

Министерство образования Республики Беларусь

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«ГРОДНЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ ЯНКИ КУПАЛЫ»

МЕЖДУНАРОДНАЯ АКАДЕМИЯ НООСФЕРЫ

Балтийское отделение

В.З. Аладьев, В.К. Бойко, Е.А. Ровба

*Программирование в пакетах Maple и
Mathematica: Сравнительный аспект*

Гродно 2011

Содержание

Предисловие	5
Глава 1. Краткий экскурс в историю систем компьютерной алгебры	17
Глава 2. Общие сведения по языкам программирования <i>Maple</i> и <i>Mathematica</i>	27
Глава 3. Базовые управляющие структуры пакетов <i>Maple</i> и <i>Mathematica</i>	37
3.1. Управляющие структуры ветвления пакетов <i>Maple</i> и <i>Mathematica</i>	40
3.2. Циклические управляющие структуры пакетов <i>Maple</i> и <i>Mathematica</i>	48
3.3. Специальные типы циклических управляющих структур обоих пакетов	55
3.4. Оценка механизмов циклических конструкций <i>Maple</i> и <i>Mathematica</i>	61
Глава 4. Базовые механизмы типирования объектов в среде обоих пакетов	63
4.1. Механизмы типирования объектов в программной среде пакета <i>Maple</i>	65
4.2. Механизмы типирования объектов в программной среде пакета <i>Mathematica</i>	75
4.3. Оценка механизмов типирования объектов пакетов <i>Maple</i> и <i>Mathematica</i>	81
Глава 5. Организация обработки особых и ошибочных ситуаций	91
5.1. Обработка особых и ошибочных ситуаций в среде пакета <i>Maple</i>	91
5.2. Обработка особых и ошибочных ситуаций в среде пакета <i>Mathematica</i>	103
5.3. Оценка эффективности пакетов <i>Maple</i> и <i>Mathematica</i> по организации в них механизмов обработки особых и ошибочных ситуаций	108
Глава 6. Организация механизма процедур и модулей в <i>Maple</i> и <i>Mathematica</i>	109
6.1.1. Определения процедур в среде пакета <i>Maple</i> и их типы	110
6.1.2. Формальные и фактические аргументы <i>Maple</i> -процедур	119
6.1.3. Локальные и глобальные переменные процедур в среде пакета <i>Maple</i>	128
6.1.4. Определяющие параметры и описания <i>Maple</i> -процедур	138
6.1.5. Механизмы возврата <i>Maple</i> -процедурой результатов ее вызова	151
6.1.6. Расширенные средства <i>Maple</i> -языка для работы с процедурами	157
6.1.7. Расширение функциональных средств <i>Maple</i> -языка пакета	166
6.1.8. Иллюстративные примеры оформления <i>Maple</i> -процедур	177
6.1.9. Элементы отладки процедур и функций в среде пакета <i>Maple</i>	221
6.1.10. Организация программных модулей в среде <i>Maple</i> -языка пакета	227
6.1.11. Сохранение определений процедур и программных модулей в файлах	241
6.1.12. Создание пользовательских пакетных модулей в среде пакета <i>Maple</i>	251
6.1.13. Статистический анализ библиотек пользователя в среде пакета <i>Maple</i>	259
6.2.1. Определения процедур в среде пакета <i>Mathematica</i> и их типы	267

6.2.2. Локальные переменные процедур в среде пакета <i>Mathematica</i>	273
6.2.3. Глобальные переменные процедур и механизмы возврата результатов процедурами пакета <i>Mathematica</i>	285
6.2.4. Некоторые примеры оформления <i>Mathematica</i> -процедур	290
6.2.5. Функциональные конструкции в среде пакета <i>Mathematica</i>	360
Глава 7. Организация работы с пакетами в среде <i>Mathematica</i>	378
Глава 8. Средства ввода/вывода пакетов <i>Maple</i> и <i>Mathematica</i>	394
8.1. Средства пакета <i>Maple</i> для работы с внутренними файлами данных	394
8.2. Средства пакета <i>Mathematica</i> для работы с внутренними файлами	399
8.3. Средства пакета <i>Maple</i> для работы с внешними файлами данных	403
8.4. Средства <i>Mathematica</i> для работы с внешними файлами данных	418
8.5. Сравнение средств доступа к файлам пакетов <i>Maple</i> и <i>Mathematica</i>	440
Глава 9. Организация программного обеспечения пользователя	441
9.1. Организация программного обеспечения в среде пакета <i>Maple</i>	443
9.1.1. Классический способ создания <i>Maple</i> -библиотек в среде пакета	446
9.1.2. Специальные способы создания библиотек пользователя в <i>Maple</i>	454
9.2. Организация программного обеспечения пользователя в среде математического пакета <i>Mathematica</i>	463
9.3. Оценка эффективности пакетов <i>Maple</i> и <i>Mathematica</i> по организации пользовательского программного обеспечения	472
9.4. Программирование больших систем в среде пакетов <i>Mathematica</i> и <i>Maple</i>	480
Глава 10. <i>Maple</i> или <i>Mathematica</i> – субъективная точка зрения	486
Литература	505
Перечень процедур для <i>Maple</i> и <i>Mathematica</i>, рассматриваемых и упоминаемых в настоящей книге	510
Справка по авторам	514

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 *Maple, Mathematica, 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.

Компьютеризация – составляет одну из наиболее актуальных проблем современного общественного прогресса, технической основой решения которой является наличие **персональных компьютеров (ПК)** и развитых телекоммуникационных средств связи, позволяющих создавать весьма эффективные информационно-вычислительные сети различных уровня и назначения. Второй важнейшей компонентой данного процесса является наличие достаточно развитых программных средств, предназначенных для решения разнообразных задач из различных областей человеческой деятельности, в том числе включая средства поддержки эксплуатации самих ПК в различных режимах обработки информации и телекоммуникации.

Решение прикладных пользовательских задач в той или другой области приложений поддерживают **пакеты прикладных программ (ППП или просто пакеты)** специального, узко-специального либо общего назначения. Классификация и характеристика данного класса **программных средств (ПС)** может быть найдена в наших предыдущих книгах [1-3,36]. Естественно, квалифицированный пользователь, владеющий в достаточной мере одним из эффективных языков программирования (например, *Basic, C, Fortran, PL/I, Pascal, Lisp, Prolog* и др.), в целом ряде случаев для решения своих задач может самостоятельно написать и отладить отдельную программу или комплекс программ, позволяющих реализовать на ПК алгоритм его задачи. Причем, в ряде случаев такой подход может быть более эффективным, чем использование для этих целей готовых ПС, т.к. разработчик ПС одновременно хорошо владеет спецификой решаемой задачи и условиями ее эксплуатации. Однако, такой подход требует, как правило, серьезных трудозатрат и при нынешнем обилии различного типа и назначения средств для ПК (особенно широко распространенных **IBM-совместимых**) в значительной мере становится нецелесообразным. Вместе с тем, развитые ППП снабжаются собственным **встроенным** языком программирования того или иного уровня сложности, позволяющим в среде пакета программировать целые задачи или их отдельные фрагменты, которые может быть нецелесообразно, неэффективно, а в ряде случаев и невозможно, реализовывать

стандартными средствами пакета. Настоящая книга посвящена классу *ПС* (названному *системами компьютерной алгебры*), которые, в первую очередь, предназначены для решения задач математического характера, и, прежде всего, лидерам в данном классе пакетам *Maple* фирмы *MathSoft Inc.* и *Mathematica* фирмы *Wolfram Research 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*. К *первому* уровню могут быть отнесены такие системы программирования как *Basic*, *C*, *Pascal-XSC*, *PL/1* и др., ко *второму* – *Fortran*, *ISETL*, *Prolog* и др. *Третий* уровень может быть представлен как библиотеками математических подпрограмм (*SSP*, *NAG*, *ППП-БИМ* и др.), так и узко-специальными пакетами *MacMath*, *Phaser*, *VossPlot*, *Eureka* и др. К *четвертому* уровню можно отнести такие пакеты как *S-Plus*, *XploRe*, *SAS*, *Dynamics*, *StatGraf*, *SPSS*, *BMDP*, *PL/1-Formac*, *Systat* и др. Так, в [1-7] дан довольно подробный обзор *ПС*, ориентированных на решение задач в области математической статистики, прикладных статистик и экономического анализа математическими методами.

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 впоследствии учтено при создании последующих версий пакетов. Наш опыт эксплуатации указанных пакетов оказал существенное влияние на апробацию и использование сравниваемых здесь пакетов *Mathematica* и *Maple*. Естественно данная рубрикация программных средств во многом носит субъективный характер, носящий следы опыта нашей работы с *ПС*, в той или иной мере ориентированными на вопросы программирования 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, наследуют целый ряд их стандартов, как пионеров-эталонов ПС такого типа, что достаточно легко прослеживается при более детальном их рассмотрении, использовании и апробации.

