```

```
Out[1064]= {"a", {{1, 1}}, {"c", {{3, 3}}, {"h", {{8, 8}}, {"w", {{12, 12}}}}
```

```
In[1065]:= StringPosition1["abcdefghpstw", {x, y, z, v}]
```

```
Out[1065]= {}
```

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 *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. Фрагмент представляет код и достаточно типичные примеры использования процедуры **SubStr**.

```

In[1379]:= 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 = t <> c; Continue[], Break[]]; c,
R = "Argument p should be in range 1.." <> ToString[StringLength[S]] <>
" but received " <> ToString[p]; $Failed]]

In[1380]:= SubStr["12345abcdefg6789sewrt", 14, "3", "r", Error]
Out[1380]= "12345abcdefg6789sew"
In[1381]:= SubStr["12345abcdefg6789sewrt", 14, {"1", "2", "0"}, {"s", "w", "x"}, Error]
Out[1381]= "345abcdefg6789"
In[1382]:= SubStr["12345abcdefg6789sewrt", 22, "0", "x", Error]
Out[1382]= $Failed
In[1383]:= Error
Out[1383]= "Argument p should be in range 1..21 but received 22"

```

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 применения представляет нижеследующий фрагмент.

```

In[755]:= ExprOfStr[x_;/ StringQ[x], 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[756]:= P[x_, y_] := Module[{a, P1}, P1[z_, h_] := Module[{n}, z^2 + h^2]; x*y + P1[x, y]]
In[757]:= x = DefFunc1[ToString[P]]; {ExprOfStr[x, 44, 1, {" ", ";", ","}],
ExprOfStr[x, 39, -1, {" ", ";", ","}]}

Out[757]= {"Module[{n}, z^2 + h^2]", "P1[z_, h_]"}
In[758]:= y = "123456789abcdefghAVZ_2012"; ExprOfStr[y, 75, -1, {" ", ";", ","}]
Out[758]= ExprOfStr["123456789abcdefghAVZ_2012", 75, -1, {" ", ";", ","}]

```

```
In[759]:= ExprOfStr[x, 10, 1, {" ", ";", ", "}]
Out[759]= $Failed
In[760]:= ExprOfStr["12345678;F[(a + b)/(c + d)]; AVZ_2012", 28, -1, {"^", ";"}]
Out[760]= "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**. Следующий фрагмент представляет исходный код процедуры **ExprOfStr1** наряду с типичными примерами ее применения.

```
In[1145]:= ExprOfStr1[x_ /; StringQ[x], n_ /; IntegerQ[n] && n >= 1,
           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[1146]:= x = "123{a+b}, F[c+d]"; ExprOfStr1[x, 4, 1]
Out[1146]= "{a+b}"
In[1147]:= x = "123{a+b}, F[c+d]"; ExprOfStr1[x, 16, -1]
Out[1147]= "F[c+d]"
In[1148]:= x = "123{a+b}, [c+d]"; ExprOfStr1[x, 15, -1]
Out[1148]= $Failed
```

В частности, процедура **ExprOfStr1** дает возможность запрограммировать достаточно простой алгоритм полезной процедуры **Locals2**[*x*], которая имеет дело с локальными переменными процедуры *x*. Фрагмент представляет исходный код процедуры.

```
In[960]:= Locals2[x_ /; ProcQ[x]] := Module[{a = HeadPF[x], b = DefFunc3[x][[1]], c, d, h, p},
           c = StringReplace[b, {a <> " := Module[" -> "", a <> " := Block[" -> ""}];
           d = ExprOfStr1[c, 1, 1]; If[d == "{}", Return[{}], Null];
           h = Select[StringSplit[StringTake[d, {2, -2}], ", "], ! StringFreeQ[#, " ="] | |
```

```
SymbolQ[#] &]; h = Map[StringTake[#, {1, p = Flatten[StringPosition[#, " = "]]];
    If[p == {}, 0, p[[1]] - 1] &]; {h, d}]
```

```
In[961]:= GS[x_] := Block[{a, b, c, x}; Map[Locals2, {UprocQ, Mapp, GS}]
```

```
Out[961]= {{{"a", "b", "d", "m", "r"}, {"a = ToString1[DefFunc1[ToString[x]], b, d, m, r]}},
    {"a", "b", "c", "h", "g", "k"}, {"a = Level[Expr, 1], b = {x}, c = {}, h, g = Head[Expr], k = 1}},
    {"a", "b", "c"}, {"a, b, c}}}
```

Вызов **Locals2[x]** возвращает вложенный список, чей первый элемент – список локальных переменных процедуры *x* в строчном формате, тогда как второй – список локальных переменных с начальными значениями; в отсутствие локальных переменных у *x* вызов процедуры возвращает пустой список, т.е. {}. Данная процедура является наиболее эффективной в задачах обработки локальных переменных процедур.

В определенном отношении к процедуре **ExprOfStr** примыкает и процедура **ExprExpr**, чей вызов **ExprExpr[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[1621]:= ExprExpr[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[1622]:= ExprExpr["z=(Sin[x+y] + Log[x])+G[x,y];", 4, 13]
```

```
Out[1622]= "Sin[x + y]"
```

```
In[1623]:= ExprExpr["z=(Sin[x+y] + Log[x])+F[x,y];", 1, 21]
```

```
Out[1623]= "Log[x] + Sin[x + y]"
```

```
In[1624]:= ExprExpr["z = (Sin[x + y] + Log[x]) + F[x, y];", 1, 36]
```

```
Out[1624]= "F[x, y] + Log[x] + Sin[x + y]"
```

Предыдущий фрагмент представляет исходный код процедуры наряду с примерами ее использования. Процедура **ExprExpr** 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. Следующий фрагмент представляет несколько таких примеров.

```
In[619]:= Border[x_;/ StringQ[x]] := Module[{a = Floor[StringLength[x]/2], b = {}, c, k = 1},  
      For[k, k <= a, k++, c = StringTake[x, k]; If[c == StringTake[x, -k],  
        b = Append[b, c], Continue[]]]; If[b == {}, "", b[[-1]]]]
```

```
In[620]:= Map[Border, {"123456789", "avz75420avz", "art22kr15ddrainianart22kr15dd"}]  
Out[620]= {"", {"avz"}, {"art22kr15dd"}}]
```

```
In[624]:= CatN[s_;/ StringQ[s], n_;/ IntegerQ[n] && n >= 1] := Module[{a = "", k = 1},  
      For[k, k <= n, k++, a = a <> s]; a]
```

```
In[625]:= CatN["agn", 16]  
Out[625]= "agnagnagnagnagnagnagnagnagnagnagnagnagnagnagnagn"
```

```
In[722]:= 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[723]:= LeftFold[F, Z, "abcdefgh"]  
Out[723]= F["a", F["b", F["c", F["d", F["e", F["f", F["g", F["h", Z]]]]]]]]]
```

```
In[724]:= LeftFold[StringJoin, ToString1["75"], "abcdefgh"]  
Out[724]= "abcdefgh75"
```

```
In[725]:= 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[726]:= RightFold[F, Z, "abcdefgh"]  
Out[726]= F["h", F["g", F["f", F["e", F["d", F["c", F["b", F["a", Z]]]]]]]]]
```

```
In[727]:= RightFold[StringJoin, ToString1["75"], "abcdefgh"]  
Out[727]= "hgfedcba75"
```

```
In[799]:= OverLap[x_;/ StringQ[x], y_] := Module[{a, b, c = {}, d = {x, y}, p = 0, h = {}, k = 1},  
      If[! StringQ[d[[2]]], Return[Defer[OverLap[{x, y}]]], {a, b} = Map[StringLength, {x, d[[2]]}];  
      For[k, k <= Min[a, b], k++, If[StringTake[d[[2]], k] != StringTake[x, -k], Continue[],  
        h = Append[h, k]]; If[p == Min[a, b], Return[$Failed], If[Length[d] >= 3,  
      If[! HowAct[d[[3]]], ToExpression[ToString[d[[3]]] <> "=" <> ToString1[StringTake[d[[2]],  
        If[h == {}, Return[$Failed], h[[-1]]]]], Return[Defer[OverLap[{x, y}]]], Null]]];  
      If[h == {}, Return[$Failed], h[[-1]]]]]
```

```
In[800]:= OverLap["123456789abcd", "3456789abcd"]  
Out[800]= 11  
In[801]:= {OverLap["123456789abcd", "3456789abcd", z], z}  
Out[801]= {11, "3456789abcd"}  
In[802]:= {OverLap["123456789abcd", "3456789abcd", z], z}  
Out[802]= {OverLap[{"123456789abcd", "3456789abcd", "3456789abcd"}], "3456789abcd"}  
In[803]:= Clear[z]; {OverLap["123456789abcd", "3456789abcd", z], z}  
Out[803]= {11, "3456789abcd"}]
```

```

In[804]:= OverLap["555555555", "123456789"]
Out[804]= $Failed

In[899]:= LongestCommonSubString[x_;/; StringQ[x], y_;/; StringQ[y]] := Module[{a,b,c,d},
    If[StringLength[x] > StringLength[y], a = x; b = y, a = y; b = x];
    d = DeleteDuplicates[Select[Map[StringJoin,
    Subsets[Characters[b]]], ! StringFreeQ[a, #] && # != "" &]];
    c = Select[Select[d, ! StringFreeQ[b, #] &], ! StringFreeQ[a, #] &];
    If[c == {}, c, Sort[Select[c, StringLength[#] == Max[Map[StringLength, c]] &]]]

In[900]:= LongestCommonSubString["abdfgavzagntrehgfgtjkt", "ghtavzagnderty"]
Out[900]= {"avzagn"}

In[901]:= LongestCommonSubString["abdfgtrehgfgtjkt", "ghtf2g6derty"]
Out[901]= {"d", "e", "f", "g", "h", "r", "t"}

In[902]:= LongestCommonSubString["abdfgtrehgfgtjkt", "Z123456789"]
Out[902]= {}

In[941]:= LongestCommonSubSequence[x_;/; StringQ[x], y_;/; StringQ[y]] :=
    Module[{a, b, c}, {a, b} = {Select[Map[StringJoin, Subsets[Characters[x]]], # != "" &],
    Select[Map[StringJoin, Subsets[Characters[y]]], # != "" &]}; c = Intersection[a, b];
    If[c == {}, c, Select[c, StringLength[#] == Max[Map[StringLength, c]] &]]

In[942]:= LongestCommonSubSequence["abcdexyzthg", "uvabxycde"]
Out[942]= {"abcde"}

In[943]:= LongestCommonSubSequence["abcdeefg", "123456789"]
Out[943]= {}

In[954]:= PrefixQ[x_;/; StringQ[x], y_;/; StringQ[y]] := If[x == "", True,
    If[! StringFreeQ[y, x] && StringTake[y, StringLength[x]] == x, True, False]]

In[955]:= {PrefixQ["avz", "avzransian"], PrefixQ["", "avzransian"]}
Out[955]= {True, True}

In[956]:= SuffixQ[x_;/; StringQ[x], y_;/; StringQ[y]] := If[x == "", True,
    If[! StringFreeQ[y, x] && StringTake[y, -StringLength[x]] == x, True, False]]

In[957]:= {SuffixQ["avz", "avzransianavz"], SuffixQ["", "avzransian"]}
Out[957]= {True, True}

In[973]:= PalindromeQ[x_;/; StringQ[x]] := If[x == StringReverse[x], True, False]

In[974]:= Map[PalindromeQ, {"123456789", "abcdeffedcba"}]
Out[974]= {False, True}

In[975]:= MaximalPalindromicSubstring[x_;/; StringQ[x]] := Module[{b = {}, c, k = 1,
    a = Select[Map[StringJoin, Subsets[Characters[x]]], # != "" &]},
    For[k, k <= Length[a], k++, If[PalindromeQ[a[[k]]], b = Append[b, a[[k]], Null]];
    c = Sort[Map[StringLength, b]]][[-1]]; Select[b, StringLength[#] == c &]]

In[976]:= Map[MaximalPalindromicSubstring, {"12345436", "abcdefgh"}]
Out[976]= {"34543"}, {"a", "b", "c", "d", "e", "f", "g", "h"}

```

```
In[988]:= IsPermutation[x_ /; StringQ[x]] := Block[{a = Characters[x]}, If[Length[a] ==
Length[DeleteDuplicates[a]], True, False]]
In[989]:= Map[IsPermutation, {"abc", "abcba", "123456789", "1233456"}]
Out[990]= {True, False, True, False}
In[991]:= IsMonotonic[x_] := Block[{a = {x}, b = {x}[[1]], c = If[StringQ[{x}[[1]]],
ToCharacterCode[{x}[[1]]], "Err"}], If[c === "Err", Return[Defer[IsMonotonic[x]]],
If[c == Sort[c, #1 > #2 &], If[Length[a] > 1 && ! HowAct[a[[2]]],
ToExpression[ToString[a[[2]]] <> "=" <> "\"Decrease\"", Null];
True, If[c == Sort[c, #1 < #2 &], If[Length[a] > 1 && ! HowAct[a[[2]]],
ToExpression[ToString[a[[2]]] <> "=" <> "\"Increase\"", Null]; True, False]]]]
In[992]:= Clear[h]; {IsMonotonic["abcdefgh", h], h}
Out[992]= {True, "Increase"}
In[993]:= Clear[h]; {IsMonotonic["987654321", h], h}
Out[993]= {True, "Decrease"}
In[994]:= IsMonotonic["qwertyuiopasdfghjklzxcvbnm"]
Out[994]= False
In[995]:= IsMonotonic[qwertyuiopasdfghjklzxcvbnm, k]
Out[995]= IsMonotonic[qwertyuiopasdfghjklzxcvbnm, k]
```

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 *невывчисленным*. 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.

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). Следующий фрагмент представляет исходный код процедуры **ExtrName** наряду с типичными примерами ее использования.

```
In[6]:= ExtrName[x_;/ StringQ[x], 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[7]:= ExtrName["1234567 xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h[a+b]; gh", 16, -1]
Out[7]= "xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h"
In[8]:= ExtrName["1234567 xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h[a+b]; gh", 8, 1]
Out[8]= "xC\[CurlyPhi]z\[Alpha]\[ScriptCapitalA]h"
```

Кстати, процедура **ExtrName** довольно легко расширяется на символы, допускаемые в качестве составляющих имен объектов в программной среде пакета *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[1150]:= 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[1151]:= StrFromStr["12345\"678abc\"xyz\"910\"mnph"]
Out[1151]= {"678abc", "910"}
In[1152]:= StrFromStr["123456789"]
Out[1152]= {}
```

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[1137]:= DelSuffPref[x_;/ StringQ[x], y_;/ StringQ[y], n_;/ MemberQ[{1, 2, 3}, n]] :=
    Block[{a = StringLength[y]}, If[! SuffPref[x, y, n], x,
    StringTake[x, {{a + 1, -1}, {1, -(a + 1)}, {a + 1, -(a + 1)}}[[n]]]]
In[1138]:= DelSuffPref["avzransianArtKravz", "avz", 1]
Out[1138]= "ransianArtKravz"
In[1139]:= DelSuffPref["avzransianArtKravz", "avz", 2]
Out[1139]= "avzransianArtKr"
In[1140]:= DelSuffPref["avzransianArtKravz", "avz", 3]
Out[1140]= "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[1226]:= StrSub[x_;/ StringQ[x], y_;/ StringQ[y], z_;/ StringQ[z], n_;/ PosIntQ[n] ||
    PosIntListQ[n]] := Block[{a = StringPosition[x, y], b = Sort[Flatten[{n}]]},
    If[a == {} || Length[a] < b[[-1]], $Failed, StringReplacePart[x, z, Part[a, b]]]]
In[1227]:= StrSub["avzArtKravzRansIanavz", "avz", "agn64", {1, 3}]
Out[1227]= "agn64ArtKravzRansIanagn64"
In[1228]:= StrSub["avzArtKravzRansIanavz", "avz69", "agn64", {1, 3}]
Out[1228]= $Failed
In[1229]:= PrefCond[x_;/ StringQ[x], y_;/ StringQ[y]] :=
    Block[{a = Flatten[StringPosition[x, y]]}, If[a == {}, "", StringTake[x, {1, a[[1]] - 1}]]]
In[1230]:= s = "rans ian 450 75"; s1 = " "; {PrefCond[s, s1], PrefCond["agn", "vsv"]}
Out[1230]= {"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 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 текст процедуры **StringSplit1** с примерами ее использования. Процедура **StringSplit1** оказывается довольно полезным средством, в частности, при программировании средств обработки заголовков процедур и функций.

```
In[1250]:= 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[1251]:= 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[1251]= {"x_String", " y_Integer", " z_;/; MemberQ[{1, \"2\", \"3\", \"4\", \"5\"}, \"z\"] ||  
IntegerQ[z], \"h_\", \"s_String\", \"s_;/; StringQ[y] || StringLength[y] == 1\"}  
  
In[1252]:= 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[1252]= {"x_String", " y_Integer", " z_;/; MemberQ[{1, 2, 3, 4, 5}, z] || IntegerQ[z], \"h_\",  
\"s_String\", \"h_;/; StringQ[y] || StringLength[y] == 1\"}
```

Целый ряд задач, имеющих дело с обработкой строчных структур, делает достаточно полезной процедуру **SubsStr** $[x, y, h, t]$ , исходный код которой 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 применения.

```
In[1438]:= 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[1439]:= SubsBstr["123452333562675243655", "2", "5"]
Out[1439]= {"2345", "23335", "2675", "24365"}
In[1440]:= SubsBstr["123452333562675243655", "9", "5"]
Out[1440]= {}
```

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

```
In[1533]:= SubStrSymbolParity[x_;/ StringQ[x], y_;/ SymbolQ[y], z_;/ SymbolQ[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[1534]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 0]
Out[1534]= {"{abcdfgh}", "{rans}", "{ian}"}
In[1535]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 0, 7]
Out[1535]= {"{rans}", "{ian}"}
In[1536]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1]
Out[1536]= {"{abcdfgh}", "{rans}", "{ian}"}
In[1537]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1, 23]
Out[1537]= {"{abcdfgh}", "{rans}"}
In[1538]:= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1, -70]
Out[1538]= SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1, -70]
```

Вызов процедуры **SubStrSymbolParity** $[x,y,z,d]$  с четырьмя фактическими аргументами возвращает список подстрок строки  $x$ , ограниченных односимвольными строками  $\{y, z\}$  ( $y \neq z$ ); 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.

Вышепредставленные средства обработки *строчных* структур, аналогичные средствам пакета *Maple*, базировались на широко используемых стандартных средствах пакета *Mathematica*, наглядно демонстрируя относительную простоту программирования в среде *Math*-языка средств, аналогичных средствам пакета *Maple* как своего основного конкурента. Между тем, наличие в *Mathematica* 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 шаблонами 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 пакета *Mathematica*. Наш опыт по использованию данных пакетов 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 представлен выше. Для пакета *Maple* нами также был создан целый ряд средств обработки *строчных* структур [45], ряд из которых (*либо их аналоги*) появились впоследствии в более старших релизах пакета; 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 пакет *Mathematica* предлагает более развитые функциональные средства.

## 7.2. Дополнительные средства для работы со списочными структурами в программной среде пакета *Mathematica*

Не взирая на то, что *Mathematica* располагает достаточно большим набором средств для работы со *списочными* структурами, являющимися одними из *базовых*, с которыми работают как собственно средства пакета, так и пользовательские, зачастую возникает потребность в средствах, отсутствующих в составе стандартных средств *Mathematica*. Некоторые из таких средств представлены настоящим разделом; среди них имеются как простые, так и более сложные, которые появились в процессе программирования задач различного назначения в качестве сопутствующих процедур либо функций. В разделе представлены исходные коды средств с кратким их описанием и примерами.

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[1355]:= 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[1356]:= DelEl[{a, a, a, a, a, 63, 68, 43, 14, 42, 47, 22, a, a, a, a, a}, a, 1]
Out[1356]= {63, 68, 43, 14, 22, a, a, a, a, a}
In[1357]:= DelEl[{a, a, a, a, a, 63, 68, 43, 14, 22, 42, 47, a, a, a, a, a}, a, 2]
Out[1357]= {a, a, a, a, a, 63, 68, 43, 14, 22}
In[1358]:= DelEl[{a, a, a, a, a, 64, 70, 44, 16, 23, 42, 47, a, a, a, a, a}, a, 3]
Out[1358]= {64, 70, 44, 16, 23, 42, 47}
```

Следующий фрагмент иллюстрирует достаточно простую процедуру **SortNL**[*L*, *p*, *b*], производящую сортировку *числового* списка *ListList*-типа по элементам в *p*-позиции его подсписков. Сортировка *L* производится по убыванию (*Greater*) либо возрастанию (*Less*) значений элементов в *p*-позиции подсписков. Основной интерес представляет в этом фрагменте программная обработка всех основных *особых* и *ошибочных* ситуаций.

```
In[1039]:= SortNL[L_/, ListQ[L], p_ /; IntegerQ[p], b_] := Module[{},
  If[DeleteDuplicates[Map[NumberQ, Flatten[L]]] == {True} &&
  DeleteDuplicates[Map[ListQ, L]] == {True} &&
  Length[DeleteDuplicates[Map[Length, L]]] == 1,
  If [p >= 1 && p <= Length[First[L]],
    If[MemberQ[{Greater, Less}, b], Sort[L, b[#1[[p]], #2[[p]]] &],
  Print[SortNL::"incorrect third argument, should be {Less, Greater}, but has received ", {b}]],
  Print[SortNL::"incorrect second argument, should lay in interval ", {1, Length[First[L]}]]],
```

```

Print[SortNL::incorrect first argument, should be numeric listlist, but has received ", {L}]]]
In[1040]:= L := {{42, 70, 450, 75}, {47, 64, 16, 23}, {67, 43, 23, 16}, {62, 48, 15, 22}}
In[1041]:= SortNL[L, 2, Less]
Out[1041]= {{67, 43, 23, 16}, {62, 48, 15, 22}, {47, 64, 16, 23}, {42, 70, 450, 75}}
In[1042]:= SortNL[L, 2, Greater]
Out[1042]= {{42, 70, 450, 75}, {47, 64, 16, 23}, {62, 48, 15, 22}, {67, 43, 23, 16}}
In[1043]:= SortNL[L, 15, Less]
SortNL::incorrect second argument, should lay in interval {1, 4}
In[1044]:= SortNL[{L, h}, 2, Less]
SortNL::incorrect first argument, should be listlist, but has received
{{{42, 70, 450, 75}, {47, 64, 16, 23}, {67, 43, 23, 16}, {62, 48, 15, 22}}, Decrease}
In[1045]:= SortNL[L, 2, Art_Kr]
SortNL::incorrect third argument, should be {Less, Greater},
but has received {Art_Kr}
In[1046]:= SortNL[{{a, b, c}, {69, 75, 450}, {64, 44, 16}}, 2, Greater]
SortNL::incorrect first argument, should be numerical listlist, but has received
{{a, b, c}, {69, 75, 450}, {64, 44, 16}}
In[1062]:= SortNL1[L_ /; ListQ[L], p_ /; IntegerQ[p], b_] :=
If[DeleteDuplicates[Map[ListQ, L]] == {True} &&
Length[DeleteDuplicates[Map[Length, L]]] == 1,
If[p >= 1 && p <= Length[First[L]], If[MemberQ[{Greater, Less}, b],
Sort[L, b[GC[#1[[p]]], GC[#2[[p]]]] &],
Print[SortNL1::incorrect 3-rd argument, should be {Less | Greater}
but had received ", {b}]],
Print[SortNL1::incorrect 2-nd argument, should lay in interval ",
{1, Length[First[L]]}],
Print[SortNL1::incorrect first argument, should has ListList-type
but had received ", L]]
In[1063]:= SortNL1[{{42, Gs, Ps}, {15, 22, 75}, {16, Sv, 75}}, 2, Greater]
Out[1063]= {{16, Sv, 75}, {42, Gs, Ps}, {15, 22, 75}}
In[1123]:= SortLpos[L_ /; ListQ[L], n_ /; IntegerQ[n], SF_] :=
If[! 1 <= n <= Min[Map[Length, L]], $Failed,
If[! MemberQ[{Greater, Less}, SF], $Failed,
Sort[L, SF[#2[[n]], #1[[n]]] &]]
In[1124]:= L = {{63, 68, 43}, {14, 22, 48}, {42, 47, 67}, {96, 72, 420}}; L1 = {{42, 47}, {14, 22, 72}}
In[1125]:= SortLpos[L, 2, Greater]
Out[1125]= {{14, 22, 48}, {42, 47, 67}, {63, 68, 43}, {96, 72, 420}}
In[1126]:= SortLpos[L, 2, Less]
Out[1126]= {{96, 72, 420}, {63, 68, 43}, {42, 47, 67}, {14, 22, 48}}
In[1127]:= {SortLpos[L, 2, GreaterEqual], SortLpos[L, 7, Less], SortLpos[L1, 2, Greater]}
Out[1127]= {$Failed, $Failed, {{14, 22, 72}, {42, 47}}}

```

Достаточно полезной модификацией процедуры **SortNL** является функция **SortNL1**, вызов которой **SortNL1[L, p, b]** возвращает результат сортировки списка *L* типа *ListList* согласно элементов в *p*-позиции его подсписков на основе их уникальных десятичных кодов, определяемых GC-процедурой; **b = {Greater | Less}**. В определенном отношении функция **SortNL1** расширяет предыдущую процедуру **SortNL**.

В отличие от процедуры **SortNL** процедура **SortLpos[L, n, SF]**, представленная концом фрагмента, обеспечивает сортировку числового вложенного списка необязательно типа *ListList* 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 все возникающие основные особые и ошибочные ситуации.

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** пакета не решают данной задачи, поэтому для этих целей достаточно успешно можно использовать процедуру **Range1**, **Range2** или **Range3**, исходные коды которых наряду с типичными примерами применения представляет нижеследующий фрагмент.

```
In[1406]:= 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[1407]:= Range1[$Kr_Art1, $Kr_Art7]
Out[1407]= {$Kr_Art1, $Kr_Art2, $Kr_Art3, $Kr_Art4, $Kr_Art5, $Kr_Art6, $Kr_Art7}
In[1408]:= Range1[$Rans_Ian1, $Rans_Ian6]
Out[1408]= {$Rans_Ian1, $Rans_Ian2, $Rans_Ian3, $Rans_Ian4, $Rans_Ian5, $Rans_Ian6}
In[1409]:= 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[1410]:= Range2[Kr, 14]
Out[1410]= {Kr1, Kr2, Kr3, Kr4, Kr5, Kr6, Kr7, Kr8, Kr9, Kr10, Kr11, Kr12, Kr13, Kr14}
In[1411]:= a = Range2[x, 9]; a = {1, 2, 3, 4, 5, 6, 7, 8, 9}; {x1, x2, x3, x4, x5, x6, x7, x8, x9}
Out[1411]= {x1, x2, x3, x4, x5, x6, x7, x8, x9}
In[1412]:= 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[1413]:= Agn[Range3[z, 10]] := Module[{}, (z1 + z3 + z5 + z7 + z9)/(z2 + z4 + z6 + z8 + z10)]
In[1414]:= Range3[h, 15]
Out[1414]= {h1_, h2_, h3_, h4_, h5_, h6_, h7_, h8_, h9_, h10_, h11_, h12_, h13_, h14_, h15_}
In[1415]:= Agn[Range[16]]
Out[1415]= Agn[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}]

```

Вызов **Range1**[*J*, *p*] возвращает список переменных в форме {*J*1, *J*2, *J*3, ..., *J**p*}, 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: {*J*1, *J*2, *J*3, ..., *J**p*}; с другой стороны, вызов процедуры **Range3**[*J*, *p*] возвращает список в форме {*J*1\_, *J*2\_, *J*3\_, ..., *J**p*\_}, где *J* - идентификатор и *p* - целое число. При этом, процедура **Range3** в ряде случаев создает довольно хорошие предпосылки удобного определения функций/процедур с большим числом формальных аргументов, как иллюстрирует пример предыдущего фрагмента. Все эти процедуры наряду с другими процедурами, представленными в настоящей книге, можно найти в свободно распространяемом пакете «AVZ\_Package» [90], содержащем более 320 программных средств (процедур, функций, переменных).

Из ряда приложений, в частности, из примера применения процедуры **Range2** видно, что принятого в пакете присвоения списку переменных значений не производится и данную задачу решает довольно полезная процедура **AssignToList**[*L*, *Z*, *n*], чей вызов присваивает значения элементов списка *L* соответствующим элементам программно сгенерированного списка {*Z*1, *Z*2, ..., *Z**n*} (*n* ≥ 1). Следующий фрагмент представляет исходный код процедуры **AssignToList** с типичными примерами ее использования.

```

In[1368]:= 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[1369]:= {AssignToList[Range[9], z, 9], z1, z3, z5, z6, z9}
Out[1369]= {{1, 2, 3, 4, 5, 6, 7, 8, 9}, 1, 3, 5, 6, 9}
In[1370]:= {AssignToList[CharacterRange["A", "H"], x, 7], x1, x2, x3, x4, x5, x6, x7}
Out[1370]= {{A, B, C, D, E, F, G}, A, B, C, D, E, F, G}

```

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 применения.

```

In[1859]:= 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[1860]:= ListListGroup[{{75, 2}, {450, 6}, {15, 2}, {22, 2}, {69, 6}}, 2]
Out[1860]= {{{75, 2}, {15, 2}, {22, 2}}, {{450, 6}, {69, 6}}}
In[1861]:= ListListGroup[{{75, 2}, {450, 6}, {15, 2}, {22, 2}, {69, 6}}, 6]
Out[1861]= ListListGroup[{{75, 2}, {450, 6}, {15, 2}, {22, 2}, {69, 6}}, 6]
```

На недопустимых фактических аргументах вызов возвращается невычисленным.

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 ситуация, как довольно наглядно иллюстрируют *первые* четыре примера следующего фрагмента. С целью устранения данного 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 *большого* 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[2810]:= Sin[{a, b, c, d, h}]
Out[2810]= {Sin[a], Sin[b], Sin[c], Sin[d], Sin[h]}
In[2811]:= {x, y, z, h}/{a, b, c, d}
Out[2811]= {x/a, y/b, z/c, h/d}
In[2812]:= {x, y, z, h} + {a, b, c, d, g}
Thread::tdlen: Objects of unequal length in {x, y, z, h} + {a, b, c, d, g}
cannot be combined. >>
Out[2812]= {x, y, z, h} + {a, b, c, d, g}
In[2813]:= 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[2814]:= ListOp[{x, y, z, h, w}, {a, b, c, d, e}, #1 + #2 &]
Out[2814]= {a + x, b + y, c + z, d + h, e + w}
In[2815]:= ListOp[{x, y, z, h, w}, {a, b, c}, #1^#2 &]
Out[2815]= {x^a, y^b, z^c, h, w}
In[2816]:= ListOp[{x, y, z, h, w}, {a, b, c, d, e}, #1 * #2 &]
Out[2816]= {a x, b y, c z, d h, e w}
In[2817]:= ListOp[{x, y, z, h, w}, {}, #1 * #2 &]
Out[2817]= {x, y, z, h, w}
In[2818]:= {True, False, True} && {True, True, False}
Out[2818]= {True, False, True} && {True, True, False}
In[2819]:= ListOp[{True, False, True}, {True, True, False}, #1 && #2 &]
Out[2819]= {True, False, False}
In[2820]:= ListOp[{True, False, True}, {True, True, False}, #1 || #2 &]
Out[2820]= {True, True, True}

```

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 применения.

```

In[7]:= 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[8]:= ListToString[{a + b, {"Agn", 64}, Kr, 15, Art, 23, "RANS", {{Avz || 70}}}, "&"]
Out[8]= "a + b&\Agn\&64&Kr&15&Art&23&\RANS\&Avz || 70"
In[9]:= ListToString["a + b&\Agn\&64&Kr&15&Art&23&\RANS\&Avz || 70", "&"]
Out[9]= {a + b, "Agn", 64, Kr, 15, Art, 23, "RANS", Avz || 70}

```

В задачах, использующих *вложенные* списки, довольно актуальным вопросом является определение *уровней* их вложенности. 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. При этом, в завершение фрагмента, приведенного ниже, представлены как процедура **MemberQ1**, в определенной мере расширяющая стандартную функцию **MemberQ** на *вложенные* списки, так и процедура **MemberQ2** $[L, x, y]$ , которая расширяет эту же стандартную 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 **False** возвращается. В случае возврата **True** через третий аргумент  $y$  возвращается 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**. Более того, при возврате значения **True** через 3-й аргумент  $y$  возвращается число вхождений  $x$ -значения в список  $L$ . В нижеследующем фрагменте представлены исходные коды упомянутых процедур с примерами их применения.

```
In[721]:= MaxNestLevel[L_ /; ListQ[L]] := Module[{a = Flatten[L], b = L, c = 0},
      While[! A == b, b = Flatten[b, 1]; c = c + 1]; c];
In[722]:= L = {{a, {b, {m, {x, y}, n}, x}, c, {{{{{{{64, 69}}}}}}}}}; Map[MaxNestLevel, {L, {a, b, c}}]
Out[722]= {8, 0}
In[723]:= L = {m, {m, {b, {m, {x, y}, n}, x}, c, {{{{{{{64, m, 69}}}}}}}}}; MemberQ[L, x]
Out[723]= False
In[724]:= MemberQ1[L_ /; ListQ[L], 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[725]:= {{MemberQ1[L, x, y], y}, {MemberQ1[L, 69, z], z}, {MemberQ1[L, m, h], h}}
Out[725]= {{True, {2, 4}}, {True, {8}}, {True, {0, 1, 3, 8}}}
In[726]:= 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[727]:= L = {42, 47, 64, 69, 64, 69, 69, 75, {64, 69, {69}}, 450, 69}; {MemberQ2[L, 69, g], g}
Out[727]= {True, 6}
```

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 подсписков которого определяет номер уровня вложенности вложенного списка  $L$  первым своим элементом и число вхождений выражения  $x$  в данный уровень вторым элементом.

```
In[1826]:= MemberT[L_ /; ListQ[L], x_] := Length[Select[Flatten[L], # == x &]]
In[1827]:= 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], b = Append[b, d], Null]]];
      h = Append[h, {c, p}]; a = Flatten[b, 1]; b = {}; p = 0; h]
```

```

In[1828]:= L = {a, b, {c, d, {d, x, d}, x}, {d, x, h, d}, p, {{d, {d, {a, d, c, d}}}, d, w}
Out[1828]= {a, b, {c, d, {d, x, d}, x}, {d, x, h, d}, p, {{d, {d, {a, d, c, d}}}, d, w}
In[1829]:= {MemberLN[L, d], MemberLN[L, t], MemberT[{a, h, {c, h, g}, s}, h]}
Out[1829]= {{{1, 1}, {2, 3}, {3, 3}, {4, 1}, {5, 2}}, {{1, 0}, {2, 0}, {3, 0}, {4, 0}, {5, 0}}, 2}
In[1830]:= {MemberLN[{{{{{{x, {{{y}}}}}}}], 2], MemberT[{a, h, {c, {{{y}}}}, h, v}, s], 4]}
Out[1830]= {{{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 *L* имеет уровень вложенности 1. Средства достаточно полезны в работе со списками.

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 процедуры **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*. Следующий фрагмент представляет исходные коды процедур **MinusList**, **MinusList1** наряду с наиболее типичными примерами их использования.

```

In[1784]:= MinusList[x_;/; ListQ[x], y_;/; ListQ[y]] := Module[{a, b, c = x, d = {}, k = 1, j},
    For[k, k <= Length[y], k++, b = Flatten[Position[c, y[[k]]]];
    If[b == {}, Continue[], For[j = 1, j <= Length[c], j++,
        d = Append[d, If[j == b[[1]], Null, c[[j]]]]; c = d; d = {}];
    Select[c, ! (# === Null) &]]

In[1785]:= 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[1785]= {"a", "P3", "b", Avz, "m", Kr, Agn, "n", "P3", Art, g}
In[1786]:= {MinusList[{}, {}], MinusList[{a, b, c, d}, {a, b, c, d}]}
Out[1786]= {{}, {}}

In[1798]:= MinusList1[x_;/; ListQ[x], y_;/; ListQ[y]] := Module[{a, b = x, c = y, d, k, j},
    Label[d]; For[k = 1, k <= Length[c], k++,
        For[j = 1, j <= Length[b], j++,
            If[c[[k]] == b[[j]], {b[[j]], c[[k]]} = {Null, Null}, Continue[]];
    b = Select[b, ! (# === Null) &]; a = Select[c, ! (# === Null) &];
    If[a == c, Return[b], c = a; Goto[d]]]

In[1799]:= MinusList1[{x, a, x, y, b, c, 75, x, d, a, b, x, y, 75}, {a, b, c, 75, x, y, a, b, x}]
Out[1799]= {x, d, x, y, 75}
In[1800]:= {MinusList1[{}, {}], MinusList1[{a, b, c, d}, {a, b, c, d}]}
Out[1800]= {{}, {}}

```

Обе процедуры представляют интерес и как вспомогательное средство при решении ряда наших процедур [90], а также самостоятельный интерес в работе со списками.

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 применения.

```
In[1019]:= 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[1020]:= ListAssign[{a + b, Sin[x], "agn", {m, n}}, y]
Out[1020]= {{y$3231, "y$3232", "y$3233", "y$3234"}, {a + b, Sin[x], "agn", {m, n}}}
In[1021]:= ListAssign[{44, 23, 15, 64, 70}, h]
Out[1021]= {"h$2549", "h$2550", "h$2551", "h$2552", "h$2553"}, {44, 23, 15, 64, 70}]
```

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

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 применения.

```
In[1228]:= 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[1229]:= ListStrToStr[{"a", "b", "c", "d", "h", "t", "k", "Art", "Kr"}]
Out[1229]= "a, b, c, d, h, t, k, Art, Kr"
In[1230]:= ListStrToStr[{"(a+b)", ".*", "Sin[x]", " - ", "(c - d)", ".*", "Cos[y]", " == ", "450"}, 6]
Out[1230]= "(a+b)*Sin[x] - (c - d)*Cos[y] == 450"
```

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 типичными примерами ее использования.

```
In[80]:= SymbolToList[x_/: SymbolQ[x]] := Map[ToExpression, Characters[ToString[x]]]

In[81]:= SymbolToList[Art23Kr15]
Out[81]= {A, r, t, 2, 3, K, r, 1, 5}
In[82]:= SymbolToList[$RANS23IAN15AVZArt2012]
Out[82]= {$, R, A, N, 2, 3, I, A, N, 1, 5, A, V, Z, A, r, t, 2, 0, 1, 2}
```

Списочная структура *Mathematica* позволяет достаточно легко симулировать работу со структурами других *CAS*, например, *Maple*. Так, в среде *Maple* одной из наиболее важных структур является табличная структура, довольно 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]. Для симулирования основных операций с *табличной* организацией, аналогичной пакету *Maple*, в среде *Mathematica* может быть использована процедура *Tbl[L, x]*, рассматриваемая в качестве *таблицы* список *L* типа *ListList*, чьи 2-элементные подписки  $\{x, y\}$  отвечают  $\{index, entry\}$  *Maple*-таблиц соответственно. Тогда как в качестве *второго* *x*-аргумента выступает список  $\{a, b\}$ , или слово  $\{ "index" | "entry" \}$ , или выражение иного типа. При этом, в первом случае при наличии в *L* подписка с *первым* элементом *a* он заменяется на  $\{a, b\}$ , в противном случае он дополняет *L*; тогда как во втором случае возвращает список  $\{indices | entries\}$  соответственно; наконец, в третьем случае процедура возвращает вход для *x*-индекса, если такой в данной таблице реально существует.

```
In[2506]:= Tab1 := {{a, a72}, {b, b42}, {c, c47}, {d, d14}, {h, h22}}
In[2507]:= Tbl[L_ /; ListListQ[L] && Length[L[[1]]] == 2, x_] := Module[{a = {}, c, d = {},
    k = 1, b = Length[L]}, For[k, k <= b, k++, a = Append[a, L[[k]][[1]]];
    d = Append[d, L[[k]][[2]]]; {a, d} = Map[DeleteDuplicates, {a, d}];
    If[Length[a] == Length[d] && Length[a] == b, {a, d} = {{}, {}},
    Return[Defer[Tbl[L, x]]]; If[MemberQ[{"index", "entry"}, x],
    For[k = 1, k <= b, k++, a = Append[a, L[[k]][[If[x == "index", 1, 2]]]];
    Return[a], Null]; If[ListQ[x] && Length[x] == 2 && ! ListListQ[x],
    For[k = 1, k <= b, k++, If[L[[k]][[1]] == x[[1]], c = ReplacePart[L, k -> x];
    Return[c], Continue[]]; c = Append[L, x]; Return[c], Null];
    If[ListListQ[x], Return[Defer[Tbl[L, x]]],
    For[k = 1, k <= b, k++, If[L[[k]][[1]] == x, Return[L[[k]][[2]], Continue[]]; $Failed]]
In[2508]:= Tbl[Tab1, {a, 68}]
Out[2508] = {{a, 68}, {b, b42}, {c, c47}, {d, d14}, {h, h22}}
In[2509]:= Tbl[Tab1, {Art, Kr}]
Out[2509] = {{a, a72}, {b, b42}, {c, c47}, {d, d14}, {h, h22}, {Art, Kr}}
In[2510]:= {Tbl[Tab1, c], Tbl[Tab1, "index"], Tbl[Tab1, "entry"]}
Out[2510] = {c47, {a, b, c, d, h}, {a72, b42, c47, d14, h22}}
```

Следующий фрагмент определяет функцию **BinaryListQ[L]**, возвращающую **True** на *бинарном* списке *L* (список может быть вложенным), в противном случае возвращается **False**. В дальнейшем этот тест, определяя, по сути дела, *min* может применяться при тестировании *фактических* аргументов в целом ряде важных процедур и функций.

```
In[1017]:= BinaryListQ[L_] := ListQ[L] &&
    MemberQ[{0, 1, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]]
In[1018]:= BinaryListQ[{{1, 0, 1, 0, {1, 1, 0, 1}, {0, 1}}}]
Out[1018] = True
```

Следующая процедура несколько расширяет стандартную функцию **Gather** пакета и оказывается 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 обоих процедур наряду с некоторыми типичными примерами их использования.

```
In[670]:= 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[671]:= L = {{42, V, 1}, {47, G, 2}, {67, S, 1}, {68, V, 2}, {63, G, 3}, {43, S, 2}}; Gather1[L, 2]
Out[671]= {{{42, V, 1}, {68, V, 2}}, {{47, G, 2}, {63, G, 3}}, {{67, S, 1}, {43, S, 2}}}
In[672]:= L = {{42, V, 1}, {47, G, 2}, {64, S, 1}, {69, V, 2}, {64, G, 3}, {44, S, 2}}; Gather1[L, 3]
Out[672]= {{{42, V, 1}, {64, S, 1}}, {{47, G, 2}, {69, V, 2}, {44, S, 2}}, {{64, G, 3}}}
In[98]:= Gather2[x_;/; ListQ[x]] := 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[99]:= Gather2[{"a", 450, "a", 75, "y", 75, "d", "h", "c", "d", 75, 450}]
Out[99]= {"a", 2}, {450, 2}, {75, 3}, {"d", 2}
In[100]:= {Gather2[{"a", "g", "y"}], Gather2[{75, "a", 75, "g", "y", 75]}}
Out[100]= {{}, {75, 3}}
```

Вызов *Maple*-процедуры **PartialSums(L)** возвращает список частичных сумм элементов списка *L*. Тогда как следующая одноименная процедура **PartialSums[L]** в *Mathematica* выполняет ту же функцию с тем только отличием, что при кодировании символа *L* в строчном формате вызов процедуры обновляет исходный список *L* на месте. Пример представляет исходный код процедуры **PartialSums** и некоторые ее применения.

```
In[1973]:= 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[1974]:= PartialSums[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}]
Out[1974]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
In[1975]:= GS = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; PartialSums["GS"]
Out[1975]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
```

```
In[1976]:= GS
Out[1976]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
In[1977]:= SV = {a, b, c, d, e, f}; PartialSums["SV"]
Out[1977]= {a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
In[1978]:= SV
Out[1978]= {a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
```

Следующие 2 простые функции достаточно полезны в работе с числовыми списками. Вызов **PosIntQ[n]** возвращает **True**, если  $n$  – целое положительное число, в противном случае возвращается **False**. Тогда как вызов **PosIntListQ[W]** возвращает **True**, если  $W$  – список целых положительных чисел, в противном случае возвращает **False**. Фрагмент представляет исходные коды обеих функций с примерами их применения.

```
In[1210]:= PosIntQ[n_] := If[IntegerQ[n] && n > 0, True, False]
In[1211]:= Map[PosIntQ, {23, 15, a + b, 69, 47.64, -450, 0}]
Out[1211]= {True, True, False, True, False, False, False}
In[1212]:= PosIntListQ[L_ /; ListQ[L]] :=
    If[DeleteDuplicates[Map[PosIntQ, L]] == {True}, True, False]
In[1213]:= Map[PosIntListQ, {{23, 15, 69, 64, 44, 6}, {23, 15, a + b, 69, 47.64, -450, 0}}]
Out[1213]= {True, False}
```

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 применения.

```
In[1308]:= ListPosition[x_ /; ! NestListQ[x], y_ /; ListQ[y]] := Module[{a = {}, c = Length[y],
    k = 1}, While[k <= c, AppendTo[a, Flatten[Position[x, y[[k]]]]; k++]; a]
In[1309]:= ListPosition[{1, 2, 3, A, G, 2, V, 1, 3, S, G, K, G, 1, G}, {1, 2, 3, G}]
Out[1309]= {{1, 8, 14}, {2, 6}, {3, 9}, {5, 11, 13, 15}}
In[1310]:= ListPosition[{1, 2, 3, {A, G, 2, V, 1, 3, S, G}, K, G, 1, G}, {1, 2, 3, G}]
Out[1310]= ListPosition[{1, 2, 3, {A, G, 2, V, 1, 3, S, G}, K, G, 1, G}, {1, 2, 3, G}]
```

Пакет не поддерживает присвоений типа  $x = y$  при разных длинах списков  $x$  и  $y$ , тогда как эта операция поддерживается вызовом **ListsAssign[x, y]**, возвращая значение  $x$ .