В среде отечественных пользователей CAS 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*. На сегодня на данном сайте форум *Maple* имеет 2215 тем и 9598 сообщений, тогда как форум *Mathematica* лишь 812 тем и 2722 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 CAS на постсоветском пространстве относительно невелика. В беседах, проведенных нами на целом ряде мастер-классов по CAS, 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 специально был проанализирован целый ряд тем и сообщений на указанном сайте в форумах по обоим пакетам и картина сложилась следующая. На обоих этих форумах, в массе своей, обитают студенты, пытающиеся решить свои задачки чужими силами, и если *Maple* им по каким-то причинам не удалось освоить на относительно низком уровне, то более сложный для восприятия неподготовленному пользователю второй пакет для своего освоения требует больших усилий. Тогда как квалифицированные пользователи обоих пакетов, как правило, не посещают форумов такого уровня. Итак, приведенная статистика с определенной долей достоверности может быть отнесена к студенческой среде, хотя и она представляет существенный пласт пользователей CAS.

Кратко охарактеризуем оба рассматриваемых пакета. *Maple* – система *компьютерной математики*, рассчитанная на весьма широкий круг пользователей. Довольно часто ее называют системой *компьютерной алгебры* ввиду ее ориентации прежде всего на выполнение символьных вычислений и преобразований. Однако, это не совсем так, т. к. на самом деле система способна довольно эффективно выполнять не только сугубо

символьные, но и численные вычисления в сочетании с довольно развитыми средствами графического отображения и подготовки электронных документов, отражающих ход решения пользовательской задачи в среде пакета. Более того, росту эффективности в направлении *численных* вычислений способствовало не только улучшение численных алгоритмов средств собственно *самого* пакета, но и имплантация в его *вычислительную* среду высоко эффективных матричных вычислений известного пакета *NAG (Numeric Algorithms Group)*. *Maple* – довольно хорошо продуманная система, которая с равным успехом может использоваться как для простых, так и для достаточно сложных задач, связанных как с символьными, так и с численными алгоритмами вычислений. Пакет в настоящее время с успехом используется в самых различных областях.

Исследователи используют пакет *Maple* как важный инструмент для решения разных проблем, связанных с их исследованиями. Пакет идеален (*по нынешним понятиям*) для формулировки, решения и исследования *различных* математических моделей. Так, его алгебраические средства существенно расширяют диапазон проблем, которые могут быть достаточно эффективно решены на качественном уровне. Тогда как педагоги в средних школах, колледжах и университетах обновляют свои традиционные учебные планы, вводя в них задачи и упражнения, которые используют *диалоговую* математику и физику *Maple*. Студенты могут сконцентрироваться на важных концепциях, а не на утомительных алгебраических вычислениях и преобразованиях. Наконец, инженеры и специалисты в промышленности используют пакет *Maple* как весьма эффективный инструмент, заменяющий много традиционных ресурсов, в частности, справочников, калькуляторов, крупноформатных таблиц, редакторов и языков программирования. Эти пользователи достаточно *легко* решают весьма широкий диапазон математически ориентированных задач, разрабатывая проекты и объединяя результаты вычислений (*как числовые, алгебраические, так и графические*) в профессиональные отчеты довольно высокого качества. Отмеченные возможности системы *Maple* в значительной степени обусловлены ее интегрированностью, включающей такие основные компоненты, как:

- мощный встроенный императивный язык программирования процедурного типа;
- редактор для подготовки и редактирования документов и программ;
- современный развитый графический пользовательский интерфейс;
- развитая справочная система по всем средствам с множеством примеров;
- эффективные алгоритмы и правила преобразования математических выражений;
- символьный и численный процессоры;
- довольно развитая система диагностики;
- библиотеки встроенных и дополнительных функций;
- большой набор пакетных модулей, ориентированных на различные приложения;
- большой набор пакетных модулей, разработанных широким кругом пользователей, ориентированных на различные приложения и свободно распространяемых, и т.д.

Система *Maple* воплощает новейшую технологию символьных вычислений, числовых вычислений с любой точностью, расширяемой технологии интерфейса пользователя (*Maplets*), наличие инновационных *Web*-компонентов, довольно развитых алгоритмов для решения сложных математических задач и др. В настоящее время у пакета *Maple*

более, чем 3 миллиона пользователей: ученые, студенты, специалисты, исследователи из различных областей. Многие ведущие университеты и научно-исследовательские институты в мире, включая такие как MIT, Cambridge, Stanford, Oxford, Waterloo и целый ряд др., используют пакет для учебных и исследовательских целей. В промышленных целях пакет используется и такими ведущими корпорациями как Boeing, Bosch, Canon, NASA, Toyota, Sun Microsystems, Hewlett Packard, Motorola, General Electric, Daimler-Chrysler, Ford и др. История пакета **Maple** занимает более 30 лет развития и апробации; за этот период он развился в весьма сложный многофункциональный продукт. Пакет **Maple** реализован на больших ЭВМ, рабочих станциях, ПК, работающих с операционными системами **Unix**, **Windows** и др. Такой обширный набор операционных платформ и довольно широкий круг пользователей весьма положительно сказались на отработке пакета. Более того, упрощенная версия **Maple** для операционной системы **Windows CE** используется, в частности, и в миниатюрных компьютерах фирмы **Casio**. Ряд других математических пакетов поддерживают интерфейс с пакетом **Maple** для обеспечения символьных вычислений и преобразований в своей среде.

Вместе с тем современное развитие пакета **Maple** вызывает и ряд серьезных опасений, которые в общих чертах можно определить следующим образом. Качество хорошего программного обеспечения *определяется* целым рядом характеристик, среди которых можно отметить такие важные как: совместимость снизу-вверх, устойчивость работы относительно операционных платформ, качественные сопровождение и поддержка, и др. По данным показателям релизы пакета **Maple**, начиная с 7-го, оставляют желать лучшего, а именно. Из релиза в релиз переходят довольно существенные ошибки (из которых многие неоднократно отражались как в наших книгах и статьях, так и во многих других источниках, включая многочисленные форумы по **Maple**). Отсутствует приемлемая совместимость релизов снизу-вверх. О несовместимости релизов **Maple** мы отмечали неоднократно в своих книгах и статьях. Кое-что для улучшения совместимости нами было сделано (в частности, посредством упоминаемой в книге нашей библиотеки [45]), но, однако, не все. Между тем, для релиза 9 была обнаружена несовместимость уже среди его клонов. Как известно, **Maple 9** поддерживает два режима работы – стандартный и классический (ядра *maplew9.exe* и *cwmaple9.exe* соответственно). Оказывается, эти клоны несовместимы даже на уровне своих встроенных функций. Так, если в классическом режиме функция ``system`` выполняется корректно, то в стандартном режиме, выдавая код возврата 0, она некорректно выполняет некоторые команды/программы MS DOS. Именно поэтому, процедуры, использующие эту функцию и отлаженные в релизах 8 и ниже, а также в классическом **Maple 9**, в стандартном режиме **Maple 9** выполняются некорректно (часто иницилируя непредсказуемые ошибочные ситуации). Естественно, что такого типа нарушения требований к качественному программному обеспечению не допустимы для программных средств подобного типа и вполне могут инициировать нежелательные для пакета **Maple** последствия. Причиной же подобных ситуаций, как нам кажется, является то, что разработчики развивают пакет, как говорится, без учета того, что новшества могут несогласовываться со средствами предыдущих релизов, что в результате и приводит к отмеченным ситуациям по несовместимости релизов. Нам не хотелось бы думать, что несовместимость релизов обуславливается коммерческими соображениями, но и другие соображения не делают чести разработчикам пакета.

Тем временем, наш эксплуатационный опыт в течение **1997 – 2010 г.г.** с пакетом *Maple* релизов **4 – 11** позволил нам не только оценить его преимущества относительно ряда других подобных пакетов, но также выявил ряд ошибок и недостатков, устраненных нами. При этом, пакет *Maple* не поддерживал целый ряд довольно важных процедур обработки информации, алгебраических и численных вычислений, включая также и средства доступа к файлам данных. В свете сказанного, в процессе работы с пакетом *Maple* мы разработали достаточно много эффективного программного обеспечения различного назначения (*процедуры, программные модули*), целым рядом характеристик расширяющих базовые и по выбору возможности пакета. Программное обеспечение было организовано в виде Библиотеки [45], структурно подобной *главной* библиотеке *Maple* и обеспеченной развитой справочной системой, аналогичной подобной *Help*-системе пакета *Maple* и органично с ней связанной. Достаточно детальное описание данной Библиотеки может быть найдено в наших работах [9-14,28-37,42-45].

Программные средства, составляющие Библиотеку, в большинстве своем имеют дело именно с *базовой* средой пакета *Maple*, что пролонгирует их актуальность на текущие релизы, начиная с шестого, и на последующие релизы пакета. В данной связи вполне уместно обратить внимание на один *довольно* существенный момент. При достаточно частом объявлении о новой продукции *MapleSoft*, тем временем, уделяет *недостаточно* внимания устранению имеющихся ошибок и недостатков, переходящих от релиза к релизу. Некоторые из которых являются достаточно существенными. В своих книгах мы неоднократно отмечали данное обстоятельство, этому вопросу посвящено немало замечаний и членов *MUG*. При этом, расширению инструментальных средств *базовой* среды пакета также уделяется явно недостаточное внимание, что особенно заметно в режиме продвинутого программирования в его среде. Объявленная нами библиотека содержит расширение инструментальных средств, прежде всего, *базовой* среды *Maple*, что пролонгирует их актуальность также на последующие релизы пакета, достаточно существенно упрощая программирование целого ряда задач в его *среде* и обеспечивая более высокий уровень совместимости релизов **6 – 11**.

Хочется надеяться, что наши издания по пакету *Maple*, содержащие многочисленные замечания, рекомендации и описания процедур, как расширяющих его возможности, так и устраняющих замеченные недостатки пакета, наряду с нашей Библиотекой [45], расширяющей функциональные возможности пакета и устраняющей целый ряд его существенных недоработок, сыграли вполне определенную роль в деле дальнейшего развития и усовершенствования *Maple*. По меньшей мере косвенные данные говорят в пользу данного предположения. В частности, особенно это заметно в плане средств доступа к файлам данных, хотя и здесь отсутствуют подвижки по целому ряду весьма важных позиций, эффективно разрешенных средствами нашей Библиотеки [45], и в определенной мере отмеченных здесь в контексте сравнения *Maple* и *Mathematica*.

Пакет *Mathematica* наряду с уже упомянутым пакетом *Maple* является одной из *CAS*, 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;
- 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 *звук*ом 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 программирования, давая возможность пользователю создания средств, выполнимых только в среде самого пакета (*т.е. в полной мере невозможно создание exe- и com-файлов*), что существенно ограничивает переносимость созданных таким образом средств.

Оба пакета имеют массу весьма существенных как ошибок, так и недоработок, немало таких моментов отражено на форумах по данным средствам. При этом, имеющиеся у них ошибки, как правило, не пересекаются, т.е. некоторое вычисление в среде пакета *Mathematica* возвращает корректный результат, тогда как эквивалентное вычисление в среде пакета *Maple* возвращает некорректный результат либо не вычисляется вовсе,

и наоборот. С другой стороны, одна и та же задача не решается ни одним из пакетов, как это иллюстрирует следующий достаточно простой фрагмент, а именно:

```
> 0^0;  => 1
In[521]:= 0^0
Power::indet: Indeterminate expression 0^0 encountered. >>
Out[521]= Indeterminate
> evalf(int(cos(x)*sin(x)/x^2, x=10^(-128) .. 10000));  => 294.4605291
In[522]:= NIntegrate[Cos[x]*Sin[x]/x^2, {x, 10^(-128), 10000}]
NIntegrate::ncvb: NIntegrate failed to converge to prescribed accuracy after 9 ... .>>
Out[522]= 3.23881
In[611]:= Integrate[1 + 1/(x^2 + 1/(x^2 + 1/x^2)), x^2]
Integrate::ilim: Invalid integration variable or limit(s) in x^2. >>
Out[611]= Integrate[1 + 1/(x^2 + 1/(1/x^2 + x^2)), x^2]
In[612]:= Int[1 + 1/(x^2 + 1/(x^2 + 1/x^2)), x^2]
Out[612]= 1/4 (4 x^2 + 2 Log[x^2] + Log[2 + x^4])
> int(1+ 1/(x^2 + 1/(x^2 + 1/x^2)), x^2);
Error, (in int) wrong number (or type) of arguments
> inttt(1 + 1/(x^2 + 1/(x^2 + 1/x^2)), x^2);  => x^2 + 1/2*ln(x^2) + 1/4*ln(x^4 + 2)
```

В частности, последние два примера фрагмента наглядно иллюстрируют *идентичный* для обоих пакетов недостаток, а именно невозможность интегрирования выражения по переменной *интегрирования*, в качестве которой выступает не простая переменная, а подвыражение интегранда. Для устранения данного недостатка в обоих пакетах мы создали довольно простые процедуры **Df** и **inttt** соответственно, успешно решающие в достаточно широких пределах задачи данного типа. Не лишены серьезных ошибок и неработок и вновь включаемые в пакеты средства. В частности, введенные в версию *7 Mathematica* средства *распараллеливания* в полной мере не выполняют свои функции, о чем свидетельствует тот факт, что если аргумент функции *ParallelTable* определяет довольно сложное выражение, то многоядерность не используется. Подобное можно в полной мере отнести и к средства *распараллеливания*, включенным в *Maple 15*. Еще немало недостатков обоих систем можно было бы отметить, которые с ростом версий пакетов только количественно растут, вызывая растущее недовольство пользователей обоих систем, что весьма четко прослеживается по целому ряду тем и сообщений на форумах по обоим пакетам. Однако, не взирая на довольно большое число ошибок в обоих пакетах, в целом ряде случаев *недопустимых* для средств подобного класса, они на сегодня *реально* являются жестко конкурирующими лидерами среди средств этого класса, что, возможно, в значительной мере определяет и многие из их недостатков.

По обоим пакетам можно было бы еще немало представить их свойств, однако не это входит в задачу настоящей книги. Все эти вопросы можно найти в соответствующих изданиях, на сайтах разработчиков и в интернете. Отметим только, каждая из данных двух систем имеет свои особенности, однако по большому счету обе эти лидирующие системы практически равноценны. Постоянно конкурируя друг с другом, оба пакета взаимно развиваются и совершенствуются. Например, появившиеся в *Mathematica 7*

средства распараллеливания, были подхвачены и пакетом *Maple 15*. В то же самое время данная конкуренция наряду с положительными моментами имеет и свои негативные стороны, отмеченные выше. В целом, с полным основанием оба пакета можно считать лидерами среди класса универсальных CAS.

В заключение настоящей преамбулы весьма кратко изложим наше личное мнение по сравнительной оценке пакетов *Maple* и *Mathematica*. Оба данные пакета несомненно являются на сегодня лидерами среди CAS, однако они изобилуют многочисленными ошибками (в целом ряде случаев недопустимыми для систем подобного рода), устранению которых разработчиками как *MapleSoft*, так и *Wolfram Research* уделяется сравнительно небольшое внимание. Вероятно, исходя из коммерческих соображений, разработчики часто весьма необоснованно выпускают новые релизы, сохраняющие старые ошибки и привнося в ряде случаев как новые ошибки, так и различного рода «архитектурные» излишества. Данный вопрос неоднократно поднимался как в наших изданиях, так и непосредственно перед разработчиками. Однако, если разработчики *Maple* в режиме открытого диалога с пользователями в какой-то мере пытаются решить эту проблему, то *Wolfram Research* весьма болезненно воспринимает любую (совершенно обоснованную в подавляющем большинстве случаев) критику в свой адрес. Между тем, *Wolfram Research*, с другой стороны, ведет весьма агрессивную маркетинговую политику, совершенно не адекватную качеству ее продукции. Именно это, прежде всего, объясняет ее временные количественные преимущества в среде пользователей CAS, которые с определенного момента вполне могут уменьшиться. Сравнивая отклики пользователей этих пакетов наряду с нашим опытом работы с ними, можно вполне однозначно констатировать, что вторые при использовании пакета имеют несколько больше проблем.

Из нашего опыта довольно глубокого использования и апробирования обоих пакетов отметим, что *Maple* – существенно более дружелюбная и открытая система, которая в качестве программной среды предоставляет достаточно развитый встроенный *Pascal*-подобный императивный язык процедурного типа, что весьма существенно упрощает освоение пакета пользователю, имеющему опыт современного программирования в среде одного из процедурных языков. Тогда как *Mathematica* имеет в определенной мере «архаичный» (точнее, скорее непривычный) и не столь изящный язык, в целом ряде отношений, отличный от популярных языков программирования. Наконец, *Maple* по ряду показателей имеет более развитые средства (например, для решения дифференциальных уравнений в частных производных, предоставления пользователю используемого алгоритма для решения задачи, настройки графического интерфейса пользователя на конкретные приложения и др.), а также довольно широкий спектр бесплатных приложений во многих областях. Оба эти пакета не являются универсальными с точки зрения систем программирования, не давая возможности пользователю создавать средства, выполнимые вне среды самого пакета (т.е. в полной мере их программная среда не позволяет создавать {*exe* | *com*}-файлы с программными средствами, созданными в ней), что довольно существенно ограничивает мобильность созданных таким образом средств. Однако, здесь необходимо отметить, что далеко не все так однозначно, прежде всего, относительно встроенного языка обоих пакетов. Именно этому аспекту применения пакетов *Maple* и *Mathematica* в качестве программной среды разработки и посвящается настоящая книга.