```
In[95]:= ListsAssign[x_ /; ListQ[x], y_ /; ListQ[y]] := 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[96]:= L1 = {x, 75, a + b, Sin, t, s}; L2 = {a, b, c, w, 70}; ListsAssign[L1, L2]
Out[96]= {a, 75, a + b, Sin, 70, s}
```

Фрагмент содержит исходный код процедуры **ListsAssign** с примером применения.

### 7.3. Дополнительные средства для работы со структурами типа последовательность в программной среде пакета Mathematica

Последовательности выражений (или просто последовательности) в среде многих языков образуются на основе оператора запятой (,) и служат основой для определения многих типов данных (запросы процедур, списки, множества, индексы и др.). Между тем, в пакете *Mathematica* данная структура в качестве самостоятельной отсутствует и в качестве ее выступает списочная структура; некоторые языки придерживаются той же концепции. Следующий фрагмент представляет процедуры, обеспечивающие 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[997]:= A := Seq[a, b, c, d, h, r, x, y, z]
In[998]:= SeqQ[x_] := Block[{a = ToString[x]}, If[StringLength[a] >= 4 &&
StringTake[a, {1, 4}] == "Seq[" && StringTake[a, {-1, -1}] == "]", True, False]]
In[999]:= Map[SeqQ, {A, {a, b, c}, Agn}]
Out[999]= {True, False, False}
In[1000]:= SeqToList[x_]; SeqQ[x] := ToExpression["{" <> StringTake[ToString[x],
{5, -2}] <> "}"]
In[1001]:= SeqToList[A]
Out[1001]= {a, b, c, d, h, r, x, y, z}
In[1002]:= ListToSeq[x_]; ListQ[x] := ToExpression["Seq[" <> StringTake[ToString[x],
{2, -2}] <> ""]
In[1003]:= ListToSeq[{42, 47, 69, 15, 23}]
Out[1003]= Seq[42, 47, 69, 15, 23]
In[1004]:= 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[1005]:= SeqIns[A, {75, 450}, 23]
Out[1005]= Seq[a, b, c, d, h, r, x, y, z, 75, 450]
In[1006]:= SeqIns[A, Seq[42, 47, 69, 15, 23], 5]
Out[1006]= Seq[a, b, c, d, h, 42, 47, 69, 15, 23, r, x, y, z]
```

```
In[1007]:= SeqDel[x_ /; SeqQ[x], y_] := Module[{a = SeqToList[x], b = If[SeqQ[y],
      SeqToList[y], y]}, ListToSeq[Select[a, ! MemberQ[Flatten[{b}], #] &]]]
In[1008]:= SeqDel[A, Seq[a, b, c, d]]
Out[1008]= Seq[h, r, x, y, z]
```

Дополнительно к функции **SeqToList** можно отметить простую функцию, чей вызов **SeqToList1[a, b, c, ...]** возвращает список ее фактических аргументов, а именно:

```
In[1009]:= SeqToList1[x___] := {x}
In[1010]:= x = 75; {SeqToList1[a, b, c, d, r, h, p], SeqToList1[]}
Out[1010]= {{a, b, c, d, r, h, p}, {}
In[1011]:= SeqToString[h___] := StringTake[ToString1[{h}], {2, -2}]
In[1012]:= {SeqToString[a, c + d, Sin[x], a^2 + (x + y)/75], SeqToString[], "VGS"}
Out[1012]= {"a, c + d, Sin[x], a^2 + (x + y)/75", "", "VGS"}
```

А также простую функцию, вызов которой **SeqToString[a, b, c, ...]** возвращает список фактических аргументов в строчном формате, как иллюстрирует последний пример предыдущего фрагмента. Как отмечалось выше, механизм, поддерживающий работу с объектами типа *последовательность* (*sequence*), в *Mathematica* отсутствует, что создает определенные затруднения. Между тем, пакет предоставляет функцию **Sequence[a, b, c, ...]**, определяющую последовательность аргументов, автоматически передаваемых произвольной функции/процедуре. И в данном контексте процедура **SequenceQ[s]** обеспечивает тестирование объектов, которые созданы на основе функции **Sequence**, возвращая **True**, если *s*-объект определен этой функцией, и **False** в противном случае; при этом, имя *s*-объекта должно кодироваться в строчном формате.

```
In[799]:= S = Sequence[a, b]; G = Sequence[x, y]; Art := Sequence[c, d]; Kr = Sequence[z];
In[800]:= SequenceQ[s_String] := Module[{a, b = " := ", c = " = ",
      d = Quiet[ToString[Definition[s]]], Quiet[Check[If[StringTake[
      StringReplace[d, {s <> b -> "", s <> c -> ""}], {1, 9}] ==
      "Sequence[" , True, False], False]]]
In[801]:= Map[SequenceQ, {"S", "G", "Art", "Kr", "GrGu"}]
Out[801]= {True, True, True, True, False}
In[802]:= SeqUnion[x_] := Sequence[x]
In[803]:= SeqUnion[a, b, c, Art, Kr, S, G]
Out[803]= 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 пакетом не поддерживается и для работы с которой *Mathematica* стандартными средствами не располагает.

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. Фрагмент представляет исходный код процедуры и ее применение.

```
In[1994]:= 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[1995]:= {F[Sequence[{x, y, z}]], F[Sequences[{x, y, z}]], F[Sequences[x, y, z]]}
Out[1995]= {F[{x, y, z}], F[x, y, z], F[x, y, z]}
```

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  $s$ . Итак, вызовы процедуры **ListStrList** функционируют по принципу переключателя.

```
In[1604]:= 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[Mapp[StringJoin, Map[ToString1, x], a]], {1, -2}], x]]]
In[1605]:= L = ListStrList[{Avz, 70, Agn, 65, Art, 23, Kr, 16, Vsv, 45}]
Out[1605]= "Avz□70□Agn□65□Art□23□Kr□16□Vsv□45"
In[1606]:= ListStrList[ListStrList[{Avz, 70, Agn, 65, Art, 23, Kr, 16, Vsv, 45}]]
Out[1606]= {Avz, 70, Agn, 65, Art, 23, Kr, 16, Vsv, 45}
In[1607]:= ListStrList["Avz70Agn65Art23Kr16Vsv45"]
Out[1607]= "Avz70Agn65Art23Kr16Vsv45"
In[1608]:= ListStrList[ListStrList[{"Avz", 70, "Agn", 65, "Art", 23, "Kr", 16, "Vsv", 45}]]
Out[1608]= {"Avz", 70, "Agn", 65, "Art", 23, "Kr", 16, "Vsv", 45}
In[1609]:= Map[ListStrList, {"□", "□□", "□□□", "□ □", "□ □□"}]
Out[1609]= {{}, {}, {}, {Null}, {Null}}
```

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*.

## 7.4. Дополнительные средства для работы с выражениями в программной среде пакета Mathematica

В настоящем разделе представлен ряд средств, расширяющих стандартные функции пакета, ориентированные на работу с выражениями в широком их понимании.

В ряде случаев требуется выполнить *обмен* значений переменных с соответствующим обменом всех их атрибутов. Например, переменные *x* и *y*, имеющие значения 68 и 63, должны получить значения 64 и 69 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.

```
In[1545]:= 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[1546]:= Agn = 64; Avz := 69; Art := 23; Kr = 15; SetAttributes["Agn", Protected];
SetAttributes["Art", Protected];

In[1547]:= Map[Attributes, {"Agn", "Avz", "x", "y", "Art", "Kr"}]
Out[1547]= {{Protected}, {}, {}, {}, {Protected}, {}}

In[1548]:= VarExch[{{"Avz", "Agn"}, {"x", "y"}, {"Art", "Kr"}}]
VarExch::Both actual arguments should be defined but
uncertainty had been detected: {x,y}

In[1549]:= {Avz, Agn, Art, Kr}
Out[1549]= {64, 69, 15, 23}

In[1550]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[1550]= {}, {Protected}, {}, {Protected}
```

```

In[1551]:= Rename[x_String /; HowAct[x], y_ /; !HowAct[y]] := Module[{a = Attributes[x],
    ClearAttributes[x, a]; ToExpression[ToString[y] <>
    StringTake[ToString1[DefFunc[x]], {StringLength[x] + 1, -1}]];
    SetAttributes[y, a]; Clear[x]]

In[1552]:= fm = "Art_Kr"; SetAttributes[fm, {Protected}]; {fm, Attributes[fm]}
Out[1552]= {"Art_Kr", {Protected}}
In[1553]:= Rename["fm", Tampere]
In[1554]:= {Tampere, Attributes[Tampere], fm}
Out[1554]= {"Art_Kr", {Protected}, fm}
In[1555]:= s = HowAct[Q]; SetAttributes[Q, {Listable}]; {s, Attributes[Q]}
Out[1555]= {False, {Listable}}
In[1556]:= Rename[Tampere, Q]
Out[1556]= Rename["Art_Kr", Q]

In[1557]:= 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[1558]:= Agn = 64; Avz := 69; Art := 23; Kr = 15; SetAttributes["Agn", Protected];
    SetAttributes["Art", Protected];
In[1559]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[1559]= {{Protected}, {}, {Protected}, {}]
In[1560]:= {$Art$, $Kr$} = {75, 450}; VarExch1[{"Agn", "Avz"}, {"x", "y"}, {"Art", "Kr"}]
In[1561]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]}
Out[1561]= {{69, 64, 15, 23}, {{}, {Protected}, {}, {Protected}}}
In[1562]:= {x, y, $Art$, $Kr$}
Out[1562]= {x, y, 75, 450}
    
```

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.

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 пакета *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 пакета процедур, содержащих *глобальные* переменные, в частности, из *nb*-файлов с последующей активизацией их содержимого.

В качестве примера приведем фрагмент, в котором загружается в текущий сеанс наш пакет *AVZ\_Package* [90] с процедурой **VarExch1**, использующей две переменные **\$Art\$** и **\$Kr\$** в качестве глобальных, после чего данная процедура используется для обмена значений у переменных *Agn*, *Avz*, *Art* и *Kr*, из которых 2 переменные *Agn* и *Art* имеют *Protected*-атрибут. Из результата вызова процедуры **VarExch1** следует, что исходные значения (*до вызова процедуры*) глобальных переменных **\$Art\$** и **\$Kr\$** не сохраняются в результате выполнения процедуры **VarExch1**, да и значения для переменных *Agn*, *Avz* получаем некорректными (*причина здесь не детализируется*). Поэтому для устранения данной ситуации предлагается процедура, чей вызов **NbCallProc**[*x*] переактивирует в текущем сеансе процедуру/функцию *x*, чье определение находилось в загруженном и активированном в текущем сеансе *nb*-файле (*в данном случае в файле AVZ\_Package.nb*).

```
In[1009]:= Agn = 64; Avz := 69; Art := 23; Kr = 15; SetAttributes["Agn", Protected];
SetAttributes["Art", Protected];
In[1011]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[1011]= {{Protected}, {}, {Protected}, {}}
In[1012]:= {$Art$, $Kr$} = {75, 450}; VarExch1[{"Agn", "Avz"}, {"x", "y"}, {"Art", "Kr"}]
```

```

In[1013]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]}
Out[1013]= {{450, 75, 15, 23}, {{}, {Protected}}, {{}, {Protected}}}
In[1014]:= {x, y, $Art$, $Kr$}
Out[1014]= {x, y, $Art$, $Kr$}
In[1015]:= NbCallProc[x_ /; ProcQ[x] || Args[x] != {}] :=
    Module[{a = SubsDel[StringReplace[ToString1[DefFunc[x]], "\n\n" -> ";",
        "" <> ToString[x] <> "", {"[", ",", "-1"}], Clear[x]; ToExpression[a]}
In[1016]:= NbCallProc[VarExch1]
In[1017]:= Unprotect[Agn, Avz, Art, Kr, $Art$, $Kr$]
Out[1017]= {"Avz", "Kr"}
In[1018]:= Clear[Agn, Avz, Art, Kr, $Art$, $Kr$]
In[1019]:= Agn = 64; Avz := 69; Art := 23; Kr = 15; SetAttributes["Agn", Protected];
    SetAttributes["Art", Protected];
In[1021]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[1021]= {{Protected}}, {{}, {Protected}}, {{}}
In[1022]:= {$Art$, $Kr$} = {75, 450}; VarExch1[{"Agn", "Avz"}, {"x", "y"}, {"Art", "Kr"}]
In[1023]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]}
Out[1023]= {{69, 64, 15, 23}, {{}, {Protected}}, {{}, {Protected}}}
In[1024]:= {x, y, $Art$, $Kr$}
Out[1024]= {x, y, 75, 450}

```

Следует отметить, что вызов процедуры **NbCallProc**[*x*] переактивизирует в текущем сеансе все определения процедур/ функций 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 *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 пакета.

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[494]:= Agn = 64; SetAttributes[Agn, Protected]; Attributes[Agn];
In[495]:= Art23[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
Return[Defer[Art23[p]]]], b = 450, c], c = N[Sin[75]]; {a, b, c}}
In[496]:= Art23[{"Agn", 69}]
Out[496]= {Art23[{"Agn", 69}], 450, -0.387782}
In[497]:= Art23[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
Print[Defer[Art23[p]]]; Abort[]], b = 450, c], c = N[Sin[75]]; {a, b, c}}
In[498]:= Art23[{"Agn", 69}]
Art23[{Agn, 69}]
Out[498]= $Aborted
In[499]:= Art23[p_List] := Module[{a = Quiet[Check[Map[Attributes, p], $Aborted]],
b = 450, c], If[a == $Aborted, Return[Defer[Art23[p]], Null]; c = N[Sin[75]]; {a, b, c}}
In[500]:= Art23[{"Agn", 69}]
Out[500]= Art23[{"Agn", 69}]

```

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 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 {}. Следующий простой пример иллюстрирует сказанное.

```

In[455]:= Nvalue[x_] := Module[{a = {}, b = Names["*"], k = 1}, For[k, k <= Length[b], k++,
If[ToExpression[b[[k]]] == x, a = Append[a, b[[k]], Next[]]; a]
In[456]:= Map[Nvalue, {69, 64, 15, 23, 75, 450, Null, 2012}]
Out[456]= {"Agn", "Avz", "Art", "Kr", "Varn", "$Art$", "Vsv", "$Kr$", "Grsu", {}}

```

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, вызов которой **ProtectedQ[x]** возвращает **True**, если объект *x* имеет атрибут **Protected**, и **False** в противном случае. Функционально эквивалентная процедура **ProtectedQ1[x]** также возвращает **True**, если объект *x* имеет атрибут **Protected**, и **False** в противном случае. Процедура **ProtectedQ1** представлена с целью иллюстрации применения механизма обработки *особых и ошибочных* ситуаций для получения требуемого результата. Следующий фрагмент представляет исходные коды обоих средств наряду с некоторыми примерами их использования.

```
In[1535]:= ProtectedQ[x_] := If[MemberQ[Attributes[x], Protected], True, False]
In[1536]:= HS=75; Protect[HS]; Map[ProtectedQ, {Sin, Protect, AVZ, HowAct, Map, "HS"}]
Out[1536]= {True, True, False, False, True, True}
In[1537]:= ProtectedQ1[x_] := Module[{h}, Quiet[Check[Clear[x], h = "error",
Clear::wrsym]]; If[h == "error", True, False]]
In[1538]:= Avz = 75; S = a+b; Protect[ArtKr]; Map[ProtectedQ1, {"Avz", "S", "Sin", "ArtKr"}]
Out[1538]= {False, False, True, True}
In[1539]:= {Avz, S, Sin, ArtKr}
Out[1539]= {75, a + b, Sin, ArtKr}
```

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 типичными примерами ее использования.

```
In[1016]:= ToList[expr_] := Module[{a, b, c = {}, d, k = 1, n}, If[StringQ[expr],
Return[Characters[expr]], If[ListQ[expr], Return[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[1017]:= ToList[(a*Sin[x] + g[b])/(c + d) + (d + c)/(Cos[y] + h)]
Out[1017]= {(c + d)/(h + Cos[y]), (g[b] + a Sin[x])/(c + d)}
In[1018]:= ToList[(a*Sin[x] + g[b])*(c + d) + (d + c)/(Cos[y] + h)]
Out[1018]= {(c + d)/(h + Cos[y]), (c + d) (g[b] + a Sin[x])}
In[1019]:= x = "qwertyuiopasdfghjklzxcvbnm"; ToList[x]
Out[1019]= {"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"}
In[1020]:= x = {a, b, c, d, e, f, g, h}; ToList[x]
Out[1020]= {a, b, c, d, e, f, g, h}
```

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 применения приведены ниже.

```
In[730]:= 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[731]:= {RhsLhs[(a + b)*d -> c, "Lhs", x], x}
Out[731]= {(a + b) d, Rule}
In[732]:= RhsLhs[a + b -> c^5, "Rhs"]
Out[732]= c^5
In[733]:= {RhsLhs[a + b^c + d, "Rhs", y], y}
Out[733]= {b^c + d, Plus}
In[734]:= {RhsLhs[a/d + b^c, "Rhs", z], z}
Out[734]= {a/d, Plus}
In[735]:= {RhsLhs[a == c + d, "Rhs", h], h}
Out[735]= {c + d, Equal}
In[736]:= {RhsLhs[a/c^d, "Rhs", t], t}
Out[736]= {c^d, Times}
In[737]:= {RhsLhs[Art_Kr, "Rhs", g], g}
Out[737]= {RhsLhs[Art_Kr, "Rhs", g], g}
In[738]:= {RhsLhs[75 ;; 450, "Rhs", s], s}
Out[738]= {450, Span}
```

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 выражение  $x$  на левую и правую части. Фрагмент представляет исходный код и примеры применения.

```
In[1100]:= 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[1101]:= RhsLhs1[Sin[x]]
Out[1101]= RhsLhs1[Sin[x]]
In[1102]:= Mapp[RhsLhs1, {a > b, a + b, a^b, a*b, a -> b, a <= b, a | | b}, "Rhs"]
Out[1102]= {b, b, b, b, b, b, b}
In[1103]:= Mapp[RhsLhs1, {a > b, a + b, a^b, a*b, a -> b, a <= b, a | | b}, "Lhs"]
Out[1103]= {a, a, a, a, a, a, a}
In[1104]:= {RhsLhs1[a | | b, "Rhs", w], w}
Out[1104]= {b, {Or, " | "}}
```

При этом, в случае невозможности выделения частей вызов процедур возвращается невычисленным, как иллюстрируют 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  $y$  возвращается тип выражения  $x$ . Следующий фрагмент представляет исходный код процедуры с примерами ее применения.

```
In[834]:= 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[835]:= {TwoHandQ[a3 <= w, h], h}
Out[835]= {True, LessEqual}
In[836]:= {TwoHandQ[a -> b, t], t}
Out[836]= {True, Rule}
In[837]:= {TwoHandQ[a == b && c, z], z}
Out[837]= {True, And}
In[838]:= {TwoHandQ[a != b, p], p}
Out[838]= {True, Unequal}
In[839]:= Clear[z]; {TwoHandQ[a < b && c, z], z}
Out[839]= {True, And}
In[840]:= {TwoHandQ[a >= b, r], r}
Out[840]= {True, GreaterEqual}
In[841]:= Clear[p]; {TwoHandQ[a || b + c, p], p}
Out[841]= {True, Or}

```

В среде *Maple* определен тип индексированных выражений, что в целом ряде случаев дает возможность более эффективно обрабатывать *индексированные* выражения. В [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 полезными при обработке ряда индексированных выражений.

```

In[1306]:= IndexedQ[x_] := Module[{a = Quiet[ToString[x]], b}, b = StringPosition[a, "[[[";
If[StringTake[a, {-2, -1}] == "[[" && +
SymbolQ[StringTake[a, {1, b[[1]][[1]] - 1}]], True, False]]
In[1307]:= {IndexedQ[Quiet[15[[2 ;; 4]]], IndexedQ[Quiet[ArtKr[[15 ;; 23]]]]}
Out[1307]= {False, True}
In[1308]:= Index[x_/, IndexedQ[x]] := Module[{a = Quiet[ToString[x]], b, c},
b = Flatten[StringPosition[a, "[[[";
ToExpression[StringTake[a, {b[[2]] + 1, -3}]]]
In[1309]:= {Index[Quiet[VSV[[15 ;; 23]]], Index[Quiet[SVEGAL[[75]]]]}
Out[1309]= {15 ;; 23, 75}
In[1310]:= Index[Quiet[450[[2 ;; 4]]]]
Out[1310]= Index[450[[2 ;; 4]]]
In[1311]:= H[1] = 75; H[2] = 47; H[6] = 15; H[9] = 23; H[15] = 42; H[23] = 69;
In[1312]:= (H[6] + H[2] + H[1])/(H[9] + H[23] + H[15])
Out[1312]= 137/134
In[1313]:= ArrayInd[x_/, StringQ[x]] := Module[{a = StringSplit[ToString[InputForm[
DefFunc[x]]], "\n\n"]], If[a == {"Null"}, {}, a]]

```

```
In[1314]:= Map[ArrayInd, {"H", "G"}]
Out[1314]= {"H[1] = 75", "H[2] = 47", "H[6] = 15", "H[9] = 23", "H[15] = 42", "H[23] = 69"}, {}
```

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 процедуры **ArrayInd** возвращает пустой список либо список с определением объекта  $H$ , заданного в строчном формате.

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 найден в текущем сеансе пакета, что подтверждает и результат вызова функции **Names**. И лишь после его вычисления в текущем сеансе, обеспечиваем его ввод в «поле зрения» пакета. Этим *Mathematica*, на наш взгляд, достаточно существенно отличается от пакета *Maple* и, прежде всего, при разработке средств *системного* характера, ориентированных на массовое применение.

```
In[787]:= ArrayInd["W"]
          Definition::notfound: Symbol W not found. >>
Out[787]= {}
In[788]:= Names["*"]
Out[788]= {"H", "P"}
In[789]:= W; ArrayInd["W"]
Out[789]= {}
In[790]:= Names["*"]
Out[790]= {"H", "P", "W"}
In[74]:= Characters[Art22Kr15]
Out[74]= Characters[Art22Kr15]
```

Весьма простая функция **GC** может оказаться довольно полезной в ряде приложений, связанных, в частности, с задачами сортировки. Вызов **GC** $[x]$  возвращает уникальный десятичный код  $x$ -выражения. При этом, из примеров применения функции следует, что результаты, возвращаемые на выражениях идентичного содержания в *символьном*

и строчном форматах различны. Следующий фрагмент представляет исходный код и довольно типичные примеры использования данной функции.

```
In[213]:= GC[x_] := ToExpression[StringReplace[ToString[ToCharacterCode[ToString[
      InputForm[x]]], {"{" -> "", "," -> "", "}" -> ""}]]]
In[214]:= Map[GC, {"RANS_IAN_RAE_RAC", 75, 42.47, 2012, Avz}]
Out[214]= {1662408248110021633239686070000000, 2915, 342056000, 5880000, 935740}
In[215]:= Map[GC, {{GS, "GS"}, {a/b, "a/b"}, {a + b, "a + b"}}]
Out[215]= {869057467240512000, 4995361808033786112000, 4384378684678734486700032000}
```

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[1289]:= 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[1290]:= Map[Ind, {Sin[x], (a + b)/(c + d), Sin[x] + Cos[x], AG[a + b], SV[x, y, z]}]
Out[1290]= {{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 довольно типичными примерами ее использования.

```
In[720]:= OP[expr_] := Module[{a = ToString[InputForm[expr]], b = {}, c, d, k, h},
      If[StringTake[a, {-1, -1}] == "]", a = Flatten[Ind[expr]],
      a = DeleteDuplicates[Quiet[ToList[expr]]];
      Label[450]; d = Length[a];
      For[k = 1, k <= Length[a], k++, h = a[[k]]; c = Quiet[ToList[h]];
      If[MemberQ[DeleteDuplicates[c], $Failed], b = Append[b, Ind[h]],
      b = Append[b, c]]; a = DeleteDuplicates[Flatten[b]];
      If[d == Length[a], Sort[a], b = {}; Goto[450]]]
In[721]:= OP[(a + b)/(c + g[d])]
Out[721]= {-1, a, b, c, d, g}
In[722]:= OP[Cos[x] + Sin[y]*Tan[z]]
Out[722]= {Cos, Sin, Tan, x, y, z}
In[723]:= OP[Sqrt[(a + b)/(c + d)] + Sin[x]*Cos[y]]
Out[723]= {-1, 1/2, a, b, c, Cos, d, Sin, x, y}
```

```
In[724]:= OP[(Log[(a + b)/(c + d)] + Sin[x]*Cos[y])/(G[h, g, t] - w^2)]
Out[724]= {-1, 2, a, b, c, Cos, d, g, G, h, Log, Sin, t, w, x, y}
In[725]:= Map[OP, {{Sin[x]}, G[h, g, t], A[m, p]/G[t, q]}]
Out[725]= {{Sin, x}, {g, G, h, t}, {-1, A, G, m, p, q, t}}
```

В целом ряде приложений несомненный интерес представляет определенный аналог *Maple*-процедуры *whattype(e)*, возвращающей тип выражения *e*, который может быть одного из 44 (*Maple 11*) базовых типов данных, как 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[864]:= 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[ProcQ1[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[ProcQ1[x, t]],
If[MemberQ[{"Module", "DynamicModule", "Block"}, t], Return[{t, t = b}][[1]],
t = b; ToString[Quiet[Head[ToExpression[x]]]], t = b; "Undefined"]
In[865]:= t = 450; x = 75; y := 42.47; z = a + b; Map[WhatType, {"Kr", "x", "y", "z", "ProcQ",
"Sin", "F[r]", "WhatType"}]
Out[865]= {"Undefined", "Integer", "Real", "Plus", "Module", "System`", "F", "Module"}
In[866]:= Map[WhatType, {"a^b", "a**b", "3 + 5*I", "{42, 47}", "a&b", "a -> b"}]
Out[866]= {"Power", "NonCommutativeMultiply", "Complex", "List", "And", "Rule"}
In[867]:= Map[WhatType, {"a /; b", "a <> b", "a <-> b", "a | b", "a`b"}]
Out[867]= {"Condition", "StringJoin", "UndirectedEdge", "Alternatives", "Undefined"}
```

Предыдущий фрагмент представляет исходный текст процедуры и ряд примеров по ее применению для выявления *типов* различных объектов. Между тем, тут же следует отметить, что процедура **WhatType** не поддерживает исчерпывающего тестирования типов, однако на ее основе несложно расширять класс проверяемых типов данных.

Функция **ReplaceAll (/.)** пакета *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  $y \equiv \{y_1, y_2, \dots, y_n\}$  и  $z \equiv \{z_1, z_2, \dots, z_n\}$ , определяя множество подстановок  $\{y_1 \rightarrow z_1, y_2 \rightarrow z_2, \dots, y_n \rightarrow z_n\}$ ,

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 (/.)** пакета *Mathematica*.

```
In[868]:= 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[869]:= (c + x^2)/x^2 /. X^2 -> a
Out[869]= (a + c)/x^2
In[870]:= Subs[(c + x^2)/x^2, x^2, a]
Out[870]= (a + c)/a
In[871]:= (c + b^2)/x^2 /. X^2 -> Sqrt[z]
Out[871]= (b^2 + c)/x^2
In[872]:= Subs[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[872]= (b^2 + c)/Sqrt[z]
In[874]:= (a + x^2)/(b + a/x^2) /. X^2 -> Sqrt[a + b]
Out[874]= (a + Sqrt[a + b])/(b + a/x^2)
In[875]:= Subs[(a + x^2)/(b + a/x^2), x^2, Sqrt[a + b]]
Out[875]= (a + Sqrt[a + b])/(b + a/Sqrt[a + b])
In[876]:= (a + x^2)/(b + 1/x^2) /. X^2 -> Sqrt[a + b]
Out[876]= (a + Sqrt[a + b])/(b + 1/x^2)
In[877]:= Subs[(a + x^2)/(b + 1/x^2), x^2, Sqrt[a + b]]
Out[877]= (a + Sqrt[a + b])/(b + 1/Sqrt[a + b])
In[878]:= Replace[1/x^2 + 1/y^3, {{x^2 -> a + b}, {y^3 -> c + d}}]
Out[878]= {1/x^2 + 1/y^3, 1/x^2 + 1/y^3}
In[879]:= Subs[1/x^2 + 1/y^3, {x^2, y^3}, {a + b, c + d}]
Out[879]= 1/(a + b) + 1/(c + d)
In[880]:= Replace[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {{x^2 -> a + b}, {y^3 -> c + d}}]
Out[880]= {Sqrt[Cos[1/y^3] + Sin[1/x^2]], Sqrt[Cos[1/y^3] + Sin[1/x^2]]}
In[881]:= Subs[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {x^2, y^3}, {a + b, c + d}]
Out[881]= Sqrt[Cos[1/(c + d)] + Sin[1/(a + b)]]
In[882]:= With[{x = a + c, y = b}, Module[{}, x^2 + y]
Out[882]= b + (a + c)^2
In[883]:= 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[883]= With[{x^2 = a + c, y = b}, Module[{}, x^2 + y]]
In[884]:= Subs[Module[{}, x^2 + y], {x, y}, {a + c, b}]
Out[884]= b + (a + c)^2
In[885]:= Subs[Module[{}, x^2 + y], {x^2, y}, {a + c, b}]
Out[885]= a + b + c
In[886]:= Replace[(a + x^2/y^3)/(b + y^3/x^2), {{y^3 -> m}, {x^2 -> n}}]
Out[886]= {(a + x^2/y^3)/(b + y^3/x^2), (a + x^2/y^3)/(b + y^3/x^2)}
In[887]:= Subs[(a + x^2/y^3)/(b + y^3/x^2), {y^3, x^2}, {m, n}]
Out[887]= n (a m + n)/(m (m + b n))
In[888]:= Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]
In[889]:= Df[(a + x^2)/(b + a/x^2), x^2]
Out[889]= (a^2 + 2 a x^2 + b x^4)/(a + b x^2)^2
In[890]:= Df[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]
Out[890]= (-4 x - 2 Sqrt[y] + y)/((2 + Sqrt[y])^3 y^(3/2))
In[881]:= D[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]
General::ivar: Sqrt[y] is not a valid variable. >>
Out[881]=  $\partial_y ((x + \sqrt{y})/(y + 2 \sqrt{y})^2)$ 
In[882]:= Df[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[882]= ((d + 2 Sqrt[x])/Sqrt[a + Sqrt[x]] - 8 (Sqrt[a + Sqrt[x]] + x))/(2 (d + 2 Sqrt[x])^3)
In[883]:= Df[(x + Sqrt[x + b])/(d + 2*Sqrt[x + b])^2, Sqrt[x + b]]
Out[883]= (d - 2 (2 x + Sqrt[b + x]))/(d + 2 Sqrt[b + x])^3
In[884]:= 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[885]:= {ReplaceAll[c/x^2 + x^2, x^2 -> t], ReplaceAll[(1 + c/x^2)/(b + x^2), x^2 -> t]}
Out[885]= {t + c/x^2, (1 + c/x^2)/(b + t)}
In[886]:= {ReplaceAll1[c/x^2 + x^2, x^2, a+b], ReplaceAll1[(1 + c/x^2)/(b + x^2), x^2, c+d]}
Out[886]= {a + b + c/(a + b), (1 + c/(c + d))/(b + c + d)}
In[887]:= 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[888]:= Df2[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
Simplify[ReplaceAll1[D[ReplaceAll1[x, y, a], a], a, y]]]]
In[889]:= Df1[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[889]= ((d + 2 Sqrt[x])/Sqrt[a + Sqrt[x]] - 8 (Sqrt[a + Sqrt[x]] + x))/(2 (d + 2 Sqrt[x])^3)

```

```

In[890]:= Df2[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[890]= ((d + 2 Sqrt[x])/Sqrt[a + Sqrt[x]] - 8 (Sqrt[a + Sqrt[x]] + x))/(2 (d + 2 Sqrt[x])^3)
In[891]:= Df2[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[891]= -(((a - c) x^2)/(1 + c)^2)
In[892]:= Df1[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[892]= -(((a - c) x^2)/(1 + c)^2)
In[893]:= Df[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[893]= -((2 (a - c) x^6)/(1 + c x^4)^2)
In[894]:= Df2[(a + b)/(Sin[x] + Cos[x]), Sin[x] + Cos[x]]
Out[894]= -((a + b)/(Cos[x] + Sin[x])^2)
In[895]:= Df2[Cos[x]/(Sin[x] + Cos[x]), Cos[x]]
Out[895]= 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** пакета *Mathematica*.

Вполне естественным представляется получение подобного расширения также и для стандартной функции **Integrate** пакета, которая имеет существенные ограничения по использованию выражений в качестве переменной интегрирования. С этой целью мы предложили 2 варианта такого расширения в виде процедур **Int** и **Int1**, чьи исходные коды с примерами их применения представляет следующий фрагмент, а именно:

```

In[5]:= Int[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y],
                                     Simplify[Subs[Integrate[Subs[x, y, a], a], a, y]]]
In[6]:= Int1[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y],
                                     Simplify[ReplaceAll1[Integrate[ReplaceAll1[x, y, a], a], a, y]]]
In[7]:= {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[7]= {-Cos[a + 1/x^2] + c Log[1/x^2], c/x^4 - Cos[a + 1/x^2]}
In[8]:= {Int[Sin[n/x^2] + m/x^2, x^2], Int1[Sin[n/x^2] + m/x^2, x^2]}
Out[8]= {-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[9]:= {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[9]= {(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[10]:= 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[10]= Integrate[(b/x^2 + a*x^2)/(d/x^2 + c*x^2), x^2]
In[11]:= 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[11]= Integrate[c/x^2 + Sin[a + 1/x^2], 1/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 пакета *Mathematica*.

Вызов простой функции *GV[]* возвращает список *специальных* символов и глобальных переменных пакета текущего релиза; в частности, в используемом релизе 8 их общее количество составляет 178. Следующий простой фрагмент представляет *исходный* код функции *GV[]* наряду с некоторыми примерами ее использования, а именно:

```
In[1783]:= GV[] := Select[Names["*"], StringTake[#, 1] == "$" &]
In[1784]:= GV[]
Out[1784]= {"$Aborted", "$ActivationGroupID", "$ActivationKey", ..., "$VersionNumber"}
In[1785]:= Length[%]
Out[1785]= 178
In[1786]:= Length[Select[GV[], Attributes[#] != {} &]]
Out[1786]= 90
In[1787]:= Length[Select[GV[], Attributes[#] == {Protected} &]]
Out[1787]= 34
In[1788]:= DeleteDuplicates[Select[Map[Attributes, Select[GV[], Attributes[#] != {} &]],
                                     # != {Protected} &]]
Out[1788]= {{Locked, Protected}, {Protected, ReadProtected}, {HoldAll, Protected}}
```

Так, последние 3 примера фрагмента констатируют, из 178 глобальных переменных и специальных символов пакета 90 наделены специальными атрибутами, например, 34 наделены только *Protected*-атрибутом. Остальные приходятся на три типа атрибутов {Locked, ReadProtected, HoldAll}, представленных последним примером фрагмента.

Следующая процедура *SEQ* служит аналогом одноименной процедуры пакета *Maple*.

```
In[2134]:= 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[2135]:= SEQ[F[k], k, 15 ;; 23]
Out[2135]= {F[15], F[16], F[17], F[18], F[19], F[20], F[21], F[22], F[23]}
In[2136]:= SEQ[F[t], t, 1 ;; 75 ;; 6]
Out[2136]= {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[2137]:= SEQ[F[t], t, 100 ;; 90]
Out[2137]= {F[100], F[99], F[98], F[97], F[96], F[95], F[94], F[93], F[92], F[91], F[90]}
In[2138]:= SEQ[F[t], t, 42.69 ;; 100 ;; 6.47]
Out[2138]= {F[42.69], F[49.16], F[55.63], F[62.1], F[68.57], F[75.04], F[81.51], F[87.98], F[94.45]}
In[2139]:= {SEQ[F[k], k, 42 ;; 69 ;; 0], SEQ[F[k], k, 42 ;; 69 ;; 6]}
Out[2139]= { SEQ[F[k], k, 42 ;; 69 ;; 0], {F[42], F[48], F[54], F[60], F[66]}}
In[2140]:= SEQ[F[k], k, 42 ;; 69 ;; -6]
Out[2140]= SEQ[F[k], k, 42 ;; 69 ;; -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. Предыдущий фрагмент содержит исходный код процедуры и примеры ее использования, из которых весьма наглядно следует сам принцип формирования списка значений в зависимости от формата 3-го аргумента. При нулевом или отрицательном значении 3-го аргумента вызов процедуры **SEQ**[*x*,*y*,*z*] возвращается невычисленным, как иллюстрируют последние примеры фрагмента.

Следующая достаточно простая процедура **FreeQ1** довольно существенно расширяет стандартную функцию **FreeQ**, обеспечивая расширенное тестирование вхождений в выражение подвыражений. **FreeQ1**[*x*, *y*] возвращает значение **True**, если выражение *x* не содержит подвыражений *y*, иначе вызов возвращает **False**. Фрагмент представляет исходный код процедуры **FreeQ1** с примерами ее применения по сравнению с **FreeQ**.

```
In[1202]:= FreeQ1[x_, y_] := Block[{h}, Quiet[FreeQ[Subs[x, y, h = Unique["ArtKr"]], h]]]
In[1203]:= {FreeQ1[1/(x^2 + b), x^2], FreeQ[1/x^2, x^2]}
Out[1203]= {False, True}
In[1204]:= {FreeQ1[1/x^3 + b, x^3], FreeQ[1/x^3 + b, x^3]}
Out[1204]= {False, True}
In[1205]:= {FreeQ1[a/Sqrt[x], Sqrt[x]], FreeQ[a/Sqrt[x], Sqrt[x]]}
Out[1205]= {False, True}
In[1206]:= {FreeQ1[{42, 64, Sqrt[x], 15, 23}, Sqrt[x]], FreeQ[{42, 64, Sqrt[x], 15, 23}, Sqrt[x]]}
Out[1206]= {False, False}
```

В целом ряде приложений процедуры раздела оказываются достаточно полезными.

## 7.5. Некоторые полезные процедуры в среде пакета Mathematica

В контексте использования стандартных функций *Nest* и *Map* для определения *новых* чистых функций на основе уже имеющихся в качестве весьма полезных расширений был предложен ряд процедур, из которых выше были рассмотрены процедуры **Mapp** и **Map4 ÷ Map6**. Здесь мы представим еще три несложные процедуры данного типа. В следующем фрагменте представлены исходные коды данных процедур с примерами их применения, которые на формальном уровне иллюстрируют результаты вызовов.

```
In[1025]:= Map1[F_;/ ListQ[F], args_;/ ListQ[args]] := Module[{a, b, c, k = 1},
    {a, b, c} = {Map[ToString, F], {}, ToString[args]};
    For[k, k <= Length[a], k++, b = Append[b,
        ToExpression[StringReplace[a[[k]] <> c, {"{" -> "[", "}" -> "]" }]]]; b]
In[1026]:= Map1[{F, G, H, V}, {x, y, z, h, p, t}]
Out[1026]= {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[1027]:= Map2[F_, c_;/ ListQ[c], d_;/ ListQ[d]] := Module[{a, b, t, k = 1},
    {a, b, t} = {Map[ToString, c], {}, ToString[d]};
    For[k, k <= Length[a], k++, b = Append[b, ToExpression[StringReplace[
        ToString[F] <> "[" <> a[[k]] <> "," <> t <> "]", {"{" -> "", "}" -> ""}]]]; b]
In[1028]:= Map2[F, {a, b, c, d, e, g}, {x, y, z, p, q, h}]
Out[1028]= {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[1029]:= Map3[F_, g_, L_;/ ListQ[L]] := Module[{a = {}, k = 1},
    For[k, k <= Length[L], k++, a = Append[a, F @@ {g, L[[k]]}]]; a]
In[1030]:= Map3[F, H, {x, y, z, h, p, h}]
Out[1030]= {F[H, x], F[H, y], F[H, z], F[H, h], F[H, p], F[H, h]}
```

Не усложняя справочной информации по процедурам, 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, ...], G[x, y, z, ...], H[x, y, z, ...], ...}, {F[a, x, y, z, ...], F[b, x, y, z, ...], F[c, x, y, z, ...], ...},
{F[H, x], F[H, y], F[H, z], F[H, h], ...}.
```

Из представленной информации достаточно прозрачен вид возвращаемых данными процедурами результатов их вызова, не требуя дополнительных разъяснений.

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 пакета *Mathematica* в виде трех процедур **MapInSitu**, **MapInSitu1** и **MapInSitu2**.

```

In[1438]:= MapInSitu1[x_, y_] := Module[{}, ToExpression[StringTake[ToString[Args[
      MapInSitu][[2]], {1, -2}] <> "=" <> ToString[Map[x, y]]]]
In[1439]:= y = {{75.42, 23.56}, {78.45, 78.89}}; MapInSitu1[Sin, y]
Out[1439]= {{0.0217746, -0.9999998}, {0.0896956, -0.34307}}
In[1440]:= y
Out[1440]= {{0.0217746, -0.9999998}, {0.0896956, -0.34307}}
In[1447]:= MapInSitu[x_, y_ /; StringQ[y]] := Module[{}, ToExpression[y <> "=" <>
      ToString[Map[x, ToExpression[y]]]]
In[1454]:= y = {a, b, c}; h = {{4.2, 6.6}, {4.7, 7.5}}; {MapInSitu[G, "y"], MapInSitu[Sin, "h"]}
Out[1454]= {{G[a], G[b], G[c]}, {{-0.871576, 0.311541}, {-0.999923, 0.938}}}
In[1455]:= {y, h}
Out[1455]= {{G[a], G[b], G[c]}, {{-0.871576, 0.311541}, {-0.999923, 0.938}}}
In[722]:= {H, G} = {{7.5, 47.64, 15.23}, {7.5, 47.64, 15.23}}
Out[722]= {{7.5, 47.64, 15.23}, {7.5, 47.64, 15.23}}
In[719]:= 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[720]:= MapInSitu2[Sin, {7.5, 47.64, 15.23}]
Out[720]= {0.938, -0.493501, 0.459972}
In[721]:= {H, G}
Out[721]= {{0.938, -0.493501, 0.468828}, {0.938, -0.493501, 0.468828}}

```

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 процедурой *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 процедурой *MapInSitu1[x, y]* в предположении, что для нее аргумент *y*, определяющий выражение, обрабатываемое функцией *Map*, остается постоянным.

Процедура *MapInSitu* для обоих пакетов характеризуется предпосылкой, что второй аргумент при ее вызове указывает на идентификатор в строчном формате, которому ранее было присвоено некоторое значение, обновляемое *на месте* после его обработки *{map | Map}* функцией. 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 процедуры **MapInSitu1** в среде пакета *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.

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[33]:= Bits[x_, P_;/; IntegerQ[P]] := 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[34]:= {Bits["A", 0], Bits["A", 2], Bits[{1, 0, 0, 0, 0, 0, 1}, 0], Bits["A", 9], Bits[{1, 1, 1, 1, 0, 1}, 0]}
Out[34]= {{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* [99].

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[926]:= 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[927]:= {BitSet1[450, {{3, 1}, {6, 0}, {9, 1}}], BitSet1[75, {4, 0}], BitSet1[75, {7, 1}]}
Out[927]= {451, 67, 75}
In[928]:= BitSet1[450, {{3, 1}, {6, 0}, {9, 2}}]
Out[928]= BitSet1[450, {{3, 1}, {6, 0}, {9, 2}}]
In[9]:= BitSet1[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, 75, {1, 5, 7}], h, BitGet1[75, {1, 5, 7}], BitGet1[p, 450, {1, 3, 5, 7, 9}], p}
Out[10]= {{1, 0, 1}, 7, {1, 0, 1}, {1, 1, 0, 0, 0}, 9}

```

Примеры применения процедуры **BitSet1** весьма наглядно иллюстрируют сказанное. Следует отметить, что функционально процедура **BitSet1** перекрывает стандартные функции как **BitSet**, так и **BitClear** пакета *Mathematica*. Тогда как вызов **BitGet1**[*x*, *n*, *p*] функционально перекрывает стандартные функции **BitGet** и **BitLength** пакета; 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.

Вызов стандартной функции **Tuples**[*L*, *n*] возвращает список всевозможных кортежей длины *n*, составленных из элементов списка *L*. Для получения сравнительной оценки реактивности пакетов *Mathematica* и *Maple* именно алгоритм, реализующий данную процедуру в среде обоих пакетов, был нами положен в основу данного сравнения. И действительно, в целом, *реактивность* того либо иного алгоритма определяется как реактивностью самого пакета, 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[1021]:= Tuples1[x_ /; ListQ[x], n_ /; IntegerQ[n]] := Module[{c = "", d = "", f = Res, h = {},
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[1022]:= Tuples1[{1, 2, 3, 4}, 3]
Out[1022]= {"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[1023]:= t = TimeUsed[]; Tuples1[{0, 1}, 15]; TimeUsed[] - t
Out[1023]= 124.047

In[1024]:= t = TimeUsed[]; Tuples[{0, 1}, 15]; TimeUsed[] - t
Out[1024]= 0.625

In[1025]:= t = TimeUsed[]; Tuples[{"0", "1", "2"}, 15]; TimeUsed[] - t
Out[1025]= 8.047

In[1026]:= t = TimeUsed[]; Tuples[{0, 1, 2, 3}, 15]; TimeUsed[] - t
General::nomem: The current computation was aborted because there was
insufficient memory available to complete the computation.
No more memory available.
Mathematica kernel has shut down.
Try quitting other applications and then retry.