Оба пакета на сегодня являются многофункциональными средствами, включающими большое число возможностей для решения широкого круга задач. Естественно, дать *полный* анализ данным средствам в отведенных книгой рамках просто невозможно. И здесь акцентируется внимание на одном их аспекте – *модульном программировании*, поддерживаемом обоими пакетами. Этот аспект имеет особую значимость не только для решения прикладных задач, но, прежде всего, он достаточно важен при создании средств, расширяющих часто используемые средства пакета и/или устраняющих их недостатки, либо дополняющих пакет новыми средствами. Именно в этом контексте и представляют интерес *возможности* пакетов по созданию такого типа процедур или функций. Итак, программирование в среде обоих пакетов – многоаспектная тема и в ней мы акцентируем внимание только на вопросах разработки процедур/функций, которые составляют *основные* программные объекты для наиболее частоиспользуемых средств пользователя, так и для средств, расширяющих и улучшающих стандартные средства пакетов, в программной среде самих пакетов, т.е. реализованные средствами встроенных языков пакетов. Именно в данном контексте и можно в довольно полной мере оценить программные среды обоих пакетов, не взирая на *субъективные* моменты, прежде всего, пользовательские предпочтения и привычки. Естественно, эти моменты играют довольно существенную роль для пользователя, имеющего *определенный* опыт работы с языками процедурного типа, тогда как для начинающего это не столь остро.

Книга состоит из **10** глав и затрагивает довольно обширный материал по *программной* среде пакетов *Maple* и *Mathematica* в контексте *сравнения* их возможностей *модульного* программирования. Между тем, ограниченные 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 пакетов *Maple* и/или *Mathematica*. Для удобства их использования именно в этом качестве предоставляется возможность загрузки содержащих их библиотеки для *Maple* [45] и пакета для *Mathematica* [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. Тогда как *вторая* глава на содержательном уровне представляет общие сведения по

языкам программирования пакетов *Maple* и *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. Завершается глава 3 оценкой механизмов управляющих структур пакетов *Maple* и *Mathematica*. 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 этот аспект рассмотреть и в контексте сравнения рассматриваемых пакетов *Maple* и *Mathematica*. И именно с точки зрения развитости поддерживаемых ими механизмов типизации объектов оба пакета рассматриваются в *четвертой* главе, как наиболее развитые и популярные на сегодня универсальные CAS.

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 контексте сравнения обоих пакетов в *пятой* главе рассматриваются средства, предоставляемые пакетами *Maple* и *Mathematica* для организации обработки особых и ошибочных ситуаций, которые в значительной степени можно отнести к вопросам робастности программных средств.

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 механизма процедур и модулей в пакетах *Maple* и *Mathematica*. Рассматриваются такие вопросы, как: определения процедур и их типы, формальные и фактические аргументы, локальные и глобальные переменные наряду с механизмами возврата процедурой результатов ее вызова и ряд других. Для пакета *Maple* рассматриваются также *расширенные* средства языка для работы с процедурами и организация программных и пакетных модулей, а также создание пользовательских пакетных модулей и сохранение процедур и программных модулей в файлах. Глава рассматривает также целый ряд примеров оформления процедур и функций в среде обоих пакетов, а также элементы функционального программирования *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*, либо программы решения задач пользователя; эти пакеты ориентированы на конкретные прикладные области. Глава 7 рассматривает основные вопросы организации пакетов, их создания, сохранения в файлах специального формата и использования в программной среде *Mathematica*.

Языки программирования *Maple* и *Mathematica* 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 форматов и организации. Поэтому в *восьмой* главе рассмотрены средства обоих пакетов для работы как с внутренними, так и с внешними файлами. В завершении главы проводится сравнение средств доступа пакетов к файлам данных.

Учитывая важность не только разработки программных средств, но и их сохранения в файлах специального формата для возможности последующего их использования, *девятая* глава рассматривает вопросы организации пользовательского программного обеспечения в среде *Maple* и *Mathematica*, специальные способы создания библиотек пользователя и представляет оценку эффективности пакетов *Mathematica* и *Maple* по организации программного обеспечения пользователя в их среде. Предложенные для иллюстрации целого ряда аспектов *программных* сред обоих пакетов многочисленные примеры во многих отношениях многофункциональны по назначению, их основное назначение довольно детально обсуждается в завершающей части настоящей книги.

Наконец, *десятая* глава представляет нашу, до определенной степени субъективную, точку зрения на предпочтительность каждого из рассматриваемых пакетов в той или иной области приложений. Не взирая на определенную субъективность, заключение по пакетам лежит в русле соображений достаточно широкого круга пользователей. В приложенном списке литературы нашли свое отражение работы наиболее известных отечественных авторов по системам компьютерной алгебры *Maple* и *Mathematica*.

В процессе подготовки наших изданий по пакетам *Maple* [8-44] и *Mathematica* [1-3,6,7, 36] наряду с библиотекой для *Maple* [45] и пакетом для *Mathematica* [90] нами были, в основном, использованы лицензионные версии 4 - 11 и 2.2, 7.0 - 8.0.4 (License # L4704-9074) для вышеуказанных пакетов соответственно, предоставленные разработчиками из MapleSoft Inc. и Wolfram Research Inc. Все вопросы, замечания и предложения по этой книге можно отсылать в любой из адресов, указанных на сайте www.aladjev.narod.ru.

Maple V, Maple 6, Maple 7, Maple 8, Maple 10, Maple 11 - торговые марки MapleSoft Inc.
Mathematica 2, Mathematica 7, Mathematica 8 - торговые марки Wolfram Research Inc.

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*, ориентированным на численные вычисления и созданном лишь в 1979 г. в университете Нью Мехико (*New Mexico*). Из первых же наиболее известных *CAS* общего назначения необходимо особо отметить систему *REDUCE*, во многом ориентированную на физическую область приложений.

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

- разложение и упорядочивание многочленов и рациональных функций;
- подстановки и сравнения шаблонов в весьма широком разнообразии форм;
- автоматическое и управляемое пользователем упрощение выражений;
- вычисления с символьными матрицами;
- поддержка произвольной точности арифметики целых и действительных чисел;
- возможности определения новых функций и расширения программного синтаксиса;
- символьные дифференцирование и интегрирование выражений;
- факторизация полиномов;
- 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*, *REDUCE*, *Mathematica* и *Maple*; причем книги [5-6] вообще были *первыми* массовыми книгами по *MatCAD*, *REDUCE* и *Mathematica*.

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

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

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

других вычислениях, а не зависеть лишь от одной системы, да и вряд ли одна даже и очень большая (зачастую громоздкая и требующая больших ресурсов) система обладает в равной степени эффективными алгоритмами. Немало «архитектурных» излишеств в *CAS Maple* и *Mathematica* с большой натяжкой можно отнести к *CAS*-проблематике. И здесь вполне уместно 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 *невывчисленным*. Тогда как *Axiom* возвращает интеграл *невывчисленным* лишь в случае, когда может доказать, что он не берется в элементарных функциях. К тому же, подобно *Maxima*, *Axiom* обладает достаточно высоким уровнем абстракции, предоставляемым пользователю. Обе системы поддерживают т.н. *Rule-преобразования*, позволяя выполнять преобразования не только над математическими выражениями, но 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.

Тут же вполне уместно отметить и отечественные работы по системам компьютерной алгебры, осуществленные в Институте кибернетики АН УССР под руководством акад. В.М. Глушкова. В рамках этих работ в 1968 г. был реализован язык программирования *Аналитик*, являющийся развитием языка *АЛМИР-65* с сохранением совместимости с ним. Отличительными чертами языка *Аналитик* являются абстрактные типы данных, вычисления в произвольных алгебрах, аналитические преобразования; он реализован на компьютерах *МИР-2*. Затем была разработана версия *Аналитик-74*, реализованная на компьютерах *МИР-3*. В настоящий момент язык *Аналитик* входит в состав системы компьютерной алгебры *АНАЛИТИК-2010*, совместно разрабатываемой и Институтом проблем математических машин и систем НАН Украины, и Полтавским национальным техническим университетом.

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*.

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 CAS можно найти в [81], однако некоторые его аспекты представляются нам довольно спорными. Между тем, анализ в целом представляет общую картину в этой области. Рассмотрим несколько более детально историю развития рассматриваемых здесь пакетов **Maple** и **Mathematica**, которая сама по себе достаточно поучительна.

Истоки создания *CAS Maple* восходят к концу 1980, когда исследователи из *Waterloo University* (Канада) вместо приобретения мощного компьютера под систему *Macsyma* решили разработать свою собственную *CAS*, способную функционировать на менее мощных компьютерах. Эта идея получила весьма быстрое развитие и уже в декабре 1980 появилась первая пробная версия системы. В основу подхода к созданию проекта была положена концепция разработки развивающейся и достаточно открытой *CAS*. Получив название *Maple* (в соответствии с кленовым листом – символом Канады), пакет, начиная с 1982, вполне успешно демонстрировался на различных конференциях, что принесло свои результаты – уже к концу 1983 г. более 50 университетов приступили к его использованию, а в 1984 г. разработчики запатентовали и начали распространять свой пакет. В 1988 разработчики пакета основали компанию *Waterloo Maple Inc.*, чьи функции не ограничились распространением пакета, но и проводились работы по его дальнейшему развитию. Между тем, хотя основная доля разработок по пакету *Maple* и принадлежит именно этой фирме, достаточно существенное развитие пакета было проведено и проводится по сей день в целом ряде других исследовательских центров и университетах, например: *Ontario Research Centre for Computer Algebra at the University of Western Ontario*, *Symbolic Computation Laboratory at the University of Waterloo*, и во многих других университетах и научно-исследовательских организациях, а также обширным кругом пользователей пакета. В рамках тематики по Системам компьютерной алгебры в *Baltic Branch of International Academy of Noosphere* (Таллинн, Эстония) проводились работы по апробации и применению пакета *Maple* версий 5–11, результатом чего явилось не только проведение целого ряда курсов и мастер-классов по пакету в разных странах, подготовка и издание в США, Белоруссии, Литве, России и Эстонии ряда монографий и книг по пакету, статей в журналах и докладов на международных конференциях, но и разработка обширной Библиотеки [45] новых программных средств для *Maple*, как расширяющих, дополняющих, так и улучшающих стандартные средства пакета.