> t := time(): tuples({1, 2}, 15): time() - t;  => 65.356                                     #Maple 11
    
```

Полученные временные оценки показывают, что на данном алгоритме *Maple* имеет довольно существенные преимущества, растущие как с ростом длины  $n$  кортежей, так и с ростом мощности алфавита  $A$ . В целом было замечено, что с ростом вложенности циклических конструкций временные издержки пакета *Mathematica* растут заметно быстрее, чем на тех же конструкциях в среде пакета *Maple*. С этой целью эксперимент с простыми процедурами **NestCycles**, запрограммированными в среде обоих пакетов и чей алгоритм состоит в реализации простейшей *циклической* конструкции заданной глубины вложенности  $n$ , со всей очевидностью подтверждает вышесказанное.

```

In[1599]:= NestCycles[n_ /; IntegerQ[n]] := Module[{a = "", b = "", t, j = 1},
            For[j = 1, j <= n, j++, a = a <> "For[t" <> ToString[j] <> "=1, t" <> ToString[j] <>
            " <= " <> ToString[n] <> ", t" <> ToString[j] <> "++,"; b = b <> "];";
            ToExpression[a <> "75" <> b]]

In[1600]:= t = TimeUsed[]; NestCycles[9]; TimeUsed[] - t
Out[1600]= 933.254

> t := time(): NestCycles(9): time() - t;  => 96.128
    
```

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 последние примеры предпоследнего фрагмента. Таким образом, оба пакета весьма чувствительны к размеру доступной памяти в их рабочей области.

В качестве аналога полезной *Maple*-процедуры *AllMatrices*(*m*, *n*, *A*, *G*), вызов которой обеспечивает возврат списка *всевозможных* (*m**x**n*)-матриц, чьи элементы принадлежат конечному алфавиту *A*, можно привести *Mathematica*-процедуру *AllMatrices*[*m*, *n*, *A*], с исходным текстом которой и примером применения можно ознакомиться ниже.

```
In[1281]:= AllMatrices[m_;/ IntegerQ[m] && m > 0, n_;/ IntegerQ[n] && n > 0,
          A_;/ ListQ[A]] := Module[{a = Tuples[A, m*n], b = {}, h, c, k = 1, p},
  For[k, k <= Length[a], k++, c = a[[k]]; h = {}; For[p = 1, p <= n, p++,
    h = Append[h, Take[c, {(p - 1)*m + 1, p*m}]]]; b = Append[b, h]];
  Map[MatrixForm, b]]
```

```
In[1282]:= AllMatrices[3, 2, {1, 2}]
```

Out[1282]=

[illegible]

```
In[1346]:= AllMatrices1[m_;/ IntegerQ[m] && m > 1, n_;/ IntegerQ[n] && n > 1,  
A_;/ ListQ[A]] := Map[MatrixForm, Tuples[Tuples[A, m], n]]
```

```
In[1347]:= AllMatrices1[3, 2, {1, 2}]
```

Out[1347]=

[illegible]

Алгоритм данной процедуры весьма существенно использует функцию **Tuples**[*A*, *n*] для генерации всех кортежей длины *n* из элементов алфавита *A*. Между тем, ввиду ее специфики возникает аварийная ситуация для достаточно больших значений *n* и/или **Length**[*A*] из-за недостатка оперативной памяти. Аналогичная ситуация возникает и с одноименной *tuples*-процедурой в *Maple*, как достаточно наглядно иллюстрируют примеры из [99]. Однако в случае *Maple* компьютер, как правило, «зависает», говоря о более развитой диагностике подобных ситуаций в пакете *Mathematica* по сравнению с пакетом *Maple*. Завершает предыдущий фрагмент простая функция, вызов которой **AllMatrices1**[*m*, *n*, *A*] возвращает список всевозможных (*m*×*n*)-матриц (*n* – число строк и *m* – число столбцов), элементы которых принадлежат конечному алфавиту *A*. Функция **AllMatrices1**, являясь чистой функцией, существенно проще процедуры **AllMatrices**.

Для исследования обратной динамики одномерных классических структур [85,91-94] в ряде случаев полезной оказывается процедура **Predecessors**[*Ltf*, *Co*, *n*], возвращающая на основе списка *Ltf*, определяющего локальную функцию перехода структуры, размера ее шаблона соседства *n* и начальной конфигурации *Co* конечного блока из состояний элементарных автоматов структуры множество конфигураций-предшественников. В следующем фрагменте представлен исходный код процедуры **Predecessors** наряду с примерами ее применения для определения предшественников 1-мерных структур.

```
In[2538]:= Predecessors[Ltf_/, ListQ[Ltf], Co_/, StringQ[Co], n_/, IntegerQ[n]] :=
Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]},
a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
For[k=1, k <= Length[a], k++, L[StringTake[First[a[[k]]], -1]] =
Map2[StringDrop, Map[ToString1, a[[k]]], {-1}]; b = L[StringTake[Co, 1]];
For[k = 2, k <= d, k++, c = L[StringTake[Co, {k, k}]];
For[i = 1, i <= Length[b], i++, For[j = 1, j <= Length[c], j++,
If[StringEnd[b[[i]], StringTake[c[[j]], n - 1], h = Append[h, b[[i]] <>
StringTake[c[[j]], -1], Null]]];
b = h; h = {}; If[Length[b] != (n - 1)^Length[a],
Print["Structure possesses the nonconstructability of NCF-type"], Null]; b]

In[2539]:= Ltf = {"0000", "0011", "0101", "0110", "1000", "1010", "1100", "1111"}
Out[2539]= {"0000", "0011", "0101", "0110", "1000", "1010", "1100", "1111"}
In[2540]:= Ltf1 = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[2540]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[2541]:= Predecessors[Ltf, "11001110", 3]
Structure possesses the nonconstructability of NCF-type
Out[2541]= {"0010111110"}
In[2542]:= Predecessors[Ltf, "11001110110101", 3]
Structure possesses the nonconstructability of NCF-type
Out[2542]= {}
In[2543]:= Predecessors[Ltf1, "1100111001101", 3]
Out[2543]= {"001011111010001", "010000100001010", "100110010111100", "111101001100111"}
```

В частности, возврат процедурой **Predecessors** пустого списка говорит о наличии для соответствующей структуры неконструируемости *NCF*-типа и позволяет определять по меньшей мере один из видов неконструируемых конфигураций [85,91-94]. Более того, процедура **Predecessors** в ряде случаев позволяет определять существование для исследуемой структуры неконструируемости *NCF*-типа на основе числа элементов у списка предшественников *блочной* конфигурации *Co*. Фрагмент представляет аналог *Maple*-процедуры **Predecessors** в среде пакета *Mathematica*. Алгоритм такого аналога не базируется на табличных структурах, а параллельные подстановки  $x_1 x_2 \dots x_n \rightarrow x'_1$ , определяющие локальную функцию перехода структуры, определяются не *таблицей*, а списком *Ltf* строк вида " $x_1 x_2 \dots x_n x'_1$ ". Аргументы *Co* и *n* идентичны одноименным аргументам процедуры-прообразу из *Maple*. Тут же вполне уместно подчеркнуть, что в среде *Maple* несколько более удачна организация табличных структур, позволяя во многих случаях эффективно определять табличные объекты различного назначения и оперировать с ними в составе более простых и прозрачных алгоритмов. Известно, наряду с теоретическим исследованием классических однородных структур (*Cellular Automata*) достаточно широко [85] используется их экспериментальное исследование с использованием средств компьютерного моделирования. С данной целью был создан большой набор программных средств различного назначения и сложности. Целый ряд средств данного типа представлен в [85,91-94]. В частности, в рамках общей задачи по исследованию проблемы неконструируемости возникает вопрос генерации для любой блочной конфигурации *Co* предшественников, т.е. блочных конфигураций, которые под действием *глобальной функции* перехода структуры в следующий момент времени переводятся в конфигурацию *Co*. Оценка числа данных предшественников позволяет на основе критерия *Аладьева-Маруока-Кимура* в целом ряде случаев решать вопрос наличия для структуры неконструируемости *NCF*-типа [85,91-94]. С иллюстративной целью здесь представлен ряд *Mathematica*-процедур, решающих данную задачу для случая *1*-мерных классических структур с произвольными алфавитом *W* внутренних состояний и индексом соседства *X* классической однородной структуры. Процедура **Predecessors**, представленная выше – одна из их числа, тогда как 2 следующие можно рассматривать в качестве достаточно полезных модификаций для среды *Mathematica*. Следующий фрагмент представляет исходные коды обеих процедур с примерами.

```
In[2558]:= PredecessorsR[Ltf_;/; ListQ[Ltf], Co_;/; StringQ[Co], n_;/; IntegerQ[n],
    Cf_;/; SymbolQ[Cf]] := Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]},
    a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
    For[k = 1, k <= Length[a], k++, L[StringTake[First[a[[k]]], -1]] =
    Map2[StringDrop, Map[ToString1, a[[k]]], {-1}]];
    b = L[StringTake[Co, 1]];
    For[k = 2, k <= d, k++, c = L[StringTake[Co, {k, k}]];
    For[i = 1, i <= Length[b], i++, For[j = 1, j <= Length[c], j++,
    If[StringEnd[b[[i]], StringTake[c[[j]], n - 1]],
    h = Append[h, b[[i]] <> StringTake[c[[j]], -1], Null]];
    b = h; h = {}; Cf = b; Length[b]]
```

```

In[2559]:= PredecessorsL[Ltf_/, ListQ[Ltf], Co_/, StringQ[Co], n_/, IntegerQ[n],
           Cf_/, SymbolQ[Cf]] := Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]},
           a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
           For[k = 1, k <= Length[a], k++, L[StringTake[First[a][[k]], -1]] =
           Map2[StringDrop, Map[ToString1, a[[k]], {-1}]];
           b = L[StringTake[Co, -1]];
           For[k = d - 1, k >= 1, k--, c = L[StringTake[Co, {k, k}]];
           For[i = 1, i <= Length[b], i++, For[j = 1, j <= Length[c], j++,
           If[StringTake[b[[i]], n - 1] === StringTake[c[[j]], {2, -1}],
           h = Append[h, StringTake[c[[j]], 1] <> b[[i]], Null]];
           b = h; h = {}; Cf = b; Length[b]]

In[2560]:= Ltf = {"0000", "0011", "0101", "0110", "1000", "1010", "1100", "1111"}
Out[2560]= {"0000", "0011", "0101", "0110", "1000", "1010", "1100", "1111"}
In[2561]:= Ltf1 = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[2561]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[2562]:= Clear[Cf]; {PredecessorsR[Ltf, "010100", 3, Cf], Cf}
Out[2562]= {0, {}}
In[2563]:= Clear[Cf]; {PredecessorsL[Ltf, "010100", 3, Cf], Cf}
Out[2563]= {0, {}}
In[2564]:= Clear[Cf]; {PredecessorsL[Ltf1, "010100", 3, Cf], Cf}
Out[2564]= {4, {"01110000", "10101011", "00011101", "11000110"}}
In[2565]:= Clear[Cf]; {PredecessorsR[Ltf1, "010100", 3, Cf], Cf}
Out[2565]= {4, {"00011101", "01110000", "10101011", "11000110"}}
In[2566]:= Clear[Cf]; {PredecessorsL[Ltf, "0101", 3, Cf], Cf}
Out[2566]= {1, {"111001"}}
In[2567]:= Clear[Cf]; {PredecessorsR[Ltf, "0101", 3, Cf], Cf}
Out[2567]= {1, {"111001"}}

```

Процедуры **PredecessorsR[Ltf, Co, n, Cf]** и **PredecessorsL[Ltf, Co, n, Cf]** идентичны и их различие сводится только к тому, что процесс генерации предшественников для **Co**-конфигурации в структуре с шаблоном соседства размера **n** и локальной функцией перехода, определяемой списком строк вида " $x_1x_2\dots x_nx_1^`$ " (эти строки отвечают набору параллельных подстановок  $x_1x_2 \dots x_n \rightarrow x_1^` \{ x_n, x_1^` \notin A \}$ , определяющих локальную функцию перехода структуры), производится слева направо (**R**) или справа налево (**L**); результат вызова процедур на идентичных фактических аргументах **[Ltf, Co, n, Cf]**, естественно, аналогичен. Вызов каждой из этих процедур в качестве результата возвращает число предшественников для блочной конфигурации **Co**, тогда как через 4-й аргумент **Cf** в качестве дополнительного возвращается список всех предшественников конфигурации **Co**. Именно на основе данной информации можно судить о том, будет ли блочная **Co**-конфигурация являться  $\gamma$ -конфигурацией [85,91-94], и что в случае положительного ответа на основе критерия **Аладьева-Маруока-Кимура** [85] можно говорить о наличии для классической 1-мерной однородной структуры неконструируемости **NCF**-типа.

Для экспериментального исследования предшественников блочных конфигураций в 1-мерных классических структурах дополнительно к представленным определенным интерес представляет и процедура **TestOnNCF**[*Ltf*, *n*, *m*], существенно базирующаяся на одной из предыдущих процедур, а именно **PredecessorsR**. **TestOnNCF**[*Ltf*, *n*, *m*] для 1-мерной классической структуры с шаблоном соседства размера *n* и локальной *Ltf*-функцией перехода возвращает *первую* из блочных конфигураций размера *m*, которая является  $\gamma$ -конфигурацией, и печатает об этом соответствующее сообщение. В случае отсутствия  $\gamma$ -конфигураций размера *m* вызов процедуры **TestOnNCF** возвращает *Null* и печатает об этом соответствующее сообщение. Следующий фрагмент представляет исходный код процедуры и наиболее типичные примеры ее использования.

```
In[1558]:= TestOnNCF[Ltf_;/; ListQ[Ltf], n_;/; IntegerQ[n], m_;/; IntegerQ[m]] :=
Module[{a = {}, b, c, Cf, d, k = 1},
  For[k, k <= Length[Ltf], k++, a = Append[a, StringTake[Ltf][[k]], -1]];
  a = DeleteDuplicates[a]; b = Map[StringJoin, Tuples[a, m]]; d = (n - 1)^Length[a];
  For[k = 1, k <= Length[b], k++, Clear[Cf]; c = PredecessorsR[Ltf, b[[k]], n, Cf];
  If[c != d, Print["Block configuration <" <> ToString[b[[k]]] <>
  "> is gamma-configuration"]; Return[b[[k]], Null]];
  Print["All block configurations of size " <> ToString[m] <> " possess " <>
  ToString[d] <> " different predecessors"]]
```

```
In[1559]:= Ltf = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[1559]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[1560]:= Ltf1 = {"0000", "0011", "0101", "0111", "1001", "1010", "1101", "1110"}
Out[1560]= {"0000", "0011", "0101", "0111", "1001", "1010", "1101", "1110"}
In[1561]:= TestOnNCF[Ltf1, 3, 15]
Block configuration <0000000000000000> is gamma-configuration
Out[1561]= "0000000000000000"
In[1562]:= TestOnNCF[Ltf, 3, 8]
All block configurations of size 8 possess 4 different predecessors
```

Учитывая, в целом, большую реактивность *Mathematica* относительно *Maple*, казалось бы вполне естественным использовать именно пакет *Mathematica* для симулирования динамики и исследования ряда свойств однородных структур. Однако, это не совсем так, а именно. Следующий фрагмент исследует *Maple*-аналог процедуры **TestOnNCF** на предмет ее реактивности в среде *Maple*. И *Maple*-аналог и ее прообраз **TestOnNCF** в среде *Mathematica* используют, практически, идентичные процедуры **Predecessors** и **PredecessorsR** наряду с функцией **Tuples** и процедурой *tuples*, которые обеспечивают основу генерации множества всевозможных кортежей заданной длины *m*, состоящих из символов некоторого алфавита *A*. Между тем, выясняется, что функция **Tuples** во много раз более реактивна (на несколько порядков), чем *Maple*-процедура; в частности, на кортежах длины 15 и алфавите мощностью 2 эта разница составляет порядка 3050 раз в пользу **Tuples**-функции. Тогда как в целом *Maple*-процедура **TestOnNCF** более, чем в 3 раза оказывается реактивней *Mathematica*-процедуры **TestOnNCF**. Итак, для задач компьютерного исследования однородных структур вопрос целесообразности

использования того или иного пакета не столь однозначен и требует более детальной проработки реализуемого пакетами вычислительного алгоритма. Так, при указанной весьма значительной разнице в реактивности функции **Tuples** и процедуры *tuples* их использования в одних и тех же алгоритмах обработки могут достаточно значительно уменьшать эту разницу, порой, и вовсе меняя местами средства по этому показателю.

```
In[2567]:= Ltf = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[2567]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[2568]:= t = TimeUsed[]; TestOnNCF[Ltf, 3, 15]; TimeUsed[] - t
All block configurations of size 15 possess 4 different predecessors
Out[2568]= 626.36
In[2569]:= t = TimeUsed[]; Tuples[{"0", "1"}, 15]; TimeUsed[] - t
Out[2569]= 0.031
> t := time(): tuples({"0", "1"}, 15): time() - t;    => 94.671
```

Процедура **ActiveProcess** служит для оперативного получения активных процессов в текущем сеансе компьютера; так ее вызов **ActiveProcess[]** возвращает список *активных* в текущем сеансе процессов. Поскольку процедура использует внешнюю программу 'tlist.exe', то ее отсутствие инициирует завершение процедуры с выводом сообщения. Следующий фрагмент иллюстрирует исходный код процедуры с ее применением.

```
In[1141]:= ActiveProcess[] := Module[{a, b, c, f, s, h}, a = SearchFile["tlist.exe"];
If[a == {}, Return["A possibility of receiving of active processes is absent"],
b = a[[1]]; f = $HomeDirectory <> "\\Art23Kr15.txt";
c = Run[b <> " > ", StrStr[f]]; If[c != 0, Return[$Failed], a = OpenRead[f]];
h[x_ /; StringQ[x]] := Module[{a, b, c},
{a, b} = {Flatten[StringPosition[ToLowerCase[x], ".com"]],
Flatten[StringPosition[ToLowerCase[x], ".exe"]]}];
If[a == {}, c = Flatten[b][[2]], c = Flatten[a][[2]]; c];
c = {}; Label[s]; b = Read[a, String];
If[b === EndOfFile, Close[a]; DeleteFile[f];
Return[Sort[DeleteDuplicates[Flatten[c]]], b = StringTrim[b];
If[StringEnd[b, "FreeCommander"], c = Append[c, "FreeCommander.exe"]; Goto[s],
If[StringPosition[ToLowerCase[b], ".exe"] != {} ||
StringPosition[ToLowerCase[b], ".com"] != {},
c = Append[c, StringTake[b, {Flatten[StringPosition[b, " "][[1]] + 1, h[b]}]];
Goto[s], Goto[s]]]]]
In[1142]:= ActiveProcess[]
Out[1142]= {"alg.exe", "AmplusnetPrivacyTools.exe", "ati2evxx.exe", "avgchsvx.exe",
"avgcsrvx.exe", "avgemcx.exe", "AVGIDSAgent.exe", "AVGIDSMonitor.exe",
"avgnsx.exe", "avgrsx.exe", "avgtray.exe", "avgwdsvc.exe", "cmd.exe",
"csrss.exe", "ctfmon.exe", "DrgToDsc.exe", "explorer.exe", "javaw.exe",
"jqs.exe", "FreeCommander.exe", "lsass.exe", "LvAgent.exe", "MsPMSPSv.exe",
```

```
"Mathematica.exe", "MathKernel.exe", "services.exe", "Skype.exe", "smss.exe",
"soundman.exe", "spoolsv.exe", "svchost.exe", "tlist.exe", "vprot.exe",
"vsmon.exe", "winlogon.exe", "WINWORD.EXE", "zldclient.exe"}
```

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[788]:= ActRemObj[x_ /; StringQ[x], y_ /; MemberQ[{"Act", "Rem"}, y]] :=
Module[{a = $HomeDirectory <> "\\\" <> ToString[x] <> ".$ArtKr$", b, c},
  If[HowAct[x] && y == "Rem", b = OpenWrite[a];
  WriteString[a, ToString[DefFunc[x]]]; Close[b]; Remove[x];
  "Remove", If[! HowAct[x] && y == "Rem", "Remove",
  If[FileExistsQ[a], b = OpenRead[a];
  Map[ToExpression, ReadList[b, String]]; Close[b]; DeleteFile[a];
  "Activate", Return[Defer[ActRemObj[x, y]]]]]]]

In[789]:= A[x_] := Module[{a = 450}, x + a]; A[x_, y_] := Module[{a = 75}, x + y + a]
In[790]:= {A[100], A[100, 200]}
Out[790]= {550, 375}
In[800]:= ActRemObj["A", "Rem"]
Out[800]= "Remove"
In[801]:= {A[100], A[100, 200]}
Out[801]= {A[100], A[100, 200]}
In[802]:= ActRemObj["A", "Act"]
Out[802]= "Activate"
In[803]:= {A[100], A[100, 200]}
Out[803]= {550, 375}
In[815]:= Fam := {69, 64, 44, 15, 23}; Fam
Out[815]= {69, 64, 44, 15, 23}
In[816]:= {ActRemObj["Fam", "Rem"], Fam}
Out[816]= {"Remove", Removed["Fam"]}
In[817]:= ActRemObj["Fam", "Act"]; Fam
Out[817]= {69, 64, 44, 15, 23}
In[822]:= GrSu = 2012; {ActRemObj["GrSu", "Rem"], GrSu}
Out[822]= {"Remove", Removed["GrSu"]}
In[823]:= ActRemObj["GrSu", "Act"]; GrSu
Out[823]= 2012
In[624]:= ActRemObj["GrSu", "Act"]
Out[624]= ActRemObj["GrSu", "Act"]
```

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.

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 довольно типичным примером ее применения.

```
In[1216]:= 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[1217]:= W = ParVar[GS, {69, 64, 44, 23, 15}]
Out[1217]= {"GS$283382, GS$283383, GS$283384, GS$283385, GS$283386"}
In[1218]:= ToExpression[W]
Out[1218]= {69, 64, 44, 23, 15}
```

Пакет *Mathematica* располагает средствами работы с системным буфером обмена на уровне двух функций **CopyToClipboard[x]** и **Paste[]**, обеспечивая запись выражения  $x$  в системный буфер (*clipboard*) и его последующее чтение из буфера соответственно.

```
In[2091]:= CopyToClipboard[23!]; Paste[]
Out[2091]= 25 852 016 738 884 976 640 000
In[2092]:= CopyToClipboard[23!]; Paste[]
               25 852 016 738 884 976 640 000
In[2093]:= %
Out[2093]= Notebook[{Cell[BoxData["25852016738884976640000"], "Output"]},
               FrontEndVersion -> "8.0 for Microsoft Windows (32-bit) (February 23, 2011)",
               StyleDefinitions -> "Default.nb"]
In[2094]:= Prev[_] := Block[{a = ToString[InputForm[%]], b, c}, b = StringReplace[a,
               "Notebook[{Cell[BoxData[" -> ""]; StringTake[StringReplace[StringTake[b,
               {1, StringPosition[b, "], \"Output\\\""}][[1]][[1] - 1]], "\\\" -> ""], {2, -2}]]
In[2095]:= CopyToClipboard[(a + b)/(c + d) + z]; Paste[]
               (a + b)/(c + d) + z
In[2096]:= ToExpression[Prev[%]]
Out[2096]= (a + b)/(c + d) + z
```

Между тем, считанное по **Paste[]** выражение  $x$  не возвращается в **Out**-параграфе для возможности его последующей обработки, как указано *первым* примером фрагмента, взятого из справки по пакету *Mathematica*, и при попытке получить его оператором % возвращается инициированная *вызовом* функции **Paste** соответствующая структурная

конструкция текущего документа. С целью возможности получать результаты вызова функции **Paste** в **Out**-параграфе предложена достаточно простая процедура **Prev[\_]**, чей исходный код представлен предыдущим фрагментом. Процедура в композиции с функцией **ToExpression** решает эту задачу, как достаточно наглядно иллюстрирует последний пример предыдущего фрагмента.

Вызов стандартной функции **Clear[x1, x2, x3, ..., xn]** очищает символы {x1, x2, x3, ..., xn} 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 {x1, x2, x3, ..., xn} от присвоенных им выражений с сохранением всех их атрибутов, тогда как при **h=2** очищая символы {x1, x2, x3, ..., xn} от присвоенных им выражений и *всех* атрибутов. Следующий фрагмент представляет исходный код процедуры **Clear1** с типичными примерами ее применения.

```
In[1158]:= 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] <> ", " <>
    ToString1[d] <> "]" <> "; Clear" <> If[x == 1, "", "All"] <> "[" <>
    ToString1[b] <> "]""]; If[x == 2, Null, Quiet[Check[ToExpression[
    "SetAttributes[" <> ToString1[b] <> ", " <> ToString1[d] <> "], $Failed]]]]]
In[1159]:= 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[1159]= {{Protected}, 30}
In[1160]:= ClearAll[S]; {Attributes[S], S[5, 6]}
ClearAll::wrsym: Symbol S is Protected. >>
Out[1160]= {{Protected}, 30}
In[1161]:= {x, y} = {75, 450}; Clear1[2]; Clear1[2, "75"]; {x, y}
Out[1161]= {75, 450}
In[1162]:= SetAttributes[G, Listable]; G[x_, y_] := x*y; SetAttributes[G, Protected];
Clear1[1, "G"]; {G, Attributes[G]}
Out[1162]= {G, {Listable, Protected}}
In[1163]:= SetAttributes[V, Listable]; V[x_, y_] := x*y; SetAttributes[V, Protected];
Clear1[2, "V"]; {V, Attributes[V]}
Out[1163]= {V, {}}
In[1164]:= Clear1[2, "V", "G"]; {V, G, Map[Attributes, {V, G}]}
Out[1164]= {V, G, {}, {}}
In[1165]:= {Clear1[2], Clear1[3, "x"], Clear1[], Clear1[1, 450]}
Out[1165]= {Null, Clear1[3, "x"], Clear1[], $Failed}
```

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 пакета присвоено значение  $x$ . Следующий фрагмент представляет исходный код **Nvalue1** с примерами.

```
In[1243]:= Nvalue1[x_] := Module[{a = {}, b = Select[Names["*"], StringFreeQ[#, "$"] &],
    c, k = 1}, For[k, k <= Length[b], k++, c = ToExpression["Attributes[" <>
    ToString1[b[[k]]] <> ""];
    If[! MemberQ[c, Protected], a = Append[a, b[[k]], Null]];
    a = Select[a, ToExpression[#] === x &]; a]

In[1244]:= {x, y, z, t, h, g, w, s, u} = {45, 78, 23, 15, 15, 15, 15, 15, 15}; Nvalue1[15]
Out[1244]= {"g", "h", "s", "t", "u", "w"}
In[1245]:= {Avz, Agn, Vsv, Art, Kr} = {69, 64, 44, 44, 44}; Nvalue1[44]
Out[1245]= {"Art", "Kr", "Vsv"}
In[1246]:= T = TimeUsed[]; {Avz, Agn} = {1966, 1966}; Nvalue1[1966]; TimeUsed[] - T
Out[1246]= 3.265
```

В частности, из последнего примера фрагмента следует, что процедура **Nvalue1** имеет *не вполне* удовлетворительные *временные* характеристики по причине, что ее алгоритм базируется на анализе более **4500** объектов как системных, так и пользовательских.

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[1185]:= GroupNames[L_] := Module[{a = If[ListQ[L], 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[1186]:= GroupNames[{Sin, Cos, ProcQ, Locals2, 75, Prev, StrStr, 67/42, Nvalue1, a + b}]
Out[1186]= {{System, Sin, Cos}, {Procedure, ProcQ, Locals2, Prev, Nvalue1}, {Integer, 75},
    {Function, StrStr}, {Rational, 67/42}, {Plus, a + b}}

In[1187]:= GroupNames[Head1]
Out[1187]= {Function, Head1}
In[1188]:= GroupNames[Names["*"]]
Out[1188]= {{Global System, "\[FormalA]", ..., "CallPacket"}, {Function, "AcNb", ...,
    "$ProcName"}, {String, "ActionMenu", ..., "GroebnerBasis"}, {Procedure, "ActiveProcess",
    ..., "WhichN"}, {System, "CanberraDistance", ..., "$VersionNumber"}}

In[1189]:= Map[Length, L] - 1
Out[1189]= {392, 142, 27, 250, 3717}
```

В частности, из 2-х последних примеров применения **GroupNames** следует, что *имена* текущего сеанса отнесены к пяти группам, а именно: **Global`System**, **Function**, **String**, **Procedure** и **System**, число элементов в которых 392, 142, 27, 250 и 3717 соответственно. Между тем, получение данного результата требует значительных временных затрат.

Выше рассмотрены полезные средства работы со *списочными* структурами, в качестве которых выступают **Map** и **Map1 ÷ Map7**. В качестве еще одного подобного средства определенный интерес представляет довольно несложная 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, а именно:

**{F[a, b, c, ..., v], G[a, b, c, ..., v], H[a, b, c, ..., v], ..., V[a, b, c, ..., v]}**

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Следующий фрагмент представляет исходный код функции **Map8** с примерами ее применения.

```
In[1412]:= Map8[x_, y_;/; ListQ[y]] := Map[ToExpression, Block[{a = {x}, c = {}, d, k = 1,
                                b = ToString1[y]}, While[k <= Length[a], d = a[[k]];
                                c = Append[c, ToString[d] <> "[" <> StringTake[b, {2, -2}] <> ""]; k++]; c]]

In[1413]:= Map8[x, y, z, h, {a, b, c, d}]
Out[1413]= {x[a, b, c, d], y[a, b, c, d], z[a, b, c, d], h[a, b, c, d]}
```

Функция **Map8** оказывается довольно полезной, например, в организации сравнений результатов вызовов функционально подобных процедур/функций на идентичных кортежах фактических аргументов. Тогда как процедура **Map9**, чей вызов **Map9[F, {a, b, ..., v}, {a1, b1, ..., v1}]**, где *F* – символ, тогда как *{a, b, c, ..., v}, {a1, b1, c1, ..., v1}* – списки произвольных выражений, возвращает результат следующего формата, а именно:

**{F[a, a1], F[b, b1], F[c, c1], ..., F[v, v1]}**

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Следующий фрагмент представляет исходный код процедуры **Map9** с примерами ее применения.

```
In[1477]:= 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], c = Append[c, f <> "[" <> a[[k]] <> "," <> b[[k]] <> ""]; k++];
                                ToExpression[c]]]

In[1478]:= Map9[F, {a, b, c}, {x, y, z}]
Out[1478]= {F[a, x], F[b, y], F[c, z]}
In[1479]:= Map9[Rule, {"70a", "65b", "45c"}, {"a", "b", "c"}]
Out[1479]= {"70a" -> "a", "65b" -> "b", "45c" -> "c"}
In[1480]:= Map9[Rule, {a, b, c}, {x, y, z}]
Out[1480]= {a -> x, b -> y, c -> z}
```

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, а *c1, c2, ..., cn* – необязательные аргументы, возвращает результат следующего формата:

$\{F[x, a, c1, c2, \dots], F[x, b, c1, c2, \dots], F[x, c, c1, c2, \dots], \dots, F[x, v, c1, c2, \dots]\}$

не требуя каких-либо дополнительных пояснений ввиду прозрачности. Следующий фрагмент представляет *исходный* код процедуры **Map10** с примерами ее применения.

```
In[1326]:= Map10[F_/, ProcQ[F] || SysFuncQ[F] || SymbolQ[F], x_, L_/, ListQ[L], y___] :=
Module[{a = ToString[F] <> "[" <> ToString1[x] <> ",",
b = If[{y} == {}, "", ", " <> StringTake[ToString1[{y}], {2, -2}]] <> "]",
c = {}, d = Map[ToString1, L], h, k = 1},
h = Map3[StringJoin, a, d]; ToExpression[Mapp[StringJoin, h, b]]]
In[1327]:= Map10[F, x, {a, "b", c}, y, "z", h]
Out[1327]= {F[x, a, y, "z", h], F[x, "b", y, "z", h], F[x, c, y, "z", h]}
In[1328]:= Map10[F, "x", {a, "b", c}]
Out[1328]= {F["x", a], F["x", "b"], F["x", c]}
In[1329]:= Map10[SuffPref, "C:\89b8fcf17cbdce3\i386\mxdwdrv.dll", {"nb", ".m"}, 2]
Out[1329]= {False, False}
```

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 по всей файловой системе компьютера при вызове процедуры с 1 аргументом. Следующий фрагмент представляет *исходный код* процедуры **FindFileObject** с некоторыми типичными примерами ее применения.

```
In[1394]:= 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 > ArtKr"];
While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]];
If[StringTake[c, {-2, -1}] == ".m", b = Append[b, c]];
Continue[]]; Quiet[Close[f]]; c = ""; k++; k = 1;
While[k <= Length[b], If[Select[ReadList[b[[k]], String],
! StringFreeQ[#, h] && StringFreeQ[#, p] &] != {},
s = Append[s, b[[k]]]; k++]; {DeleteFile[f], s}][[2]]]
In[1395]:= FindFileObject[ProcQ, "D"]
Out[1395]= {"d:\avz_package\aladjevproceduresandfunctions.m",
"d:\avz_package\avz_package.m", "d:\avz_package\myfunctions.m",
"d:\avz_package\rans.m"}
In[1396]:= Mapp[FindFileObject, {Map10, AvzAgn}, "D"]
Out[1396]= {{ "d:\avz_package\avz_package.m"}, FindFileObject[AvzAgn, "D"]}
```

## Глава 8. Организация работы с пакетами в среде Mathematica

Подобно пакету *Maple*, *Mathematica* поддерживает достаточно развитые средства для расширения *своих* функциональных возможностей. Действительно, 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 этом случае *Mathematica* предоставляет возможность написания на ее языке специального пакета, содержащего определения требуемых функций. Такие пакеты написаны на *Math*-языке и содержат наборы определений функций, которые не входят в набор стандартных средств *Mathematica*; данные *пакеты* ориентированы на конкретные прикладные области, зачастую представляя специфический интерес. В то же время, оставляя традиционную терминологию по отношению к документам под названием «*пакеты*», мы считаем ее не вполне строгой в терминологическом аспекте; с обсуждением данного вопроса можно ознакомиться, в частности, в книгах [1-3,44]. В случае необходимости вызова функции, определение которой находится в пакете, он должен быть сначала загружен в текущий сеанс *Mathematica*. Загрузка необходимого пакета в текущий сеанс производится по команде `<<Имя_пакета``, например:

```
In[1942]:= << NumericalCalculus`
In[1943]:= {Nlimit[Sin[x]/x, x -> 0], Nlimit[Sin[x]/(x /Cosh[x] + Tan[x]), x -> -Infinity]}
Out[1943]= {1., -0.00125447 - 0.00173918 I}
```

Подробнее вопросы организации пакетов в среде *Mathematica* и работы с ними будут рассмотрены несколько ниже. Прежде всего, предопределенная переменная `$Packages` дает список контекстов пакетов, загруженных в текущий сеанс, например:

```
In[47]:= $Packages
Out[47]= {"NumericalCalculus`", "NumericalMath`Nlimit`", "NumericalMath`", "Global`",
  "PrimalityProving`", "NumberTheory`", "ResourceLocator`", "PacletManager`",
  "DocumentationSearch`", "AladjevProceduresAndFunctions`", "Jlink`",
  "WebServices`", "System`"}
```

Отметим, что в связи с понятием «*пакета*» в *Mathematica* дополнительно вводится ряд новых символов, предназначенных для обеспечения интерфейсных функций с таким пакетом. Например, эти символы могут соответствовать новым функциям или новым объектам, определенным в пакете. При загрузке пакета в текущий сеанс *все* входящие в него символы, ассоциированные с его именем, отражаются в системной переменной `$ContextPath`, определяющей пути поиска контекстов, например:

```
In[821]:= << PrimalityProving`
In[822]:= ProvablePrimeQ[2012]
Out[822]= False
In[823]:= $Packages
Out[823]= {"PrimalityProving`", "NumericalCalculus`", "DocumentationSearch`", "Jlink`",
  "AladjevProceduresAndFunctions`", "ResourceLocator`", "GetFEKernelInit`",
  "PacletManager`", "WebServices`", "System`", "Global`"}
```

```
In[824]:= $ContextPath
```

```
Out[824]= {"PrimalityProving`", "NumericalCalculus`", "PacletManager`", "System`",  
"AladjevProceduresAndFunctions`", "WebServices`", "Global`"}
```

Тогда как переменная *\$Context* определяет текущий контекст, который по умолчанию имеет значение «**Global`**», которое совпадает со значением вызова **Context[]**. Вызов же **Context[Имя]** возвращает контекст, присущий указанному имени, например:

```
In[312]:= Sv[x_, y_] := Module[{x + y}; F[x_] := x^2; G[x_, y] := 15*x + 23*y &
```

```
In[313]:= Map[Context, {Sv, Df, F, G, Sin}]
```

```
Out[313]= {"Global`", "AladjevProceduresAndFunctions`", "Global`", "Global`", "System`"}
```

т.е., имена *F* и *G* носят в текущем сеансе *глобальный* характер, тогда как *Df* определено в загруженном пакете с указанным контекстом, а функция **Sin** является стандартной. При этом, следует иметь ввиду, что вполне возможны и конфликтные ситуации при вызовах функций, чьи идентичные имена определены в различных пакетах. В любом случае, при вызове подобных функций *Mathematica* выводит сообщения о наличии неоднозначности такой функции, используя функцию, определенную последней. В таком случае следует использовать функцию **Remove[N1, N2, N3, ...]**, по которой все объекты с именами {*N1, N2, N3, ...*} в текущем сеансе более не распознаваемы. В этой связи определенный интерес может представить и достаточно несложная процедура, вызов которой **LocObj[x]** возвращает *трех*-элементный список, чей первый элемент определяет сам объект *x*, второй определяет его тип в разрезе {"**Procedure**", "**Function**", "**Expression**"}, тогда как третий – его изначальное *местоположение* в разрезе {"**Global**" – текущий сеанс, "**System**" – библиотека либо ядро *Mathematica*, и "**Пакет**" – загруженный в текущий сеанс системный или пользовательский пакет, содержащий определение объекта *x*}. Следующий фрагмент представляет исходный код процедуры наряду с примерами.

```
In[1104]:= LocObj[x_] := Module[{a = ToString1[Head1[x]], b},  
                                b[y_] := StringTake[Context[y], {1, -2}];  
                                If[a == "Procedure", {x, "Procedure", b[x]},  
                                If[a == "Function", {x, "Function", b[x]},  
                                If[SystemQ[x], {x, "Function", b[x]}, {x, "Expression", "Global"}]]]  
In[1105]:= Map[LocObj, {AllMatrices1, ProcQ, Sin, a + b, 70, LocObj}]  
Out[1105]= {{AllMatrices1, "Function", "AladjevProceduresAndFunctions"},  
            {ProcQ, "Procedure", "AladjevProceduresAndFunctions"},  
            {Sin, "Function", "System"}, {a + b, "Expression", "Global"},  
            {70, "Expression", "Global"}, {LocObj, "Procedure", "Global"}}
```

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 довольно типичным примером ее использования.

```
In[2056]:= 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[2057]:= Contexts1[]
Out[2057]= {"AladjevProceduresAndFunctions`", "Algebra`", "AlphaIntegration`",
            "Assumptions`", "AssumptionsDump`", "BinningUtilities`", "BoxForm`",
            "BoxFormat`", "BrowserCategoryLoad`", "CcodeGenerator`", .....,
            "Visualization`", "Wavelets`", "WebServices`", "WrappersDump`", "XML`"}
In[2058]:= Length[%]
Out[2058]= 141

```

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 пакетом либо определенные нами, в частности, определенные нами типы {"Procedure", "Function"} [90,99].

```

In[2087]:= 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 = "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, ! #[[2]] === Symbol &], 2]];
    a = ToExpression[StringReplace[ToString1[Map[DeleteDuplicates,
    Map[Sort, Map[Flatten, a]]], "AladjevProceduresAndFunctions`TypeActObj`" -> ""]];
    Append[{}, Do[a[[k]][[1]] = StringTake[a[[k]][[1]], {2, -1}], {k, Length[a]}]]; a]
In[2088]:= TypeActObj[]
Out[2088]= {"Symbol", "A", "B", "g", "H3", "m", "n", "Procedure", "System", "z"},
            {"Procedure", "Art", "Asv", "Kr"}, {"Function", "G", "V"}, {"List", "xyz"}]

```

Вызов процедуры **WhatObj[x]** возвращает местоположение объекта *x*, активированного (*доступного*) в текущем сеансе, а именно: "*Current Session*" – объект, чье определение вычислено в текущем сеансе пакета, "*Standard*" – стандартная функция пакета, *None* – неопределенный объект, "*Контекст*" – контекст, определяющий некоторый пакет, активированный в текущем сеансе и содержащий *определение x*-объекта, в противном случае вызов процедуры **WhatObj** возвращается *невычисленным*. Данная информация оказывается достаточно полезной при программировании целого ряда приложений.

```

In[739]:= WhatObj[x_ /; SymbolQ[x]] := Module[{a = ToString[Definition[x]], b,
    c = "" <> ToString[x] <> "", h, t}, If[a == "Null", None,
    b = Flatten[StringPosition[a, c]]; If[b == {}, If[MemberQ[Names["*"], ToString[x]],
    "Current Session", "Standard"], h = $Art23Kr15$; Clear[$Art23Kr15$];
    t = SubStr[a, b[[1]] - 1, {"[", "{"}, "", $Art23Kr15$] <> ""; $Art23Kr15$ = h; t]]]

```

```
In[740]:= Map[WhatObj, {Map3, Break, Agn, Sin, 6, GS, Continue, Goto, StrStr}]
Out[740]= {"AladjevProceduresAndFunctions`", "Standard", "Current Session", "Standard",
WhatObj[6], None, "Standard", "Standard", "AladjevProceduresAndFunctions`"}
```

Предыдущий фрагмент представляет исходный код процедуры с примерами. Между тем, для данных целей предпочтение следует отдавать все же процедуре **LocObj**.

Предоставляя механизм *функционального* расширения на основе пакетов, *Mathematica* при использовании как пользовательских, так и стандартных средств не делает между ними различий. Более того, существенное число стандартных функций расположены в пакетах *Mathematica*; при этом, ряд из этих пакетов загружаются сразу при запуске *Mathematica*, обеспечивая доступ к содержащимся в них функциям. Что до функций пользователя, определения которых находятся в его пакетах, то для их использования необходимо указать или автоматический режим загрузки в текущий сеанс пакета с их определениями, или делать это целенаправленно, *загружая* конкретный пакет по мере надобности содержащихся в нем определений пользовательских функций. При этом, если пользователь определяет пакет для автоматической загрузки, то содержащиеся в нем функции выступают на уровне стандартных функций *Mathematica* с тем только отличием, что *справки* по ним отсутствуют в справочной базе пакета. Вполне уместно отметить различие между пакетами и *notebooks* (блокнотами). Обе компоненты могут быть сохранены в файле с последующей их загрузкой в текущий сеанс. Если *notebook* служит в качестве визуализируемого документа с соответствующим пользовательским интерфейсом, то пакет служит лишь для загрузки в текущий сеанс с *вычислением* всех входящих в него определений функций. При этом, *блокноты* могут содержать секции, которые можно рассматривать в качестве пакетов, тогда как возможно поддерживать некоторые автоматически загружаемые пакеты в качестве блокнотов.

Вкратце рассмотрим структуру типичного пакета. В таком пакете выделяются 2 типа имен – *экспортируемые* (используемые вне пакета) и *внутренние* (используемые внутри пакета), которые различаются лишь контекстуально. Для возможности обращения по именам к определяемым ими пакетным функциям, как правило, пакеты должны быть предварительно загружены в текущий сеанс, например, следующим вызовом:

```
Get["<Путь к подкаталогу с пакетом>\\"<Основное имя пакета>.{m | mx}"]
```

или с *предварительным* переопределением в качестве текущего каталога, содержащего пакеты в форматах {«.m», «.mx»}, например:

```
SetDirectory["<Путь к подкаталогу с пакетом>"]; << Имя_пакета`
```

После загрузки пакета в *текущий* сеанс обращение к функциям/процедурам, которые содержатся в нем, можно обеспечить по их именам. Более того, определения имен, не предназначенных для использования *вне* пакета, кодируются в *его* контексте с именем **Package`Private`**, где **Package** – имя самого пакета. Данный контекст не добавляется в *глобальную* переменную **\$ContextPath**, определяющую путь поиска контекстов, посему прямого обращения к функциям/процедурам, определения которых представлены в контексте **`Private`**, не допускается. Для доступа к ним следует указывать *полный* путь, например, в следующем формате, а именно:

**Package`Private`Имя функции/процедуры[Аргументы]**

Следующий весьма простой фрагмент довольно наглядно иллюстрирует сказанное:

```
In[1032]:= BeginPackage["Test`"]
           ArtKr::usage = "The call ArtKr[x, y] returns  $x^2 + y^2 + (x + y)/z^3$ ."
           Begin["`Private`"]
           GS[x_, y_, z_] :=  $x^2 + y^2 + \text{Kr}[x, y, z]$ 
           Kr[x_, y_, z_] :=  $(x + y)/z^3$ 
           End[]
           Begin["`ArtKr`"]
           ArtKr[a_, b_, c_] := Test`Private`GS[a, b, c]
           End[]
           EndPackage[]

Out[1032]= "Test`"
Out[1033]= "The call ArtKr[x, y] returns  $x^2 + y^2$ ."
Out[1034]= "Test`Private`"
Out[1037]= "Test`Private`"
Out[1038]= "Test`ArtKr`"
Out[1040]= "Test`ArtKr`"
In[1042]:= 18*ArtKr[15, 23, 6]
Out[1042]= 81451
In[1043]:= {GS[x, y, z], Kr[x, y, z]}
Out[1043]= {GS[x, y, z], Kr[x, y, z]}
In[1044]:= $ContextPath
Out[1044]= {"Test`", "AladjevProceduresAndFunctions`", "PacletManager`",
           "WebServices`", "System`", "Global`"}
In[1045]:= {Test`Private`GS[x, y, z], Test`Private`Kr[x, y, z]}
Out[1045]= { $x^2 + y^2 + (x + y)/z^3$ ,  $(x + y)/z^3$ }
```

При необходимости использования пакетом средств, определения которых находятся в других контекстах, загружать их можно по функции **Needs**, одного из следующих 3 достаточно простых форматов, а именно:

**Needs["Контекст"]** – загружает соответствующий файл, если отвечающий ему контекст отсутствует в глобальной переменной **\$Packages**;

**Needs["Контекст", F]** – загружает файл F, если отвечающий ему контекст отсутствует в глобальной переменной **\$Packages**;

**Needs["F"]** – загружает файл F, в предположении, что его имя имеет вид «F.m».

В любом из перечисленных случаев в текущий сеанс загружается файл, отвечающий заданному контексту и содержащий определения средств, необходимых сеансу, в том числе и пакету, содержащему **Needs**. После загрузки объекты, определения которых находятся в файле, становятся доступными в текущем сеансе. Следующий достаточно простой фрагмент весьма наглядно иллюстрирует вышесказанное, а именно:

```
In[1046]:= SetDirectory["D:\\AVZ_Package"]
Out[1046]= "D:\\AVZ_Package"
In[1047]:= Needs["AladjevProceduresAndFunctions`"]
Needs::nocont: Context AladjevProceduresAndFunctions` was not created
when Needs was evaluated. >>
Out[1047]= {Temporary}
In[1048]:= Df[(Sin[x] + Cos[y])/(a + 1/x^3), 1/x^3]
Out[1048]= -((Cos[y] + Sin[x])/(a + 1/x^3)^2)
In[1049]:= $ContextPath
Out[1049]= {"Test`", "AladjevProceduresAndFunctions`", "PacletManager`",
"WebServices`", "System`", "Global`"}
```

Из примера следует, загрузив файл "AVZ\_Package.m" из каталога "D:\\AVZ\_Package", мы обеспечили доступность содержащихся в нем средств, без внесения его контекста в глобальную переменную **\$ContextPath**, что во многих случаях вполне достаточно. В общем случае принципиальная структурная схема пакета пользователя сможет быть представлена следующим образом, как иллюстрирует следующая схема, а именно:

```
BeginPackage["Имя_пакета`"]
P1::usage = "Справка по функции P1."
P2::usage = "Справка по функции P2."
=====
Pn::usage = "Справка по функции Pn."
Needs["`Контекст1`", "`Контекст2`", ..., "`КонтекстN`"]
Begin[["`Private`"]]
    <Определения внутренних процедур/функций>
End[]
=====
Begin["`P1`"]
P1[x_, ...] := Module[{...}, <Тело функции/процедуры>]
End[]
=====
Begin["`Pn`"]
Pn[x_, ...] := Module[{...}, <Тело функции/процедуры>]
End[]
EndPackage[]
```

Тогда как следующий достаточно простой фрагмент представляет пример типичной структурной организации пакета. Расширенная версия этого пакета находится в [90] в файле "AVZ\_Package.m", тогда как краткая справка по входящим в него средствам после его загрузки доступна по глобальной переменной **\$Help** (возвращается список имен всех средств, определения которых содержатся в пакете), тогда как расширенная справка выводится по вызову **Help[]** (возвращается справка по всем средствам с их определениями).

**BeginPackage["MyFunctions`"]**

**UprocQ::usage** = "The call UprocQ[x] returns False if x is not a procedure; otherwise, two-element list of the format {True, {Module | Block}} is returned."

**SymbolQ::usage** = "The call SymbolQ[x] returns True if x is a symbol; otherwise, False is returned."

**IsFile::usage** = "The call IsFile[x] returns True if x is a really existing datafile; otherwise, False is returned."

**FileQ::usage** = "The call FileQ[x] returns True if x is a really existing datafile; otherwise, False is returned."

**StrOfSymbQ::usage** = "The call StrOfSymbQ[x, A] returns True if a string x contains only symbols of a list A; otherwise, False is returned."

**Ind::usage** = "The call Ind[x] returns a list of form {Symbol, {Expression}} if x has format Symbol[Expression], and a simplified expression x otherwise. A sequence can be as an Expression."

**DuplicateLocalsQ::usage** = "The call DuplicateLocalsQ[P] returns True in the presence in definition of a procedure P of duplication of local variables, otherwise False is returned. In addition, at return of True through the second optional argument the list, simple or of ListList-type, whose elements define names of duplicated local variables with their multiplicities of occurrences in the list of local variables is returned."

**Uprocs::usage** = "The call Uprocs[] returns a list of procedures names activated in the current session. In the absence of such procedures the empty list is returned."

**TypeActObj::usage** = "The call TypeActObj[] returns the nested list whose sublists contain names of objects activated in the current session (in Input-mode) in string format and as the first element - their type, recognized by the package or defined by us, in particular, {"Procedure"}, {"Function"}."

**ProcQ::usage** = "The call ProcQ[x] returns True if x is a procedure and False otherwise."

**SubProcs::usage** = "The procedure call SubProcs[P] returns the 2-element nested list of ListList-type whose the first element is the list of headings of subprocedures composing a main procedure P of Module-type whereas the second element is the list of the generated names of all these subprocedures including a main procedure P which are activated in the current session P. Between the both lists one-to-one correspondence exists."

**SubsProcQ::usage** = "The call of procedure SubsProcQ[x, y] returns True if y is a subprocedure of a procedure x, and False otherwise; in addition, through 3-rd optional argument the call of procedure returns the nested list, whose sublists by the first element contain heading of procedure with name x, but with the different headings, and by the second - headings of subprocedures corresponding to them with a name y."

**Op::usage** = "The procedure call Op[x] extracts operands from an expression x. The procedure is an analog of built-in function of Maple accurate within axiomatics of both packages."

**HeadPF::usage** = "The call HeadPF[F] returns the heading in string format of an object activated in the current session with name F of type {procedure, function or block}. In addition, for a system object is returned its name F, whereas in all other cases the call is returned unevaluated."

**BitSet1::usage** = "The call BitSet1[n, p] returns the result of setting into positions of binary representation of an integer n which are defined by the first elements of sublists of a nested list p values {0 | 1}; in case of non-nested list p replacement of value only in a unique position of number n is made."

**BitGet1::usage** = "The call BitGet1[x, n, p] returns the list of bits in positions of binary representation of an integer n, that are defined by a list p; in case of an integer p the bit in a position p is returned. Whereas the call BitGet1[x, n, p] through a symbol x in addition returns number of bits in binary representation of an integer n."

**ProcQ1::usage** = "The call ProcQ1[x, y] returns True if x is an object on the basis of Block, Module or DynamicModule, and False or \"Others\" otherwise. In addition, through the second argument y the type of the object x is returned."

**SubsDel::usage** = "The call of procedure SubsDel[S, x, y, p] returns the result of deleting from a S-string of all substrings which are limited on the right (at the left) by substring x and at the left (on the right) by the first met symbol in string format of a list y; moreover, search of a y-symbols is made to the left (p = -1) or to the right (p = 1). In addition, the deleted substrings will contain substring x from one end along with the first symbol up to met of y from other end. Furthermore, if not symbols of the list y that were not found, then the rest in the string S is deleted."

**NbCallProc::usage** = "The call NbCallProc[x] returns Null, i.e. nothing, over-activating the definition of a procedure/function x in the current session, whose definition was in a nb-file loaded and was previously activated in the current session. It is necessary to note, that the call of procedure NbCallProc[x] over-activates all definitions of procedures/functions with an identical name x and with various headings in the current session."

**Sequences::usage** = "The function Sequences[x] generalizes the standard function Sequence[x], allowing a list {a, b, c, ...} as an actual argument x and returning Sequence[a, b, c, ...]."

**ArgsProc::usage** = "The procedure call ArgsProc[P] returns the list of formal arguments of a procedure P with the conditions attributed to them, i.e. in the form «x\_ /; Test». If argument P does not define a procedure activated in the current session, then the call ArgsProc[P] is returned unevaluated; the given result takes place and for a system function as an actual argument P. In case of impossibility to determine formal arguments the call of procedure ArgsProc[P] returns the value Undefined."

**Tuples1::usage** = "The procedure call Tuples1[A, n] returns the list of all possible tuples in string format of length n, composed from symbols of an alphabet A."

**GC::usage** = "The call GC[x] returns the unique decimal code of arbitrary x-expression."

**ExprOfStr::**usage = "The call ExprOfStr(x, m, n, L) returns the result of extracting from a string w, limited by its m-th position and the end, of the first correct expression on condition that search is made to the left (n = -1)/to the right (n = 1) from the given position and a symbol following or in front of a found expression should belong to a list L. The result of the procedure call is returned in string-format; at absence of a correct expression \$Failed is returned, while the call on inadmissible actual arguments is returned unevaluated."

**ListStrToStr::**usage = "Procedure ListStrToStr represents indubitable interest in work with lists in the string format, more precisely, the call of procedure ListStrToStr[x] with single factual argument x where argument x has format {"a", "b", "c", ...}, converts x into string of the format "a, b, c, ... "; if the procedure call uses any additional expression p as the second argument, the call returns a string of the format "abcd...".

**Nvalue::**usage = "The procedure call Nvalue[w] returns the list of global names to which in the current session the value w has been assigned."

**ProcsAct::**usage = "The procedure call ProcsAct[] returns the nested 4-element list in which the sublists as the first element define types of objects in the context of {"Block", "Module", "DynamicModule" and "Others"}, whereas other elements of sublists define names of objects that have been activated in the current session and have the appropriate type."

**RemProcOnHead::**usage = "The procedure call RemProcOnHead[x] returns value "Done", having removed a procedure with heading x, given in string-format, from the current session of the package; otherwise, value \$Failed is returned."

**LocObj::**usage = "The call LocObj[x] returns the three-element list, whose first element defines an object x, the second defines its type in the context of {"Procedure", "Function", "Expression"}, whereas the third - its initial location in the context of {"Global" - the current session, "System" - library or kernel of Mathematica, and "Package" - a system or user package, which is loaded into the current session and contains the definition of the object x}."

**ActRemObj::**usage = "The procedure call ActRemObj[x, y] depending on value {"Act", "Rem"} of the second factual argument deletes from the current session of the package an object defined by its name in string-format, or activates it in the current session or in other session accordingly. Successful removal of an object from the current session of the package returns the value "Remove" whereas its restoration in the current session of the package returns the value "Activate". If the file containing definition of a removed object, has not been found in the user home directory \$HomeDirectory the procedure call ActRemObj is returned unevaluated."

**ProcCalls::**usage = "The procedure call ProcCalls[P] returns the list of headings in string-format of all procedures with name P, which have been activated in the current session of the package."

**MemberQ1::usage** = "The procedure call MemberQ1[L, x, y] returns True if x is an element of any level of nesting of a list L (provided that a non-nested list has level of nesting 0); otherwise, False is returned. In case of return True through the third argument y the list of levels of the list L which contain occurrences of x-value is returned. The procedure MemberQ1 in the certain degree expands the standard function MemberQ onto the nested lists."

**MemberT::usage** = "The procedure call MemberT[L, x] returns the total number of occurrences of an expression x into a list L."

**MinusList1::usage** = "The call of procedure MinusList1[x, y] returns result of subtraction of the list y from the list x which consists in parity removal from the list x of all occurrences of elements from the list y."

**EmptyFileQ::usage** = "The call EmptyFileQ[F] returns True if F is an empty datafile, and False otherwise; at that, if F is not a datafile the procedure call is returned unevaluated."

**Rename::usage** = "In the regular mode the procedure Rename[x, y] returns Null-value, i.e. nothing, providing replacement of a name x of a defined object onto a name y with preservation of all attributes of the x-object. If y-argument defines a name of an defined object or of an undefined name with attributes attributed to it, the procedure call is returned unevaluated."

**Subs::usage** = "The procedure call Subs[x, y, z] returns the result of substitutions into an expression x of expressions z instead of occurrences in it of subexpressions y. In addition, if as x any correct expression admitted by Math-language then as a single substitution or their set coded as y = {y1, y2, ..., yn} and z = {z1, z2, ..., zn} are allowable as the second and third arguments determining substitutions of the format y -> z, defining a set of substitutions {y1 -> z1, y2 -> z2, ..., yn -> zn} carried out consistently."

**Arity::usage** = "The call Arity[x] returns the arity of a procedure/function defined by x-name; in addition; for x with undefined number of arguments the call returns the value Undefined, on standard functions x the value System is returned, whereas on other expressions x the call Arity[x] returns value \$Failed."

**ArgsLocals::usage** = "The call ArgsLocals[x] returns the nested 2-element list whose first element-sublist defines formal arguments of a procedure x whereas the second element-sublist defines its local variables with initial values in string format. At absence of local variables the empty sublist is returned whereas on standard functions the call of ArgsLocals returns the value System; on other actual arguments x the call ArgsLocals[x] returns an expression x in the reduced form."

**DefFunc3::usage** = "The call DefFunc3[x] returns the list of definitions, given in string format, of x-procedures of the same name with the various headings which were activated in the current session of the package; In addition, the order of definitions of the returned list corresponds to the order of their output by standard function Definition of the package."

**Df::usage** = "The call Df[x, y] returns the result of differentiation of x on y. The procedure

expands standard function D."

**Int::usage** = "The call Int[x, y] returns the result of integration of x on y. The procedure expands standard function Integrate."

**Mapp::usage** = "The call Mapp[F, L, g] generalizes standard function Map on number of arguments the greater than 2. For example, Mapp[a, {b, c, d}, x, y, z] == {a[b, x, y, z], a[c, x, y, z], a[d, x, y, z]}; Mapp[StringReplace, {"812345265\", \"72345957\"}, {"2\" -> \"V\", \"5\" -> \"G\"}] == {"81V34GV6G\", \"7V34G9G7\"}. In addition, the user procedure/function, a standard function or a symbol F can be as a factual argument F."

**TwoHandQ::usage** = "The procedure call TwoHandQ[x] returns True if an expression x has one of the following types, namely: {"+"\", \">=\", \"<=\", \"&&\", \"| |\", \"-\", \"^\", \"\*\*\", \"<\", \"==\", \"!=\", \">\", \"->\"}, and False otherwise. In addition, in case of the call TwoHandQ[x, y] through the second argument y the type of x is returned if main result of the call is True."

**Begin["`ExtrExpr`"]**

```
ExtrExpr[S_/, StringQ[S], N_/, IntegerQ[N], M_/, IntegerQ[M]] := Module[{b, c, d, k = -1,
  Res = {}, a = StringLength[S]}, 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]]
```

**End[]**

**Begin["`ExtrCall`"]**

```
ExtrCall[N_, P_] := Module[{a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]],
  b, x}, 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[#] &]]
```

**End[]**

**Begin["`TypeActObj`"]**

```
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 = "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, ! #[[2]] === Symbol &], 2]];
  a = ToExpression[StringReplace[ToString1[Map[DeleteDuplicates,
  Map[Sort, Map[Flatten, a]]], "AladjevProceduresAndFunctions`TypeActObj" -> ""]]];
```

```

Append[{}, Do[a[[k]][[1]] = StringTake[a[[k]][[1]], {2, -1}], {k, Length[a]}]; a
End[]

Begin["`ArgsProc`"]
ArgsProc[x_ /; ProcQ[x]] := Module[{a = HeadPF[x], b, c = ToString[x] <> "["},
    b = Flatten[StringPosition[a, {c}]];
    a = ToExpression["{" <> StringTake[a, {b[[2]] + 1, -2}] <> "}"];
    If[a == {}, Undefined, a]]
End[]

Begin["`ArgsLocals`"]
ArgsLocals[x_] := Module[{a = "" <> ToString[x] <> "", b, c, d, h},
    If[MemberQ[{And, Complex, Integer, List, Not, Or, Plus, Power,
        Rational, Real, String, Times}, Head[x]], Return[x],
    If[SystemQ[x], Return[ToExpression[SubsDel[ToString[System], a, {" ", ",", "["], -1}]],
    b = SubsDel[HeadPF[x], a, {"[", ",", " "}, -1]; c = SubsDel[Locals[x], a, {"[", ",", " "}, -1];
    b = ToExpression["{" <> StringTake[b, {StringLength[ToString[x]] + 2, -2}] <> "}"];
    c = Flatten[StringSplit[c, " "]; c[[1]] = StringTake[c[[1]], {2, -1}];
    c[[-1]] = StringTake[c[[-1]], {1, -2}]; {b, If[c == {""}, {}, c]}]
End[]

Begin["`TwoHandQ`"]
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]]
End[]

Begin["`ExprOfStr`"]
ExprOfStr[x_ /; StringQ[x], 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]
End[]

Begin["`RhsLhs1`"]

```

```

RhsLhs1[x_, y_] := Module[{a = Head[x], b = {x, y}, d, c = {{Greater, ">"}, {And, "&&"},
    {GreaterEqual, ">="}, {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]]]]
End[]

Begin["`Head1`"]
Head1[x_] := Quiet[ToExpression[SubsDel[ToString[If[SystemQ[x], System,
    If[ProcQ[x], Procedure, If[HeadingQ[ToString1[HeadPF[x]],
    Function, Head[x]]]], "`Head1`", {"(", "(", "(", -1]]]
End[]

Begin["`LocObj`"]
LocObj[x_] := Module[{a = ToString1[Head1[x]], b}, b[y_] := StringTake[Context[y], {1, -2}];
    If[a == "Procedure", {x, "Procedure", b[x]}, If[a == "Function", {x, "Function", b[x]},
    If[SystemQ[x], {x, "Function", b[x]}, {x, "Expression", "Global"}]]]]
End[]

Begin["`SubsProcQ`"]
SubsProcQ[x_] := Module[{a = {x}, b = Length[{x}], c, d, k = 1, j = 1, Res = {}},
    If[b >= 2 && ProcQ[a[[1]]] && ProcQ[a[[2]]],
    {c, d} = {StringSplit[ToString[InputForm[DefFunc[a[[1]]]], "\n\n"],
    StringSplit[ToString[InputForm[DefFunc[a[[2]]]], "\n\n"],
    Defer[SubsProcQ[x]]]; {c, d};
    For[k, k <= Length[d], k++, For[j, j <= Length[c], j++,
    If[! StringFreeQ[c[[j]], d[[k]], Res = Append[Res, {StringTake[c[[j]],
    {1, Flatten[StringPosition[c[[j]], " := "][[1]] - 1}], StringTake[d[[k]],
    {1, Flatten[StringPosition[d[[k]], " := "][[1]] - 1}], Continue[]]];
    If[b > 2 && ! HowAct[a[[3]]],
    Quiet[ToExpression[ToString[a[[3]]] <> " = " <> ToString1[ Res]], Null];
    If[Res == {}, False, True]]
End[]

Begin["`MemberQ1`"]
MemberQ1[L_ /; ListQ[L], 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]]
End[]

Begin["`GotoLabel`"]

```

```

GotoLabel[P_;/ ProcQ[P]] := Module[{a = ToString1[DefFunc[P]], b = {}, c = {}, d, h, p, k = 1,
    f = Union[CharacterRange["a", "z"], CharacterRange["A", "Z"]], j, g, l},
    If[P === GotoLabel, Return[{{}, {}, {}, {}], {g, l} = {{}, {}];
    {d, h} = Map3[StringPosition, a, {"Goto", "Label"}];
    d = Select[d, ! MemberQ[f, StringTake[a, {#[[1]] - 1, #[[1]] - 1}]] &];
    h = Select[h, ! MemberQ[f, StringTake[a, {#[[1]] - 1, #[[1]] - 1}]] &];
    For[k, k <= Length[h], k++, p = StringTake[a, h[[k]]];
    For[j = h[[k]][[2]] + 1, j < Infinity, j++, p = p <> StringTake[a, {j, j}];
    If[Quiet[ToExpression[p]] === $Failed, Continue[], l = Append[l, p]; Break[]];
    For[k = 1, k <= Length[d], k++, p = StringTake[a, d[[k]]];
    For[j = d[[k]][[2]] + 1, j < Infinity, j++, p = p <> StringTake[a, {j, j}];
    If[EvenQ[StringCount[p, {"[", "]" }]], g = Append[g, p]; p = {}; Break[]; Continue[]];
    {g, l} = {DeleteDuplicates[StringReplace[Map[ToString,
    Map[ToExpression, StringReplace[g, "Goto" -> "Goto1["],
    "Goto1" -> "Goto["], Map[ToString, Map[ToExpression, l]]];
    p = DeleteDuplicates[StringReplace[l, "Label" -> "Goto["];
    {g, l, Select[g, ! MemberQ[p, #] &], MinusList[l, DeleteDuplicates[l]]]
End[]
Begin["ActRemObj`"]
ActRemObj[x_;/ StringQ[x], y_;/ MemberQ[{"Act", "Rem"}, y]] :=
    Module[{a = $HomeDirectory <> "\\ " <> ToString[x] <> ".$ArtKr$", b, c},
        If[HowAct[x] && y == "Rem", b = OpenWrite[a];
        WriteString[a, ToString[DefFunc[x]]]; Close[b]; Remove[x]; "Remove",
        If[! HowAct[x] && y == "Rem", "Remove",
        If[FileExistsQ[a], b = OpenRead[a];
        Map[ToExpression, ReadList[b, String]]; Close[b]; DeleteFile[a];
        "Activate", Return[Defer[ActRemObj[x, y]]]]]]
End[]
Begin["HeadPF`"]
HeadPF[F_;/ SymbolQ[F]] := Module[{a = ToString[InputForm[DefFunc[F]]], b, c = " := "},
    If[StringFreeQ[a, c], Return[HoldForm[F]], b = Flatten[StringPosition[a, c]];
    b = StringTake[a, {1, b[[1]] - 1}];
    SubsDel[If[! StringFreeQ[b, ToString[F] <> "["], b, HoldForm[F]],
    "" <> ToString[F] <> "", {"[", ",", " ", -1]]
End[]
Begin["Rename`"]
Rename[x_String;/ HowAct[x], y_;/ ! HowAct[y]] := Module[{a = Attributes[x]},
    ClearAttributes[x, a]; ToExpression[ToString[y] <> StringTake[ToString1[DefFunc[x]],
    {StringLength[x] + 1, -1}]; SetAttributes[y, a]; Clear[x]
End[]
Begin["VarExch`"]

```

```

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]]]]]; ;
End[]
Begin["`Arity`"]
Arity[x_] := Module[{a = If[SystemQ[x], System, ArgsProc[x]]}, If[a === System, System,
  If["InputForm[ArgsProc[" <> ToString1[x] <> "]]" ===
    ToString1[InputForm[a]], Return[$Failed], If[ListQ[a],
  If[Length[Select[Map[ToString, a], ! StringFreeQ[#, "___"] &]] != 0,
    Undefined, Length[a]], Undefined]]]
End[]
Begin["`ObjType`"]
ObjType[x_] := Module[{a, b, c, d = {}, h},
  If[ToString1[HeadPF[x]] === "HeadPF[" <> ToString1[x] <> "]" ||
    SymbolQ[HeadPF[x]], Return[Head[x]], b = DefFunc3[x]; c = Length[b]];
  Do[d = Append[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]]]]}]]]], {k, c}]; d]
End[]
Begin["`IsFile`"]
IsFile[x_ /; StringQ[x]] := If[FileExistsQ[x], If[! DirectoryQ[x], True, False], False]
End[]
EndPackage[]

```

Представленный пример пакета **"MyFunctions"** после вычисления в текущем сеансе обеспечивает возможность доступа ко всем объектам, в частности, процедурам либо функциям, определения которых в нем содержатся, но при том условии, что каждому определению объекта должна предварительно быть задана по нему справка формата **Имя\_объекта::usage = "Справка"**, как иллюстрирует первая часть фрагмента, которая следует за открывающей скобкой **BeginPackage["MyFunctions"]** пакета **MyFunctions**. В результате вычисления пакета в текущем сеансе возвращается имя пакета и тексты всех справок по содержащимся в нем объектам, как иллюстрирует фрагмент ниже:

```
Out[1]= "MyFunctions`"
Out[2]= "The call UprocQ[x] returns False if x is not a procedure; otherwise, two-element
        list of the format {True, {Module | Block}} is returned."
Out[3]= "The call SymbolQ[x] returns True if x is a symbol; otherwise, False is returned."
        =====
Out[45]= "The procedure call TwoHandQ[x] returns True if an expression x has one of the
        following types, namely: { "+", ">=", "<=", "&&", "|", "-", "^", "*", "<", "=", ">",
        "!=", "->" }, and False otherwise. In addition, in case of the call TwoHandQ[x, y]
        through the second argument y the type of x is returned if main result of the call
        is True."
In[46]:= <Исходные коды вычисленных определений объектов пакета MyFunctions>
Out[46]= "MyFunctions`ExtrExpr`"
Out[48]= "MyFunctions`ExtrExpr`"
Out[49]= "MyFunctions`ExtrCall`"
Out[51]= "MyFunctions`ExtrCall`"
Out[52]= "MyFunctions`TypeActObj`"
Out[54]= "MyFunctions`TypeActObj`"
Out[55]= "MyFunctions`ArgsProc`"
Out[57]= "MyFunctions`ArgsProc`"
Out[58]= "MyFunctions`ArgsLocals`"
Out[60]= "MyFunctions`ArgsLocals`"
Out[61]= "MyFunctions`TwoHandQ`"
Out[63]= "MyFunctions`TwoHandQ`"
Out[64]= "MyFunctions`ExprOfStr`"
Out[66]= "MyFunctions`ExprOfStr`"
Out[67]= "MyFunctions`RhsLhs1`"
Out[69]= "MyFunctions`RhsLhs1`"
Out[70]= "MyFunctions`Head1`"
Out[72]= "MyFunctions`Head1`"
Out[73]= "MyFunctions`LocObj`"
Out[75]= "MyFunctions`LocObj`"
Out[76]= "MyFunctions`SubsProcQ`"
Out[78]= "MyFunctions`SubsProcQ`"
```

```

Out[79]= "MyFunctions`MemberQ1`"
Out[81]= "MyFunctions`MemberQ1`"
Out[82]= "MyFunctions`GotoLabel`"
Out[84]= "MyFunctions`GotoLabel`"
Out[85]= "MyFunctions`ActRemObj`"
Out[87]= "MyFunctions`ActRemObj`"
Out[88]= "MyFunctions`HeadPF`"
Out[90]= "MyFunctions`HeadPF`"
Out[91]= "MyFunctions`Rename`"
Out[93]= "MyFunctions`Rename`"
Out[94]= "MyFunctions`VarExch`"
Out[96]= "MyFunctions`VarExch`"
Out[97]= "MyFunctions`Arity`"
Out[99]= "MyFunctions`Arity`"
Out[100]= "MyFunctions`ObjType`"
Out[102]= "MyFunctions`ObjType`"
Out[103]= "MyFunctions`IsFile`"
Out[105]= "MyFunctions`IsFile`"
In[110]:= $Packages
Out[110]= {"MyFunctions`", "ResourceLocator`", "DocumentationSearch`", "Jlink`",
           "PacletManager`", "GetFEKernelInit`", "WebServices`", "System`", "Global`"}

```

Итак, для возможности последующего доступа к процедурам/ функциям (*определения которых со справками находятся в нем*) как в текущем сеансе, так и с целью сохранения в файле форматов, например, {«.nb», «.m»}, предварительно требуется активировать; иными словами вычислить, в результате чего возвращается информация, указанная в предыдущем фрагменте, именно: (1) имя пакета *"MyFunctions"*, (2) справки (*если они определены в пакете*) по всем содержащимся в пакете функциям/процедурам, (3) коды всех процедур/ функций пакета и (4) парные контекстные имена функций в формате *"MyFunctions`Имя\_функции"*. При этом, порядок определения справок в их области не обязательно должен совпадать с порядком исходных кодов объектов в их области, тогда как исходный код без соответствующей ему справки в текущем сеансе не будет *активирован*, т.е. такой объект останется недоступным в текущем сеансе. Организация такого типа с логической точки зрения вполне оправдана, если исходить из того, что без информации по объекту работать с ним весьма проблематично. Простой пример пакета иллюстрирует взаимосвязь между определением объекта и справкой по нему.

```

In[1]:= BeginPackage["AVZ`"]
        A::usage = "A[x] := x^2"
        B::usage = "B[x] := x^3"
        Begin["`A`"]
        A[x_] := x^2
        End[]

```

```

Begin["`H`"]
H[x_] := x^3
End[]
EndPackage[]

Out[1]= "AVZ`"
Out[2]= "A[x] := x^2"
Out[3]= "B[x] := x^3"
Out[4]= "AVZ`A`"
Out[6]= "AVZ`A`"
Out[7]= "AVZ`H`"
Out[9]= "AVZ`H`"
In[11]:= ?B
      B[x] := x^3
In[12]:= ?A
      A[x] := x^2
In[13]:= ?H
      Information::notfound: Symbol H not found. >>
In[14]:= {A[75], B[75], H[75]}
Out[14]= {5625, B[75], H[75]}
In[15]:= $Packages
Out[15]= {"AVZ`", "ResourceLocator`", "DocumentationSearch`", "GetFEKernelInit`",
      "Jlink", "PacletManager`", "WebServices`", "System`", "Global`}

```

Из представленного фрагмента следует, что пакет "AVZ" содержит определения двух простых функций *A* и *H*, тогда как справки определены для объектов *A* и *B* (последняя из которых не поддерживается соответствующим определением объекта *B*). Вычисление этого пакета показывает, что по конструкции *?h* выводятся все имеющиеся в пакете справки безотносительно наличия в пакете соответствующих определений объектов, тогда как доступными становятся только объекты, для которых закодированы и определения, и справки. Вызовы объектов *A*, *B* и *H* наглядно иллюстрируют сказанное. Итак, пакеты в среде *Mathematica* должны содержать и определения объектов, и соответствующие им справки; при этом, при наличии нескольких справок для одного и того же объекта воспринимается лишь определенная последней. Интересно отметить, что для пакета *Maple* имеет место совершенно иная организация пользовательских библиотек, когда определения и справки по объектам разнесены, предоставляя возможность создавать доступные программные средства, непосредственно не ассоциируя их со справочной информацией по ним. При этом, программное средство может быть как без справки по нему, так и со справкой, организованной аналогично справочной системы пакета, обеспечивая доступ к ней на уровне справки по пакетным средствам. Данный прием можно, в частности, использовать для сокрытия некоторых программных средств. Так, в *Maple* допустимо использование встроенной функции *goto* безусловного перехода

> ParProc(goto); ⇒ builtin function, 226

(Maple 11)

Между тем, на пользовательском уровне справочная информация по *goto* недоступна [44]. На наш взгляд, подобная организация пользовательских средств более удобна, в значительной степени лучше интегрируя их в программную среду пакета.

Завершаются оба предыдущих фрагмента вызовом системной переменной **\$Packages**, которая возвращает *списки* контекстов, отвечающих пакетам, загруженным в текущий сеанс *Mathematica*; первым в них указан контекст пакета "MyFunctions`" либо "AVZ`". В результате активации пакета в текущем сеансе становятся *доступными* все функции и/или процедуры, которые определены в нем вместе со справками по ним. Однако, *наибольший* интерес представляет возможность сохранения пакета в файлах формата {«.m», «.mx»}, что в последующем обеспечивает возможность загрузки его в текущий сеанс с активацией определений содержащихся в нем процедур/функций, тем самым расширяя, порой, существенно возможности программной среды *Mathematica*. Ниже довольно простой фрагмент иллюстрирует сохранение пакета "MyFunctions`" в двух файлах форматов {«.m», «.mx»} с возможностью последующей загрузки его в новую сессию с проверкой доступности в *новой* сессии функций/процедур, чьи определения содержатся в пакете наряду со справками по ним.

```
In[128]:= Directory[]
Out[128]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[129]:= SetDirectory["D:\\AVZ_Package"]
Out[129]= "D:\\Math_myLib"
In[130]:= Save["D:\\AVZ_Package\\MyFunctions.mx", "MyFunctions`"]
In[131]:= Save["D:\\AVZ_Package\\MyFunctions.m", "MyFunctions`"]
In[132]:= DumpSave["D:\\AVZ_Package\\MyFunctions.mx", "MyFunctions`"]
Out[132]= {"MyFunctions`"}
```

Предварительно проверяется текущий каталог, где по умолчанию будет произведено сохранение пакета, затем для удобства текущий каталог переопределяется на каталог "D:\\AVZ\_Package", где по функциям **Save** и **DumpSave** сохраняется искомый пакет в файлах сразу в *двух* форматах {«.m», «.mx»}. Следует отметить, что по функции **Save** сохранение производится в идентичных по содержанию файлах *входного* формата, но с разными *расширениями* имени, тогда как по функции **DumpSave** сохранение пакета производится в *mx*-файле бинарного формата с возвратом списка с именем пакета в контекстном формате. Сохранения по **Save** и **DumpSave** принципиально различны.

По функции **Save** независимо от указания расширения имени получаем идентичные файлы *входного* формата, тогда как по функции **DumpSave** получаем файл *бинарного* формата, который оптимизирован для загрузки в текущий сеанс. Загружаются такие файлы по функции **Get**, однако они жестко привязаны к вычислительной платформе, т.е. *непереносимы*. Поэтому для обеспечения переносимости файлов с пакетами нужно сохранять их по функции **Save**. После сохранения пакета производится перезагрузка *Mathematica* и в новом сеансе работы с ним выполняется загрузка пакета с *выборочной* проверкой как *доступности* содержащихся в нем функций/процедур, так и *справок* по ним. Нижеследующий фрагмент представляет протокол проведенной проверки на предмет корректности загруженного пакета "MyFunctions`".

```

In[1772]:= SetDirectory["D:\\AVZ_Package"]
Out[1772]= "D:\\AVZ_Package"
In[1773]:= << MyFunctions`
In[1774]:= $Packages
Out[1774]= {"MyFunctions`", "ResourceLocator`", "DocumentationSearch`", "Jlink`",
            "GetFEKernelInit`", "PacletManager`", "WebServices`", "System`", "Global`}
In[1775]:= $ContextPath
Out[1775]= {"MyFunctions`", "PacletManager`", "WebServices`", "System`", "Global`}
In[1776]:= Context[ExprOfStr]
Out[1776]= "MyFunctions`"
In[1777]:= ExprOfStr["abccdfg(Sin[x] + x*Cos[x])mnp", 7, 1, {"m"}]
Out[1777]= "(Sin[x] + x*Cos[x])"
In[1778]:= ?ExprOfStr
            The call ExprOfStr(x, m, n, L) returns the result of extracting from a string w,
            limited by its mth position and the end, of the first correct expression on
            condition that search is made to the left (n = -1)/to the right (n = 1) from the
            given position and a symbol following or in front of a found expression should
            belong to a list L. The result of the procedure call is returned in string-format; at
            absence of a correct expression $Failed is returned, while the call on
            inadmissible actual arguments is returned unevaluated.
In[1779]:= ?Df
            Df[x, y] returns the result of differentiation of x on y. The procedure expands
            standard function Df.
In[1780]:= Df[(a + Sin[x])/Cos[x], Cos[x]]
Out[1780]= -Sec[x]^2 (a + Sin[x])
In[1781]:= NamesProc[]
Out[1781]= {}
In[1782]:= ?NamesProc
            NamesProc[] returns the list of names of the user procedures activated in the
            current session.

```

Определяя справки (*usage*) по всем процедурам/функциям, чьи определения входят в пакет, обеспечивается корректность экспорта их в контексте пакета **MyFunctions`**, а также возможность получения справок по  $\{?N | ??N\}$ , где *N* – имя средства из пакета. О взаимосвязи определений средств и справок по ним говорилось несколько выше.

По функции **Needs["Пакет`"]** производится вызов **Get["Пакет`"]**, где по соглашению в качестве файла, загружаемого таким образом, выступает файл, содержащий пакет с контекстом "Пакет`". Как правило, по функции **Needs["Имя`"]** в текущий сеанс будет загружаться файл с именем «Имя.m», расположенный в текущем подкаталоге сеанса. Если контекст "Пакет`" отсутствует в списке, определяемом переменной **\$Packages**, то 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» 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 представлен в следующем достаточно простом и прозрачном фрагменте, а именно:

```
In[1]:= Need[x_] := Module[{a = $Path, b = If[Length[{x}] > 1, {x}[[2]], "Null"], c, p,
                                d = {x}[[1]]},
    If[! StringQ[d] && StringTake[d, -1] === "", Return[False],
    If[MemberQ[$Packages, d], Return[True],
    c = Quiet[Check[Needs[d], False, Needs::nocont]]];
    If[ToString[c] != "False", Return[True],
    If[b === "Null", Return[False], ClearAttributes[$Path, Protected];
    $Path = DeleteDuplicates[Append[$Path, b]];
    Quiet[Check[Needs[d], p = False, {Get::noopen, Needs::nocont}]];
    SetAttributes[$Path, Protected]; If[p === False, True, False]]]

In[2]:= $Packages
Out[2]= {"ResourceLocator", "DocumentationSearch", "GetFEKernelInit", "Jlink",
    "PacletManager", "WebServices", "System", "Global"}
In[3]:= Need["MyFunctions", "D:\\AVZ_Package"]
Out[3]= True
In[4]:= Need["AVZ", "D:\\AVZ_Package"]
Out[4]= True
In[5]:= Need["AVZAGN", "D:\\AVZ_Package"]
Out[5]= True
In[6]:= Need["AVZ70", "D:\\AVZ_Package"]
Out[6]= False
In[7]:= $Packages
Out[7]= {"AVZAGN", "AVZ", "MyFunctions", "ResourceLocator", "GetFEKernelInit",
    "DocumentationSearch", "Jlink", "PacletManager", "WebServices", "System",
    "Global"}
In[8]:= Map[Context, {ExprOfStr, A1, A2}]
Out[8]= {"MyFunctions", "AVZAGN", "AVZ"}
In[9]:= {ExprOfStr["abcdfg(Sin[x] + x*Cos[x])mnp", 7, 1, {"m"}], A[5], A1[25]}
Out[9]= {"(Sin[x] + x*Cos[x])", 25, 625}
```

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 пакетной переменной **\$Path**, с возвратом **True**; в противном случае возвращается значение **False**. Тогда как вызов процедуры **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, определяемых пакетной переменной **\$Path**, или в каталоге, определенном аргументом *y*, с возвратом значения **True**; в противном случае возвращается значение **False**.

Таким образом, создав *nb*-документ с определениями объектов, снабдив их справками и сохранив его по функции {*Save* | *DumpSave*} в файле формата {«.m» | «.mx»}, получаем возможность в последующих сеансах загружать его в текущий сеанс функцией **Needs** либо представленной процедурой **Need** с получением доступа как к содержащимся в нем программным объектам, так и к справкам по ним.

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[837]:= DeclarePackage["MyFunctions`", {"ProcQ", "Df", "Locals", "Uprocs", "SortNL"}]
Out[837]= "MyFunctions`"
In[838]:= $Packages
Out[838]= {"ResourceLocator`", "DocumentationSearch`", "GetFEKernelInit`", "Jlink`",
           "PacletManager`", "WebServices", "System", "Global`}
In[839]:= Df[(Sin[x] + Cos[x])/Log[x]^2, Log[x]^2]
Out[839]= -((Cos[x] + Sin[x])/Log[x]^4)
In[840]:= Map[ProcQ, {Locals, Avz_68, Df, Mapp, SortNL}]
Out[840]= {True, False, True, False, True}
In[841]:= ?Df
           Df[x, y] returns the result of differentiation of x on y. The procedure expands the
           standard function Df.
```

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 этого символа.

Представляется довольно интересным получение списка имен объектов, определения которых находятся в пакете и по которым имеются в пакете справки (*usage*). Данную задачу решает достаточно простая процедура **PackNames["Пакет`"]**, возвращающая список имен вышеуказанных объектов в заданном пакете "Пакет`" при условии, пакет

располагается в текущем подкаталоге или в подкаталоге, определенном в **\$Path**. Ниже даны исходный код процедуры и пример ее применения для пакета "MyFunctions".

```
In[942]:= PackNames[x_] := Module[{a, b, c, d, h, p, R, St}, {a, R} = {FindFile[x], {}};
      If[a === $Failed, Return["File with package does not exist in $Path"],
        If[StringTake[a, {-2, -1}] != ".m",
          Return["File with package has mx-format"], b = OpenRead[a]]];
      Label[St]; d = Read[b, String]; If[d === EndOfFile, Close[b];
        Return[Map[ToExpression, DeleteDuplicates[R]]],
        c = StringPosition[d, "::usage = "];
      If[c == {}, Goto[St], c = StringReplace[StringTake[d, {1, First[First[c]] - 1}], "/" -> ""]];
      d = StringLength[c]; If[EvenQ[d], p = StringTake[c, d/2]; h = StringTake[c, -d/2], Null];
      If[p == h, R = Append[R, p], Null]; Goto[St]]

In[943]:= PackNames["MyFunctions`"]
Out[943]= "File with package does not exist in $Path"
In[944]:= UpdatePath["D:\\AVZ_Package"]
In[945]:= PackNames["MyFunctions`"]
Out[945]= "File with package has mx-format"
In[946]:= PackNames["MyFunctions`"]
Out[946]= {Rename, ActRemObj, FileQ, ProcQ, SubsDel, ProcsAct, ArgsProc, HeadPF,
  StrOfSymblQ, MemberT, IsFile, LocObj, Subs, SymbolQ, DefFunc3, MemberQ1,
  SubProcs, EmptyFileQ, TypeActObj, ProcCalls, Nvalue, SubsProcQ, ExprOfStr,
  RemProcOnHead, Arity, ListStrToStr, Ind, Int, Op, GC, Mapp, Df, MinusList1,
  UprocQ, Uprocs, ArgsLocals, Sequences, ProcQ1, BitSet1, NbCallProc,
  TwoHandQ, BitGet1, DuplicateLocalsQ, Tuples1}

In[542]:= UpdatePath[x_ /; StringQ[x] || ListQ[x] && DeleteDuplicates[Map[StringQ, x]]
      == {True}] := Module[{}, ClearAttributes[$Path, Protected];
      $Path = DeleteDuplicates[Flatten[Append[$Path, x]]];
      SetAttributes[$Path, Protected]]

In[543]:= UpdatePath["D:\\AVZ_Package"]
In[544]:= UpdatePath[{"D:\\AVZ_Package", "C:/Temp"}]
In[545]:= $Path
Out[545]= {"C:\\Program Files\\Wolfram Research\\Mathematica\\8.0\\SystemFiles\\
  Links", "C:\\Documents and Settings\\Aladjev\\Application Data\\
  Mathematica\\Kernel", ....., "D:\\AVZ_Package", "C:/Temp"}
```

Во фрагменте первый вызов процедуры **PackNames["MyFunctions`"]** завершается без результата с возвратом сообщения об отсутствии файлов форматов  $\{m | mx\}$  с пакетом в каталогах, список которых определяется переменной **\$Path**. Поэтому последующий вызов процедуры **UpdatePath["D:/AVZ\_Package"]** расширяет список **\$Path** на каталог "D:\\AVZ\_Package", содержащий указанные файлы. Как уже отмечалось выше, при наличии файлов двух форматов *первым* используется именно *mx*-файл, как наиболее оптимальный для данной *операционной среды*. Поэтому повторный вызов **PackNames**

также завершается без результата с возвратом сообщения об обнаружении *mx*-файла с пакетом. И только после удаления из каталога данного *mx*-файла вызов **PackNames** возвращает список имен объектов, сохраненных в пакете "MyFunctions", находящемся в файле *m*-формата. Завершает фрагмент исходный код процедуры, успешный вызов **UpdatePath[x]** которой возвращает *Null*, т.е. ничего, расширяя список **\$Path** каталогов на каталог либо список каталогов *x*.

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[24]:= ExtrPackName[F_ /; StringQ[F], N_ /; StringQ[N]] := Module[{a, b, c, d, Art, Kr},
    If[FileExistsQ[F] && FileExtension[F] == "m", a = OpenRead[F], $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[25]:= ExtrPackName["D:\\AVZ_Package\\AVZ_Package.m", "Df"]
In[26]:= ExtrPackName["D:\\AVZ_Package\\AVZ_Package.m", "Subs"]
In[27]:= ExtrPackName["D:\\AVZ_Package\\AVZ_Package.m", "HowAct"]
In[28]:= ExtrPackName["D:\\AVZ_Package\\AVZ_Package.m", "ArtKr"]
Out[28]= "Definition of ArtKr is absent in file <D:\\AVZ_Package\\AVZ_Package.m>"
In[29]:= Df[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2]
Out[29]= x^2 (- (-1 + x^2) Cos[1/x^2] - (1 + x^2) Sin[1/x^2])
In[30]:= Subs[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2, h]
Out[30]= (Cos[h] + Sin[h])/h
```

Данная процедура обеспечивает активацию в текущем сеансе конкретной функции либо процедуры, находящейся в *m*-файле, без загрузки такого файла полностью. Но, с другой стороны, определение загруженной функции/процедуры может содержать и другие пользовательские средства, находящиеся либо в том же самом пакете, что и загруженная функция/процедура, либо в других пакетах пользователя. Поэтому для последующего корректного использования такого *загруженного* средства необходимо дозагружать и все содержащиеся в нем средства пользователя, не активизированные в текущем сеансе. В помощь решению данной задачи предлагается (*полезная также и в*

качестве самостоятельного средства) процедура **CallsInProc1**, успешный вызов которой **CallsInProc1[x, y]** обеспечивает возврат вложенного списка имен в строчном формате как системных средств (1-й подписок), так и пользовательских средств (2-й подписок), содержащихся в процедуре *x*, тогда как через второй аргумент *y* возвращается список контекстов, соответствующих пользовательским средствам. Исходный код процедуры **CallsInProc1** с примерами ее использования представлены следующим фрагментом.