Если до версии 4.3 *Maple* для интерфейса с пользователем использовал только режим командной строки, то уже с этой версией для *Macintosh* в 1989 был использован первый графический пользовательский интерфейс, тогда как улучшенная версия интерфейса под *Windows* была представлена в 1990 с *Maple V*. В 1999 пакет *Maple 6* включал в себя часть широко известной библиотеки *NAG* (*The Numerical Algorithms Group Ltd., Oxford, UK*) для задач линейной алгебры и был довольно существенно улучшен для работы с арифметикой произвольной точности. В версию *Maple 9* дополнительно к прежнему, т.н. классическому интерфейсу, который допустим на платформах *Windows* и *UNIX*, и требует меньше памяти, был добавлен более развитый и современный стандартный интерфейс. При этом, в среде *Windows* пользователь имеет возможность выбора типа интерфейса простым переключением. Текущий вид «стандартного» интерфейса был введен в 2003, интерфейс написан на *Java* и частично на встроенном *Maple*-языке, но на первых порах вызвал немало критики из-за весьма низкой реактивности на малой памяти, что требовало от пользователя вновь обращаться к классическому интерфейсу, который разработчиками пакета более не поддерживается.

Несмотря на существенное улучшение интерфейса, в период 1995 – 2005 пакет *Maple* потерял в пользу конкурентов определенную часть своего рынка из-за более слабого

пользовательского интерфейса. По этой причине в версию **Maple 10** был введен *новый «режим документа»* как часть *стандартного* интерфейса. Главная особенность данного режима состоит в том, что математические выражения вводятся в обычном 2-мерном формате, подобно используемому в книгах. Более того, в интерфейс версии **Maple 12** (2008) был добавлен ряд дополнительных возможностей, позаимствованных из пакета **Mathematica**. При этом, *классический* интерфейс написан на **C**, тогда как интерфейсы *стандартный* и с калькулятором написаны на языке **Java**. Более того, пакет допускает создание *интерфейсов*, определяемых пользователем. По-видимому, дополнительные возможности добавлялись (*в том числе*) для более приемлемого использования пакета с пакетом **MATLAB**. Пакет **Maple** располагает процедурным *Pascal*-подобным языком программирования *императивного* типа, поддерживая интерфейсы с такими языками программирования, как **Java**, **Fortran**, **C** и **Visual Basic**, а также с пакетами **MATLAB** и **Excel**. В архитектурном контексте **Maple** базируется на относительно небольшом ядре, написанном на языке **C**, которое поддерживает *встроенный Maple-язык*. Большинство функциональных возможностей пакета обеспечивается библиотеками, созданными в различных организациях и *отдельными* пользователями пакета. В частности, большое число процедур для численных вычислений поддерживаются такими библиотеками, как **ATLAS**, **NAG** и **GMP**. Тогда как большинство библиотек собственно самого пакета написано на *Maple-языке* с открытым для просмотра исходным кодом, что довольно важно со многих точек зрения, включая возможность редактирования средств **Maple**, создания пользовательских на их основе, а также использования их в качестве весьма прекрасных примеров для осваивающих программирование в среде **Maple**. При этом, за период **1982 – 2011** было создано **37** версий пакета, из которых *первой* доступной для массового пользователя явилась версия **Maple 3.3** (1985), тогда как на сегодня текущей является версия **Maple 15** (2011). Отметим, что в сентябре **2009** пакет **Maple** совместно с фирмой **MapleSoft Inc.** были приобретены фирмой **Cybernet Systems Ltd.** (Япония).

В свою очередь, *история* создания пакета **Mathematica** восходит к достаточно простой программе символьного манипулирования (**SMP**), являющейся довольно простой **CAS**, созданной **C. Cole** и **S. Wolfram** в **1979** и которая первоначально развивалась большой группой исследователей под руководством **S. Wolfram**. Программа **SMP** возникла под влиянием ранних **CAS Schoonschip** и **Macsyma**; так, **S. Wolfram** исследовал исходный код первой и являлся пользователем второй системы. Коммерческое *распространение* данной программы было начато в **1981 Computer Mathematics Corp.** (США), вошедшей в последующем в состав **Inference Corp.**, в рамках которой и проводились последующие развитие и распространение программы **SMP** в период с **1983** по **1988**. Есть основание 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 случаев 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 8.0.1* (2011). При подготовке этой книги нами использовались, в основном, версии *Mathematica 7.0*, *8.0.0* – *8.0.4* пакета.

Глава 2. Общие сведения по языкам программирования пакетов Maple и Mathematica

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*. В целом, *Maple*-язык является *процедурным* языком, при этом располагая только очень ограниченным количеством конструкций из базового функционального программирования.

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 доступна 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 входной язык 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.

Среда программирования пакета обеспечивается *встроеным 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.

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 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 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[#] &]
Out[547]= {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.

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 *невычисленным*, 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 *невычисленным*. 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[537]:= Sin[42.47]
```

```
Out[537]= -0.998289
```

```
In[538]:= Sin [42.47]
```

```
Out[538]= -0.998289
```

```
In[539]:= Sin[42. 47]
```

```
Out[539]= 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 искушенного пользователя.

Реальная мощь *Math*-языка обусловлена не только его управляющими структурами и структурами данных, но и всем богатством его *функциональных средств* (*встроенных,*

библиотечных, пакетных) и прикладных средств (*в виде Math-пакетов*), которые созданы к настоящему времени пользователями из различных прикладных областей, прежде всего, математических. К их числу можно с полным основанием отнести и пакет [90], содержащий более 200 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. производится сравнение замкнутых программных сред обоих пакетов.

Глава 3. Базовые управляющие структуры Maple и Mathematica

Для описания произвольного вычислительного алгоритма структур, базирующихся на сугубо последовательностных операциях, совершенно недостаточно без средств по управлению вычислительным процессом. Настоящая глава служит задаче сравнения управляющих структур встроенных языков пакетов *Maple* и *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.

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 заикливаний 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 или 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.

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» *Maple*-языка понимается конструкция следующего простого вида, а именно:

Maple-выражение {;|:}

где в качестве выражения допускается любая корректная с точки зрения *Maple*-языка конструкция, например: `A:= sqrt(68 + x): evalb(42 <> 68); sin(x) + x; `Tallinn-2011`:= 6; # Вызов и др.` В рассмотренных ниже иллюстративных примерах приведено довольно много разных примеров относительно простых предложений в рамках управляющей структуры *следования*, которая достаточно прозрачна и особых пояснений не требует. Предложения 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 языка будем называть в соответствии с их определяющим назначением, например, предложение *присваивания*, *комментария*, *вызова функции*, *while-предложение*, *restart-предложение*, *if-предложение* и т.д. Мы сделаем лишь одно замечание к предложению *присваивания*.

Наиболее широко употребительно определение предложения *присваивания* на основе одноименного (:=)-оператора, допускающего множественные присвоения. Однако, в целом ряде случаев вычислительные конструкции не допускают его использования, и в этом случае можно успешно использовать следующую процедуру *Maple*-языка:

assign(Id {,|=} Выражение)

возвращающую *NULL*-значение (*т.е. ничего*) и присваивающую *Id*-идентификатору вычисленное выражение (*которое, начиная с Maple 7, может быть и NULL*). При этом, процедура *assign* в качестве единственного аргумента допускает и *список/множество* уравнений формата *Id = выражение*, определяющих соответствующие множественные присваивания. Например, *списочная* структура следующего примера допустима лишь с использованием вышеупомянутой *assign*-процедуры пакета, а именно:

```
> [x := 68, y := 63*x, z := evalf(sqrt(x^2 + y^2)), x + y + z];
Error, `:=` unexpected
> [assign(x=68), assign(y, 63*x), assign(z, evalf(sqrt(x^2 + y^2))), x+y+z]; => [8636.539649]
> [assign(x = 42), assign(y, 47*x), assign(z, evalf(sqrt(x^2 + y^2))), x + y + z];
Error, (in assign) invalid arguments
```

```
> [assign(['x' = 42, 'y' = 47*x, 'z' = evalf(sqrt(x^2 + y^2))], x + y + z); ⇒ [7522.539649]
```

Итак, первый пример иллюстрирует недопустимость использования $(:=)$ -оператора, а три последующих – реализацию этой же списочной структуры на основе процедуры *assign*. При этом, последний пример демонстрирует использование невычисленных идентификаторов, обеспечивающих корректность вычислений. Тогда как 3-й пример иллюстрирует недопустимость повторного применения *assign* для переопределения вычисленных переменных. Процедура *assign* достаточно широко использовалась при создании целого ряда эффективных процедур, включенных в Библиотеку [45], тогда как в книгах [8-14,28-38,42-44] представлен ряд ее полезных расширений.