```
In[101]:= CallsInProc1[x_ /; ProcQ[x], y_ /; SymbolQ[y]] := Module[{b, c = {}, h, k = 1, p = {},
    a = DefFunc3[x][[-1]], t = ArgsLocals[x][[2]]},
    b = DeleteDuplicates[Flatten[StringPosition[a, "["]]];
    For[k, k <= Length[b], k++, h = b[[k]]; c = Append[c, ExtrName[a, h, -1]]];
    If[t == {}, Null, For[k = 1, k <= Length[t], k++, h = t[[k]];
        p = Append[p, If[StringFreeQ[h, " = "], h,
            StringTake[h, {1, Flatten[StringPosition[h, " = "][[1]] - 1]}]]];
    t = Select[DeleteDuplicates[c], ! MemberQ[Flatten[{System, Symbol, p}], Head1[#]] &&
        # != "" && # != ToString[x] &];
    ToExpression[ToString[y] <> " = " <> ToString1[Map[Context, t]]];
    Map[Sort, {Select[DeleteDuplicates[c], SystemQ[#]] &&
        ToString[Definition[#]] != "Null" &&
        ! MemberQ[{"Block", "Module", "DynamicModule"}, #] &}, t]]

In[102]:= Clear[Con]; CallsInProc1[ArgsLocals, Con]
Out[102]= {"Flatten", "Head", "If", "MemberQ", "Return", "StringJoin", "StringLength",
    "StringSplit", "StringTake", "ToExpression", "ToString",
    {"HeadPF", "Locals", "SubsDel", "SystemQ"}}

In[103]:= Con
Out[103]= {"AladjevProceduresAndFunctions`", "AladjevProceduresAndFunctions`",
    "AladjevProceduresAndFunctions`", "AladjevProceduresAndFunctions`"}

In[104]:= Clear[Con]; CallsInProc1[ProcQ, Con]
Out[104]= {"Definition", "First", "If", "InputForm", "Quiet", "StringJoin", "StringPosition",
    "StringTake", "ToString"}, {"HeadPF", "SubsDel"}}

In[105]:= Con
Out[105]= {"AladjevProceduresAndFunctions`", "AladjevProceduresAndFunctions`"}
```

Наряду с отмеченным назначением процедура **CallsInProc1** представляет достаточно существенный интерес при анализе пользовательских процедур/функций на предмет наличия в них вызовов как системных, так и пользовательских программных средств.

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 либо препроцессора *Mathematica*.

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 8.0.4.0* 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\Aladjev\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\Aladjev\Application Data\Mathematica\FrontEnd"
```

Для удобства автоматической загрузки пакета пользователя при запуске *Mathematica* можно воспользоваться простой процедурой **AutoLoadPack[x, y]**, которая производит загрузку пакета *x*, находящегося в произвольном подкаталоге файловой системы ПК, в текущий сеанс. Исходный код процедуры с примером ее применения даны ниже.

```
In[1]:= AutoLoadPack[x_, y_] := Module[{}, SetDirectory[y]; Quiet[Needs[x];  
ResetDirectory[]; Null]  
In[2]:= AutoLoadPack["MyFunctions`, "D:\\AVZ_Package"]  
In[3]:= Map[ProcQ, {ProcQ, Spos, BlockQ, FunctionQ, Subs, StringEnd, Locals, SortNL,  
Mapp, NamesProc, Df, Uprocs, FunCompose, UprocQ}]  
Out[3]= {True, True, True, True, True, True, True, True, True, True, True, True, True, True}
```

Данная процедура помещается в инициализационный файл, например, в «*init.m*» из подкаталога, предназначенного для инициализации ядра для текущего пользователя

"C:\\Documents and Settings\\Aladjev\\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 - 29.05.2012 **)
AutoLoadPack[x_, y_] := Module[{}, SetDirectory[y]; Quiet[Needs[x]]; ResetDirectory[]; Null]
AutoLoadPack["MyFunctions`, "D:\\AVZ_Package"]
```

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.

Следующий фрагмент представляет результат *тестирования* успешности загрузки в текущий сеанс (*сразу после запуска Mathematica*) определений функций и процедур из пакета **"MyFunctions"**, 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, FunctionQ, Subs, StringEnd, Locals, SortNL,
      NamesProc, Mapp, Df, Uprocs, FunCompose, UprocQ}]
Out[2]= {True, True, True, True, True, True, True, True, True, False, True, True, True, True}
In[3]:= Directory[]
Out[3]= "C:\\Documents and Settings\\Aladjev\\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;
- выполняет команды, указанные в опциях *-run*, передаваемых ядру пакета;
- 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»*;
- начинает выполнение своего основного цикла.

В деталях с вопросами использования файлов *инициализации* возможно ознакомиться по справочной системе *Mathematica* либо в соответствующей литературе по пакету.

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 "MyFunctions`" 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 - 29.05.2012 **)
UpdatePath[x_/: StringQ[x] || ListQ[x] && DeleteDuplicates[Map[StringQ, x]]
      == {True}] := Module[{}, ClearAttributes[$Path, Protected];
      $Path = DeleteDuplicates[Flatten[Append[$Path, x]]];
      SetAttributes[$Path, Protected]]
UpdatePath[{"D:\\AVZ_Package", "D:\\UserLib6789", "D:\\New_Book"}]
<< MyFunctions`
=====
In[1]:= Map[ProcQ, {ProcQ, Spos, BlockQ, FunctionQ, Subs, StringEnd, SortNL, Locals,
      NamesProc, Mapp, Df, Uprocs, FunCompose, UprocQ}]
Out[1]= {True, True, True, True, True, True, True, True, True, False, True, True, True, True}
In[2]:= a = Length[$Path]; {$Path[[a - 2]], $Path[[a - 1]], $Path[[a]]}
Out[2]= {"D:\\AVZ_Package", "D:\\UserLib6789", "D:\\New_Book"}
In[3]:= Map[UprocQ, {ProcQ, Uprocs, FunCompose, UprocQ}]
Out[3]= {{True, Module}, {True, Module}, {True, Module}, {True, Module}}
In[4]:= $Packages
Out[4]= {"ResourceLocator`, "DocumentationSearch`, "GetFEKernelInit`, "Jlink`,
      "PacletManager`, "WebServices`, "System`, "Global`"}
```

При этом, следует отметить, что автоматически загруженный в процессе запуска ядра контекст пакета "MyFunctions`, не нашел отражения в переменной **\$Packages**. Эту функцию обеспечивает процедура **UpdatePackages**, представленная фрагментом:

```
In[337]:= UpdatePackages[x_] := Module[{}, ClearAttributes[$Packages, Protected];
      $Packages = DeleteDuplicates[Flatten[ReleaseHold[Append[$Packages, x]]]];
      SetAttributes[$Packages, Protected]; Null]
In[338]:= UpdatePackages["MyFunctions`"]
In[339]:= $Packages
Out[339]= {"ResourceLocator`, "DocumentationSearch`, "GetFEKernelInit`, "Jlink`,
      "PacletManager`, "WebServices`, "System`, "Global`, "MyFunctions`"}
In[340]:= Attributes[$Packages]
Out[340]= {Protected}
```

```
In[341]:= UpdateContextPaths[x_] := Module[{}, $ContextPath =
DeleteDuplicates[Flatten[ReleaseHold[Append[$ContextPath, x]]]]; Null]
In[342]:= UpdateContextPaths["MyFunctions`"]; $ContextPath
Out[342]= {"PacletManager`", "WebServices`", "System`", "Global`", "MyFunctions`"}
```

Данная процедура вполне может сочетаться с ранее представленными средствами по работе с пакетами пользователя, упрощая их программирование и применение.

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 подписка 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 *Mathematica* активизированы.

```
In[2213]:= NamesCS[P_;/! HowAct[P], Pr_;/! HowAct[Pr], Pobj_;/! HowAct[Pobj]] :=
Module[{a = 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]= {"AladjevProceduresAndFunctions`", "ResourceLocator`", "GetFEKernelInit`",
"DocumentationSearch`", "Jlink`", "PacletManager`", "WebServices`",
"System`", "Global`"}, {AvzAgn, Svetla, NamesCS, ProcName, Art23Kr15,
NoosphereAcademy, SveGal, GrSU, VgTU, Ranslan}}
```

```
In[2216]:= Pobj
```

```
Out[2216]= {"AladjevProceduresAndFunctions`", ActiveProcess, ActRemObj, Affiliate,
AllMatrices, AllMatrices1, Args, Args0, Args1, Args2, ArgsLocals, ArgsProc,
```

Arity, ArrayInd, AssignToList, AtomicQ, Attrib, BinaryListQ, BitGet1, Bits, BitSet1, Border, CALL, CallsInProc, CallsInProc1, CatN, CharacterQ, Contexts1, CopyDir, DeCod, Decomp, Defaults, DefFunc, DefFunc1, DefFunc2, DefFunc3, DelEl, DelSubStr, DirName, DO, DuplicateLocalsQ, Email, Email1, ExpFunc, EmptyFileQ, ExpFunc1, ExpFunc2, ExpLocals, ExprOfStr, ExprQ, ExtrCall, ExtrExpr, ExtrName, ExtrPackName, FileFormat1, FileOpenQ, FileQ, FreeSpaceVol, FunCompose, Gather1, Gather2, GC, Globals, Globals1, GotoLabel, Head1, HeadingQ, HeadPF, Help, HowAct, Iff, Ifk, Ifk1, Ind, Index, IndexedQ, InsertN, IsFileOpen, IsMonotonic, LeftFold, ListAssign, ListOp, ListListGroup, ListListQ, ListStrToStr, ListToSeq, ListToString, LoadFile, LoadNameFromM, Locals1, LocObj, Map1, Map2, Map3, Map4, Map5, Map6, Mapp, MaxNestLevel, MemberLN, MemberQ1, MemberQ2, MemberT, MinusList, MinusList1, NamesCS, NamesN, NamesNbPackage1, NbCallProc, NbName, NestCycles, NestListQ, NestQL, Nobj, Npackage, Nproc, ObjType, Op, OP, OpenFiles, OverLap, PackNames, PartialSums, Predecessors, PredecessorsL, PredecessorsR, PrefixQ, Prev, ProcCalls, ProcQ1, ProcsAct, ProtectedQ, ProtectedQ1, PureFuncQ, Range1, Range2, Range3, ReadFullFile, Rename, ReplaceAll1, RETURN, RhsLhs, RhsLhs1, RightFold, SearchDir, SearchFile, Seq, SeqDel, SeqIns, SeqQ, SeqToList, SeqToList1, SequenceQ, Sequences, SeqUnion, SortLpos, SortNL, SortNL1, SortQ, Spos, SortString, StrDelEnds, StringMultiple, StringMultipleD, StringPosition1, StringReplace1, StringReplace2, StringTake1, StringTake2, StrOfSymblQ, SubDelStr, SubProcs, Subs, SubsProcQ, SubStr, SuffixQ, SuffPref, SymbolQ, SysFuncQ, SysFuncQ1, SystemQ, Tbl, TestArgsTypes, TestOnNCF, TestProcCalls, ToList, ToString1, Try, Tuples1, TwoHandQ, Type, TypeActObj, Un, UnevaluatedQ, UprocQ, Uprocs, UserLib, VarExch, VarExch1, WhatObj, WhatType, WhichN}, {"ResourceLocator", Private}, {"DocumentationSearch", Information, Private}, {"Jlink", Private, CallPrivate, Information, InstallPrivate, vm1}, {"PacletManager", Private, Information}, {"WebServices", Information}, {"System", BesselParamDerivative, ComplexExpand, Commo, Converter, MathM, Tabl, Cros, DiscretePlo, ArgumentCount, GeoLocatio, Environment, ParameterValidation, F, FileExportLis, GroebnerBasi, Inf, InputOutput, IntegerPartition, InterpolatingFunction, LanguageEnhancements, Parallel, LaplaceTransfor, PowerReduc, Private, Serie, Utilities}, {"Global", Private}}

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[1107]:= Npackage[x_;/ StringQ[x]] := If[MemberQ[Contexts1[], x],
Sort[Select[Names[x <> ""], StringTake[#, -1] != "$" &&
```

```
ToString[Definition[#]] != "Null" &]], $Failed]
```

```
In[1108]:= Npackage["AladjevProceduresAndFunctions`"]
```

```
Out[1108]= {"AcNb", "ActiveProcess", "ActRemObj", "Adrive", "Affiliate", "AllMatrices",  
            "AllMatrices1", "Aobj", "Args", "Args0", "Args1", "Args2", "ArgsLocals", .... ,  
            "WhatObj", "WhatType", "WhichN", "$HeadProc", "$Help", "$ProcName"}
```

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[1080]:= 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[1081]:= G = 64; Map[Affiliate, {"ProcQ1", "Sin", "G", "Z", "Affiliate"}]
```

```
Out[1081]= {"AladjevProceduresAndFunctions`", "System`", "Global`", "Undefined",  
            "AladjevProceduresAndFunctions`"}
```

```
In[1082]:= {V, G = 75, 75}; Map[Affiliate, {"V", "G", "75", "Sin[15]", "Q", "Map"}]
```

```
Out[1082]= {"Undefined", "Global`", "Undefined", "Undefined", "Undefined", "System`"}
```

Достаточно интересной представляется и процедура, вызов которой **ContOfContext[x]** возвращает вложенный 2-элементный список, первый элемент которого определяет подсписок *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 подсписок *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 пример ее применения относительно контекста "AladjevProceduresAndFunctions`". В завершение вычисляется *длина* обоих подсписков возвращаемого результата наряду с выборочной проверкой по **Definition** определений средств из обоих подсписков.

```
In[1038]:= ContOfContext[x_/, StringQ[x] && StringTake[x, -1] == "" ] :=  
            Module[{a = Contexts[], b, c},  
            If[! MemberQ[a, x], $Failed, b = Select[a, SuffPref[#, x, 1] &];  
            b = Select[Mapp[StringReplace, b, {x -> "", "" -> ""}], # != "" &];  
            c = Select[Names[x <> ""], Context[#] == x && ToString[Definition[#]] != "Null" &];  
            {b, Select[MinusList[c, b], ToString[Head1[#]] != "System" &]}
```

```
In[1039]:= ContOfContext["AladjevProceduresAndFunctions`"]
```

```
Out[1039]= {"ActiveProcess", "ActRemObj", "Affiliate", "AllMatrices", "AllMatrices1",
```

```

"Args", "Args0", "Args1", "Args2", "ArgsLocals", "ArgsProc", "Arity", ...,
"Un", "UnevaluatedQ", "UprocQ", "Uprocs", "UserLib", "VarExch",
"VarExch1", "WhatObj", "WhatType", "WhichN", "$ProcName",
{"AcNb", "Adrive", "Aobj", "AutoLoadPack", "Avg", "BlockQ", "Cdir",
"CloseAll", "Closes", "ComplexQ", "Df", "Df1", "Df2", "DirQ", "FunctionQ", ...,
"UpdatePackages", "UpdatePath", "Ver", "$HeadProc", "$Help"}}
In[1040]:= Map[Length, %]
Out[1040]= {202, 56}
In[1056]:= Definition["DirName"]
Out[1056]= DirName[AladjevProceduresAndFunctions`DirName`F_String] :=
    If[DirQ[AladjevProceduresAndFunctions`DirName`F], None,
    If[! FileExistsQ[AladjevProceduresAndFunctions`DirName`F], $Failed,
    Quiet[Check[FileNameJoin[FileNameSplit[
        AladjevProceduresAndFunctions`DirName`F][[1; -2]]], None]]]]
In[1057]:= Definition["StrStr"]
Out[1057]= StrStr[x_] := If[StringQ[x], StringJoin["\""], x, "\""], ToString[x]]

```

Во фрагменте представлено использование процедуры **ContOfContex** для контекста вполне реального *nb*-документа, загруженного и вычисленного в текущем сеансе. В качестве одного из возможных приложений этой процедуры можно отметить задачи, имеющие дело с исходными кодами программных средств пакетов пользователя.

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 пакета. Таким образом, пользователь получает вполне реальную возможность довольно эффективно проводить разработку *своих* собственных средств, включая средства, расширяющие возможности самого пакета. Действительно, на базе целого ряда таких глобальных переменных и ряда довольно развитых средств можно разрабатывать оригинальные средства, разработка аналогов которых в среде пакета *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 пакетов представляет задача определения *имени* текущего документа {*mws-файла*, *nb-файла*}. В среде *Maple* нами для такой цели была создана процедура *mwsname*, разработка которой потребовала нестандартного подхода. Тогда как разработка подобного средства для *Mathematica* оказалась намного более простой, не потребовав какого-либо *нестандартного* подхода, что прекрасно иллюстрирует следующая достаточно простая процедура **NbName**.

```

In[1942]:= NbName[] := Module[{a, b, c, d, k}, {a, d} = {ToString[Notebooks[]], {}};
    {b, c} = {StringPosition[a, "<<"], StringPosition[a, ">>"]};
    For[k = 1, k <= Length[b], k++, d = Append[d, StringTake[a, {b[[k]][[2]] + 1,

```

```
c[[k]][[1]] - 1]]]; Select[d, StringEnd[#, ".nb"] &]]
```

```
In[1943]:= NbName[]
```

```
Out[1943]= {"Package_67.nb", "AVZ_Package.nb", "Ver.nb", "Files$Directories.nb"}
```

```
In[2487]:= NbName[]
```

```
Out[2487]= {"SearchFile.nb", "LoadFile.nb", "ActiveProcess.nb", "Integral.nb", "Ver.nb"}
```

```
In[2493]:= AcNb[] := StringSplit[NotebookFileName[], {"\\", "/" }][[-1]]
```

```
In[2494]:= AcNb[]
```

```
Out[1494]= "Package_67.nb"
```

Вызов `NbName[]` процедуры 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 `AcNb[]` простой функции возвращает имя текущего документа, ранее сохраненного в файле *nb*-формата. Процедура `NamesNbPackage[F]` возвращает список *имен* в строчном формате всех средств, находящихся в файле *F nb*-формата с пакетом, снабженных краткими справками «*usages*» по использованию в среде пакета *Mathematica*. Следующий фрагмент представляет исходный код `NamesNbPackage` с примером ее применения к *nb*-файлу "D:\\Math\_myLib\\AVZ\_Package.nb". Тогда как вызов процедуры `NamesNbPackage1[F]` – довольно эффективной модификации предыдущей процедуры – возвращает аналогичный список *имен* в строчном формате всех средств, находящихся в файле *F nb*-формата с пакетом; при этом, полагается, что все средства снабжены краткими справками (*usages*) по использованию в среде пакета *Mathematica*, в отсутствие таких средств возвращается пустой список, т.е. {}.

```
In[1041]:= NamesNbPackage[F_;/; IsFile[F] && StringTake[F, -3] == ".nb"] :=
```

```
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[1042]:= NamesNbPackage["D:\\Math_myLib\\AVZ_Package.nb"]
```

```
Out[1042]= {"AcNb", "ActiveProcess", "ActRemObj", "Adrive", "Affiliate", "AllMatrices",
  "AllMatrices1", "Aobj", "Args", "Args0", "Args1", "Args2", "ArgsLocals", ...,
  "WhatObj", "WhatType", "WhichN", "$HeadProc", "$Help", "$ProcName"}
```

```
In[1088]:= NamesNbPackage1[F_;/; IsFile[F] && StringTake[F, -3] == ".nb"] :=
```

```
Module[{a = OpenRead[F], c, d, g = "\": \"", 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[1088]= NamesNbPackage1["D:\\AVZ_Package\\AVZ_Package.nb"]
Out[1088]= {"AcNb", "ActiveProcess", "ActRemObj", "Adrive", "Affiliate", "AllMatrices",
            "AllMatrices1", "Aobj", "Args", "Args0", "Args1", "Args2", "ArgsLocals", ...,
            "WhatObj", "WhatType", "WhichN", "$HeadProc", "$Help", "$ProcName"}
In[1089]:= Length[%]
Out[1089]= 254

```

Следующая процедура является аналогом предыдущих процедур **NamesNbPackage** и **NamesNbPackage1** для случая *пользовательских* пакетов *m*-формата. 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 ее применения.

```

In[2133]:= NamesMPackage[F_;/; IsFile[F] && StringTake[F, -2] == ".m"] :=
            Module[{a = OpenRead[F], c, d, s = "":usage="", Res = {}},
            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[2134]:= NamesMPackage["D:\\AVZ_Package\\AVZ_Package.m"]
Out[2134]= {"AcNb", "ActiveProcess", "ActRemObj", "Adrive", "Affiliate", "AllMatrices",
            "AllMatrices1", "Aobj", "Args", "Args0", "Args1", "Args2", "ArgsLocals", ...,
            "WhatObj", "WhatType", "WhichN", "$HeadProc", "$Help", "$ProcName"}
In[2135]:= Length[%]
Out[2135]= 259

```

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. Фрагмент содержит исходный код функции **ContextQ** с типичными примерами ее применения.

```

In[1370]:= ContextQ[x_] := StringQ[x] && StringLength[x] > 1 &&
            Quiet[SymbolQ[Symbol[StringTake[x, {1, -2}]]]] && StringTake[x, {-1, -1}] == ""
In[1371]:= ContextQ["AladjevProceduresAndFunctions`"]
Out[1371]= True
In[1372]:= ContextQ["75AladjevProceduresAndFunctions`"]
Out[1372]= False

```

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** $[x, y, \dots]$  с *необязательными* аргументами  $\{y, \dots\}$ , в качестве которых выступают имена в строчном формате устройств внешней памяти прямого доступа, обеспечивают поиск искоемых 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[1384]:= FindFileContext[x_;/ ContextQ[x], y___] := Module[{b = {}, c = "", s = {}, d, k = 1,
    a = If[{y} == {}, Adrive[], {y}], f = "Art23Kr16.txt",
    h = "(*BeginPackage[" <> ToString1[x] <> "*)"},
    While[k <= Length[a], Run["Dir ", a[[k]] <> ":\\", "/B/S/L > Art23Kr16.txt"];
    While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]];
    If[StringTake[c, {-2, -1}] == ".m", b = Append[b, c]];
    Continue[]]; Quiet[Close[f]]; c = ""; k++; k = 1;
    While[k <= Length[b], If[Select[ReadList[b[[k]]], String], ! StringFreeQ[#, h] &] != {},
    s = Append[s, b[[k]]]; k++; {DeleteFile[f], s}][[2]]

In[1385]:= FindFileContext["AladjevProceduresAndFunctions`", "D"]
Out[1385]= {"d:\\avz_package\\aladjevproceduresandfunctions.m",
    "d:\\avz_package\\avz_package.m", "d:\\avz_package\\rans.m"}
In[1386]:= FindFileContext["AladjevProceduresAndFunctions`"]
Out[1386]= {"d:\\avz_package\\aladjevproceduresandfunctions.m",
    "d:\\avz_package\\avz_package.m", "d:\\avz_package\\rans.m"}
In[1387]:= FindFileContext["AvzAgnSvetArtKr`", "D"]
Out[1387]= {}
```

Следует иметь в виду, что поиск в рамках всей файловой системы компьютера может потребовать достаточно существенных временных затрат.

Итак, вообще говоря, даже уже представленной информации вполне достаточно не только для понимания принципиальной организации пакетов в среде *Mathematica*, но даже для написания собственных относительно несложных пакетов, содержащих определения процедур и/или функций. В данном контексте, если *блокноты (notebooks)* с полным основанием можно отождествлять с документами пакета *Maple* ( $\{mws | mw\}$ -*файлами*), пакеты могут ассоциироваться с пакетными модулями *Maple*. Однако, если пакетные модули в организационном плане размещаются в библиотеках, структурно аналогичных главной *Maple*-библиотеке, обеспечивая весьма высокую унификацию *пакетных* и *пользовательских* средств, то пакеты *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.

## 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/I*, *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 круг пользователей математических приложений пакета. 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 пакета *Mathematica*, вполне может обратиться к документации, поставляемой с пакетом. Нами с целью развития способов доступа к файловой системе компьютера создан ряд достаточно эффективных средств, которые представлены в пакете [90]. В настоящей главе акцентируется внимание на средствах, расширяющих стандартные средства пакета *Mathematica* для обеспечения работы с файловой компьютером. Ряд из них весьма полезны для практического применения.

### 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 пакет имеет дело с 3-мя различными типами *внутренних* файлов, из которых отметим файлы, имеющие расширения {"*.nd*", "*.m*", "*.mx*"}, структура которых распознается пакетом *его* стандартными средствами и которые достаточно важны уже на первых этапах работы с пакетом. 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 упоминавшегося пакета *Maple*, за исключением того, что если в *Maple* для *СФ* допускается формат {*string* | *symbol*}-типа, то для *СФ* в пакете *Mathematica* только формат *string*-типа является допустимым.

По функции **Directory[]** возвращается *активный* подкаталог текущего сеанса пакета, 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** пакета, 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 функция **SetDir[w]** обеспечивает возможность определять в

качестве текущих также и *несуществующие* подкаталоги *w*. Вызов функции **SetDir[w]** 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[381]:= Directory[]
Out[381]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[382]:= SetDirectory["D:\\Math_myLib"]
Out[382]= "D:\\Math_myLib"
In[383]:= SetDirectory[$UserDocumentsDirectory]
Out[383]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[384]:= SetDirectory[]
Out[384]= "C:\\Documents and Settings\\Aladjev"
In[385]:= SetDirectory["D:\\Math_myLib\\rans"]
      SetDirectory::cdir: Cannot set current directory to D:/Math_myLib/rans. >>
Out[385]= $Failed

In[86]:= 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.

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 в наш пакет, готовящейся в настоящее время. При этом, файлы любого из указанных форматов с сохраненными в них по функции *Save* определениями объектов в результате загрузки в последующих сеансах таких файлов по функции *Get* (<<) обеспечивают доступность данных объектов, что иллюстрирует достаточно наглядно следующий весьма простой фрагмент, а именно:

```
In[2221]:= Art1 := #^2 &; Art2 := #^3 &; Art3 := #^4 &; Art4 := #^5 &
In[2222]:= Save["D:/avz_package\\Art1", Art1]; Save["D:/avz_package\\Art2.m", Art2];
           Save["D:/avz_package/Art3.mx", Art3]; Save["D:/avz_package/Art4.nb", Art4]
           ===== Новый текущий сеанс =====
In[821]:= SetDirectory["D:\\AVZ_Package"]
Out[821]= "D:\\avz_package"
In[822]:= << "Art1"; << "Art2.m"; << "Art3.mx"; << "Art4.nb"
In[823]:= {Art1[69], Art2[69], Art3[69], Art4[69]}
Out[823]= {4761, 328509, 22667121, 1564031349}
```

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 типичными примерами их использования.

```
In[2120]:= Art1 := #^2 &; Art2 = #^3 &; Art3 := #^4 &; Art4 = #^5 &; Kr = 15;
In[2121]:= Save["d:/avz_package/Obj.m", {Adrive, SetDir, Art1, Art2, Art3, Art4, Nobj, Kr}]
In[2241]:= Nobj[x_;/ FileExistsQ[x] && StringTake[x, -2] == ".m", y_;/! HowAct[y]] :=
Module[{a, b, c, d, p, h, t}, If[FileExistsQ[x] &&
MemberQ[{"Table", "Package"}, Quiet[FileFormat[x]]],
{a, b, d, h} = {OpenRead[x], {}, "69", {}};
Label[c]; d = Read[a, String]; If[d === EndOfFile, Close[a];
Return[{Quiet[ToExpression[ToString[y] <> " = " <> ToString1[h] <> ";"], b] [[2]]],
If[SuffPref[d, " ", 1] || ! StringFreeQ[d, "::usage"], Goto[c, Null]]];
t = Flatten[StringPosition[d, {" := ", " = " }]];
If[t == {}, Goto[c], h = Append[h, StringTake[d, t[[1]] - 1]]];
p = StringTake[d, t[[1]] - 1];
b = Append[b, If[StringFreeQ[p, "["], p,
StringTake[p, Flatten[StringPosition[p, "["] [[1]] - 1]]]; Goto[c, $Failed]]
In[2242]:= Clear[y]; Nobj["D:\\AVZ_Package\\Obj.m", ArtKr]
Out[2242]= {"Adrive", "SetDir", "SetDir", "Art1", "Art2", "Art3", "Art4", "Nobj", "Kr"}
In[2243]:= ArtKr
Out[2243]= {"Adrive[]", "SetDir[x_String]", "SetDir[x_;/ StringQ[x]]", "Art1", "Art2", "Art3",
"Art4", "Nobj[x_String]", "Kr"}
In[1114]:= Aobj[x_;/ FileExistsQ[x] && StringTake[x, -2] == ".m", y_;/ StringQ[y]] :=
Module[{a, b = "(*End[*)", c = "(*EndPackage[*)", d = "",
p = OpenRead[x], h, l1, l2, t = {}},
If[Read[p, String] != "(* ::Package:: *)", Close[p];
Return[$Failed], a = "(*Begin[\\ "" <> y <> \\ ""*)"];
Label[l1]; h = Read[p, String]; If[h == c, Close[p];
Return[t = Map[ToExpression, t]; If[t == {}, t, t[[1]]],
If[h != a, Goto[l1], Label[l2]; h = Read[p, String];
If[h != b, d = d <> h; Goto[l2], t = Append[t, d]; d = ""; Goto[l1]]];
For[d = 1, d <= Length[t], d++, t[[d]] = StringTake[t[[d]], {3, -3}]]]]
In[1115]:= Aobj["D:\\AVZ_Package\\RANS.m", "Mapp"]
In[1116]:= Aobj["D:\\AVZ_Package\\RANS.m", "Rans"]
Out[1116]= {}
```

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. 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 процедуры *Adrive*, находящейся в ранее созданном файле. Проверка подтверждает доступность указанной процедуры в текущем сеансе. Процедуры *Nobj* и *Aobj* обрабатывают основные ошибочные ситуации, возвращая на них **\$Failed**. При этом, обе процедуры могут быть расширены путем загрузки их новыми функциями. Имеется целый ряд других интересных процедур для обеспечения работы с файлами входного формата *Mathematica*, чьи имена имеют расширения {".nb", ".m", ".txt"} и др. Все они базируются на основе анализа *внутренней* структуры файлов такого формата. Ряд из них позволяет создавать достаточно эффективные библиотеки пользователя, содержащие определения *Mathematica*-объектов. Эти и другие средства оформлены специальной библиотекой, поддерживающей релизы 4 – 8 пакета *Mathematica*.

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. Такую задачу решает процедура **Save1**, чей исходный код представлен следующим фрагментом.

```
In[10]:= A[x_] := x^2; A[x_, y_] := x + y; A[x_, y_, z_] := x + y + z; A[x_] := {x}; DefFunc3[A]
Out[10]= {"A[x_] := x^2", "A[x_, y_] := x + y", "A[x_, y_, z_] := x + y + z", "A[x_] := {x}"}
In[11]:= Save1[x_/, StringQ[x], y_/, DeleteDuplicates[Map[StringQ, Flatten[{y}]]][[1]] :=
Module[{Rs, t = Flatten[{y}], k = 1},
Rs[n_, m_] := Module[{b, c = "$Art23$$$Kr15$",
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[12]:= Save1["D:\\AVZ_Package\\rans_ian.m", {"A[x_, y_, z_]", "A[x_]"}]
In[13]:= Clear[A]; DefFunc3[A]
Out[13]= DefFunc3[A]
```

```

In[14]:= << "D:\AVZ_Package\rans_ian.m"
In[15]:= DefFunc3[A]
Out[15]= {"A[x_, y_, z_] := x + y + z", "A[x_] := {x}"}
In[16]:= Agn = 64; Save1["C:/Temp/Herson.m", {"A[x_, y_, z_]", "B", "A[x_]"}, "Agn"]
In[17]:= Clear[A, B, Agn]; Map[DefFunc3, {A, B, Agn}]
In[18]:= {DefFunc3[A], DefFunc3[B], Agn}
Out[18]= {DefFunc3[A], DefFunc3[B], Agn}
In[19]:= << "C:\\Temp\\Herson.m"
Out[19]= 64
In[20]:= DefFunc3[A]
Out[20]= {"A[x_, y_, z_] := x + y + z", "A[x_] := {x}"}
In[21]:= {DefFunc3["B"], Agn}
Out[21]= {"B[x_] := x^2", 64}

```

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 пакета.

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 пакета.

```

In[378]:= DumpSave["D:\\AVZ_Package/Objects.mx", {Adrive, SetDir, Art1, Art2, Art3,
           Art4, Kr, Nobj, Aobj}]
Out[378]= {Adrive, SetDir, #1^2 &, #1^3 &, #1^4 &, #1^5 &, 14, Nobj, Aobj}
===== Новый текущий сеанс =====
In[281]:= Get["D:\\AVZ_Package\\Objects.mx"]

```

```
In[282]:= Nobj["D:\\AVZ_Package\\Objects.m", t]
Out[282]= {"Adrive", "SetDir", "Art1", "Art2", "Art3", "Art4", "Nobj", "Kr"}
```

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.

```
In[1123]:= 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[1124]:= LoadFile["AVZ_Package.m"]
Out[1124]= "File <D:\\Math_myLib\\AVZ_Package.m> has been loaded;
    $Load$Files$ defines the list with full paths to the found files."

In[1125]:= $Load$Files$
Out[1125]= {"C:\\Temp\\AVZ_Package.m", "D:\\AVZ_Package\\AVZ_Package.m"}
In[1126]:= LoadFile["Example70.m"]
Out[1126]= "File <Example70.m> has not been found"
In[1127]:= LoadFile["AVZ_Package69.m"]
Out[1127]= "File <C:\\Temp\\AVZ_Package69.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.

## 9.2. Средства Mathematica для работы с внешними файлами

По такому достаточно важному показателю, как средства доступа, пакет *Mathematica*, на наш взгляд, обладает рядом преимуществ по сравнению с пакетом *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 пакет *Mathematica 8* в общей сложности поддерживает *более 190* различных форматов данных самого различного назначения. Тогда как в качестве базовых форматов следует отметить следующие, а именно:

"Bit" – последовательность битов  
 "BMP" – Microsoft *bmp*-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  
 "Complex64" – комплексные IEEE числа обычной точности  
 "Complex128" – комплексные IEEE числа двойной точности  
 "Complex256" – комплексные IEEE числа учетверенной точности  
 "DBF" – формат файлов базы данных *dBase*  
 "Directory" – иерархия каталогов файловой системы  
 "EPS" – *eps*-формат (*Encapsulated PostScript format*)  
 "GIF" – *gif*-формат  
 "HTML" – {*htm*, *html*}-формат  
 "Integer8" – последовательность 8-битных целых чисел со знаком  
 "Integer16" – последовательность 16-битных целых чисел со знаком  
 "Integer24" – последовательность 24-битных целых чисел со знаком  
 "Integer32" – последовательность 32-битных целых чисел со знаком  
 "Integer64" – последовательность 64-битных целых чисел со знаком  
 "Integer128" – последовательность 128-битных целых чисел со знаком  
 "JPEG" – {*jpeg*, *jpg*}-формат (*JPEG raster image format*)  
 "LaTeX" – *latex*-формат (*LaTeX format*)  
 "Package" – *m*-формат пакетов *Mathematica*  
 "PDF" – *pdf*-формат (*Adobe Acrobat PDF format*)  
 "Real32" – действительные IEEE числа обычной точности  
 "Real64" – действительные IEEE числа двойной точности  
 "Real128" – действительные IEEE числа учетверенной точности  
 "RTF" – *rtf*-формат (*Microsoft Rich Text Format*)  
 "String" – формат строчных данных пакета  
 "Table" – формат произвольных табличных данных

"TeX" - *tex*-формат (*TEX document format*)  
 "Text" - *txt*-формат (*ASCII format*)  
 "TIFF" - *tiff, tif*-формат (*TIFF raster image format*)  
 "UnsignedInteger8" - последовательность 8-битных целых чисел без знака  
 "UnsignedInteger16" - последовательность 16-битных целых чисел без знака  
 "UnsignedInteger24" - последовательность 24-битных целых чисел без знака  
 "UnsignedInteger32" - последовательность 32-битных целых чисел без знака  
 "UnsignedInteger64" - последовательность 64-битных целых чисел без знака  
 "UnsignedInteger128" - последовательность 128-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[1057]:= Map[FileFormat, {"D:/AVZ/AVZ_Package.nb", "D:/AVZ/AVZ_Package.m"}]
Out[1057]= {"NB", "Package"}
In[1058]:= FileFormat["D:\\AVZ_Package\\Art1"]
Out[1058]= "Text"
In[1059]:= FileExtension["D:\\AVZ_Package\\Art1"]
Out[1059]= ""
In[1060]:= FileFormat["Art1"]
FileFormat::nffil: File not found during FileFormat[Art1]. >>
Out[1060]= $Failed
In[1061]:= FileFormat["D:\\AVZ_Package\\AVZ_Package"]
FileFormat::nffil: File not found during
FileFormat[D:\\AVZ_Package\\AVZ_Package]. >>
Out[1061]= $Failed
In[1062]:= Map[FileFormat, {"C:/Temp/AVZ_P", "C:/Temp/AVZ_P1", "C:/Temp/Der"}]
Out[1062]= {"NB", "Package", "XLS"}
In[1063]:= FileFormat["C:\\Temp\\Der.doc"]
Out[1063]= "DOC"
In[1087]:= FileFormat1[x_ /; StringQ[x]] := Module[{a = 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[1088]:= FileFormat1["AVZ_Package"]
Out[1088]= {"C:\\Temp\\AVZ_Package", "NB"}, {"D:\\AVZ_Package", "Directory"},
{"D:\\AVZ_Package\\AVZ_Package", "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 пакету либо в соответствующей литературе [100,104,110,113].

*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 ошибочных сообщений с возвратом значения *Null*, т.е. ничего, или имени или 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 *v/v*>]**

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 применения их и рассмотренных функций доступа к файлам данных.

```
In[42]:= Streams[]
Out[42]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[43]:= S1 = OpenRead["D:/Math_myLib/testfile"]
Out[43]= InputStream["D:/Math_myLib/testfile", 33]
In[44]:= S2 = OpenWrite["D:/Math_myLib/testfile.txt"]
Out[44]= OutputStream["D:/Math_myLib/testfile.txt", 34]
In[45]:= S3 = OpenAppend["D:/Math_myLib/testfile1.txt"]
Out[45]= OutputStream["D:/Math_myLib/testfile1.txt", 35]
In[46]:= Streams[]
Out[46]= {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[47]:= Close[S3]
Out[47]= "D:/Math_myLib/testfile1.txt"
In[48]:= Streams[]
Out[48]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
          InputStream["D:/Math_myLib/testfile", 33],
          OutputStream["D:/Math_myLib/testfile.txt", 34]}
In[49]:= OpenWrite[]
Out[49]= OutputStream["C:\\Documents and Settings\\Aladjev\\Local Settings\\Temp
          \\m-0a9fa8c8-b787-4e47-958b-c392c61f793c", 36]
In[50]:= Close["D:/Math_myLib/testfile1.txt"]
          General::openx: D:/Math_myLib/testfile1.txt is not open. >>
Out[50]= Close["D:/Math_myLib/testfile1.txt"]
In[51]:= Close["D:/Math_myLib/test72.txt"]
          General::openx: D:/Math_myLib/test72.txt is not open. >>
Out[51]= Close["D:/Math_myLib/test72.txt"]
In[52]:= Closes[x_] := Quiet[Check[Close[x], Null]]
In[53]:= Closes["D:/Math_myLib/test72.txt"]
In[54]:= Closes["D:/Math_myLib/testfile1.txt"]
In[55]:= SP = Streams[]
Out[55]= {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[56]:= StreamsU[] := Select[Streams[], !MemberQ[{"[stdout", "[stderr]",
          StringTake[ToString[#1], {13, 19}]]] &]
```

```

In[57]:= StreamsU[]
Out[57]= {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[58]:= Close["D:/Math_myLib/testfile"]
Out[58]= "D:/Math_myLib/testfile"
In[59]:= StreamsU[]
Out[59]= {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 пакета. 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* пакета 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. Типичными установками для **FormatType** являются:

- **Automatic**                    автоматический выбор формата вывода
- **InputForm**                    формат, пригодный для ввода с клавиатуры
- **OutputForm**                    символьный двумерный формат
- **StandardForm**                    стандартный двумерный формат
- **TraditionalForm**                    приближенный к традиционной математической нотации

При этом, по умолчанию вывод выражений в файл выполняется в строчном формате отдельными записями (*строками*). Следующий простой фрагмент довольно наглядно иллюстрирует применение функций **OpenWrite**, **Write**, **StreamsU** и **CloseAll**.

```

In[1048]:= S = OpenWrite["D:/AVZ_Package/file75"]
Out[1048]= OutputStream["D:/AVZ_Package/file75", 24]
In[1049]:= For[k = 1, k <= 25, k++, Write[S, "RANS_" <> ToString[k]]]

```

```

In[1050]:= {Streams[], Close[S], StreamsU[]}
Out[1050]= {{OutputStream["stdout", 1], OutputStream["stderr", 2],
             OutputStream["D:/AVZ_Package/file75", 24]}, "D:/AVZ_Package/file75", {}
In[1051]:= Write["D:/AVZ_Package/file75", "International Academy of Noosphere"]
In[1052]:= StreamsU[]
Out[1052]= {OutputStream["D:/AVZ_Package/file75", 25]}
In[1053]:= Close[OutputStream["D:/AVZ_Package/file75", 25]]
Out[1053]= "D:/AVZ_Package/file75"
In[1054]:= StreamsU[]
Out[1054]= {}
In[1055]:= Write["D:/AVZ_Package/File450", "International Academy of Noosphere"]
In[1056]:= StreamsU[]
Out[1056]= {OutputStream["D:/AVZ_Package/File450", 26]}
In[1057]:= {StreamsU[], Close[OutputStream["D:/AVZ_Package/file450", 26]], StreamsU[]}
Out[1057]= {{OutputStream["D:/AVZ_Package/file450", 26]}, "D:/AVZ_Package/file450", {}
In[1058]:= CloseAll[] := Map[Close, StreamsU[]]
In[1059]:= CloseAll[]
Out[1059]= {}
In[1060]:= For[k = 1, k <= 3, k++, OpenWrite["C:/<" <> ToString[k]]]
In[1061]:= StreamsU[]
Out[1061]= {OutputStream["C:/1", 35], OutputStream["C:/2", 36], OutputStream["C:/3", 37]}
In[1062]:= CloseAll[]
Out[1062]= {"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 пакета. 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, завершённую символом перевода строки и возврата каретки;

**Word** – считывает последовательность символов, завершённую разделителем слов.

Смысл данных типов достаточно прозрачен и может быть детализирован в справке по пакету. Подобно пакету *Maple* пакет **Mathematica** также даёт возможность открывать один и тот же файл на различных 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 довольно простой фрагмент иллюстрирует применение такого 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[7]:= F = "D:\\AVZ_Package\\file75"; {S, S1} = {OpenRead[F],
      OpenRead[If[UpperCaseQ[StringTake[F, 1]], ToLowerCase[F], ToUpperCase[F]]]}
Out[7]= {InputStream["D:/Math_myLib/file75", 44], InputStream["d:/math_mylib/file75", 45]}
In[8]:= R = {}; For[k = 1, k <= 7, k++, R = Append[R, {Read[S], Read[S1]}]]; Flatten[R]
Out[8]= {"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[426]:= Read["D:\\Math_myLib\\file75"]
Out[426]= "RANS_1"
In[427]:= StreamsU[]
Out[427]= {InputStream["D:\\Math_myLib\\file75", 59]}
In[428]:= Read["D:\\Math_myLib\\file450"]
      Read::openx: D:\\Math_myLib\\file450 is not open. >>
Out[428]= Read["D:\\Math_myLib\\file450"]
In[429]:= Read["D:\\Math_myLib/file75", {Byte, Byte, Byte, Byte, Byte, Byte}]
Out[429]= {34, 82, 65, 78, 83, 95}
In[430]:= {h, S} = {"", {}}; While[h != EndOfFile, h = Read["D:\\Math_myLib\\file450",
      Word]; S = Append[S, h]]; CloseAll[]; Select[S, #1 != EndOfFile &]
Out[430]= {"International", "Academy", "of", "Noosphere", "RANS", "Tallinn", "May_2012"}
```

Между тем, следует иметь ввиду, что как и в случае пакета *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 пакету *Maple*, пакет *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** пакета *Mathematica*.

```
In[704]:= R = OpenRead["D:\AVZ_Package\file75"]; {h, S} = {"", {}};
          While[h != EndOfFile, h = Read[R, String]; S = Append[S, h];
          Skip[R, String, 2]]; CloseAll[]; Select[S, #1 != EndOfFile &]
Out[704]= {"RANS_1", "RANS_4", "RANS_7", "RANS_10", "RANS_13", "RANS_16"}
In[705]:= FileOpenQ[F_]; StringQ[F]] := Module[{a = FileType[F], b, d, x = inputstream,
          y = outputstream, c = Map[ToString1, StreamsU[]],
          f = ToLowerCase[StringReplace[F, "\\\" -> "/" ]]},
          If[MemberQ[{Directory, None}, a], Return[$Failed],
          Clear[inputstream, outputstream];
          d = ToExpression[ToLowerCase[
          StringReplace[ToString1[Map[Args2, StreamsU[]]], "\\\" -> "/" ]]];
          a = Select[d, #[[2]] === f &];
          If[a == {}, {inputstream, outputstream} = {x, y}; {}; False,
          {ReplaceAll[a, {inputstream -> "read", outputstream -> "write"}],
          {inputstream, outputstream} = {x, y}][[1]]]]]
```

```

In[706]:= Write["rans.ian"]; Write["C:\\Temp/Summ.doc"]; Write["d:\\Book/Grin.pdf"]
In[707]:= Map[FileOpenQ, {"rans.ian", "D:\\Book\\Grin.pdf", "Tallinn.docx",
                        "C:\\Temp/Summ.doc", "Rans.Ian", "C:\\HTTP.html"}]
Out[707]= {{{"write", "rans.ian", 362}}, {"write", "d:/book/grin.pdf", 364}}, $Failed,
          {"write", "c:/temp/summ.doc", 363}}, {"write", "rans.ian", 362}}, False}
In[708]:= Write["rans.ian"]; Read["Rans.Ian"]; Write["RANS.IAN"];
In[709]:= FileOpenQ["rans.ian"]
Out[709]= {"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. Рекомендуется 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.

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 нашем небольшом пакете **AVZ\_Package** [90].

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[9]:= FileNames["*", "d:/Book"]
Out[9]= {"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 2011 16:00:44"
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[1098]:= 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[1098]= {"KML", "KML"}

In[1143]:= 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[1144]:= Map[FileFormat2, {"C:/", "C:\\", "C:/Temp", "C:\\Temp"}]
Out[1144]= {"Directory", "Directory", "Directory", "Directory"}
In[1145]:= FileFormat2["Obj.m"]
Out[1145]= {"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.

Кроме перечисленных для работы с файлами пакет **Mathematica** располагает целым рядом довольно полезных функций, здесь нами не рассматриваемых, но с которыми заинтересованный читатель сможет ознакомиться в справочной системе пакета либо в соответствующей литературе [100,104,110]. Между тем, здесь вполне уместно сделать одно достаточно существенное замечание. Подавляющее большинство стандартных функций доступа пакета **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 искомым файлам 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>**.

Выше были кратко рассмотрены функции *нижнего* 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 имеющемся у нас релизе **8.0.4.0** пакета *Mathematica* данные функции доступа в полной мере не поддерживаются, о чем свидетельствует простой пример:

```
In[294]:= Export["D:/Math_myLib/Example.txt", "Academy of Noosphere, April 2012"]
```

General::unavail: Export is not available in this version of Mathematica.

```
Out[294]= Export["D:/Math_myLib/Example.txt", "Academy of Noosphere, April 2012"]
```

```
In[295]:= Export["D:\\Aladjev.jpg", <Портрет>]
```

General::unavail: Export is not available in this version of Mathematica.

```
Out[295]= Export["D:\\Aladjev.jpg", <Портрет>]
```

хотя в справке по пакету все выглядит вполне прилично. Но не все примеры справки выполняются успешно в среде упомянутой версии пакета. Функция **Import** – обратная к функции **Export**, поддерживает следующие форматы кодирования, а именно:

**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. данная функция имеет ряд достаточно существенных ограничений, как иллюстрирует следующий достаточно простой пример:

```
In[372]:= Import["D:\\UserLib6789\\CopyRight_Info\\Miscellaneous\\Aladjev1.jpg"]
```

```
Out[372]= <Портрет>
```

```
In[373]:= Import["http://www.aladjev.narod.ru/"]
```

General::unavail: Export is not available in this version of Mathematica.

```
In[374]:= Import["D:/Math_myLib/ArtKr.txt"]
```

```
Out[374]= "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]]"
```

```
In[375]:= ArtKr[42.68, 47.63]
```

```
Out[375]= ArtKr[42.68, 47.63]
```

В частности, по глобальным переменным **\$ImportFormats** и **\$ExportFormats** возможно получать списки форматов текущего релиза пакета, поддерживаемых соответственно функциями **Import** и **Export**. Пакет декларирует **158** форматов для функции **Import** и **134** для функции **Export**, тогда как, например, формат *jpeg (jpg)*, включенный в списки для обеих функций, поддерживается для **Import** и не поддерживается для **Export**, как достаточно наглядно иллюстрируют приведенные выше примеры.

Функция доступа **Get**[*F*] (<< *F*) читает данные из файла *F*, вычисляя все содержащиеся в них выражения и возвращая результат вычисления последнего из этих выражений, т.е. определения данных выражений активизируются в текущем сеансе. Эта функция в этой книге рассматривается неоднократно, поэтому более детального рассмотрения здесь не требует. Обратная для нее функция доступа **Put** (>>) допускает 3 формата:

**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[760]:= F = "D:\Math_myLib\IAN"; {Put[F], StreamsU[], FileByteCount[F]}
Out[760]= {Null, {}, 0}
In[761]:= DeleteFile[F]
In[762]:= {OpenWrite[F], Close[F], StreamsU[], FileByteCount[F]}
Out[762]= {OutputStream["D:\Math_myLib\IAN", 43], "D:\Math_myLib\IAN", {}, 0}
In[763]:= R[x_] := Sin[x]*Cos[x]; Put[R, "D:/Math_myLib/R"]; StreamsU[]
Out[763]= {}
In[764]:= Get["D:/Math_myLib/R"]; {StreamsU[], Definition[R]}
Out[764]= {{}, Definition[R]}
In[765]:= R[19.42]
Out[765]= 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]** активирует 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[247]:= ReadList["D:\Math_myLib\exp.txt", {Word, Number}, 7]
Out[247]= {"rans", 2012}, {"ian", 420}, {"tru", 72}, {"avz", 68}, {"agn", 63}, {"art", 21}, {"kr", 14}
In[248]:= ReadList["D:\Math_myLib\exp.txt", {Word, Number}]
Out[248]= {"rans", 2012}, {"ian", 420}, {"tru", 72}, {"avz", 68}, {"agn", 63}, {"art", 21}, {"kr", 14}
In[249]:= ReadList["D:\Math_myLib\exp.txt", String]
Out[249]= "rans 2012 ian 420 10 2 rags 72 avz 68 agn 63 art 21 kr 14"
In[250]:= StreamsU[]
Out[250]= {}
```

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[345]:= BinaryReadList["D:\\Math_myLib\\expert.txt"]
Out[345]= {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[346]:= StreamsU[]
Out[346]= {}
```

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[4]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\exp.txt"]; If[h ===
                                EndOfFile, S, S = Append[S, h]; Goto[A]]
Out[4]= {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[5]:= {StreamsU[], CloseAll[]}
Out[5]= {{InputStream["D:\\Math_myLib\\exp.txt", 35]}, {"D:\\Math_myLib\\exp.txt"}}
In[6]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\exp.txt", "Integer16"];
                                If[h === EndOfFile, S, S = Append[S, h]; Goto[A]]
Out[6]= {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[7]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\exp.txt", "Integer16"];
                                If[h === EndOfFile, S, S = Append[S, h]; Goto[A]]
In[8]:= For[k = 42, k <= 65, k++, BinaryWrite["D:\\Math_myLib\\w.txt", k];
                                                {StreamsU[], CloseAll[]}]
```

```

Out[8]= {{OutputStream["D:\\Math_myLib\\w.txt", 46]}, {"D:\\Math_myLib\\w.txt"}}
In[9]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\w.txt"]; If[h === EndOfFile,
S, S = Append[S, h]; Goto[A]]
Out[9]= {42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65}
In[10]:= {StreamsU[], CloseAll[]}
Out[10]= {{InputStream["D:\\Math_myLib\\w.txt", 47]}, {"D:\\Math_myLib\\w.txt"}}
In[11]:= BinaryWrite["D:\\Math_myLib\\w.txt", Sin[x]*Cos[x], "Character8"];
{StreamsU[], CloseAll[]}
Out[11]= {{OutputStream["D:\\Math_myLib\\w.txt", 44]}, {"D:\\Math_myLib\\w.txt"}}
In[12]:= Read["D:\\Math_myLib\\w.txt"]
Out[12]= Cos[x] Sin[x]
In[13]:= BinaryWrite["D:\\Math_myLib\\Z.txt", "426847636743624889148922"]
Out[13]= "D:\\Math_myLib\\Z.txt"
In[14]:= {StreamsU[], CloseAll[]}
Out[14]= {{OutputStream["D:\\Math_myLib\\Z.txt", 53]}, {"D:\\Math_myLib\\Z.txt"}}
In[15]:= Read["D:\\Math_myLib\\Z.txt", Record]
Out[15]= "426847636743624889148922"
In[16]:= {StreamsU[], CloseAll[]}
Out[16]= {{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[5]:= FindFile["init.m"]
Out[5]= "C:\Documents and Settings\Aladjev\Application Data\Mathematica\Kernel\F"
In[6]:= FileExistsQ["D:\\Math_myLib\\expert.txt"]
Out[6]= True
In[7]:= FileNameSplit["D:\\Math_myLib\\AVZ_Package.nb"]
Out[7]= {"D:", "Math_myLib", "AVZ_Package.nb"}
In[8]:= FileNameJoin[%]
Out[8]= "D:\\Math_myLib\\AVZ_Package.nb"
In[9]:= FileBaseName["D:\\Math_myLib\\AVZ_Package.nb"]
Out[9]= "AVZ_Package"
In[10]:= ExpandFileName["AVZ_Package.nb"]
Out[10]= "C:\\Documents and Settings\\Aladjev\\My Documents\\AVZ_Package.nb"
In[11]:= FileExistsQ[%]
Out[11]= False
In[12]:= FileNameDepth["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb"]
Out[12]= 5
In[13]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb"]
Out[13]= "AVZ_Package.nb"
In[14]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", 3]
Out[14]= "D:\\Book\\Aladjev"
In[15]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", -2]
Out[15]= "Math_myLib\\AVZ_Package.nb"
In[16]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", {2, 4}]
Out[16]= "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[1072]:= 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]]] <> " = " <> ToString1[a], Null]; True]]
In[1073]:= FileExistsQ1["Rans.ian"]
Out[1073]= False
In[1074]:= FileExistsQ["NewBook.doc"]
Out[1074]= True
In[1075]:= {FileExistsQ1["NewBook.doc", h], h}
Out[1075]= {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 пакет располагает набором средств для работы 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[77]:= DirectoryQ["C:/Documents and Settings/Aladjev/My Documents/My Videos/"]
Out[77]= False
In[78]:= DirQ[d_ /; StringQ[d]] := DirectoryQ[StringReplace[d, "/" -> "\\"])
In[79]:= DirQ["C:/Documents and Settings/Aladjev/My Documents/My Videos/"]
Out[79]= True
In[80]:= DirQ["C:/Documents and Settings/Aladjev/My Documents/My Videos\\"]
Out[80]= 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 пакетной переменной *\$Path*, а именно:

```
In[7]:= Map[DirectoryName, {"D:/Math_myLib/AVZ_Package.nb", "D:\\Math_myLib"}]
Out[7]= {"D:\\Math_myLib\\", "D:\\"}
In[8]:= DirName[F_ /; StringQ[F]] := If[DirQ[F], None, If[! FileExistsQ1[F], $Failed,
Quiet[Check[FileNameJoin[FileNameSplit[F][[1; -2]]], None]]]
In[9]:= Map[DirName, {"D:/Math_myLib/AVZ_Package.nb", "D:\\Math_myLib", "H:/"}]
Out[9]= {"Math_myLib", 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 пакета с возвратом пути к нему:

```
In[9]:= CDir[d_;/ StringQ[d]] := Module[{a = ADrive[]}, Quiet[If[StringTake[d, {2, 2}] == ":",
    If[MemberQ[a, StringTake[d, 1]], CreateDirectory[d,
    CreateDirectory[a[[-1]] <> StringTake[d, {2, -1}]]], CreateDirectory[d]]]]
In[10]:= CreateDirectory["H:\\Temp\\GSV\\ArtKr"]
CreateDirectory::nffil: File not found during CreateDirectory .... >>
Out[10]= $Failed
In[11]:= CDir["H:\\Temp\\GSV\\ArtKr"]
Out[11]= "G:\\Temp\\GSV\\ArtKr"
In[12]:= CDir["G:\\Temp\\AVZ\\Tallinn\\IAN\\Grodno"]
Out[12]= "D:\\Temp\\AVZ\\Tallinn\\IAN\\Grodno"
```

**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, но с тем отличием, что при наличии принимающего каталога каталог **d1** копируется в качестве подкаталога **d2**, например:

```
In[539]:= CopyDirectory["D:/AVZ_Package", "C:/Temp/Temp"]
CopyDirectory::filex: Cannot overwrite existing file C:/Temp/Temp. >>
Out[539]= $Failed
In[540]:= CopyDir[d_;/ StringQ[d], p_;/ StringQ[p]] := CopyDirectory[d, If[DirQ[p],
    p <> "\\ " <> FileNameSplit[p][[-1]], p]]
In[541]:= CopyDir["D:/AVZ_Package", "C:/Temp/Temp"]
Out[541]= "C:\\Temp\\Temp\\Temp"
```

**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 процедура **Attrib**, чей исходный текст с примерами применения дается ниже:

```
In[1224]:= 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 = "ArtKr", b = "attrib ", c, d = " ", h = "attrib.exe", p, f, g = Clear[$Art23$Kr16$]},
    If[StringLength[F] == 3 && DirQ[F], f = F,
    If[FileExistsQ1[StrDelEnds[StringReplace[F, "/" -> "\\ "], "\\ ", 2],
    $Art23$Kr16$], g = $Art23$Kr16$; f = g[[1]];
```

```

Clear[$Art23$Kr16$], Return["<" <> F <> "> is not a directory or a datafile"];
Quiet[If[DeleteDuplicates[Mapp[SuffPref, $Path, h, 2]] == {False}, p = SearchFile[h];
If[p === {}, Return["Utility <Attrib.exe> can not be found"], UpdatePath[p]], Null]];
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]}]]];
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[1225]:= Attrib["dereks.doc", "Attr"]
Out[1225]= {"C:\\Temp\\dereks.doc"}
In[1226]:= Attrib["dereks.doc", {"+A", "+S", "+R"}]
In[1227]:= Attrib["dereks.doc", "Attr"]
Out[1227]= {"A", "S", "R", {"C:\\Temp\\dereks.doc"}}
In[1228]:= Attrib["dereks.doc", {}]
In[1229]:= Attrib["dereks.doc", "Attr"]
Out[1229]= {"C:\\Temp\\dereks.doc"}
In[1230]:= Attrib["dereks.doc", {"+A"}]
In[1231]:= Attrib["dereks.doc", "Attr"]
Out[1231]= {"A", {"C:\\Temp\\dereks.doc"}}
In[1232]:= Attrib["D:\\", "Attr"]
Out[1232]= "Drive D:\\
In[1233]:= Attrib["H:\\", "Attr"]
Out[1233]= "<H:\\> is not a directory or a datafile"
In[1234]:= Attrib["RANS.IAN", "Attr"]
Out[1234]= "<RANS.IAN> is not a directory or a datafile"
In[1235]:= Attrib["dereks.doc", "Attr"]
Out[1235]= {"A", "S", "H", "R", {"C:\\Temp\\dereks.doc"}}
In[1236]:= Attrib["dereks.doc", {"-S", "-R", "-H"}]
In[1237]:= Attrib["dereks.doc", "Attr"]
Out[1237]= {"A", {"C:\\Temp\\dereks.doc"}}
In[1238]:= Attrib["c:/temp\\", "Attr"]
Out[1238]= {"A", {"C:\\Temp"}}
In[1239]:= Attrib["temp\\", "Attr"]
Out[1239]= {"A", {"C:\\Temp", ..., "G:\\Temp"}}

```

Успешный вызов процедуры **Attrib**[*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 **Attrib**[*F*, {}] возвращает **Null**, т.е. *ничего*, отменяя все атрибуты для обрабатываемого файла/каталога *F*, в то время как вызов **Attrib**[*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 **Attrib**[*F*, *x*] возвращает соответствующие *сообщения*. Данная процедура представляется нам достаточно полезной при работе с файловой системой компьютера. Процедура в отличие от стандартных средств доступа пакета позволяет производить обработку и файла данных, и каталога, расположенных в любом месте файлофой системы *ПК*. В реализации процедуры существенно использовалась функция **Run** пакета, а именно:

**Run**[*S1*, *S2*, ..., *Sn*] – выполняет в базовой операционной среде (например, *MS DOS*) команду, сформированную из выражений *Sj* (*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[1151]:= Run["Dir ", "C:\\", " > ", "C:\\\\Dir"]
```

```
Out[1151]= 0
```

```
Volume in drive G is WORK_FLASH
```

```
Volume Serial Number is 1C41-8474
```

```
Directory of G:\
```

```
12/26/2011 08:35 PM <DIR> AVZ_Package
```

```
12/26/2011 08:36 PM <DIR> New_Book
```

```
12/26/2011 03:30 PM 31,114 http.html
```

```
12/29/2011 07:30 PM <DIR> Rasnoe
```

```
01/07/2012 11:01 AM <DIR> USERLIB6789
```

```
1 File(s) 31,114 bytes
```

```
5 Dir(s) 8,128,442,368 bytes free
```

```
In[1152]:= Run["Attrib ", "D:/AVZ_Package\\AVZ_Package.m", " > ", "C:\\\\Attrib"]
```

```
Out[1152]= 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 иллюстрирует *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.

```
In[1081]:= Attrib["dereks.doc", "Attr"]
```

```
Out[1081]= {"A", "S", "H", "R", {"C:\\Temp\\dereks.doc"}}
```

```
In[1082]:= Attrib["dereks.doc", "Attr"]
```

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[1131]:= LoadExtProg[x_ /; StringQ[x]] := Module[{a = $InstallationDirectory, b, c,
    d = Directory[]}, If[SetDirectory[a]; FileExistsQ[x], SetDirectory[d]; Return[x],
    SetDirectory[a]; Clear[$Art23$Kr16$]; b = FileExistsQ1[x, $Art23$Kr16$];
    c = $Art23$Kr16$; Clear[$Art23$Kr16$];
    If[! b, SetDirectory[d]; $Failed, SetDirectory[a];
    CopyFile[c[[1]], x]; SetDirectory[d]; c[[1]]]]
```

```
In[1132]:= LoadExtProg["attrib.exe"]
Out[1132]= "C:\\WINDOWS\\$NtServicePackUninstall$\\attrib.exe"
In[1133]:= FileExistsQ[$InstallationDirectory <> "\\\" <> "attrib.exe"]
Out[1133]= True
In[1134]:= Attrib1["dereks.doc", "Attr"]
Out[1134]= {"A", "S", "H", "R", {"C:\\Temp\\dereks.doc"}}
```

Успешный вызов процедуры **LoadExtProg**[*x*] отыскивает в файловой системе файл *x* и копирует его в каталог, определяемый переменной **\$InstallationDirectory**, возвращая *x*, если файл уже находился в данном каталоге, и полный путь к нему, из которого он был в каталог скопирован. При невозможности найти искомый файл с программой *x* возвращается **\$Failed**. Заранее по вызову **LoadExtProg**[*x*] можно обеспечить доступ к *x*. Таким образом, используя данную процедуру в сочетании с функцией **Run**, можно в программной среде пакета *Mathematica* выполнять целый ряд весьма полезных {*exe* | *com*}-программ самого различного назначения, тем самым достаточно существенно расширяя функциональные возможности самого пакета.

Ниже приводится модификация процедуры **Attrib**, которая существенно использует наши процедуры **LoadExtProg**, **StandPath**, **StrDelEnds**, **FileExistsQ1** и **StandPath** [90].

```
In[1071]:= Attrib1[F_;/; StringQ[F], x_;/; ListQ[x] && DeleteDuplicates[Map3[MemberQ,
    {"-A", "-H", "-S", "-R", "+A", "+H", "+S", "+R"}, x]] == {True} || x == {} || x == "Attr" :=
    Module[{a = "$Art23_Kr16$", b = "attrib ", c, d = "> ", h = "attrib.exe", p, f, g},
    If[LoadExtProg["attrib.exe"] === $Failed, Return[$Failed], Null]; If[StringLength[F] == 3
    && DirQ[F], f = StandPath[F], If[FileExistsQ1[StrDelEnds[StringReplace[F, "/" -> "\\",
    "\\\", 2], $Art23$Kr16$], g = $Art23$Kr16$; f = StandPath[g[[1]]]; Clear[$Art23$Kr16$],
    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[$InstallationDirectory <> "\\\" <> "attrib.exe"];
    h = Flatten[h /. {"HR" -> {"H", "R"}, "SH" -> {"S", "H"},
    "SHR" -> {"S", "H", "R"}, "SRH" -> {"S", "R", "H"},
    "HSR" -> {"H", "S", "R"}, "HRS" -> {"H", "R", "S"},
    "RSH" -> {"R", "S", "H"}, "RHS" -> {"R", "H", "S"}}];
    If[h === {"File", "not", "found", "-"} ||
    MemberQ[h, "C:\\Documents", "Drive " <> f, Append[h, g]]]
In[1072]:= Attrib1["dereks.doc", "Attr"]
Out[1072]= {"A", "S", "H", "R", {"C:\\Temp\\dereks.doc"}}
In[1073]:= FileExistsQ[$InstallationDirectory <> "\\\" <> "attrib.exe"]
Out[1073]= False
```

Процедура **Attrib1**[*F*,*x*] функционально полностью аналогична процедуре **Attrib**[*F*,*x*], имеет аналогичную справку, но с несколько иной реализацией, базирующейся как на

использовании стандартной функции **Run**, так и нашей процедуры **LoadExtProg**; при этом, после вызова процедуры файл «*attrib.exe*» удаляется из **\$InstallationDirectory**. В этом отношении файловая система пакета остается неизменной после вызова **Attrib1**. Таким образом, декларируемая пакетом возможность расширения системы каталогов, определяемая переменной **\$Path**, в общем случае не работает уже для *внешних* команд **MS DOS**, как 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 *Windows XP*, а именно:

```
In[1065]:= Run["tlist", ">", "C:\\tlist"]
```

```
Out[1065]= 1
```

```
In[1066]:= LoadExtProg["tlist.exe"]; Run["tlist", ">", "C:\\tlist"]
```

```
Out[1066]= 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          New_Book - Microsoft Word
4348 avgrsx.exe
4380 avgcsrvx.exe
5248 Mathematica.exe       Wolfram Mathematica 8.0 - [Running.AVZ_Package.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 пакета, определяемый переменной **\$InstallationDirectory**, файла «*tlist.exe*», позволяя успешно выполнить посредством **Run** внешнюю команду **tlist** с сохранением в файле результата ее выполнения, 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 **Run**-функцию пакета.

```

In[693]:= 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[StringEnd[a, h],
c = Append[c, If[FileExistsQ[v], v, Null]]; Goto[b], Goto[b]]];
For[k = 1, k <= Length[a], k++, b = Append[b, dir[a[[k]]]]; Flatten[b]]

In[694]:= SearchFile["AVZ_Package.nb"]
Out[694]= {"C:\\Book\\AVZ_Package.nb", "D:\\Book\\AVZ_Package.nb",
"D:\\Math_myLib\\AVZ_Package.nb", "G:\\Book\\AVZ_Package.nb"}

In[695]:= SearchFile["init.m"]
Out[695]= {"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\\8.0\\SystemFiles\\Links\\XMLSchema\\Kernel\\init.m"}

In[696]:= Length[%]
Out[696]= 91
    
```

```

In[697]:= SearchFile["MapleMathem.doc"]
Out[697]= {"C:\\Book\\MapleMathem.doc", "D:\\Book\\MapleMathem.doc",
           "G:\\Book\\MapleMathem.doc"}
In[698]:= SearchFile["new_book.doc"]
Out[698]= {"D:\\New_Book\\New_Book.doc", "G:\\New_Book\\New_Book.doc"}

```

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[1158]:= SearchDir[D_;/ StringQ[D]] := Module[{a, b, c, d, f, h, dir, k},
  {a, b, f} = {Map[ToUpperCase[#] <> ":\\" &, ADrive[]], {}, "$$$$15Dir23$$$$.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[1159]:= SearchDir["book"]
Out[1159]= {"C:\\Book", "C:\\Program Files\\Wolfram Research\\Mathematica\\8.0\\
SystemFiles\\FrontEnd\\StyleSheets", "C:\\Program Files\\Wolfram Research\\
Mathematica\\8.0\\SystemFiles\\FrontEnd\\StyleSheets\\Book", "D:\\Book",
"G:\\Book", "G:\\Book\\Fonts", "G:\\Book\\Fonts\\Type1", "G:\\Book\\Fonts\\
TrueType", "G:\\Book\\Fonts\\SVG"}

In[1160]:= SearchDir["Old Grodno"]
Out[1160]= {"C:\\ARCHIVE", "C:\\ARCHIVE\\OLD GRODNO", "G:\\ARCHIVE",
"G:\\ARCHIVE\\OLD GRODNO", "G:\\RANS_IAN_2011",
"G:\\RANS_IAN_2011\\OLD GRODNO"}

In[1161]:= SearchDir["Photo"]
Out[1161]= {"C:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS",
"C:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS\\PHOTO",
"G:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS",
"G:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS\\PHOTO"}

In[1162]:= SearchDir["UserLib"]
Out[1162]= {"C:\\Program Files\\Maple 11\\lib", "C:\\Program Files\\Maple 11\\
lib\\userlib", "D:\\USERLIB6789\\USERLIB10", "D:\\USERLIB6789\\

```

```
USERLIB10\\userlib", "D:\\USERLIB6789\\USERLIB11",
"D:\\USERLIB6789\\USERLIB6\\USERLIB", "D:\\USERLIB6789\\USERLIB7",
"D:\\USERLIB6789\\USERLIB9\\userlib", "G:\\USERLIB6789\\USERLIB9",
"G:\\USERLIB6789\\USERLIB9\\userlib", "G:\\USERLIB6789\\USERLIB8",
"G:\\USERLIB6789\\USERLIB11\\userlib", "G:\\USERLIB6789\\USERLIB10",
"G:\\USERLIB6789\\USERLIB10\\userlib"}
```

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 и в таком случае можно воспользоваться простой функцией *Ver*, чей вызов *Ver[]* возвращает строку с уточнением типа операционной платформы. Следующий фрагмент представляет исходный код функции и пример применения.

```
In[2245]:= {$System, $SystemID, $OperatingSystem}
Out[2245]= {"Microsoft Windows (32-bit)", "Windows", "Windows"}
In[2246]:= Ver[] := Block[{a, b}, a = "Art####$Kr####.txt"; Run["Ver > " <> a];
           b = Read[a, String]; Close[a]; DeleteFile[a]; b]
In[2247]:= Ver[]
Out[2247]= "Microsoft Windows XP [Version 5.1.2600]"
```

Вызов довольно простой функции *DelFile[x]* удаляет файл *x*, заданный именем либо полным путем к нему, возвращая *Null*, т.е. ничего, и расширяя процедуру *DelDirFile* на открытые файлы. Фрагмент представляет исходный код с примером применения.

```
In[1947]:= DelFile[x_;/ FileExistsQ[x]] := If[Closes[x]; SameQ[Quiet[DeleteFile[x]],
                                           $Failed], DelDirFile[x], Null]
In[1948]:= DelFile1[x_;/ SameQ[Closes[x]] && FileExistsQ[x]] :=
           If[SameQ[Quiet[DeleteFile[x]], $Failed], DelDirFile[x], Null]
In[1949]:= DelFile2[x_;/ FileExistsQ[x]] := If[SameQ[Quiet[DeleteFile[Closes[x]; x]],
                                           $Failed], DelDirFile[x], Null]
In[1955]:= DelFile["C:\\Temp\\Book\\CoreLang.pdf"]
```

Фрагмент представляет две модификации функции, содержащие полезные приемы.

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. Таким образом, подобный класс задач в программной среде пакета *Maple* пользователь вынужден достаточно существенно ограничивать.

### 9.3. Некоторые полезные средства доступа в среде Mathematica

В предыдущих разделах главы представлен ряд достаточно полезных средств доступа к элементам файловой системы компьютера, как расширяющих набор стандартных средств, так и расширяющих сферу их применимости. Здесь представлен ряд средств, представляющих определенный интерес для пользователя, чьи задачи имеют дело с обработкой файлов данных, но не нашедших по той либо иной причине отражения в предыдущих разделах главы. Между тем, использованные в этих средствах приемы могут оказаться достаточно полезными при программировании в среде *Mathematica* задач подобного типа. С рядом других подобных средств можно ознакомиться в [90].

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[2518]:= OpenRead["C:/Temp/Attention.doc"]; Read["C:/Temp/Attention!.doc", Word];
Write["D:/Art.22", S]; Write["C:/Temp/Agn.63", "TS"]; OpenRead["C:/Temp/Fultus.doc"];
In[2519]:= 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[2520]:= OpenFiles[]
Out[2520]= {"read", "C:/Temp/Attention.doc", "C:/Temp/Fultus.doc", {"write", "D:/Art.23",
    "C:/Temp/Agn", "C:/Academy/RANS/IAN/Tallinn.txt"}}
In[2521]:= {CloseAll[], OpenFiles[]}
Out[2521]= {"C:/Temp/Attention.doc", "C:/Temp/Fultus.doc", "D:/Art.23", "C:/Temp/Agn",
    "C:/Academy/RANS/IAN/Tallinn.txt"}, {}
```

При этом, под полным путем понимается либо реально полный путь к файлу, либо его полное имя, если он находится в текущем каталоге, определяемом **Directory[]**.

```
In[1141]:= 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[1142]:= Read["C:/Temp\\dereks.doc"]; Mapp[Write, {"RANS", "D:/Agn", "Academy"}];
```

```
In[1143]:= StreamFiles[]
Out[1143]= {"in", "C:/Temp\\dereks.doc"}, {"out", "RANS", "D:/ Agn"}
In[1144]:= CloseAll[]; StreamFiles[]
Out[1144]= "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 **IsFileOpen[F, h]**, вызов которой возвращает **True**, если определенный своим именем или полным путем файл **F** открыт, и **False** в противном случае. Если аргумент **F** не определяет файл и/или второй аргумент **h** является *определенным*, т.е. **DefFunc[h]** возвращает значение, отличное от **Null**, то вызов процедуры **IsFileOpen** возвращает **\$Failed**. При этом, через второй аргумент в случае возврата **True** возвращается список режимов открытия файла данных **F**, а именно **{"read", "write"}**. Следующий фрагмент представляет исходный код процедуры наряду с примерами ее использования.

```
In[503]:= IsFileOpen[F_ /; FileExistsQ[F], h_ /; ! HowAct[h]] := Module[{a = OpenFiles[],
    b, d = {}, c = StringReplace[ToUpperCase[F], "\\\" -> "/"], k = 1, t = False},
    If[a == {}, Return[False], b = Map[ToUpperCase, a]];
    b = Map4[StringReplace, Map[ToString1, b], ToString1["\\\" -> "/]];
    If[! FreeQ[b[[1]], c], d = Append[d, "read"]; t = True, Null];
    If[! FreeQ[b[[2]], c], d = Append[d, "write"]; t = True, Null]; h = d; t]
In[504]:= {IsFileOpen["C:\\Temp\\Fultus.doc", G], G}
Out[504]= {True, {"read"}}
```

Процедура **FileOpenQ1** является достаточно полезным расширением рассмотренной выше процедуры **FileOpenQ**. Вызов процедуры **FileOpenQ1[F]** возвращает *вложенный* список формата **{{R, x, y, ..., z}, {{R, x1, y1, ..., z1}}}**, если файл **F** открыт для чтения либо записи (**R = {"in" | "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, ..., z}** и **{x1, y1, ..., 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 применения.

```
In[1183]:= Write["dereks1.doc", "ian"]; Write["D:/dereks.doc", "Kr"]; Write["RANS", "ian"]
In[1184]:= Write["C:/TEMP\\DEREKS.doc", 45]; Read["C:/temp/dereks.doc"];
In[1185]:= StreamsU[]
```

```

Out[1185]= {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[1186]:= 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[1187]:= FileOpenQ1["dereks.doc"]
Out[1187]= {"in", "C:/Temp\\dereks.doc", "C:/temp/dereks.doc"},
  {"out", "D:/dereks.doc", "C:/TEMP\\DEREKS.doc"}
In[1188]:= FileOpenQ1["rans"]
Out[1188]= {"in"}, {"out", "RANS"}
In[1189]:= FileOpenQ1["Tallinn_Tampere_ArtKr.pdf"]
Out[1189]= False

```

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[1135]:= 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[1136]:= Map[PathToFileQ, {"C:", "C:/", "G:/AVZ_Package", "H:\\\\AGN", "C:/Temp"}]
Out[1136]= {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 следующем фрагменте представлена процедура, чей вызов **EmptyFileQ[F]** возвращает **True**, если файл данных  $F$  пуст, и **False** в противном случае.

```

In[1283]:= EmptyFileQ[F_;/; StringQ[F], y_] := Module[{a, b}, If[FileExistsQ[F], b = F,
  Clear[$$ArtKr$$]; a = FileExistsQ1[F, $$ArtKr$$];
  If[! A, Return[$Failed], b = $$ArtKr$$[[1]]; Clear[$$ArtKr$$]];
  If[! Y === Null && ! HowAct[y], ToExpression[ToString[y] <>
    " = " <> ToString1[b]], Null];
  Quiet[Close[b]]; If[Read[b] === EndOfFile, Close[b]; True, Close[b]; False]]

```

```
In[1284]:= Map[EmptyFileQ, {"ArtKr.tam", "RANS", "dereks.doc", "C:/grsu.doc"}]
Out[1284]= {$Failed, True, False, False}
In[1285]:= {Clear[y]; EmptyFileQ["rans", y], y}
Out[1285]= {True, "C:\\RANS"}
```

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 тестируется первый файл из их списка. Более того, основной результат вызова процедуры **EmptyFileQ**[ $F$ ,  $y$ ] с *двумя* фактическими аргументами аналогичен вызову с *одним* аргументом, тогда как через второй *необязательный* аргумент возвращается полное имя или полный путь к файлу  $F$ . Предыдущий фрагмент представляет исходный код с примерами применения.

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 процедура **ReadFullFile**, чей исходный код с примерами применения представляет нижеследующий фрагмент.

```
In[1092]:= ReadFullFile[F_/, StringQ[F], y___] := Module[{a}, If[FileExistsQ[F], a = F,
    Clear[$$ArtKr$$]; If[! FileExistsQ1[F, $$ArtKr$$], Return[$Failed],
    a = $$ArtKr$$[[1]]; Clear[$$ArtKr$$]]; If[! Y === Null && ! HowAct[y],
    ToExpression[ToString[y] <> " = " <> ToString1[a], Null];
    Close[a]; StringReplace[StringJoin[Map[FromCharCode,
    BinaryReadList[a]]], "\r\n" -> ""]]
In[1093]:= ReadFullFile["direks.doc"]
Out[1093]= $Failed
In[1094]:= ReadFullFile["dereks.doc"]
Out[1094]= "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[1095]:= ReadFullFile["dereks.doc", h]; h
Out[1095]= "C:\\Temp\\dereks.doc"
```

Вызов **ReadFullFile**[ $F$ ] возвращает содержимое файла данных  $F$  с исключением из него символов «\r\n» возврата каретки и перевода строки, если файл данных  $F$  отсутствует в файловой системе компьютера, вызов возвращает значение **\$Failed**. При этом, вызов **ReadFullFile**[ $F$ ,  $y$ ] со вторым аргументом  $y$  через него возвращается полное имя либо полный путь к файлу данных  $F$ .

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[1161]:= 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[1162]:= Map[FreeSpaceVol, {"c", "G", "d", "A", "F"}]
Out[1162]= {{100721643520, bytes}, {8127459328, bytes}, {33992261632, bytes},
      "Drive <A> is not ready", "Drive <F> is not ready"}
```

Следующая достаточно простая процедура **DeCod**[*W*] функционирует по принципу переключателя, кодируя/декодируя файл, заданный ее фактическим аргументом *W*. Ее вызов **DeCod**[*W*] на нормальном файле *W* обеспечивает его кодирование, которое не позволяет использовать файл *W* в *оригинальном* формате. Тогда как последующий вызов процедуры **DeCod**[*W*] на закодированном файле декодирует файл в исходный оригинальный формат. Вызов **DeCod**[*W*] возвращает имя обрабатываемого файла *W* или *полный* путь к нему; при этом, обработка файла *W* производится на месте (*in situ*), а в качестве *W* может выступать *произвольный* файл данных из файловой системы *ПК*. При отсутствии файла *W* в файловой системе *ПК* возвращается **\$Failed**. Следующий фрагмент представляет *исходный* код процедуры **DeCod** с примерами ее применения.

```
In[1030]:= DeCod[F_;/ StringQ[F]] := Module[{a, b, k = 1, Kr},
      If[FileExistsQ[F], b = F, Clear[$$Art$Kr$];
      If[FileExistsQ1[F, $$Art$Kr$], b = $$Art$Kr$[[1]];
      Clear[$$Art$Kr$], Return[$Failed]];
      Kr[N_;/ IntegerQ[N]] := Module[{a, b, k = 1}, a = PadLeft[IntegerDigits[N, 2], 8];
      For[k, k <= 8, k++, a[[k]] = If[a[[k]] == 0, 1, 0]]; Sum[a[[k]]*2^(8 - k), {k, 1, 8}];
      a = BinaryReadList[b];
      For[k, k <= Length[a], k++, a[[k]] = Kr[a[[k]]]]; BinaryWrite[b, a]; Close[b]

In[1031]:= DeCod["D:/AVZ_Package/Project_HS.doc"]
Out[1031]= "D:/AVZ_Package/Project_HS.doc"
In[1032]:= DeCod["D:/AVZ_Package/Project_HS.doc"]
Out[1032]= "D:/AVZ_Package/Project_HS.doc"
In[1033]:= DeCod["Dereks.doc"]
Out[1033]= "C:\\Temp\\dereks.doc"
In[1034]:= DeCod["Dereks.doc"]
Out[1034]= "C:\\Temp\\dereks.doc"
```

Для пакета *Maple* была создана процедура *Email*, обеспечивающая детальный анализ файлов типов {*doc*, *rtf*, *htm*}, а в общем случае и *rlb*-типа [44,45] для поиска в них *email*-адресов. Ниже в качестве упрощенного аналога приведена одноименная процедура,

чей вызов **Email**[F, t] обеспечивает возврат списка корректных *email*-адресов из файла F типа {*mht*, *htm*, *html*}, тогда как через второй *d*-аргумент возвращается список *email*-конструкций, требующих дальнейшего анализа на предмет быть *email*-адресами.

```
In[2236]:= Email[F_ /; FileExistsQ[F] && MemberQ[{"html", "htm", "mht"},
  FileExtension[F]], doubt_ /; ! HowAct[doubt]] := Module[{a = {}, b, c, d, k = 1, v, p, t},
  v = Flatten[Map[ToCharacterCode, Join[CharacterRange["a", "z"],
    CharacterRange["A", "Z"], {"_", "-", "."}]]];
  c = Select[BinaryReadList[F], # != 10 || # != 13 &]; d = Flatten[Position[c, 64]];
  For[k, k <= Length[d], k++, b = "@";
    For[p = d[[k]] + 1, p <= Length[c], p++, t = c[[p]];
    If[MemberQ[v, t], b = b <> FromCharacterCode[t]; Continue[], Break[]];
    For[p = d[[k]] - 1, p >= 1, p--, t = c[[p]];
    If[MemberQ[v, t], b = FromCharacterCode[t] <> b; Continue[], Break[]];
    a = Append[a, b]; a = DeleteDuplicates[a];
    doubt = Select[a, # == "@" || StringFreeQ[StrDelEnds[#, "@", 3], "@"] ||
      StringFreeQ[#, "."] &]; Select[a, ! MemberQ[doubt, #] &]]

In[2237]:= Email["D:\\Math_myLib\\Rans_Ian.mht", Res]
Out[2237]= {"aladjev@yandex.ru", "aladjev@gmail.com", "valadjev@yahoo.com",
  "noosphere_academy@yahoo.com", "iansd.staff@inbox.ru", "kristo@live.com"}

In[2238]:= Res
Out[2238]= {"@font-face", "@SimSun", "@page", "m@", "l@", "@yandex.ru", "Art@Kr", "@"}
```

Отметим, для упрощения реализуемого процедурой **Email** алгоритма полагается, что *email*-адреса указаны в общепринятой форме без защиты от автоматического чтения. В процедуре **Email** иллюстрируется использование *фактических аргументов* процедур для обеспечения возможности возврата результатов вызова процедур дополнительно к основному результату, что во многих случаях представляется весьма удобным. При этом, следует отметить, что *способ* возврата результатов через фактические аргументы более предпочтителен (*ввиду ранее сказанного*) возврата через глобальные переменные.

Следующий фрагмент представляет *реализацию* процедуры **Email1** с использованием процедуры **SubStr**, позволяя упростить алгоритм и расширить *типы* анализируемых файлов данных на предмет наличия в них корректных *email*-адресов.

```
In[2149]:= Email1[F_ /; FileExistsQ[F], t_ /; ! HowAct[t]] := Module[{a = {}, c, d, k = 1, h, v},
  v = Join[CharacterRange["a", "z"], CharacterRange["A", "Z"], {"_", "-", "."}];
  v = Select[Map[FromCharacterCode, Range[0, 255]], ! MemberQ[v, #] &];
  c = StringJoin[Map[FromCharacterCode, Select[BinaryReadList[F],
    # != 10 || # != 13 &]]]; d = DeleteDuplicates[Flatten[StringPosition[c, "@"]]];
  For[k, k <= Length[d], k++, a = Append[a, SubStr[c, d[[k]], v, v, $Art23$Kr15$]];
  a = DeleteDuplicates[a]; t = Select[a, # == "@" || StringFreeQ[StrDelEnds[#, "@", 3],
    {"@", "."}]] || StringFreeQ[#, "."] || StringTake[#, {1, 1}] == "." ||
  StringTake[#, {-1, -1}] == "." &]; Select[a, ! MemberQ[t, #] &]]
```

```

In[2150]:= Email1["D:\\Math_myLib\\Noosphere_Academy.nb", Res]
Out[1250]= {"valadjev@yahoo.com", "vaganov@yandex.ru", "kristo@live.com"}
In[2151]:= Res
Out[2151]= {"@", "@font-face", "@page", "Art@Kr", "@SimSun", "U@", "rans@", "ian@"

```

Отметим, что в качестве первого фактического аргумента процедуры как **Email**, так и **Email1** должен выступать файл данных, находящийся либо в текущем каталоге, либо находящийся в произвольном каталоге файловой системы компьютера, однако в этом случае файл данных должен задаваться полным путем к нему. Это условие довольно легко устранимо, как это было нами реализовано, в частности, в процедуре **DeCod**.

Наряду с теоретическим исследованием классических однородных структур (*Cellular Automata*) достаточно широко [85] используется их экспериментальное исследование с использованием средств компьютерного моделирования. С данной целью был создан большой набор программных средств различного назначения и сложности. Целый ряд средств данного типа представлен в [85,91-94]. В частности, в рамках общей задачи по исследованию проблемы *неконструируемости* возникает вопрос генерации для любой блочной конфигурации *Co* предшественников, т.е. блочных конфигураций, которые под действием *глобальной функции* перехода структуры в следующий момент времени переводятся в конфигурацию *Co*. Оценка числа данных предшественников позволяет на основе критерия *Аладьева-Маруока-Кимура* в целом ряде случаев решать вопрос наличия для структуры неконструируемости *NCF*-типа [85,91-94]. С иллюстративной целью здесь представлен ряд *Mathematica*-процедур, решающих данную задачу для случая *1*-мерных классических структур с произвольными алфавитом *W* внутренних состояний и индексом соседства *X* классической однородной структуры.

Для исследования *обратной* динамики одномерных классических структур [85,91-94] в ряде случаев полезной оказывается процедура **Predecessors**[*ltf*, *Co*, *n*], возвращающая на основе таблицы *ltf*, определяющей *локальную функцию* перехода структуры, размера ее шаблона соседства *n* и начальной конфигурации *Co* конечного блока из состояний элементарных автоматов структуры множество конфигураций-предшественников. В следующем фрагменте представлен исходный код процедуры наряду с примерами ее использования для простых *1*-мерных классических бинарных структур.