В пакете *Mathematica* последовательностная структура также может состоять из 1 или более предложений, заканчивающихся точкой с запятой $\{;$, которые располагаются в одной либо нескольких строках. Более того, как пакет *Maple*, так и пакет *Mathematica* для присвоения значений использует как оператор « $:=$ », так и оператор « $=$ ». Различие обоих операторов здесь не рассматривается; оно довольно прозрачно, пользователь с ним может ознакомиться в приведенной ниже литературе. Более того, определенным аналогом для процедуры *assign*($x_1 = a_1, \dots, x_n = a_n$) в *Mathematica* вполне могут служить списки следующего вида $\{x_1 = a_1, \dots, x_n = a_n\}$ либо $\{x_1 := a_1; \dots; x_n := a_n\}$, как иллюстрирует следующий весьма простой и наглядный фрагмент, а именно:

```
In[3]:= a := 68; b := 63; c := 43; d := 47; {x, y, z} = {42, 47, 67};
In[4]:= {{a, b, c, d}, {x, y, z}}
Out[4]= {{68, 63, 43, 47}, {42, 47, 67}}
In[5]:= x1 = 42; x2 = 47; x3 = 43; x4 = 14; x5 = 22;
In[6]:= {x1, x2, x3, x4, x5}
Out[6]= {42, 47, 43, 14, 22}
In[7]:= x1 = 42; x2 = 47; x3 = 43; x4 = 14; x5 = 22;
In[8]:= {x1, x2, x3, x4, x5}
Out[8]= {42, 47, 43, 14, 22}
In[9]:= {y1 = 42, y2 = 47, y3 = 43, y4 = 14, y5 = 22}
Out[9]= {42, 47, 43, 14, 22}
In[10]:= {z1 := 42, z2 := 47, z3 := 43, z4 := 14, z5 := 22}
Out[10]= {Null, Null, Null, Null, Null}
In[11]:= {z1, z2, z3, z4, z5}
Out[11]= {42, 47, 43, 14, 22}
```

При этом, вышеотмеченная аналогия не является полной, а именно: если процедура *assign* допускает в качестве фактических аргументов лишь последовательности, чьи элементы имеют тип *имя=выражение*, в качестве элементов списка *Mathematica* могут использоваться также конструкции вида *имя:=выражение* $\{;$ $|;$, т.е. предложения языка. Более того, для возможности корректного использования *assign*-процедуры в общем случае необходимо элементы списка фактических аргументов кодировать в формате '*имя*'=*выражение*, т.е. использовать невычисленный формат. Библиотека [45] содержит ряд расширений процедуры *assign*. Детальнее с вопросами организации в языковой среде обоих пакетов последовательностных структур можно ознакомиться в справках по пакетам, а также в соответствующей литературе по пакетам.

3.1. Управляющие структуры ветвления Maple и 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 *последовательный* 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 *Maple* располагает достаточно эффективным средством, обеспечиваемым т.н. **if-предложением**, имеющим следующие четыре формата кодирования, а именно:

- (1) **if** <ЛУ> **then** <ПП> **end if** {;|:}
- (2) **if** <ЛУ> **then** <ПП1> **else** <ПП2> **end if** {;|:}
- (3) **if** <ЛУ1> **then** <ПП1> **elif** <ЛУ2> **then** <ПП2> ... **else** <ПП3> **end if** {;|:}
- (4) **`if`**(<ЛУ>, V1, V2)

В качестве *логического условия (ЛУ)* всех четырех форматов **if**-предложения выступает любое допустимое *булево* выражение, образованное на основе операторов *отношения* {<|<=|>|>=|=|<>}, логических операторов {**and**, **or**, **not**} и логических констант {**true**, **false**, **FAIL**}, 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. Во всех форматах, исключая последний, ключевая фраза «**end if**» определяет закрывающую скобку (*конец*) **if**-предложения и его отсутствие идентифицирует синтаксическую ошибку, вид которой определяется контекстом **if**-предложения. При этом, каждой **if**-фразе должна строго соответствовать своя закрывающая скобка «**end if**» либо ее краткий формат «**end**».

Первый формат **if**-предложения несет следующую смысловую нагрузку: если результат вычисления ЛУ возвращает **true**-значение, тогда выполняется указанная за ключевым **then**-словом *ПП*, в противном случае выполняется следующее за **if** предложение, т.е. **if**-предложение эквивалентно *пустому* предложению. При этом, если **if**-предложение завершается (;)-разделителем, выводятся результаты вычисления всех предложений, образующих *ПП*, независимо от типа завершающего их разделителя. Следовательно, во избежание вывода и возврата излишней промежуточной информации, завершать **if**-предложение рекомендуется (;)-разделителем.

$$R = \begin{cases} \text{ПП1, если } \text{evalb}(\text{ЛУ1}) \Rightarrow \text{true} \\ \text{ПП2, если } (\text{evalb}(\text{ЛУ1}) \Rightarrow \text{false}) \text{ and } (\text{evalb}(\text{ЛУ2}) \Rightarrow \text{true}) \\ \text{ПП3, если } (\text{evalb}(\text{ЛУ1}) \Rightarrow \text{false}) \text{ and } (\text{evalb}(\text{ЛУ2}) \Rightarrow \text{false}) \text{ and } (\text{evalb}(\text{ЛУ3}) \Rightarrow \text{true}) \\ \hline \text{ППn-1, если } (\forall k | k \leq n-2)(\text{evalb}(\text{ЛУk}) \Rightarrow \text{false}) \text{ and } (\text{evalb}(\text{ЛУn-1}) \Rightarrow \text{true}) \\ \text{ППn, in other cases} \end{cases}$$

Второй формат **if**-предложения несет следующую смысловую нагрузку: если результат вычисления ЛУ возвращает **true**-значение, то выполняется указанная за **then**-словом *ПП1*, в противном случае выполняется следующая за ключевым **else**-словом *ПП2*.

Третий формат **if**-предложения несет смысловую нагрузку, несложно усматриваемую из выше представленной общей **R**-функции, которая поясняет сам принцип выбора выполняемой **ППк** в зависимости от цепочки истинности предшествующих ей **ЛУ_j** ($j=1..k$) в предложении. А именно: **if**-предложение 3-го формата возвращает результат **R** выполнения **ППк** только тогда, когда справедливо следующее соотношение:

$$(\forall j | j \leq k-1) (evalb(ЛУ_j) \Rightarrow false) \text{ and } (evalb(ЛУ_k) \Rightarrow true)$$

Данный формат **if**-предложения является наиболее общим, допуская любой уровень вложенности. Ключевое **elif**-слово является сокращением от '**else if**', что позволяет не увеличивать числа закрывающих скобок '**end if**' в случае вложенности **if**-предложения.

Наконец, по четвертому формату **if**-предложения возвращается результат вычисления **V1**-выражения, если $evalb(ЛУ) \Rightarrow true$, и **V2**-выражения в противном случае. Данный формат **if**-предложения аналогично вызову **Maple**-функции возможно использовать в любой конструкции подобно обычному выражению или отдельным предложением. Следующий фрагмент иллюстрирует применение четырех форматов **if**-предложения:

```
> if 68 <> 63 then V:= 42: G:= 47: S:= evalf(sqrt(G - V), 6) end if: S;    => 2.23607
> if 68 = 63 then V:=42 else G:= 47: S:=evalf(sqrt(68 - G), 4) end if: [V, S]; => [42, 4.583]
> AGN:= `if`(68 <> 63, 42, 47): Ar:=sqrt(621 + `if`(14 < 22, 2, z)^2): [AGN, Ar]; => [42, 25]
> if 68 = 6 then H:= `if`(3 <> 52, 52, 57) elif 57 <= 52 then H:= `if`(57 <> 52, 52, 57) elif
    32 >= 52 then S:= 67 elif 10 = 3 then Art:= 17 else `ArtKrVGS` end if; => ArtkrVGS
> F:=x -> `if`(x < 39, S(x), `if`(39 <= x and x < 59, G(x), `if`(59 <= x and x < 64, V(x), Z(x)))):
> [F(25), F(50), F(55), F(60), F(95), F(99)]; => [4.583, 47, 47, 42, Z(95), Z(99)]
```

Так, последний пример данного фрагмента иллюстрирует применение 4-го формата **if**-предложения для определения простой кусочно-определенной функции. Нами в процессе работы с **Maple** был создан целый ряд оригинальных средств [45], в качестве которых можно отметить и такие процедуры как **IF**, **If**, **Iff**, существенно расширяющие стандартный **`if`**-оператор. В качестве примера следующий фрагмент представляет исходный текст процедуры **Iff** в двух реализациях (в форме однострочного экстракода и в стандартной) с некоторыми типичными примерами ее применения.

```
> Iff := proc(x, y)
    if nargs = 2 then if x then y else NULL end if
    elif 3 <= nargs then if x then y else args[3] end if
    else error "invalid input: Iff expects 2 or 3 arguments, but received 1"
    end if
end proc:

> Iff := (x, y) -> `if`(nargs = 2, `if`(x, y, NULL), `if`(3 <= nargs, `if`(x, y, args[3]),
    ERROR("invalid input: Iff expects 2 or 3 arguments, but received 1")));

> [seq(Iff(type(k, prime), k), k=1 .. 70)];
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]

> [seq(Iff(type(k, prime), k, 0), k=1 .. 30)];
    [0, 2, 3, 0, 5, 0, 7, 0, 0, 0, 11, 0, 13, 0, 0, 0, 17, 0, 19, 0, 0, 0, 23, 0, 0, 0, 0, 0, 29, 0]
```



```

> Iff(x = 75);
Error, (in Iff) invalid input: Iff expects 2 or 3 arguments, but received 1
In[5]:= x = 75; {If[x == 75, 420], If[x == 15, 420]}
Out[5]= {420, Null}
In[6]:= a = {}; For[k = 1, k <= 18, k++, a = Append[a, If[PrimeQ[k], k]]]; a
Out[6]= {Null, 2, 3, Null, 5, Null, 7, Null, Null, Null, 11, Null, 13, Null, Null, 17, Null}
In[7]:= a = {}; For[k=1, k <= 75, k++, a=Append[a, If[PrimeQ[k], k]]]; Select[a, # != "Null" &]
Out[7]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73}

```

В отличие от стандартного ``if``-оператора, предполагающего при вызове обязательное наличие 3-х фактических аргументов, процедура *Iff* допускает наличие *двух* или *трех* фактических аргументов, вызывая ошибочную ситуацию только на одном аргументе. Вызов процедуры *Iff(x, y)* возвращает результат вычисления *y* в случае истинности *x*, и *NULL* в противном случае. Вызов *Iff(x, y, z)* эквивалентен вызову ``if``(*x, y, z*). В целом ряде случаев *Iff*-процедура оказывается достаточно полезным средством. Между тем, вызов функции *If[x, y]* в *Mathematica* также может возвращать *Null*, однако наличие его в результатах требует предварительной обработки, как иллюстрирует фрагмент.

Синтаксис *Maple*-языка допускает также *if*-предложения следующего формата:

```
if <ЛУ1> then if <ЛУ2> then ... if <ЛУn> then <ПП> end if end if ... end if {;|:}
```

где *ЛУ_k* – логические условия и *ПП* – последовательность предложений. Конструкция *if* данного формата выполняется корректно, но результат возвращается без вывода. Его возможно получать по `{% | %%% | %%%%`–предложению или по переменной, которой он был присвоен заранее, как это иллюстрирует следующий простой фрагмент:

```

> V:=68; G:=63; if V <> G then if V = 68 then if G=63 then R:= G+V end if end if end if;
> [%, R]; => [131, 131]
> if V<>G then S:=43; if V=68 then Art:= 22; if G=63 then R:=G+V end if end if end if;
> [%, S, Art, R]; => [131, 43, 22, 131]

```

Последний пример фрагмента иллюстрирует значительно более широкую трактовку допустимости *if*-предложений представленного формата наряду с особенностями их выполнения. Между тем, следует иметь в виду, *if*-предложение возвращает значение `{true | false}` даже в случае неопределенного по сути своей результата вычисления *ЛУ*:

```

> [[a, b], `if` (eval(a) <> eval(b), true, false)]; => [[a, b], true]
> `if` (FAIL = FAIL, true, false), `if` (eval(a) = FAIL, true, false); => true, false

```

С другой стороны, как иллюстрирует последний пример фрагмента, *if*-предложение отождествляет *FAIL*-значения, сводя различные *неопределенности* к единой, что также не соответствует здравому смыслу. Однако, причина этого факта лежит не в самом *if*-предложении, а в трактовке пакетом *Maple* самого понятия *неопределенности*, которая обсуждалась нами достаточно детально еще в наших книгах [8-10].

В концепции *Maple* символ *FAIL* используется булевой логикой для определения как неизвестного, так и неопределенного значения. При этом, в случае использования его внутри булевых выражений с операторами **and**, **or** и **not**, то его поведение аналогично

случаю стандартной 3-значной логики. Многие процедуры и функции *Maple*, прежде всего тестирующего типа возвращают символ *FAIL*, если не могут с определенностью установить истинность или ложность булевого значения. При этом, булево значение *FAIL* в *if*-предложении имеет тот же смысл, что и *false*-значение. Между тем, в случае появления значения *FAIL* для *a*-выражения *if*-предложения «*if a then b else c end if*» и «*if not a then c else b end if*» не эквивалентны, например:

```
> testeq(ln(x)^2 - x^2), `if`(testeq(ln(x)^2 - x^2), b, c);  ⇒  FAIL, c
> `if`(not testeq(ln(x)^2 - x^2), c, b);  ⇒  b
```

Арифметика с участием значения *FAIL* допустима, однако ограничена и применение *FAIL* в базовых арифметических операциях суммируется нижеследующим образом:

```
> 68 + FAIL, 42*FAIL, FAIL/FAIL, FAIL*FAIL, FAIL - FAIL, FAIL + FAIL, FAIL^FAIL;
    FAIL, FAIL, FAIL, FAIL, FAIL, FAIL, FAIL
```

Mathematica в качестве основной структуры ветвления располагает *If*-функцией, чей формат кодирования имеет следующие два варианта, а именно:

(1) *If*[*ЛУ*, *ПП1*, *ПП2*] (2) *If*[*ЛУ*, *ПП1*, *ПП2*, *ПП3*]

Вызов *If*-функции первого формата возвращает результат вычисления последнего из предложений *последовательности предложений ПП1* только в том случае, если *логическое условие (ЛУ)* принимает значение True; если *ЛУ* принимает значение False, функция *If* возвращает результат вычисления последнего из предложений *последовательности предложений ПП2*. Если же пакет не может определиться с истинностью/ложью *ЛУ*, т. е. в понимании пакета *Maple* с *FAIL*-значением, то вызов *If*-функции будет возвращен невычисленным, как это иллюстрирует следующий простой фрагмент, а именно:

```
In[245]:= If[a == b, c, d]
Out[245]= If[a == b, c, d]
In[246]:= If[42 == 47, c]
```

Последний пример фрагмента иллюстрирует тот факт, что вызов *If*[*ЛУ*, *c*] возвращает значение Null (*т.е. ничего*), если *ЛУ* получает значение False. Тогда как второй формат *If*[*ЛУ*, *ПП1*, *ПП2*, *ПП3*] *If*-функции ориентирован именно на возможность получения *ЛУ* неопределенного значения, как иллюстрирует следующий пример, а именно:

```
In[414]:= If[a == b, c, d, h]
Out[414]= h
```

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

```
In[1011]:= If[68 == 68, 420 + 72*If[42 == 47, If[42 == 47, c], d], b]
Out[1011]= 420 + 72 d
In[1012]:= If[v == v, If[42 == 42, Sqrt[If[68 == 68, 9.47] + 19.42], d], b, h]
Out[1012]= 5.37494
```

В целом, процедуру *`if`*(*ЛУ*, *assign('x1'=a1,...,'xk'=ak), assign('y1'=b1,...,'yk'=bk)*) пакета *Maple* вполне можно ассоциировать с функцией *If*[*ЛУ*, *x1=a1;...;xk=ak,y1=b1;...;yk=bk*] пакета *Mathematica*, как хорошо иллюстрирует следующий фрагмент, а именно:

```

> `if`(isprime(68), assign('b'=72, 'w'=420), assign('v'=68, 'g'=63, 's'=43, 'Art'=22, 'Kr'=14));
> v, g, s, Art, Kr;  => 68, 63, 43, 22, 14
In[135]:= If[PrimeQ[68], b := 72; w := 420, v := 68; g := 63; s := 43; Art := 22; Kr := 14]
In[136]:= {v, g, s, Art, Kr}
Out[136]= {68, 63, 43, 22, 14}
In[137]:= If[PrimeQ[68], b = 72; w = 420, v = 68; g = 63; s = 43; Art = 22; Kr = 14]
Out[137]= 14
In[138]:= {v, g, s, Art, Kr}
Out[138]= {68, 63, 43, 22, 14}

```

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 фактические аргументы вычисляются/упрощаются.

```

In[844]:= 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[845]:= a = {}; For[k = 1, k <= 100, k++, Iff[PrimeQ[k], "a = Append[a, k]"]; a
Out[845]= {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}

```

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[1]:= CopyToClipboard[10!]; Paste[]
Out[1]= 3628800
In[691]:= CopyToClipboard[10!]; Paste[]
      3628800
In[692]:= %
Out[692]= Notebook[{Cell[BoxData["3628800"], "Output"]}, FrontEndVersion -> ...
In[904]:= 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[905]:= CopyToClipboard[(a + b)/(c + d) + z]; Paste[]
      (a + b)/(c + d) + z
In[906]:= ToExpression[Prev[%]]
Out[906]= (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.