```

In[2538]:= Predecessors[Ltf_List, Co_String, n_Integer] := Module[{L, a, b, c, h = {}, i, j, k,
  d = StringLength[Co]}, a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
  For[k=1, k <= Length[a], k++, L[StringTake[First[a][[k]], -1]] =
  Map2[StringDrop, Map[ToString1, a[[k]], {-1}]]; b = L[StringTake[Co, 1]];
  For[k = 2, k <= d, k++, c = L[StringTake[Co, {k, k}]];
  For[i = 1, i <= Length[b], i++, For[j = 1, j <= Length[c], j++,
  If[StringEnd[b[[i]], StringTake[c[[j]], n - 1], h = Append[h, b[[i]] <>
  StringTake[c[[j]], -1], Null]]; b = h; h = {}]; If[Length[b] != (n - 1)^Length[a],
  Print["Structure possesses the nonconstructability of NCF-type"], Null]; b]
In[2539]:= Ltf = {"0000", "0011", "0101", "0110", "1000", "1010", "1100", "1111"}
Out[2539]= {"0000", "0011", "0101", "0110", "1000", "1010", "1100", "1111"}

```

```
In[2540]:= Ltf1 = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[2540]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[2541]:= Predecessors[Ltf, "11001110", 3]
      Structure possesses the nonconstructability of NCF-type
Out[2541]= {"0010111110"}
In[2542]:= Predecessors[Ltf, "11001110110101", 3]
      Structure possesses the nonconstructability of NCF-type
Out[2542]= {}
In[2543]:= Predecessors[Ltf1, "1100111001101", 3]
Out[2543]= {"001011111010001", "010000100001010", "1001100101111100", "111101001100111"}
```

В частности, возврат процедурой *Predecessors* пустого списка говорит о наличии для соответствующей структуры неконструируемости *NCF*-типа и позволяет определять по меньшей мере один из видов неконструируемых конфигураций [85,91–94]. Более того, процедура *Predecessors* в ряде случаев позволяет определять существование для исследуемой структуры неконструируемости *NCF*-типа на основе числа элементов у списка *предшественников* блочной конфигурации *Co*. Данный фрагмент представляет аналог процедуры *Predecessors* в среде пакета *Maple*. Алгоритм данного аналога не базируется на табличных структурах, а параллельные подстановки  $x_1x_2x_3 \dots x_n \rightarrow x'_1$ , определяющие локальную функцию перехода структуры, определяются не *таблицей*, а списком *Ltf* строк вида « $x_1x_2 \dots x_nx'_1$ ». Аргументы *Co* и *n* идентичны одноименным аргументам процедуры-прообразу из *Maple*. Тут же вполне уместно подчеркнуть, что в среде *Maple* несколько более удачна организация табличных структур, позволяя во многих случаях эффективно определять табличные объекты различного назначения и оперировать с ними в составе более простых и прозрачных алгоритмов [91–94,99].

Процедуры *PredecessorsR[Ltf, Co, n, Cf]* и *PredecessorsL[Ltf, Co, n, Cf]* идентичны и их различие сводится только к тому, что процесс генерации предшественников для *Co*-конфигурации в структуре с шаблоном соседства размера *n* и локальной функцией перехода, определяемой списком строк вида " $x_1x_2\dots x_nx'_1$ " (эти строки отвечают набору параллельных подстановок  $x_1x_2 \dots x_n \rightarrow x'_1 \setminus \{x_n, x'_1 \notin A\}$ , определяющих локальную функцию перехода структуры), производится слева направо (*R*) или справа налево (*L*); результат вызова процедур на идентичных фактических аргументах [*Ltf, Co, n, Cf*], естественно, аналогичен. Вызов каждой из этих процедур в качестве результата возвращает число предшественников для блочной конфигурации *Co*, тогда как через 4-й аргумент *Cf* в качестве дополнительного возвращается список всех *предшественников* конфигурации *Co*. Именно на основе данной информации можно судить о том, будет ли блочная *Co*-конфигурация являться  $\gamma$ -конфигурацией [85,91–94], и что в случае положительного ответа на основе критерия *Аладьева-Маруока-Кимура* [85] можно говорить о наличии для классической однородной структуры неконструируемости *NCF*-типа. Следующий фрагмент представляет исходные коды обеих процедур *PredecessorsR* и *PredecessorsL* наряду с примерами их использования для анализа простых 1-мерных классических бинарных структур с шаблоном соседства размера 3. Данные процедуры существенно используют рассмотренные функцию *SymbolQ* и процедуры *Map2* и *ToString1*.

```

In[2258]:= PredecessorsR[Ltf_List, Co_String, n_ /; IntegerQ[n], Cf_ /; SymbolQ[Cf]] :=
Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]},
a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
For[k = 1, k <= Length[a], k++, L[StringTake[First[a[[k]]], -1]] =
Map2[StringDrop, Map[ToString1, a[[k]], {-1}]]; b = L[StringTake[Co, 1]];
For[k = 2, k <= d, k++, c = L[StringTake[Co, {k, k}]];
For[i = 1, i <= Length[b], i++, For[j = 1, j <= Length[c], j++,
If[StringEnd[b[[i]], StringTake[c[[j]], n - 1],
h = Append[h, b[[i]] <> StringTake[c[[j]], -1], Null]]];
b = h; h = {}; Cf = b; Length[b]]

In[2259]:= PredecessorsL[Ltf_List, Co_ /; StringQ[Co], n_Integer, Cf_ /; SymbolQ[Cf]] :=
Module[{L, a, b, c, h = {}, i, j, k, d = StringLength[Co]},
a = Gather[Ltf, StringTake[#1, -1] == StringTake[#2, -1] &];
For[k = 1, k <= Length[a], k++, L[StringTake[First[a[[k]]], -1]] =
Map2[StringDrop, Map[ToString1, a[[k]], {-1}]]; b = L[StringTake[Co, -1]];
For[k = d - 1, k >= 1, k--, c = L[StringTake[Co, {k, k}]];
For[i = 1, i <= Length[b], i++, For[j = 1, j <= Length[c], j++,
If[StringTake[b[[i]], n - 1] === StringTake[c[[j]], {2, -1}],
h = Append[h, StringTake[c[[j]], 1] <> b[[i]], Null]]];
b = h; h = {}; Cf = b; Length[b]]

In[2260]:= Ltf = {"0000", "0011", "0101", "0111", "1000", "1011", "1101", "1110"}
Out[2260]= {"0000", "0011", "0101", "0111", "1000", "1011", "1101", "1110"}
In[2261]:= Ltf1 = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[2261]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[2262]:= Clear[Cf]; {PredecessorsR[Ltf, "010100", 3, Cf], Cf}
Out[2262]= {0, {}}
In[2263]:= Clear[Cf]; {PredecessorsL[Ltf, "010100", 3, Cf], Cf}
Out[2263]= {0, {}}
In[2264]:= Clear[Cf]; {PredecessorsL[Ltf1, "010100", 3, Cf], Cf}
Out[2264]= {4, {"01110000", "10101011", "00011101", "11000110"}}
In[2265]:= Clear[Cf]; {PredecessorsR[Ltf1, "010100", 3, Cf], Cf}
Out[2265]= {4, {"00011101", "01110000", "10101011", "11000110"}}
In[2266]:= Clear[Cf]; {PredecessorsL[Ltf, "0101", 3, Cf], Cf}
Out[2266]= {1, {"111001"}}
In[2267]:= Clear[Cf]; {PredecessorsR[Ltf, "0101", 3, Cf], Cf}
Out[2267]= {1, {"111001"}}

```

Для экспериментального исследования предшественников блочных конфигураций в 1-мерных классических структурах дополнительно к представленным определенным интерес представляет и процедура **TestOnNCF[Ltf, n, p]**, существенно базирующаяся на одной из предыдущих процедур, а именно **PredecessorsR**. **TestOnNCF[Ltf, n, p]** для 1-мерной классической структуры с шаблоном соседства размера  $n$  и локальной **Ltf**-

функцией перехода возвращает *первую* из *блочных* конфигураций размера *m*, которая является  $\gamma$ -конфигурацией, и печатает об этом соответствующее сообщение. В случае отсутствия  $\gamma$ -конфигураций размера *p* вызов процедуры **TestOnNCF** возвращает **Null** и печатает об этом соответствующее сообщение. Следующий фрагмент представляет исходный код процедуры и наиболее типичные примеры ее использования.

```
In[1558]:= TestOnNCF[Ltf_/, ListQ[Ltf], n_/, IntegerQ[n], m_/, IntegerQ[m]] :=
Module[{a = {}, b, c, Cf, d, k = 1},
  For[k, k <= Length[Ltf], k++, a = Append[a, StringTake[Ltf[[k]], -1]]];
a = DeleteDuplicates[a]; b = Map[StringJoin, Tuples[a, m]]; d = (n - 1)^Length[a];
For[k = 1, k <= Length[b], k++, Clear[Cf]; c = PredecessorsR[Ltf, b[[k]], n, Cf];
  If[c != d, Print["Block configuration <" <> ToString[b[[k]]] <>
    "> is gamma-configuration"]; Return[b[[k]], Null]];
Print["All block configurations of size " <> ToString[m] <>
  " possess " <> ToString[d] <> " different predecessors"]

In[1559]:= Ltf = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[1559]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[1560]:= Ltf1 = {"0000", "0011", "0101", "0111", "1001", "1010", "1101", "1110"}
Out[1560]= {"0000", "0011", "0101", "0111", "1001", "1010", "1101", "1110"}
In[1561]:= TestOnNCF[Ltf1, 3, 15]
Block configuration <0000000000000000> is gamma-configuration
Out[1561]= "0000000000000000"
In[1562]:= TestOnNCF[Ltf, 3, 8]
All block configurations of size 8 possess 4 different predecessors
```

Учитывая, в целом, большую реактивность *Mathematica* относительно *Maple*, казалось бы вполне естественным использовать именно пакет *Mathematica* для симулирования динамики и исследования ряда свойств однородных структур. Однако, это не совсем так, как весьма наглядно иллюстрирует ряд примеров [99]. Впрочем, данные задачи с большей эффективностью решаются и в среде *Mathematica* на основе иных подходов. Однако нами такой задачи не ставилось, а тестировалась лишь эффективность обоих пакетов на примере реализации аналогов алгоритмов для ряда программных средств.

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 применения.

```
In[2230]:= 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[2231]:= BootDrive[]
```

```
Out[2231]= {"C", "Microsoft Windows XP Professional"}
```

```
In[942]:= BootDrive[]
```

```
Out[942]= {"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 7–8 на платформе **Windows {2000 | 2003 | NT | XP}** с несколькими операционными средами. В предыдущем примере она применяется в одной операционной среде и в нескольких.

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** с примерами ее довольно типичного применения.

```

In[1128]:= 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[1129]:= FindFile1["New_Book.doc"]
```

```
Out[1129]= {"C:\\New_Book\\New_Book.doc", "D:\\New_Book\\New_Book.doc",
"G:\\New_Book\\New_Book.doc"}
```

```
In[1130]:= Map[FindFile1, {"Rans.Ian", "Cinema.txt"}]
```

```
Out[1130]= {{}, "C:\\Temp\\Cinema.txt"}
```

```
In[1131]:= t = TimeUsed[]; FindFile1["Cinema.txt"]; TimeUsed[] - t
```

```
Out[1131]= 6.844
```

```
In[1132]:= t = TimeUsed[]; FileExistsQ1["Cinema.txt"]; TimeUsed[] - t
```

```
Out[1132]= 6.313
```

Как показывают 2 последних примера фрагмента, процедура **FileExistsQ1**, во многом функционально подобная процедуре **FindFile1**, во временном отношении является в некоторой степени более реактивной в аналогичной файловой системе компьютера.

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[1115]:= Adrive1[] := Block[{a = CharacterRange["A", "Z"], b = {}, c = 1, d, p = {}, h,
    t = "$Art23$Kr16$"}, 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[1116]:= Adrive1[]
Out[1116]= {"C", "D", "G"}, {"A", "E"}}
```

Вызов **HostName[]** довольно простой процедуры возвращает имя в строчном формате компьютера, на котором выполняется текущий сеанс пакета. Фрагмент представляет исходный код процедуры **HostName** с типичным примером ее применения.

```
In[1242]:= HostName[] := Module[{a = Flatten[{LoadExtProg["HostName.exe"]], b,
    c = "$hostname$.txt"}, If[a === {}, $Failed, Run["Hostname.exe > " <> c];
    b = Read[c, String]; Close[c];
    DeleteFile[{InstallationDirectory <> "\\\" <> "hostname.exe", c}]; b]]

In[1243]:= HostName[]
Out[1243]= "aladjev-ef81db6"
```

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 *z* (*произвольное выражение*) файл с программой *x*, загруженный в пакетный каталог удаляется; удаляется и файл **"\$TempFile\$"**, если он пуст либо выполнение программы *x* завершилось аварийно. Во фрагменте представлены исходный код процедуры с примерами ее применения.

```
In[1220]:= ExtProgExe[x_;/; StringQ[x], y_;/; StringQ[y], h___] :=
    Module[{a = "$TempFile$", b = FileExistsQ[x], c},
    Empty::datafile = "Datafile $TempFile$ is empty; the datafile had been deleted.";
    If[b, c = x, b = LoadExtProg[x];
    If[b === $Failed, Return[$Failed], c = If[StringQ[b], b, b[[1]]];
    c = Run[c, " ", y, " > ", a];
    If[{h} == {}, Null, DeleteFile[InstallationDirectory <> "\\\" <> x];
    If[c != 0, $Failed; DeleteFile[a],
    If[EmptyFileQ[a], DeleteFile[a]; Message[Empty::datafile], Directory[] <> "\\\" <> a]]]
```

```

In[1221]:= ExtProgExe["HostName.exe", "", 1]
Out[1221]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
In[1222]:= ExtProgExe["Rans.exe", "", 1]
Out[1222]= $Failed
In[1223]:= ExtProgExe["qprocess.exe", "", 1]
Empty::datafile: Datafile $TempFile$ is empty; the datafile had been deleted.
In[1224]:= ExtProgExe["systeminfo.exe", "/fo table", 1]
Out[1224]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
In[1225]:= ExtProgExe["tasklist.exe", "/svc ", 1]
Out[1225]= "C:\\Documents and Settings\\Aladjev\\My Documents\\$TempFile$"
In[1226]:= {G, h, a} = {{}, "", "$TempFile$"); While[h != "EndOfFile", G = Append[G,
    PrefCond[h = Read[a, String], " "]]; Close[a]; Select[G, # != "" &][[2 ;; -2]]
Out[1226]= {"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"}

```

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 применения.

```

In[1259]:= DirEmptyQ[d_ /; DirQ[d]] := Block[{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[1260]:= Map[DirEmptyQ, {"C:\\Academy", "C:\\temp", "D:/AVZ_Package", "Rans"}]
Out[1260]= {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[1213]:= 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[1214]:= DirFD["C:/Academy"]
```

```
Out[1214]= {"Addition", "Ian", "Rans"}, {"CoreLanguage.pdf", "GrSu.doc", "N_Book.doc"}
```

```
In[1215]:= Map[DirFD, {"C:/RANS", "IAN"}]
```

```
Out[1215]= {{}, DirFD["IAN"]}
```

```
In[1216]:= DirFD["C:/Program Files\\Wolfram Research/Mathematica/8.0\\
Documentation/English/Packages"]
```

```
Out[1216]= {"ANOVA", "Audio", "BarCharts", "Benchmarking", "BlackBodyRadiation",
  "Calendar", "Combinatorica", "Compatibility", "ComputationalGeometry",
  "ComputerArithmetic", "Developer", "EquationTrekker", "ErrorBarPlots",
  "Experimental", "FiniteFields", "FourierSeries", "FunctionApproximations",
  "Geodesy", "GraphUtilities", "HierarchicalClustering", "Histograms",
  "HypothesisTesting", "LinearRegression", "MultivariateStatistics", "Music",
  "NonlinearRegression", "Notation", "NumericalCalculus", "PlotLegends",
  "PieCharts", "NumericalDifferentialEquationAnalysis", "PhysicalConstants",
  "PolyhedronOperations", "Polytopes", "PrimalityProving", "Quaternions",
  "RegressionCommon", "ResonanceAbsorptionLines", "StandardAtmosphere",
  "Splines", "StatisticalPlots", "Units", "VariationalMethods", "VectorAnalysis",
  "VectorFieldPlots", "WorldPlot", "XML"}, {}
```

```
In[1217]:= Length[%[[1]]]
```

```
Out[1217]= 47
```

Если каталог *d* пуст, вызов процедуры возвращает пустой список, т.е. {}. Предыдущий фрагмент представляет исходный код процедуры с примерами ее применения. Так, в последнем примере возвращается список *каталогов* с пакетами *Mathematica 8.0*. Таким образом, с релизом *Mathematica 8.0* поставляется 47 пакетов различного назначения, которое несложно усматривается из самого названия содержащих их подкаталогов.

В продолжение процедуры **DirFD[d]** процедура **TypeFilesD[d]**, базирующаяся на ней, возвращает список типов файлов, расположенных в каталоге *d*, на файлах данных без расширения имени, возвращая значение "undefined". Следующий простой фрагмент представляет исходный код процедуры **TypeFilesD** с примерами ее применения.

```

In[1222]:= TypeFilesD[d_ /; DirQ[d]] := Block[{a = DirFD[d][[2]], b = {}, c, p, k = 1},
  If[a == {}, {}, For[k, k <= Length[a], k++, c = a[[k]]; p = StringPosition[c, "."];
    b = Append[b, If[p == {}, "undefined", StringTake[c, {Flatten[p][[-1]] + 1, -1}]]];
  DeleteDuplicates[b]]]

```

```
In[1223]:= TypeFilesD["C:\\Program Files\\Maple 11\\BIN.WIN"]
```

```
Out[1223]= {"exe", "ico", "dll", "lib", "bat", "lic", "ini", "properties", "undefined"}
```

Пакет располагает 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 эти функции было бы целесообразно оформлять единым средством, ибо в данной концепции каталоги и файлы во многом отождествляются и их обработка может производиться одними и теми же средствами. Упомянутые стандартные функции и по ограничениям вполне идентичны, а именно: для переименования имени *x* элемента файловой системы на *y* элемент с именем *y* должен отсутствовать, в противном случае возвращается **\$Failed** с выводом соответствующего сообщения. Более того, если в качестве *y* кодируется лишь новое имя без полного пути к новому элементу *y*, то производится его копирование в текущий каталог; в случае каталога *x* он со всем содержимым копируется в текущий каталог под новым именем *y*. Естественно, такая организация достаточно неудобна во многих отношениях, что и побудило нас для переименования каталогов и файлов определить единую процедуру **RenDirFile**[*x*, *y*], которая обеспечивает переименование элемента *x* (каталог либо файл) на месте с сохранением его типа и всех его атрибутов; при этом, в качестве аргумента *y* используется новое имя элемента. Успешный вызов процедуры **RenDirFile**[*x*, *y*] возвращает полный путь к переименованному элементу. В случае существования элемента *y* возвращается полный путь к исходному элементу *x*. Фрагмент представляет исходный код процедуры с примерами ее применения.

```
In[1199]:= RenameFile["C:\\Temp\\Grsu.doc", "Vgtu.doc"]
Out[1199]= "C:\\Documents and Settings\\Aladjev\\My Documents\\Vgtu.doc"
In[1200]:= RenameDirectory["C:\\Temp\\Noosphere Academy", "Rans Ian"]
Out[1200]= "C:\\Documents and Settings\\Aladjev\\My Documents\\Rans Ian"
In[1233]:= RenDirFile[x_;/; FileExistsQ[x] || DirectoryQ[x], y_;/; StringQ[y]] :=
    Block[{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[1234]:= RenDirFile["C:\\Temp\\Kino Online.txt", "Cinema Online.txt"]
Out[1234]= "C:\\Temp\\Cinema Online.txt"
In[1235]:= RenDirFile["C:\\Temp\\Noosphere Academy", "Rans Ian"]
Out[1235]= "C:\\Temp\\Rans Ian"
In[1248]:= RenDirFile["C:\\Temp\\Noosphere Academy", "Noosphere Academy"]
Out[1248]= "c:\\temp\\noosphere academy"
In[1249]:= RenDirFile["C:\\Temp\\Cinema Online.txt", "Cinema Online.txt"]
Out[1249]= "c:\\temp\\cinema online.txt"
```

Дополнительно к **DirFD** определенный интерес представляет и процедура **DirFull**.

```
In[1247]:= DirFull[x_;/; DirQ[x]] := Block[{a = "$ArtKr$", 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[1248]:= DirFull["C:\\Temp\\avz agn vsv"]
Out[1248]= "C:\\Temp\\avz agn vsv", "c:\\temp\\avz agn vsv\\Art Kr 70", "c:\\temp\\avz agn vsv\\Art Kr 70\\GRSU.doc", "c:\\temp\\avz agn vsv\\Art Kr 70\\Rans Ian", "c:\\temp\\avz agn vsv\\Art Kr 70\\Rans Ian\\Vgtu.doc"
In[1249]:= DirFull["C:\\Temp\\Aaa Bbb Ccc"]
Out[1249]= {"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 исходный код **DirFull** с примерами.

Следующая процедура **Attrib2**[*F*, *x*, *y*] во многом подобна процедуре **Attrib1**[*F*, *x*] как в функциональном, так и в описательном отношении, однако **Attrib2** имеет целый ряд отличий. Прежде всего, вызов **Attrib2**[*F*, *x*] возвращает список атрибутов в строчном формате, которые приписаны каталогу либо файлу *F*, тогда как вызов **Attrib2**[*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 с примерами ее применения.

```
In[1228]:= Attrib2[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], 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[1229]:= Mapp[Attrib2, {"C:\\temp\\aaa bbb ccc", "C:\\temp\\g r s u.doc"}, "Attr"]
Out[1229]= {"A", "R"}, {"A", "R"}
In[1230]:= Attrib2["C:\\temp\\aaa bbb ccc\\g r s u.doc", "Attr"]
Out[1230]= {"A", "S", "H", "R"}
In[1250]:= DelDirFile[x_ /; StringQ[x] && DirQ[x] || FileExistsQ[x]] :=
Block[{a = If[DirQ[x], $$$ = DirFull[x]; DeleteDirectory, $$$ = {x}; DeleteFile],
If[Quiet[If[a === DeleteDirectory, a[x, DeleteContents -> True], a[x]]] ==
```

```

$Failed, Mapp[Attrib2, $$$, {}]; Clear[$$$];
If[a === DeleteDirectory, a[x, DeleteContents -> True], a[x], Null]]
In[1251]:= DelDirFile["C:\\temp\\rans ian rac agn"]
In[1252]:= DelDirFile["C:\\temp\\rans ian rac agn/g r s u.doc"]
In[1253]:= DirQ["C:\\temp\\rans ian rac agn"]
Out[1253]= False
In[1254]:= FileExistsQ["C:\\temp\\rans ian rac agn/g r s u.doc"]
Out[1254]= False

```

В отличие от стандартных функций **DeleteFile** и **DeleteDirectory**, достаточно простая процедура **DelDirFile[x]**, завершающая предыдущий фрагмент, удаляет из файловой системы компьютера файл либо каталог *x*, возвращая **Null**, т.е. ничего. Если элемент *x* (каталог или файл) файловой системы имеет атрибут **Read-only**, то перед удалением атрибуты элемента *x* отменяются, обеспечивая корректное удаление элемента *x*, что в отличие от стандартных средств расширяет возможности по удалению элементов *x*. В предыдущем фрагменте представлен исходный текст процедуры **DelDirFile** наряду с примерами ее применения. Следует отметить, что пакет *Mathematica* не располагает стандартными средствами обработки атрибутов файлов и каталогов и предложенные нами средства **Attrib**, **Attrib1** и **Attrib2** в определенной мере заполняют эту нишу.

Следующая процедура обеспечивает поиск в файловой системе ПК либо в файловой системе заданных устройств внешней памяти прямого доступа, заданных именами в строчном формате, полных путей, содержащих подкаталог *x*, заданный именем. При этом, вызов **FindSubDir[x]** возвращает список полных путей в рамках всей файловой системы компьютера, тогда как вызов **FindSubDir[x, y, z, ...]** – рамках лишь файловой системы устройств {*y, z, ...*}. Фрагмент представляет исходный код **FindSubDir** наряду с достаточно типичным примером ее применения.

```

In[1431]:= 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 > ArtKr.txt"];
    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[1432]:= FindSubDir["Addition", "D"]
Out[1432]= {"d:\\new_book\\addition", "d:\\new_book\\addition\\academy",
    "d:\\new_book\\addition\\grinn.zip", "d:\\new_book\\addition\\G.pdf"}

```

В заключение раздела еще раз напомним, все элементы файловой системы ПК должны кодироваться с разделителями, определяемыми переменной **\$PathnameSeparator**, по умолчанию в качестве разделителя полагается обратный слэш "\", кодируемый как двойной. Между тем, в пакете *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 этого неудобства нами создана простая функция *DirQ*, рассмотренная выше. При этом, достаточно простая функция **SetPathSeparator[x]** производит установку для путей к файлам/каталогам в качестве разделителя "\" или "/" на период текущего сеанса с пакетом. Следующий фрагмент представляет исходный код функции с примерами ее применения.

```
In[1069]:= $PathnameSeparator
Out[1069]= "\\"
In[1107]:= SetPathSeparator[x_/: MemberQ[{"/", "\"\""}, x]] :=
            Block[{}, Unprotect[$PathnameSeparator]; $PathnameSeparator = x;
            SetAttributes[$PathnameSeparator, Protected]]
In[1108]:= {SetPathSeparator["/"]; $PathnameSeparator,
            SetPathSeparator["\""]; $PathnameSeparator}
Out[1108]= {"/", "\"\""}
In[1170]:= DirectoryQ["C:/Program Files/Wolfram Research/Mathematica/8.0"]
Out[1170]= True
In[1171]:= DirectoryQ["C:\\Program Files\\Wolfram Research\\Mathematica/8.0"]
Out[1171]= True
In[1172]:= DirectoryQ["C:\\Program Files\\Maple 11\\"]
Out[1172]= True
In[1173]:= DirectoryQ["C:\\Program Files\\Maple 11/"]
Out[1173]= False
In[1174]:= Map[DirQ, {"C:\\Program Files/Maple 11/", "C:/Program Files/Maple 11\\"}]
Out[1174]= {True, True}
In[1197]:= FileDirStForm[x_/: StringQ[x]] :=
            Block[{a = StringReplace[ToLowerCase[x], "/" -> "\"\""]},
            If[StringLength[a] == 3 && StringTake[a, {2, 2}] == ":", a, StrDelEnds[a, "\"\"", 2]]]
In[1198]:= FileDirStForm["D:/AVZ_Package\\AVZ_Package.nb/"]
Out[1198]= "d:\\avz_package\\avz_package.nb"
In[1199]:= Map[FileDirStForm, {"D:/", "C:"}]
Out[1199]= {"d:\\", "c:"}
In[1200]:= StandPath[x_/: PathToFileQ[x]] := Block[{b = "", c, k = 1,
            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}]]
In[1201]:= StandPath["C:/Temp/Academy\\New_Book/GRSU.doc"]
Out[1201]= "c:\\temp\\academy\\new_book\\grsu.doc"
In[1202]:= StandPath["C:/Program Files\\Wolfram Research/Mathematica/8.0"]
Out[1202]= "c:\\\\program files\\\\\\\\wolfram research\\\\\\\\mathematica\\\\8.0"
```

Таким образом, в большинстве случаев подобно *Maple*, для *Mathematica* также можно использовать оба типа разделителей элементов файловой системы, однако сказанное относится *лишь* относительно *Windows XP*, для других платформ возможны отличия, в целом ряде случаев достаточно существенные.

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[1115]:= Write["RANS"]; Close["Rans"]
          General::openx: Rans is not open. >>
Out[1115]= Close["Rans"]
```

Таким образом, корректное использование *имен* файлов и *путей* к ним предполагает, в общем случае, работу с одним и тем же их форматом, как иллюстрирует пример.

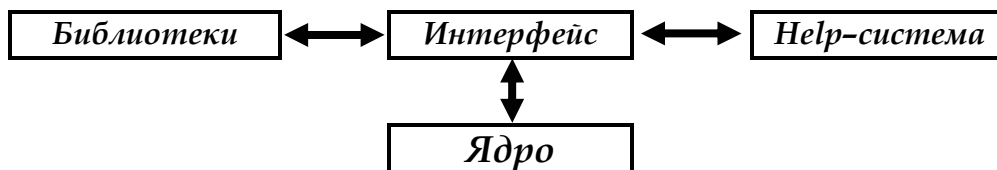
Поэтому в качестве довольно простого приема, позволяющего унифицировать имена файлов/каталогов и путей к ним, можно предложить следующий стандарт – символы, составляющие имена файлов и путей к ним, кодируются на нижнем регистре, тогда как в качестве разделителей используется стандартный "\". Данную задачу успешно решают достаточно простые функции **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, использование 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 D:\\Noosphere Academy\\New Book\\*.*
The filename, directory name, or volume label syntax is incorrect.
In[1197]:= DirFD["D:/Noosphere Academy/New Book"]
Out[1197]= {"Addition"}, {"CoreLanguage.pdf", "GRSU.doc", "New_Book.doc"}}
```

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 пакетом, облегчая программирование целого ряда весьма важных приложений, имеющих дело с файлами данных.

## Глава 10. Организация программного обеспечения пользователя в среде математического пакета Mathematica

Прежде, чем рассматривать организацию программного обеспечения пользователей, целесообразно ознакомиться со структурой файловых систем пакетов, которые могут быть, в целом, с достаточной степенью общности представлены следующей схемой.



Данная схема является общей для всех современных *систем компьютерной алгебры (СКМ)*, наиболее отчетливо просматриваемой в таких наиболее известных и широко применяемых системах, как пакеты *Maple* и *Mathematica*. Именно относительно этих двух пакетов и рассмотрим вкратце наполнение представленной 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 пакета и за представление его структур данных. Ядро обеспечивает также 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 пакета 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 визуализации различных *Maplet*-приложений.

Однопоточное выполнимое ядро на *Windows*-платформе – «*mserver.exe*» («*mserver*» на всех других платформах), и *SMP*-допустимое выполнимое ядро – «*mtserver.exe*» (или «*mtserver*» на всех других платформах). Одно из данных ядер стартует автоматически в фоновом режиме при запуске пакета *Maple*, общаясь с интерфейсом пользователя по гнезду *TCP/IP*. Данная связь безопасна, ибо она является локальной для компьютера, на котором выполняется пакет и к ней нельзя получить доступ с других компьютеров сети без особых хакерских ухищрений.

Основные принципы, заложенные при проектировании ядра пакета *Maple*, в полной мере наследуются и ядром пакета *Mathematica*, однако их реализации различаются и

характеризуются специфическими чертами каждого из пакетов, наряду с вкусами и пристрастиями разработчиков. В частности, ядро *Mathematica* реализовано языком C++, зарекомендовавшим 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 любого пакета является требование их высокой реактивности, то функции, как правило, выполняются эффективным экстракодом относительно небольшого объема, а само ядро 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 тем либо иным пакетом. Так, для *Maple* документы в файлах форматов {*txt, mpl, mws, mw*}, тогда как для пакета *Mathematica* в форматах {*m, nb, nbp*}, которые обеспечивают загрузку документов в текущий сеанс на выполнение. Данные документы наряду с библиотеками пользователя пишутся на встроенном языке того либо иного пакета, что делает возможным их подготовку обычными пользователями. Более того, расширение и функциональных, и прикладных возможностей пакетов на основе библиотек и документов не ограничено практически ничем. В частности, для пакета *Maple* характерна возможность создания библиотек пользователей, имеющих организацию как аналогичную организации главной *Maple*-библиотеке пакета, так и отличных от нее. Так, сама главная библиотека *Maple* содержит определения средств в количестве, зависящем от релиза пакета (*Maple 8 - 22501, Maple 10 - 27986, Maple 11 - 32883*), тогда как наша библиотека для релизов 6 - 11 содержит чуть более 850 средств, а наш пакет *AVZ\_Package* для *Mathematica* включает более 320 процедур и функций; при этом, все стандартные библиотеки пишутся, как правило, на встроенных языках, тогда как наши средства для обоих пакетов 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 того или иного пакета 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 самого пакета. В таком случае вопросы организации пользовательского программного обеспечения принимают специфический характер, определяемый собственно средствами пакета, предназначенными для этих целей. В этом контексте и попытаемся сравнить пакеты

*Maple* и *Mathematica*. Естественно, для пользователя, использующего пакет лишь на внешнем уровне, т.е. в виде написания документов, решающих отдельные задачи, или использующих пакет в качестве интеллектуального калькулятора для вычисления ряда математических выражений или создания на их основе графических объектов, организация программных средств не носит столь актуального характера, однако для пользователя, разрабатывающего достаточно серьезные проекты на основе пакета с использованием средств, отсутствующих в пакете или недостаточно эффективных и требующих дополнительного программирования, эта проблема довольно актуальна.

## 10.1. Организация программного обеспечения пользователя в среде математического пакета *Mathematica*

Пакет *Mathematica* не располагает средствами организации библиотек пользователя подобно случаю пакета *Maple*, создавая определенные затруднения при организации программного обеспечения, созданного пользователем в его 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 пакета, 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 пакета. 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 «>>>» или «>>>>» определения *Mathematica*-объектов сохраняются в файлах во входном формате пакета *Mathematica*, обеспечивая весьма простое чтение их обратно в среду пакета. Для сохранения же 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 пакета 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 в виду, 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 пакета (*бинарный mx-файл*), наиболее оптимальный для считывания пакетом. 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**.

Между тем, здесь целесообразно сделать целый ряд весьма существенных замечаний по использованию указанных средств пакета. 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 математических пакетов. В конкретном случае мы для этой цели использовали функции **Import** и **ToExpression** пакета.

Во-вторых, механизм обработки ошибочных и особых ситуаций представляет собой достаточно мощный инструмент программирования практически каждого довольно сложного алгоритма. Однако, в пакете *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 данный механизм в целом не работает, как иллюстрирует ниже следующий достаточно простой фрагмент, а именно:

```
In[2218]:= Import["D:\\Math_myLib\\ArtKr.m"]
          Import::nffil: File not found during Import. >>
Out[2218]= $Failed
In[2219]:= Off[Import::nffil]
In[2220]:= Import["D:\\Math_myLib\\ArtKr.m"]
Out[2220]= $Failed
In[2221]:= On[Import::nffil]
```

```
In[2222]:= F[x_] := Module[{a}, Off[Import::nffil]; a := Import[x]; On[Import::nffil]; a]
In[2223]:= F["D:\\Math_myLib\\ArtKr.m"]
Import::nffil: File not found during Import. >>
Out[2223]= $Failed
```

Таким образом, при создании собственных достаточно *сложных* процедур, в которых требуется решать вопросы блокировки вывода ряда ошибочных сообщений, средства пакета *Mathematica*, на наш взгляд, недостаточно развиты. Прежде всего это касается создания сложных процедур и библиотек пользовательских программных средств.

Принимая во внимание схему организации библиотеки пользовательских процедур, рассмотренную в наших изданиях [97–99] относительно пакета *Maple* с отличной от главной библиотеки организацией, представим реализацию подобной библиотеки пользователя для случая пакета *Mathematica*. Для этого, на первом шаге в файловой системе компьютера создается каталог, например, "D:\\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[1942]:= NF[x_] := Sin[x]*Cos[x]
In[1943]:= 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[1944]:= Save["D:\\Math_myLib\\NF.txt", NF]
In[1945]:= Save["D:\\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, например, *Notepad*.

Ниже приведен пример простой процедуры **CALL**, обеспечивающей возврат вызова процедуры/функции  $G[x, y, \dots, z]$  при условии, что ее определение находится в файле "G.txt" библиотечного каталога "D:\\Math\_myLib". Если файла с таким именем нет, то вызов процедуры  $G[x, y, \dots, z]$  возвращается невычисленным, хотя и производится вычисление и упрощение фактических аргументов функции в полном соответствии с соглашениями пакета. Приведенные примеры достаточно наглядно иллюстрируют различные случаи применения процедуры **CALL[x]** для вызовов процедур.

```
In[1771]:= CALL[x_] := Module[{b, c, d, h, t},
    b = ToString[x, InputForm]; c = StringPosition[b, "[";
    If[Length[c] == 0, Return[x], t = "D:\\Math_myLib\\";
```

```

If[Length[c] != 0, d = StringTake[b, First[First[c]] - 1], Null];
If[FindFile[t <> d <> ".txt"] == $Failed, Return[x], Null];
h := ToExpression[Import[t <> d <> ".txt", "Text"]]; h; x];

In[1772]:= CALL[42.68]
Out[1772]= 42.68
In[1773]:= CALL[Sin[Pi/2]]
Out[1773]= 1
In[1774]:= Directory[]
Out[1774]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[1775]:= CALL[AVZ[42, 47, 67, 89, 96]]
Out[1775]= AVZ[42, 47, 67, 89, 96]
In[1776]:= CALL[Log[68]*Cos[42.47]]
Out[1776]= 0.246697
In[1777]:= CALL[NF[68.42]]
Out[1777]= -0.491857
In[1778]:= CALL[ArtKr[68.42, 63.47]]
Out[1778]= 7.81774
In[1779]:= CALL[KrArt[68.42, 63.47]]
Out[1779]= KrArt[68.42, 63.47]
In[1780]:= CALL[NF[z]]
Out[1780]= Cos[z]*Sin[z]
In[1781]:= CALL[ArtKr[x, y]]
Out[1781]= Sqrt[47*Cos[x] + 42*Sin[y]]
In[1782]:= CALL[ArtKr[42.68, 47.63]]
Out[1782]= 0. + 2.80933 I

```

Для настройки пакета под свои нужды пользователь имеет возможность размещать в его каталогах специальные файлы, загружаемые в текущий сеанс либо при загрузке пакета, либо при 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 пакета. В частности, файлы **"init.m"** конфигурации ядра пакета могут содержать любые команды *Mathematica*. Такие предопределенные **\$**-переменные имеют *Protected*-атрибут, поэтому для их переопределения требуется предварительно отменить данный атрибут с последующим его восстановлением. В нашем случае интерес представляют две предопределенные переменные, а именно:

**\$BaseDirectory** – определяет основной подкаталог пакета, в который помещаются все системные файлы, автоматически загружаемые при запуске *Mathematica*;

**\$UserBaseDirectory** – определяет основной подкаталог пакета, в который помещаются все пользовательские файлы, автоматически загружаемые при запуске *Mathematica*.

Для нашего случая данные **\$**-переменные имеют следующие значения, а именно:

```

In[2211]:= $UserBaseDirectory
Out[2211]= "C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica"

```

```
In[2212]:= $BaseDirectory
```

```
Out[2212]= "C:\\Documents and Settings\\All Users\\Application Data\\Mathematica"
```

Поэтому для удобства последующего использования процедуры **CALL** помещаем ее исходный текст в файл **"init.m"**, размещаемый в одном из указанных каталогов. Этим мы обеспечиваем автоматическую загрузку файла при каждом новом запуске пакета, активируя в текущем сеансе процедуру **CALL**. В простейшем случае его содержимое принимает следующий вид, а именно:

```
(** User Mathematica initialization file **)
CALL[x_]:= Module[{b, c, d, h, t}, b = ToString[x, InputForm]; c = StringPosition[b, "["];
      If[Length[c] == 0, Return[x], t = "D:\\Math_myLib\\"];
      If[Length[c] != 0, d = StringTake[b, First[First[c]] - 1], Null];
      If[FindFile[t <> d <> ".txt"] == $Failed, Return[x], Null];
      h:= ToExpression[Import[t <> d <> ".txt", "Text"]]; h; x];
```

которое легко модифицируется любым текстовым редактором, например, *Notepad*.

Представленный выше простой подход к организации средств пользователя в среде пакета *Mathematica* является лишь одним из возможных, предоставляя возможность создания собственных библиотек процедур/функций с доступом на уровне пакетных средств. Действительно, довольно несложно поместить исходный текст отложенной процедуры в *txt*-файл с именем процедуры, сохранить его в выбранном подкаталоге библиотеки пользователя и поместить исходный текст процедуры **CALL** в «*init.m*», который находится (*размещается*) в одном из каталогов, определяемых переменными **\$BaseDirectory** и **\$UserBaseDirectory**. После чего каждая загрузка пакета *Mathematica* делает доступными в текущем сеансе все процедуры/функции, определения которых находятся в файлах библиотечного каталога пользователя, наравне со встроенными средствами пакета, посредством вызовов процедуры **CALL**, например:

```
In[1447]:= AGN = Sqrt[CALL[NF[42.47]]^2 + CALL[ArtKr[19.89, 19.96]]^3]
```

```
Out[1447]= 21.935
```

Как уже отмечалось выше, один такой *txt*-файл может содержать определения ряда процедур, однако должен быть именован одной из процедур, находящихся в файле. Тогда первый же вызов данной процедуры активирует в текущем сеансе пакета все процедуры, находящиеся в файле. После этого обращаться к ним возможно уже без использования процедуры **CALL**, т.е. традиционным для пакета способом.

Отметим, что процедура **CALL** может быть расширена по различным направлениям, например, может иметь два формальных аргумента, второй из которых определяет полный путь к каталогу файловой системы компьютера, содержащему *txt*-файлы с определениями процедур пользователя. Представляет также интерес модификация процедуры **CALL** и 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 (<<)**; более того, *mx*-формат обладает весьма существенным свойством: в отличие от *m*- либо *txt*-формата **Get**-функция позволяет

загружать и активировать в текущем сеансе определения объектов, содержащихся в загружаемых *mx*-файлах, безотносительно точки их выхода - в *Input*-режиме либо в теле процедуры. Именно это свойство, порой, существенно упрощает использование вызовов **Get**-функции в теле процедур, базирующихся на активации определений в текущем сеансе пакета, находящихся в загружаемых *mx*-файлах данных.

```
In[1]:= CALL[x_]:=Module[{b, c, d, t}, b=ToString[x, InputForm]; c=StringPosition[b, "["];
      If[Length[c] == 0, Return[x], t = "D:\\Math_myLib\\"];
      If[Length[c] != 0, d = StringTake[b, First[First[c]] - 1], Null];
      If[FindFile[t <> d <> ".mx"] == $Failed, Return[x], Null];
      Get[t <> d <> ".mx"]; x];

In[2]:= CALL[42.68]
Out[2]= 42.68
In[3]:= CALL[Sin[Pi/2]]
Out[3]= 1
In[4]:= Directory[]
Out[4]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[5]:= CALL[AVZ[42, 47, 67, 89, 96]]
Out[5]= AVZ[42, 47, 67, 89, 96]
In[6]:= CALL[Log[68]*Cos[42.47]]
Out[6]= 0.246697
In[7]:= CALL[NF[68.42]]
Out[7]= -0.491857
In[8]:= CALL[ArtKr[42.68, 47.63]]
Out[8]= 0. + 2.80933 I
In[9]:= AGN = Sqrt[CALL[NF[42.47]]^2 + CALL[ArtKr[19.89, 19.96]]^3]
Out[9]= 21.935
```

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 такие процедурные файлы.

В качестве достаточно простого примера ведения пользовательских библиотек можно представить процедуру **UserLib[L, S]**, поддерживающую ряд полезных функций.

```
In[1822]:= 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]]],
```



**{"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 пакета 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 *невычисленным*. Имеется хорошая возможность расширить процедуру рядом полезных функций такими, как: удаление из библиотеки определений указанных средств или старых их версий и т.д. 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.

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, однако назвать их эффективными в среде пакета нам не представляется возможным. В качестве одного из них можно предложить создание *hb*-файлов (*Helpbase*), содержащих описания всех процедур библиотеки пользователя и находящихся в библиотечном каталоге. Такие файлы могут загружаться в текущий сеанс по вызову **Help[Proc]**, выводя справочную информацию по процедуре/ функции с именем *Proc*. Исходный код такой простой **Help**-процедуры и пример ее типичного применения приводятся ниже, а именно:

```
In[1425]:= Help[x_] := Module[{}, FilePrint["D:\Math_myLib/" <> ToString[x] <> ".hb"]]
```

```
In[1426]:= Help[ArtKr]
```

ArtKr[x, y] – function from 2 formal arguments; as its actual arguments can be any correct expressions, namely: ArtKr[x, y]:= Sqrt[47\*Cos[x] + 42\*Sin[y]]

Ради удобства исходный текст процедуры **Help** подобно вышеописанному случаю с процедурой **CALL** помещается в инициализационный файл **«init.m»**, обеспечивая в каждой новой сессии *Mathematica* доступ к этим процедурам на уровне встроенных средств пакета. Исходные тексты обоих процедур выводятся в **Input**-режиме запросом **??{CALL | Help}**. Естественно, предложенный простой способ организации справочной базы для библиотеки пользователя далеко не единственный. Используя стандартные средства пакета можно создавать намного более развитые средства для обеспечения пользовательских средств справками. Однако это требует определенных усилий. Так, возможно создавать справочные базы на основе *nb*-файлов (*Notebooks*), оформляемых соответствующим образом. При этом, для всей библиотеки можно оформить только один справочный *nb*-файл, по мере необходимости вызывая в текущий сеанс только информацию, относящуюся к запрашиваемому средству. Для обеспечения доступа к таким справочным *nb*-файлам их необходимо размещать в специальных каталогах файловой системы пакета. Типичными здесь выступают подкаталоги, определяемые предопределенными переменными **\$InstallationDirectory** и **\$UserBaseDirectory**. Эти *nb*-файлы помещаются в каталог **\$UserBaseDirectory <> "\\Documentation/Language"** либо в каталог **\$InstallationDirectory <> "\\Documentation/Language"**, где «**Language**» определяется предопределенной переменной **\$Language**, имеющей по умолчанию значение «**English**» – одно из распознаваемых пакетом **{English, French, German}**. Между тем, средства пакета позволяют создавать и размещать справочные *nb*-файлы в любом подкаталоге файловой системы компьютера, для чего требуется запрограммировать соответствующий алгоритм обработки такой справочной базы, базирующийся как на собственно самой задаче поиска нужной справки, так и на средствах *Mathematica* для работы с файлами *nb*-формата (*Notebooks*). Мы на этом вопросе не останавливаемся. 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.

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. Пакет *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 пакетом, 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 пакетом в связи с их востребованностью по мере необходимости, возникающей в целом ряде случаев 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 и содержат ошибки. Наш опыт работы с обоими анализируемыми пакетами подтверждает сказанное. В качестве примера можно привести следующий. Несколько лет тому назад на ряде форумов по *Mathematica* и *Maple* обсуждалась возможность дифференцирования произвольного выражения не по переменной, а по некоторому подвыражению первого (например, на форуме *mapleprimes* разработчиков пакета *Maple*; [www.mapleprimes.com/search/?term=Diff](http://www.mapleprimes.com/search/?term=Diff)). Так, для дифференцирования выражений пакеты *Maple* и *Mathematica* располагают стандартными процедурой *diff* и функцией *D* соответственно, как это иллюстрирует следующий достаточно простой фрагмент:

Maple 11:

```
> diff(x*sin(x) + cos(x), x);           ⇒      x cos(x)
> diff(x*sin(x) + cos(x)/x^3, x^3);
Error, invalid input: diff received x^3, which is not valid for its 2nd argument
> diff(x*sin(x) + cos(x), x);           ⇒      x cos(x)
> diff((x*sin(x) + cos(x))/x^3, x^3);    ⇒      -cos(x)/x^6
```

Mathematica 8:

```
In[1098]:= D[x*Sin[x] + Cos[x], x]
Out[1098]= x Cos[x]
In[1099]:= D[x*Sin[x] + Cos[x]/x^3, x^3]
General::ivar: x^3 is not a valid variable. >>
Out[1099]= ∂x3 (Cos[x]/x3 + x Sin[x])
In[1100]:= Df[x*Sin[x] + Cos[x]/x^3, x^3]
Out[1100]= - Cos[x]/x6
```

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. Далее представлены исходные коды для данных процедур в *Mathematica*, именованные *Df*, с некоторыми типичными примерами их применения для дифференцирования произвольных выражений.

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 ниже 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 выражений.

```
In[2247]:= Df[x_, y_] := Module[{ArtKr, AvsAgn, Svetla, Ian, Rans},
    If[NumberQ[x], Return[0], If[NumberQ[y],
    Return[Print[Df::"incorrect second argument ", {y}]],
    AvsAgn = Denominator[x]]; Svetla := x;
    If[AvsAgn == y, Svetla = Numerator[x]/(y + Rans), Null];
    ReplaceAll[D[ReplaceAll[Svetla, y -> Ian], Ian], {Ian -> y, Rans -> 0}]]]]

In[2248]:= D[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]]
Out[2248]= Sin[x]/Sqrt[Cos[x] + Sin[x]^2]
In[2249]:= Df[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]]
Out[2249]= Sin[x]/Sqrt[Cos[x] + Sin[x]^2]
In[2250]:= D[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]^2]
General::ivar: Sin[x]^2 is not a valid variable. >>
Out[2250]=  $\partial_{\sin[x]^2} \sqrt{\cos[x] + \sin[x]^2}$ 
In[2251]:= Df[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]^2]
Out[2251]= 1/(2 Sqrt[Cos[x] + Sin[x]^2])
In[2252]:= Df[(Cos[x]^2 + x*Sin[x]^2)/Cos[x]*Sin[x]^2, Sin[x]^2]
Out[2252]= x*Sin[x]*Tan[x]
In[2253]:= D[(Cos[x]^2 + x*Sin[x]^2)/Cos[x]*Sin[x]^2, Sin[x]^2]
General::ivar: Sin[x]^2 is not a valid variable. >>
Out[2253]=  $\partial_{\sin[x]^2} (\sin[x] (\cos[x]^2 + x \sin[x]^2) \tan[x])$ 
In[2254]:= D[x, 5]
General::ivar: 5 is not a valid variable. >>
Out[2254]=  $\partial_5 x$ 
In[2255]:= Df[x, 5]
Df::incorrect second argument {5}
In[2256]:= Df[5, x]
Out[2256]= 0
In[2257]:= D[5, x]
Out[2257]= 0
In[2258]:= Df[(x^2*Cos[x]^2 + x*Sin[x]^2)/(Log[x^2] - x^2*Cos[x]*Sin[x]^2), x^2]
Out[2258]=  $\cos[x]^2 (-x^{5/2} + \log[x^2]) - x^{3/2} \sin[x]^2 + x \cos[x] \sin[x]^4 / (\log[x^2] - x^2 \cos[x] \sin[x]^2)^2$ 
In[2259]:= Df[x_, y_] := Module[{a, b, c, h}, If[NumberQ[x], Return[0], If[NumberQ[y],
    Return[Print[Df::"incorrect second argument ", {y}]], Null];
    {{a, b}, h} = {Map[ToString, Map[InputForm, {c*x, y}]], "h"};
    Simplify[ReplaceAll[D[ToExpression[StringReplace[a, b -> h]],
    Symbol[h]]/c, Symbol[h] -> y]]]]
```

```

In[2260]:= Df[(x^2*Cos[x]^2 + x*Sin[x]^2)/(Log[x^2] - x^2*Cos[x]*Sin[x]^2), x^2]
Out[2260]= (Cos[x]^2 (-x^(5/2) + Log[x^2]) - x^(3/2) Sin[x]^2 + x Cos[x] Sin[x]^4)/(Log[x^2] -
x^2 Cos[x] Sin[x]^2)^2
In[2261]:= Df[(x^2*Cos[x]^2 + x*Sqrt[x^2 + Sin[x]])/(Log[x^2] - Sqrt[x^2 + Sin[x]]),
Sqrt[x^2 + Sin[x]]]
Out[2261]= (x (x Cos[x]^2 + Log[x^2]))/(Log[x^2] - Sqrt[x^2 + Sin[x]])^2
In[2262]:= {Df[2010, x^2], Df[x^2, x^2], Df[x^2, 420], Df[72, 420]}
Out[2262]= {0, 1, 0, 0}
In[2263]:= Df[(a*Sqrt[x + Cos[x]] + b*Cos[x])/(d + Cos[x]), Cos[x]]
Out[2263]= (a (d - 2 x) - a Cos[x] + 2 b d Sqrt[x + Cos[x]])/(2 (d + Cos[x])^2 Sqrt[x + Cos[x]])
In[2264]:= Simplify[D[(a*Sqrt[x + Cos[x]] + b*Cos[x])/(d + Cos[x]), Cos[x]]]
Out[2264]= (a (d - 2 x) - a Cos[x] + 2 b d Sqrt[x + Cos[x]])/(2 (d + Cos[x])^2 Sqrt[x + Cos[x]])
In[2265]:= Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]

```

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 пакетом *Maple*, так и *Mathematica* посредством создания библиотеки для *Maple* [45] и пакета для *Mathematica* [90]. Но к сожалению, далеко не всегда данный подход оказывается достаточно продуктивным, хотя и весьма необходимым для расширения программной среды обоих пакетов.

Итак, пользовательские средства для рассматриваемых пакетов 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 пакета. 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 пакета, используя его возможности. Этот механизм представлен довольно детально в справке по пакету и вполне понятен для пользователей *Mathematica*.

Говоря об организации пользовательских библиотек в пакете *Mathematica*, нельзя не 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 пакета 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 пакета.

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 пакета *Mathematica*, где аргумент *targs* определяет список допустимых типов аргументов вызываемой функции и *rtype* определяет тип возвращаемого функцией *F* результата; загруженная таким способом *F*-функция становится доступной в текущем сеансе пакета;

**LibraryLoad**[*L*] – загружает в ядро пакета динамическую библиотеку *L*, не активируя какую-либо из содержащихся в ней функций;

**LibraryFunctionInformation**[*F*] – возвращает справочную информацию по *F*-функции, загруженной в текущий сеанс пакета вызовом функции **LibraryFunctionLoad**;

**LibraryFunctionUnload**[*F*] – выгружает из памяти компьютера библиотечную функцию *F*, после чего функция становится недоступной в текущем сеансе пакета;

**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 пакета вызовом функции **LibraryFunctionLoad**; при этом, поиск такой библиотеки производится согласно информации глобальной переменной **\$LibraryPath**, значение которой определяет список подкаталогов пакета, 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 переменной **\$LibraryPath**.

```
In[516]:= $LibraryPath
Out[516]= {"C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica\\
SystemFiles\\LibraryResources\\Windows",
=====
"C:\\Program Files\\Wolfram Research\\Mathematica\\8.0\\SystemFiles\\
Links\\TetGenLink\\LibraryResources\\Windows"}
In[517]:= Attributes[$LibraryPath]
Out[517]= {ReadProtected}
In[518]:= ModLibraryPath[x_;/; DirQ[x]] := Module[{a = $LibraryPath,
                                                    b = Attributes[$LibraryPath]},
ClearAttributes[$LibraryPath, b]; a = Insert[a, "D:\\Math_MyLib", -1];
$LibraryPath = a; SetAttributes[$LibraryPath, b]]
```

```

In[519]:= ModLibraryPath["D:\\Math_MyLib"]; $LibraryPath
Out[519]= {"C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica\\
SystemFiles\\LibraryResources\\Windows", ...
=====
"C:\\Program Files\\Wolfram Research\\Mathematica\\8.0\\SystemFiles\\
Links\\TetGenLink\\LibraryResources\\Windows", "D:\\Math_MyLib"}
In[520]:= Attributes[$LibraryPath]
Out[520]= {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 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 пакета. В настоящее время обеспечена полная поддержка выгрузки динамических библиотек на платформах *Windows* и *OS X 10.5*. Детально с принципами механизма **LibraryLink** подключения библиотек к пакету *Mathematica* можно ознакомиться в документации.

Наряду с механизмом **LibraryLink** подключения библиотек пакет *Mathematica* имеет и другие механизмы связи с *внешними* по отношению к пакету 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 пакетом *Mathematica*. Детально с механизмом **MathLink** также можно ознакомиться в справке по пакету. Но вопросы поддержки библиотек средствами пакета нами здесь не рассматриваются.

И еще на одном очень немаловажном аспекте следует акцентировать ваше внимание. Пакет *Maple* является намного более открытой системой как ввиду ранее сказанного, так и по причине доступности для пользователя исходных текстов средств из главной библиотеки пакета. Такая возможность весьма важна по двум причинам: (1) освоение

программистской технологии, используемой пакетом, (2) возможность модификации средств пакета на предмет устранения возможных ошибок (*коих в пакетах немало*) или их расширения. Тогда как в случае пакета *Mathematica* с подобными вопросами дело обстоит значительно сложнее. Для организации пользовательских библиотек нужно создавать собственные средства (*простейший пример приведен выше*), которые в полной мере нельзя считать интегрированными в пакет. Тогда как на уровне документов оба пакета по возможностям создания средств пользователя, практически, эквивалентны. Более того, в пакете *Mathematica* исходные тексты 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 пакета 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 был изначально ориентирован пакет, а именно на область приложений CAS. В частности, в пакет 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 такую работу.

Таким образом, с учетом сказанного, а также опыта использования обоих пакетов мы отдаем в этом вопросе предпочтение пакету *Maple*, который располагает удобными и эффективными средствами по организации *массовых* пользовательских программных средств, полностью интегрированных в функциональном и в справочном отношении в среду пакета, 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 пакетами.

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].

## 10.2. Программирование больших систем в среде Mathematica

Построение достаточно больших систем программного обеспечения в таких пакетах, 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 обоих пакетов – среде языков *Mathematica* и *Maple*. И первый, и второй пакет довольно эффективно используют *императивную* парадигму программирования, типичную для наиболее массово используемых языков и систем программирования, в то время как первый эффективно использует и *функциональную* парадигму, весьма существенно расширяющую возможности его 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 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 инструментальный 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 пакета *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 пакета *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 пакета. Более того, немало из них были в дальнейшем включены в виде аналогов в последующие релизы пакета. К сожалению, зачастую это были, порой, весьма запоздалые решения, а в целом ряде случаев имели и 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 пакет располагал 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 пакета выявился ряд довольно существенных его 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 пакета.

Подобная методика расширения программной среды была предложена и для пакета *Mathematica* созданием *nb*-документа, содержащего более 320 процедур и функций, которые наряду со стандартными зачастую используют и средства самого документа, предполагая для получения полноценного доступа к ним загрузку соответствующего *nb*-файла в текущий сеанс или обеспечение контекстной связи с *m*-файлом. В основу проектирования был положен наш опыт использования и разработки программных систем в других программных и операционных средах, а также в среде пакета *Maple*, однотипного с пакетом *Mathematica*. 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 *Math*-документа [90].

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 запрограммировать систему коллективом программистов различного уровня под руководством системных аналитика и программиста. Немало найдется довольно опытных 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 *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 пакета с обеспечением непосредственного доступа к ним. Имеется ряд других механизмов организации программной системы пользователя, включая и сочетания в довольно широких пределах всех вышеперечисленных механизмов.

Со своей стороны, пакет *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 пакета *Mathematica*, обладающим достаточно 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 пакета. Более того, 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 данного пакета.

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 весьма важны для разработчиков. По данной причине пакет *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 пакет *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*, пакет предоставляет ряд полезных средств (документ, рабочее место). В то же самое время рабочая документация собственно по самому пакету 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 пакета 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 пакетом, что весьма важно осваивающему программирование в его 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 обоих пакетов, а многочисленные ошибки, зачастую, не устранимые из версии в версию, не делают чести разработчикам, в целом, достаточно полезных средств. Между тем, оба пакета предоставляют достаточно 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 *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 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 клиекта 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.

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 исходного кода 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. И в этом отношении пакет *Maple* достаточно неустойчив 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 пакетом. Наш опыт довольно разностороннего использования пакета *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 пакет не располагает.

С другой стороны, пакет *Mathematica* достаточно эффективно управляет памятью, на каждом шаге собирая «муссор» и освобождая от него 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[1793]:= MemoryInUse[]
```

```
Out[1793]= 20320708
```

```
In[1794]:= 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 = 75}, (x^2 + y)/(a + b)]
```

```

In[1795]:= MemoryInUse[]
Out[1795]= 20339892
In[1796]:= Share[]
Out[1796]= 1702208
In[1797]:= MemoryInUse[]
Out[1797]= 18639260
In[2362]:= 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[2363]:= Headings[M]
Out[2363]= {"M[x_ /; x === \"avz\", y_]", "M[x_ /; x == \"avz\"]",
      "M[x_ /; IntegerQ[x], y_String]", "M[x_ /; ListQ[x], y_]",
      "M[x_, y_]", "M[x_String]", "M[x_, y_, z_]"}
In[2364]:= MemoryInUse[]
Out[2364]= 19957780
In[2365]:= Remove[M, L]; Share[]
Out[2365]= 61947

```

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 обоих пакетов, большее предпочтение для этих целей мы отдаем пакету *Mathematica*, который 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 пакета к другой. Тогда как в случае с пакетом *Maple* достаточно 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 модификации с ростом номеров релизов.

## 10.3. 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 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 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 его 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 (*Maple-язык* и *Math-язык*);
- библиотеки различного назначения и организации;
- стандартные процедуры вычислений и обработки информации;
- 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 *ПС*, оба пакета попеременно выходят в лидеры по той либо иной группе составляющих их функций, зачастую беря пример со своего конкурента. Между тем, даже на сегодня оба пакета имеют свои «коньки», по которым они превосходят конкурента. В частности, если пакет *Maple* располагает лучшими на сегодня средствами для аналитического решения диффуравнений, то *Mathematica* в определенной мере превосходит *первый* в задачах интегрального исчисления. В то же время оба пакета просто нашпигованы различного рода ошибками, недоработками и другими неприятными моментами, пожалуй, в намного большем количестве, чем *ОС Microsoft* и *Linux* вместе взятые, и в намного большем, чем можно было бы ожидать от современной системы компьютерной алгебры. Как правило, появляющиеся все новые версии обоих пакетов не только не устраняют большинство застарелых ошибок, но и добавляют немало новых, вызывающих немало раздражений у пользователя. Именно по этой причине пользователь вынужден *своими* силами устранять некоторые из них. Так, в случае с пакетом *Maple* наша Библиотека [45] преследовала, в первую очередь, именно такие цели, предоставив целый ряд средств как устраняющих ошибки *Maple*, так и расширяющих набор стандартных средств или их функциональность. Нечто в подобном роде можно сказать и о нашем небольшом пакете [90] для *Mathematica*, но в гораздо меньшей степени, т.к. основная его цель – иллюстрация к настоящей книге, однако и пользователь пакета сможет найти целый ряд полезных для себя средств.

В этом контексте представляется достаточно интересным и проект по сбору ошибок в пакете *Maple*, которыми он изобилует подобно *Mathematica* [98,99]. Неплохо было бы адаптировать его также на пакет *Mathematica*. Из 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 указанных средств. В противном же случае данные базы ошибок наряду с многочисленными ошибками, анонсированными на соответствующих форумах, так и останутся гласом вопиющего в пустыне, не оказывая влияния на работу по их устранению в последующих релизах

указанных средств. К сожалению, пока картина нам представляется именно такой. И опыт нашего сотрудничества с *MapleSoft* по подготовке и написанию книг по пакету *Maple*, выявлению *весьма* принципиальных ошибок, устранению их разработанными нами средствами, значительная часть которых помещена в нашу библиотеку [45], не оказали сколько-нибудь серьезного влияния на активность разработчиков пакета по устранению ошибок, зачастую, переходящих из релиза в релиз; между тем, как вместо такой важнейшей работы по доведению до кондиции функционирующего средства с завидной частотой появляются все новые его релизы как с дополнительными новыми ошибками, так и с немалым уровнем несовместимости между релизами. В частности, на наш взгляд, недопустимым является использование разных текстовых сообщений, иницируемых возникающими одними и теми же ошибочными ситуациями, что не позволяет корректно их обрабатывать уже созданными программными средствами в среде *Maple* предыдущих релизов, 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 релизах пакета *Maple*. Но в принципе, фирмы-разработчики помимо отделений маркетинга должны включать и отделения тестирования (*типа ОТК*) своей продукции, которые на основе результата своей работы вкупе с откликами пользователей должны осуществлять обратную связь с непосредственными разработчиками программного продукта с целью устранения выявленных недостатков. К сожалению, либо данная служба отсутствует, либо носит в значительной степени формальный характер – ее результаты не прослеживаются.

Между тем, можно сколь угодно критиковать *оба* пакета 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 обоих пакетов будет все же налажена должным образом и будет серьезно учитывать в своей работе *рекламации* пользователей и соответствующим образом доводить *оба* пакета «до ума». На сегодня же мы видим лишь гонку релизов пакетов, обусловленную не столько настоятельной необходимостью, сколько сугубо коммерчески-конкурентными соображениями.

Немалую роль в степени продвижения обоих пакетов на рынке программных средств помимо их возможностей в той или иной области играет служба маркетинга пакетов. Так, служба маркетинга *Mathematica* работает намного агрессивнее и эффективнее, чем аналогичная служба пакета *Maple*. Судя же по качеству обоих пакетов у *первого* в данной службе работает сотрудников, не менее, чем разработчиков, и квалификация их повыше, и платят им поболее. Неодинаково и отношение разработчиков пакетов к их многочисленным пользователям; так, если сетевая конференция для пакета *Maple comp.soft-sys.math.maple* не модерировается, в то время как аналогичная конференция *comp.soft-sys.math.mathematica* не только модерировается, но также и цензурируется в том отношении, что фильтруются посты, с относительно резкой критикой пакета. В плане поддержки своих пользователей от обоих пакетов следовало бы ожидать много большего, между тем, многочисленные пользователи *Maple* получают превосходные

помощь и поддержку от большого числа пользователей пакета на упомянутой выше конференции и специальных форумах пользователей. Между тем, сравнивая число постов по обоим пакетам на обоих конференциях и на ряде других форумов по этим пакетам, с большой долей уверенности можно сказать, что пользователи *Mathematica* имеют от него намного больше неприятностей, чем пользователи пакета *Maple*. Более того, из нашего личного опыта можно сделать вывод, что пакет *Maple* намного более дружелюбен к своим пользователям, чем пакет *Mathematica*. И между тем, не взирая на многочисленную критику в адрес обоих пакетов, 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 обоих пакетов на предмет их предпочтительности в разработке тех либо иных пользовательских средств. Как показывает наш опыт использования данных пакетов в различных приложениях, практически, любая задача решается в среде любого из них с той или иной степенью эффективности. Именно в контексте использования данных пакетов в качестве среды программирования наиболее массовых задач, прежде всего, оформляемых в виде процедур и функций, и лежит круг интересов данной книги.

В качестве рассматриваемых *объектов*, программируемых средствами языков *Maple* и *Math* пакетов *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 обоих пакетах для этой цели используются механизмы типирования, рассмотренные с достаточной степенью детализации в четвертой главе настоящей книги.

Пакет *Maple* поддерживает достаточно развитый механизм типизации объектов, для которого характерно наличие весьма развитого набора встроенных как простых, так и составных *типов*, распознаваемых пакетом, наряду с двумя механизмами, которые поддерживают создание пользовательских типов, также распознаваемых пакетом. 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 пакета *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 пакетом *Maple* не предусмотрена, тогда как *Mathematica* в этом отношении имеет довольно серьезные преимущества, поддерживая механизмы не только *отдельного* определения типов объектов, распознаваемых пакетом, но также дает возможность определения типов формальных аргументов непосредственно при определении 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 пакете *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[594]:= GS[x_Integer /; If[x > 72 || x <= 0, Print["Actual argument <" <x> ToString[x] <"> is inadmissible"]; False, True]] := Module[{}, x^2]
In[595]:= {GS[14], GS[22]}
Out[595]= {196, 484}
In[596]:= GS[420]
Actual argument <420> is inadmissible
Out[596]= GS[420]
In[621]:= If[SameQ[GS[44], Defer[GS[44]]], True, False]
Out[621]= False
In[622]:= If[SameQ[GS[420], Defer[GS[420]]], True, False]
Actual argument <420> is inadmissible
Out[622]= False
```

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 обоих пакетов. Они достаточно наглядно иллюстрируют возможности обоих программных сред в вопросе создания средств, тестирующих *фактические* аргументы, получаемые в точках вызова процедур/функций, на предмет их допустимости.

*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 пакете *Mathematica*; так, если в пакете *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 пакета *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_Integer /; If[x > 72 | | x <= 0, False, True]] :=
Module[{a = If[IntegerQ[x], a, b]}, x^2 + a + b]
In[3534]:= {a, b} = {14, 22}; Art[42]
Out[534]= 1800
In[3535]:= Kr[x_Integer /; If[x > 72 | | x <= 0, False, True]] := Module[{a}, x^2 + a + b]
In[3536] := {a, b} = {14, 22}; Kr[42]
Out[3536]= 1786 + a$1474
In[3537]:= ArtKr[x_Integer /; If[x > 72 | | x <= 0, False, True]] := Module[{a = 420}, x^2+a+b]
In[3538] := {a, b} = {14, 22}; Art[42]
Out[3538]= 2206
```

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 вне 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 использование 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** и **Apply**, которые играют 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 мере эквивалентные процедурным средствам пакета *Maple*. С рядом из них можно познакомиться выше и в нашем небольшом пакете [90]. Что в определенной степени уравнивает возможности обоих пакетов в программировании процедурных объектов, позволяя легче погружать *Maple*-процедуры в *Mathematica*.

Между тем, *процедурная* парадигма пакета *Mathematica* с учетом предоставляемых его *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 второго пакета, по нашему мнению, более предпочтителен. Вышеупомянутые процедурные средства, созданные в среде *Math*-языка в значительной степени уравнивают возможности обоих пакетов 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 аналогии 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. И так, 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 самого пакета.

В контексте работы с *внешними* файлами данных, без чего немыслимо использование обоих пакетов 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 пакета в данном 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 обоих 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, иницилируя *ошибочные* ситуации, допускающие программную обработку. Тогда как пакет *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 оба эти пакета имеют вполне достаточно средств. Поэтому в этом отношении вполне можно констатировать, что между пакетами имеет место определенный паритет.

Со стороны продвинутого пользователя чрезвычайно важным представляется вопрос по разработке средств, отсутствующих в пакете либо по тем или иным причинам для него неподходящими. В таком 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 пакетные

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 одного пакета в среду другого, и наоборот. В пакете *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. Следующий фрагмент иллюстрирует 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 обоими пакетами. Это еще одно соображение в пользу того, что наиболее эффективно освоение обоих пакетов для своей работы.

Достаточно важным моментом при разработке набора процедур и функций является *вопрос* их организации и имплантирования в программную среду пакета. И в данном отношении мы отдаем определенные предпочтения пакету *Maple* и вот почему. Для сохранения *ПС* пользователя, созданных в среде любого из рассматриваемых пакетов, 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 {*mw*s, *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 пакетными 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].

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 пакета. Неплохо 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 тому либо другому пакету, 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 попытаться 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 пакетной, 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 пакета *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 – пакет *Maple* либо пакет *Mathematica*, т.е. налицо субъективность.

Являясь эквивалентными в смысле универсальной вычислимости, пакеты *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 определенной мере вопрос привычки, однако пакет *Mathematica* является далеко не универсальным средством, чтобы под него серьезно подстраиваться, и такого типа разнобой с общеустоявшимся вряд ли можно считать положительным явлением.

С другой стороны, встроенный *Math*-язык пакета *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 пакета *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 знакомого с основами английского позволяет легче понимать их суть и осуществлять ассоциативный поиск средств по именам в справке по пакету.

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 пакету *Maple*, тогда как *функциональная* составляющая пакета *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 выше*), 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 столь хорош.

В данном контексте следует отметить и такую весьма важную составляющую пакетов, 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 пакета *Mathematica* скрыты от пользователя, возможно, содержа очень оригинальные алгоритмы, достойные премии *Филдса* или на худой конец *Тьюринга*, то стандартные средства пакета *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 и других соображений в пользу такой открытости пакета [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 пакет *Maple* возможно рассматривать как наиболее удачное на сегодня средство в деле освоения систем компьютерной алгебры в университетах, а также преподавания информатики, что делает его более естественная организация, встроенный язык программирования и достаточно дружелюбный пользовательский интерфейс, тогда как пакет *Mathematica* с его 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, пакеты *Maple* и *Mathematica* являются несомненными лидерами среди *CAS* общего назначения; однако, существуют *CAS* более специально ориентированные, в своей предметной области значительно превосходящие по возможностям первые две.

Естественно, оба рассмотренных пакета 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 пакетов *Mathematica* и *Maple*, реализующую простой вычислительный алгоритм, а именно:

```
In[1725]:= Log[{a, k} = {0, 1}; For[k, k <= 200, k++, a = a + If[PrimeQ[k], k, 0]]; a + 420.75]
Out[1725]= 8.44414
```

```

> In((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)(420.75));      ⇒ 8.444138511
> evalf(In(add(`if`(type(k, prime), k, 0), k=1..200) + 420.75));    ⇒ 8.444138511
In[1733]:= Log[Sum[If[PrimeQ[k], k, 0], {k, 200}] + 420.75]
Out[1733]= 8.44414

```

Уже из приведенного простого примера видно, что в среде первого пакета *композиции* функций являются более компактными и прозрачными. На примерах данного типа, приведенных в книге, эта разница прослеживается еще более наглядно. Естественно, в каждом конкретном случае соотношение может меняться на противоположное, как иллюстрирует два последних примера, однако в этом фрагменте оптимизационный фактор не принимался во внимание, отдавая в данном вопросе *предференции* наиболее типичным конструкциям, реализующим довольно сложные алгоритмы, требующие, как правило, *процедурного* оформления. Следует также отметить, что с ростом версий пакета *Maple* вопрос композиции процедур/ функций все более усложняется, в основе чего в значительной степени лежат модификации синтаксиса базовых предложений входного языка пакета. В частности, этому способствует и замена функций **RETURN** и **ERROR** соответственно предложениями **return** и **error**. Пока допустимы оба варианта кодирования, однако *первый* вариант считается *устаревшим* (*obsolete*) и в последующих версиях пакета *Maple* поддерживать его не планируется. Это решение, на наш взгляд, не обусловлено какими-либо принципиальными соображениями.

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 пакетов составят хорошую основу для расширения процедурной парадигмы для пакета *Mathematica*, с одной стороны, и *функциональной* парадигмы для пакета *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 пакету *Maple*. Тогда как в случае с пакетом *Mathematica* ввиду большей, в целом, реактивности его программной среды издержки данного типа не столь заметны.

Естественно, более существенный уровень использования парадигмы, не присущей в определяющей степени тому либо иному из рассматриваемых пакетов, предполагает

расширение, порой существенное, стандартных средств пакета *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 пакетами 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 пакетом *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 пакетом *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 *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 [90]), что обуславливает весьма неплохие предпосылки для распараллеливания вычислений, использующих такие функции, то *Maple*-язык подобными средствами не располагает (напомним, что сравнительный анализ пакетов *Maple* и *Mathematica* проводился нами [97-99], в основном, на основе их релизов 8 – 11 и 7.0 – 8.0.4.0 соответственно). При этом, при рассмотрении в настоящей книге ряда средств *Mathematica* с целью акцента на его свойствах дается и сравнение с соответствующими средствами пакета *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 его среде еще релиза 2.2 успешно выполняются в последних его релизах 7 и 8, 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 обоих пакетов, представленными нами в данной книге, в контексте использования их в качестве программной среды, прежде всего, для задач *модульного* программирования наша точка зрения на данные пакеты, лидеры в классе CAS, состоит в следующем, а именно:

– CAS *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 и освоении систем компьютерной математики. Опыт использования обоих пакетов для разработки различного рода пакетов в областях, таких как математика и физика подтверждает нашу точку зрения.

В частности, использование пакета *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 пакет *Mathematica* обладает также в разработке системных средств, расширяющих средства пакета, носящие массовый характер, либо устраняющих какие-либо их недостатки, а также расширяющих их функциональные возможности. Однако, это не самый худший вариант, к большому сожалению, оба пакета 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 пакет *Maple* выглядит несколько более впечатляюще, чем пакет *Mathematica*.

И если на *уровне* студенческих задач и относительно несложных проектов в контексте упомянутой простоты можно отдать некоторое предпочтение пакету *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 пакет *Mathematica*. И в данном контексте довольно заметно, что *Maple* со временем пытается заимствовать немалое число новаций, навеянных именно пакетом *Mathematica*.

Не взирая на достаточно жесткую конкуренцию рассмотренных выше пакетов *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 *оба* рассмотренных пакета, 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 пакетов *Maple* и *Mathematica* данная 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 *CAS*, 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 обоих пакетов со всей определенностью говорит в пользу предпочтения пакета *Mathematica*, но с одним весьма существенным «*но*», а именно. Преимущества пакета *Mathematica* сказываются лишь в том случае, если пользователь достаточно искушен в его программной среде и хорошо владеет средствами его встроенного *Math*-языка, прежде всего, функциональной парадигмой программирования. Иначе недостаточно квалифицированному пользователю придется по душе существенно более простой в освоении и использовании встроенный язык пакета *Maple*. Еще раз отметим, здесь мы акцентируем внимание именно на *программном* аспекте пакетов, 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 пакетом синтаксиса. Именно по последнему показателю пакет *Maple* отличается в лучшую сторону и привычностью используемого его языком синтаксиса, и вполне достаточной прозрачностью наряду с хорошей структурированностью исходных кодов его программных средств. Тогда как по этим показателям пакет *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 него 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 пакета. Тогда как в среде пакета *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 *большого* профессионализма, чем чтение листингов подобных средств на встроенном языке пакета *Maple*. Однако, при получении определенного навыка в работе с программной средой пакета данная проблема нивелируется, тогда как сами коды средств получаются более компактными и эффективными при выполнении.

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 пакетов *Maple* и *Mathematica*, так и лежат в русле оценок ряда других специалистов по *CAS* и опытных программистов. Надеемся, что представленные выше соображения окажут определенное влияние на решение вопроса о приоритетности того либо другого пакета для конкретной сферы приложений. Более того, рекомендуется работать с обоими пакетами, 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 пакетами не исключает ряда неприятных моментов, серьезно раздражающих пользователя, когда совместная работа с обоими пакетами, 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 средствам пакета *AVZ\_Package* [90].

## 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., Богдявичус 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., Богдявичус 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-03.- 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-Х.
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://depositfiles.com/files/iwjto8xuz>.
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.ecsocman.edu.ru/text/16089443](http://www.ecsocman.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. Коробов В.И., Очков В.Ф. Химическая кинетика: Введение с *Mathcad/Maple/MCS*.- Москва: Горячая линия-Телеком, 2009.
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/AVZ\\_Package.zip](http://www.aladjev.narod.ru/AVZ_Package.zip).

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*.– CA: 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*.– CA: USA, Palo Alto, Fultus Corporation, 2011, ISBN 978-1-59682-268-9, 418 p.
98. Aladjev V.Z., Bezrukavyy 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. Mangano S. *Mathematica Cookbook*.– CA: Sebastopol: O'Reilly Media, Inc., 2010, 828 p.
101. Kendrick D. et al. *Computational Economics*.– Princeton University Press, 2006, 406 p.
102. 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.
103. Roe T. et al. *Multisector Growth Models: Theory and Applications*.– Springer, 2009, 329 p.
104. 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.
105. Sisson P. *College Algebra, 2<sup>nd</sup> ed.*– Hawkes Learning Systems, 2008, ISBN 9781932628272.
106. 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.
107. Gregor J., Tier J. *Discovering Mathematics: A Problem-Solving Approach to Mathematical Analysis with MATHEMATICA and Maple*.– Springer, 2010, 254 p., ISBN 978-0857290540.
108. Панкратьев Е.В. Элементы компьютерной алгебры.– М.: Изд-во МГУ, 2007, 243 с.
109. Hege Hans-Christian, Polthier K. *Visualization and Mathematics III*, 2011, 457 p.
110. Albery R. *Applications of Mathematica*.– Wiley Press, 2011, ISBN 9780470639320, 456 p.
111. Shiskowski K., Frinkle K. *Principles of Linear Algebra with Mathematica*.– Wiley, 2011.
112. Terras A. *Zeta Functions of Graphs, A Stroll through the Garden*.– Cambridge Univ., 2011.
113. Kilian A. *Programmieren mit Wolfram Mathematica*.– Springer, 2010, ISBN 9783642046711
114. Hollis S. *CalcLabs with Mathematica for Multivariable Calculus*.– Brooks/Cole, 2012, 274 p.
115. Anmong Xu. *Introduction to Scientific Computing: Numerical Analysis With Mathematica*.– China Machine Press, 2010, ISBN 978-711-131-091-4.
116. *Core Language: Tutorial Collection*.– Wolfram Research Inc., 2008, 358 p.
117. Hastings K.J. *Introduction to Probability with Mathematica*.– CRC Press, 2010, 451 p.

***Monographs, textbooks and books on Computer Algebra Systems  
Maple and Mathematica, prepared and published by members of the  
Baltic Branch of International Academy of Noosphere during 1995 – 2011***

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 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 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 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 and in English*).
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 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., Bezrukavyyi 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.

## **Перечень пользовательских процедур и функций для пакета Mathematica, рассматриваемых в настоящей книге**

Ниже представлен список имен всех процедур, функций и глобальных переменных, с той либо иной детализацией рассмотренных в настоящей книге. Данные средства сгруппированы по их основному функциональному назначению; между тем, данная группировка в определенной степени носит условный характер, ибо многие из этих средств достаточно многофункциональны и имеют довольно широкое применение.

### **Средства для работы со строчными структурами и символами**

Border, CharacterQ, DelSubStr, DelSuffPref, ExprOfStr, ExprOfStr1, InsertN, IsMonotonic, IsPermutation, ListStrToStr, LongestCommonSubSequence, LongestCommonSubString, MaximalPalindromicSubstring, OverLap, PrefixQ, SortString, Spos, StrDelEnds, StrExprQ, StrFromStr, StrOfSymb1Q, StrStr, StrSub, StringEnd, StringMultiple, StringMultipleD, StringPosition1, StringReplace1, StringReplace2, StringSplit1, StringTake1, StringTake2, SubDelStr, SubStr, SubStrSymbolParit, SubsBstr, SubsDel, SubsStr, SuffPref, SuffixQ, SymbolQ, ToString1

### **Средства для работы со списочными структурами**

ArrayInd, AssignToList, BinaryListQ, DelEl, ListAssign, ListListGroup, ListListQ, ListOp, ListPosition, ListStrList, ListToSeq, ListToString, ListsAssign, MaxNestLevel, MinusList, MinusList1, NestListQ, NestQL, ParVar, PosIntListQ, SortLpos, SortNL, SortNL1, SortQ, SymbolToList, ToList

### **Средства для работы с последовательностными структурами**

SEQ, SeqDel, SeqIns, SeqQ, SeqToList, SeqToList1, SeqToString, SeqUnion, SequenceQ, Sequences

### **Средства для работы с процедурами и функциями**

\$CallProc, \$HeadProc, \$Help, \$ProcName, \$TypeProc, Args, Args0, Args1, Args2, ArgsLocals, ArgsProc, ArgsTypes, Arity, Avg, BlockQ, BlockToMod, CallsInProc, CallsInProc1, ClearAllAttributes, CorrCall, DefFunc, DefFunc1, DefFunc2, DefFunc3, DefOpt, Defaults, DelRestPF, DuplicateLocalsQ, ExpFunc, ExpFunc1, ExpFunc2, ExpLocals, ExprQ, ExtProgExe, ExtrCall, ExtrExpr, ExtrName, ExtrNames, FunCompose, FunctionQ, FunctionQ1, Globals, Globals1, Globals2, GotoLabel, HeadName, HeadPF, HeadPF1, HeadingQ, HeadingQ1, HeadingQ2, HeadingQ3, Headings, Headings1, HeadingsPF, Locals, Locals1, Locals2, MdP, ModuleQ, NamesProc, NamesProc1, Nproc, PartProc, ProcBody, ProcCall, ProcCalls, ProcContent, ProcContent1, ProcContent2, ProcFuncBlQ, ProcFuncCS, ProcLocals, ProcQ, ProcQ1, ProcQ2, ProcsAct, PureFuncQ, QFunction, RETURN,

RemProcOnHead, RemProcOnHead1, RemovePF, RemovePF1, ReplaceProcBody, ScanLikeProcs, SubProcs, SubsProcQ, SubsProcs, SyntCorProcQ, SysFuncQ, SysFuncQ1, TestArgsTypes, TestProcCalls, UprocQ, Uprocs

*Средства для работы с пакетами и документами (m- и nb-файлами)*

AcNb, Aobj, AutoLoadPack, ContOfContex, ExtrPackName, LoadNameFromM, NamesMPackage, NamesNbPackage, NamesNbPackage1, NbCallProc, NbName, Need, Npackage, PackNames, UpdatePackages

*Средства, расширяющие стандартные функции пакета Mathematica либо его программную среду в целом*

ActRemObj, ActiveProcess, Affiliate, AtomicQ, Attributes1, AttributesH, BitGet1, BitSet1, Bits, CatN, Clear1, ComplexQ, ContextQ, Contexts1, DO, Decomp, DefAttributesH, Definition1, Df, Df1, Df2, FreeQ1, GV, Gather1, Gather2, GroupNames, Head1, HowAct, IFk, IFk1, Iff, Ind, Index, IndexedQ, Int, Int1, LeftFold, LocObj, Map1, Map10, Map11, Map2, Map3, Map4, Map5, Map6, Map7, Map8, Map9, MapInSitu, MapInSitu1, MapInSitu2, Mapp, MemberLN, MemberQ1, MemberQ2, MemberT, MixCaseQ, Names1, NamesCS, Nvalue, Nvalue1, OP, ObjType, Op, PalindromeQ, PartialSums, PosIntQ, ProtectedQ, ProtectedQ1, Range1, Range2, Range3, RemoveNames, Rename, Rename1, RenameH, ReplaceAll1, RhsLhs, RhsLhs1, RightFold, Subs, SystemQ, Try, Tuples1, TwoHandQ, Type, TypeActObj, UnDef, UnDef1, UnevaluatedQ, VarExch, VarExch1, WhatObj, WhatType, WhatValue, WhichN

*Средства для работы с файловой системой компьютера*

Adrive, Adrive1, Attrib, Attrib1, Attrib2, BootDrive, CALL, CDir, CloseAll, Closes, CopyDir, DelDirFile, DelFile, DelFile1, DelFile2, DirEmptyQ, DirFD, DirFull, DirName, DirQ, EmptyFileQ, FileDirStForm, FileFormat1, FileFormat2, FileOpenQ, FileOpenQ1, FileQ, FindFile1, FindFileContext, FindFileObject, FindSubDir, FreeSpaceVol, IsFile, IsFileOpen, LoadExtProg, LoadFile, ModLibraryPath, Nobj, OpenFiles, PathToFileQ, ReadFullFile, RenDirFile, Save1, SearchDir, SearchFile, SetDir, SetPathSeparator, StandPath, StreamFiles, StreamsU, TypeFilesD, UpdateContextPaths, UpdatePath

*Некоторые средства специального назначения*

AllMatrices, AllMatrices1, DeCod, Email, Email1, GC, Help, HostName, NestCycles, Predecessors, PredecessorsL, PredecessorsR, Prev, Tbl, TestOnNCF, Un, UserLib, Ver

Довольно детальное описание средств представленного списка может быть найдено в пакете для *Mathematica* [90]. 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 [90]. Поэтому для детального их 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.nb*» с последующим его вычислением, имеющего лицензию *FreeWare* (для работы в среде *Mathematica* версий 6 – 8 [90]). 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 достаточно сложных задач.

Также следует отметить, что в представленных программах и функциях, написанных в *Mathematica*, для ряда часто используемых функций не использовались принятые в *Math*-языке эквиваленты, например, **Map** [/@], что было обусловлено желанием более удобной ассоциации с процедурами, написанными в *Maple*-языке и использующими подобные стандартные средства, например, *map* и **Map**.

С целью упрощения сравнения программных сред обоих пакетов нами была избрана следующая методика – проводилось программирование процедур и функций одного и того же назначения в среде обоих пакетов 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 пакета, 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 пакетов *Mathematica* и *Maple*, а ряд приведенных реализаций в обоих пакетах носят специальный иллюстративный характер, тогда как с рядом основных аспектов сравнительного анализа программных сред обоих пакетов можно ознакомиться в наших предыдущих книгах [97-99].

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 пакетами *Mathematica* и

*Maple*. Хотя они и могут представить интерес для начинающих в плане ознакомления с встроенными языками программирования обоих пакетов. 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 пакетам, в основе своей, носят парный характер (*реализация одного и того же алгоритма в среде обоих пакетов*). Сам же подбор средств был выбран таким образом, чтобы проиллюстрировать возможности программных сред обоих пакетов на примерах, прежде всего, системного характера, носящих достаточно массовый характер наподобие массовых стандартных пакетных 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 воспринимаем 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 пакетов 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 пакета и 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* пакетов, 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 пакета.

Включение примеров процедур 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. В принципе, не так давно порядка 25–30 лет тому назад при отсутствии *ПК* программисты работали в *пакетном* режиме, когда *отперфорированная* на носителе (*перфокарта, перфоленга*) программа передавалась на отладку в пакете с

другими программами. Получив результат *прогона* своей программы в виде *листинга*, программист проводил ее доотладку, работая с листингом, для чего ему был крайне необходим хороший навык чтения *листинга* и на основе знания того или иного языка программирования вносить в него необходимые правки, с учетом которых программа на носителе соответствующим образом корректировалась и процесс повторялся.

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, существенно сокращая многочасовые бдения перед *ПК* и преследуя одновременно важную цель сведения к минимуму негативного влияния *ПК*. Впрочем, это во многом дело привычки, однако не стоит забывать, старое далеко не всегда хуже нового, и приобретая новые возможности, мы, зачастую, теряем старые навыки, важные для работы. Вот и в нашем случае, получив весьма удобное средство общения в любой нужный момент, мы, порой, упускаем из вида эффективность кода программных средств, создавая его без особых затруднений в *интерактивном* режиме с единственной целью получить требуемый результат быстро, 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 пакета *Mathematica* исходные коды более 320 процедур/функций находятся в 3 файлах *AVZ\_Package.nb*, *AVZ\_Package.cdf* и *AVZ\_Package.m*; при этом, *первые 2* файла с *Mathematica* документом можно непосредственно загружать в *текущий* сеанс пакета по цепочке функций *GUI: Evaluation → Evaluation Notebook*, получая доступ ко всем средствам, содержащимся в нем, наряду с краткой справкой по ним, а также получать исходный код средств на экран *ПК*. Такой подход позволяет удовлетворить читателя с разными пристрастиями. Наш опыт проведения разного уровня мастер-классов по пакетам как *Mathematica*, так и *Maple* подтверждает целесообразность этого подхода. Надеемся, что наша методика предоставления материала книги найдет понимание у читателя. На данный вопрос, впрочем, имеют место право быть и иные точки зрения.

**V.Z. Aladjev, V.A. Vaganov, D.S. Grimm. Additional functional facilities for the package Mathematica.– Ukraine: Kherson, Oldi-Plus Press, 2012, 404 p.**

The *computer algebra* has found application in many fields of science such as mathematics, physics, chemistry, computer sciences, engineering, education, technology, computational biology, etc. The computer algebra systems (CAS) such as *Mathematica* are becoming more and more popular in teaching, research and industry. Researchers use well-known package *Mathematica* as an essential tool when solving problems related to their investigations. The package is ideal for formulating, solving, and exploring different mathematical models. Its symbolic manipulation facilities extend greatly over a range of problems that can be solved with its help. Educators in colleges and universities have revitalized traditional curricula by introducing problems and exercises, which use *Mathematica*'s interactive mathematics and physics. Students can concentrate on the more fundamental concepts rather than on tedious algebraic manipulations. Finally, engineers and experts in industries use *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 package *Mathematica* of releases 7 – 8 enabled us not only to estimate its advantages compared with other similar CAS, first of all *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.

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 package and/or eliminating their defects or expanding the package new means. The software presented in the book contains a series of useful and effective receptions of programming in *Mathematica*, and expands its software which enable more simply and effectively to programme in package *Mathematica* the problems of various purpose. The book is supplied by the package containing more 320 procedures and functions with *Freeware* licence. Contents of the book in the context of its chapters can be presented as follows, namely:

**Preface**

**Chapter 1.** Brief excursus into the history of computer algebra systems

**Chapter 2.** The general information concerning the builtin programming language of the package *Mathematica* (*the Math-language*)

**Chapter 3.** Basic control structures of the package *Mathematica*

**Chapter 4.** Basic mechanisms of typification of objects in the package *Mathematica*

**Chapter 5.** Processing of special and erroneous situations in the package *Mathematica*

**Chapter 6.** Organization of mechanisms of procedures and functions in *Mathematica*

**Chapter 7.** Additional means of work with basic structures of the data and with expressions in software environment of the package *Mathematica*

**Chapter 8.** Organization of work with packages in software environment of *Mathematica*

**Chapter 9.** Means of input-output in software environment of the *Mathematica*

**Chapter 10.** Organization of the user software in the package *Mathematica*

**References**

*Аладьев Виктор Захарович  
Ваганов Вячеслав Алексеевич  
Гринь Дмитрий Сергеевич*

# *Дополнительные функциональные средства для пакета Mathematica*

## *Монография*

Оригинал-макет издания подготовлен в Балтийском отделении  
Международной Академии Ноосферы (Таллинн, Эстония)

Формат 7,44 x 9,69. Бумага Офс.  
Тираж 1000.

Издательство ЧП «Олди-плюс»  
e-mail: oldi-ks@i.ua  
73033. г. Херсон, а/я № 15  
Лицензия сер. № 2 от 16.08.2000 г.