Итак, **if**-предложение представляет собой наиболее типичное средство обеспечения ветвящихся алгоритмов. В данном контексте следует отметить, что соответствующие **if**-средства *Math*-языка и *Mathematica*-языка представляются в значительной степени эквивалентными, однако в смысле читабельности несколько проще воспринимаются достаточно сложные *ветвящиеся* алгоритмы, реализованные именно предложениями **if** пакета *Maple*. И в этом отношении весьма сложно отдать кому-либо предпочтение.

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 *Maple* для организации *ветвления* алгоритмов наряду с представленным предложением **if** допускает использование и безусловных переходов на основе встроенной функции **goto**, кодируемой в формате **goto(Label)**. Между тем, по вполне понятным причинам, обусловленным структурным подходом к программированию, явно проповедуемым *Maple*, данная функция недокументирована. И мы обнаружили ее, главным образом, интуитивно. Между тем, в целом ряде случаев использование данного средства очень эффективно, в частности, при необходимости погружения в *Maple*-среду программы, использующей безусловные переходы на основе **goto**-предложения. Так, достаточно типичным примером являются *Fortran*-программы, очень широко распространенные в научных приложениях. Из нашего опыта следует отметить, что использование **goto**-функции существенно упростило погружение в *Maple* комплекса *Fortran*-программ, относящихся к инженерно-физическому направлению, очень широко использующих **goto**-конструкции. Между тем, **goto**-функция имеет смысл только в теле процедуры, обеспечивая в точке вызова **goto(Label)** переход к *Maple*-предложению, перед которым установлена отмеченная *Label*. В качестве *Label* выступает некоторый идентификатор, как это иллюстрирует следующий весьма простой фрагмент, а именно:

```
> A := proc(x) `if`(x > 68, goto(Label_1), `if`(x < 63, goto(Label_2), goto(Fin)));
    Label_1: return x^2;
    Label_2: return 14*x + 22;
    Fin: NULL
end proc;
> A(420), A(43), A(68), A(14), A(22), A(63), A(42), A(47); ⇒ 176400, 624, 218, 330, 610, 680
```

При этом, при использовании вызовов **goto(Label)** необходимо иметь в виду, что *Label* является глобальной переменной, предполагая ее выбор таким образом, чтобы она не

пересекалась с переменными текущего сеанса пакета, которые расположены вне тела процедуры, использующей *goto*-переходы. С такой целью мы определили процедуру *isplabel* [11,45], использующую весьма простой подход к защите *goto*-меток, который состоит в следующем. Если процедура использует, например, метки *L1, L2, ..., Ln*, то в начале исходного текста процедуры кодируется вызов *unassign('L1', 'L2', ..., 'Ln')*. Такой подход позволяет обеспечивать ошибкоустойчивость всех меток процедуры. Однако, данный подход требует определенной уверенности, что отмена значения *L*-метки не будет отрицательно сказываться на выполнении документа текущего сеанса, если он использует одноименную переменную *L* вне тела процедуры. Процедура *uglobal* [45] позволяет в *Maple*-процедуре работать с глобальными переменными, совершенно не заботясь о возможности наступления нежелательных последствий такого решения на глобальном уровне текущего сеанса пакета.

Действительно, поскольку *Label* является глобальной переменной, а ее определение в качестве локальной (*не тестируемое в качестве ошибки на стадии вычисления определения процедуры*) инициирует ошибочную ситуацию при вызове процедуры, то желательно обезопасить выполнение процедуры от возможных ошибок, обусловленных метками. Именно с этой целью были созданы вышеупомянутые процедуры. Между тем, с этой целью можно использовать достаточно простой прием программирования процедур, использующих *goto*-переходы. В самом начале тела процедуры, использующей метки *L1, L2, ..., Ln*, кодируется конструкция следующего простого вида, а именно:

assign(a1 = L1, a2 = L2, ..., an = Ln), unassign('L1', 'L2', ..., 'Ln');

тогда как список *local*-переменных процедуры расширяется переменными {*a1, a2, ..., an*}; затем каждый реальный выход процедуры снабжается выполнением конструкции

assign('L1' = a1, 'L2' = a2, ..., 'Ln' = an);

по которой глобальным переменным *L1, L2, ..., Ln* возвращаются начальные (*до вызова процедуры*) значения. Нижеследующий простой фрагмент иллюстрирует сказанное:

> restart; Label_1, Label_2 := 68, 63;

> A := proc(x) local L1, L2;

assign(L1 = Label_1, L2 = Label_2), unassign('Label_1', 'Label_2');

`if` (68 < x, goto(Label_1), `if` (x < 63, 42*a + 47*goto(Label_2), goto(Fin)));

Label_1; return x^2, assign('Label_1' = L1, 'Label_2' = L2);

Label_2; return 14*x + 22, assign('Label_1' = L1, 'Label_2' = L2);

Fin; assign('Label_1' = L1, 'Label_2' = L2)

end proc;

> A(420), A(43), A(68), A(14), A(22), A(63), A(42), A(47); ⇒ 176400, 624, 218, 330, 610, 680

> Label_1, Label_2; ⇒ 68, 63

Фрагмент достаточно прозрачен и особых пояснений не требует; при этом, фрагмент иллюстрирует допустимость использования вызовов *goto*-функции в выражениях. Но описанные подходы не требуются, если процедура использует, например, в качестве меток *goto*-функции *protected*-идентификаторы, как это иллюстрирует фрагмент:

```
> G:= proc(n::posint) goto(type(n, prime));
      true; WARNING("%1 is prime", n); goto(sin);
      false; WARNING("%1 is positive integer", n);
      sin; NULL; end proc;
> G(47), G(68), G(43), G(421), G(14), G(22);
Warning, 47 is prime
Warning, 68 is positive integer
Warning, 43 is prime
Warning, 421 is prime
Warning, 14 is positive integer
Warning, 22 is positive integer
```

С целью более детального прояснения вопросов использования *goto*-функции пакета и средств, сопутствующих ее использованию, можно обратиться к работам [11,44,45].

Пакет *Mathematica* располагает аналогичной функцией **Goto[W]**, которая безусловно передает управление в точку, поименованную конструкцией **Label[W]**. Как правило, **Goto**-функция используется, главным образом, в процедурных конструкциях, однако в отличие от *goto*-функции пакета *Maple* она может использоваться также во входной конструкции *Mathematica*, как это иллюстрирует простой фрагмент, а именно:

```
In[42]:= x:= 1; Label[AVZ]; Print[x++]; If[x != 31, Goto[AVZ], Null]
1
=====
30
In[43]:= x := 1; Label[Sin[42]+x]; Print[x++]; If[x != 31, Goto[Sin[42]+x], AVZ]
1
=====
30
Out[43]= AVZ
```

Более того, фрагмент иллюстрирует также тот факт, что в качестве метки может быть произвольное допустимое выражение; при этом, метка может быть как *глобальной*, так и *локальной* переменной относительно модуля, как это иллюстрирует фрагмент:

```
In[120]:= 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[121]:= ArtKr[47]
2209
In[122]:= ArtKr[42]
In[123]:= ArtKr[121]
127
```

С целью более детального прояснения вопросов применения **Goto**-функции пакета и средств, сопутствующих ее применению, можно обратиться к работам [3,6,7,57-60]. С нашей точки зрения **Goto**-функция пакета *Mathematica* несколько предпочтительнее по отношению к **goto**-функции пакета *Maple* в плане эффективности использования при программировании процедур. Итак, рассмотрев средства организации ветвления вычислительных алгоритмов, переходим к сравнению основных средств организации *циклических* конструкций в среде обоих пакетов.

3.2. Циклические управляющие структуры обоих пакетов

Предложение «while_do» пакета Maple. Циклическое предложение «*while_do*» пакета служит для многократного вычисления предложения либо их последовательности по указанным его переменным (*переменным цикла*), принимающим указанное множество значений. При этом, предложение **while_do** имеет 2 основных формата кодирования, наиболее общие из которых имеют следующий принципиальный вид, а именно:

- | | |
|-------|--|
| (1.a) | for <ПЦ> in <Выражение> while <ЛУ> do <ПП> end do {; :} |
| (1.b) | for <ПЦ> in <Выражение> do <ПП> end do {; :} |
| (1.c) | in <Выражение> while <ЛУ> do <ПП> end do {; :} |
| (1.d) | in <Выражение> do <ПП> end do {; :} |
| (1.e) | do <ПП> end do {; :} |
| (2.a) | for <ПЦ> from <A> by to <C> while <ЛУ> do <ПП> end do {; :} |
| (2.b) | for <ПЦ> from <A> to <C> while <ЛУ> do <ПП> end do {; :} |
| (2.c) | for <ПЦ> from <A> while <ЛУ> do <ПП> end do {; :} |
| (2.d) | for <ПЦ> while <ЛУ> do <ПП> end do {; :} |
| (2.e) | while <ЛУ> do <ПП> end do {; :} |

Наиболее общего вида вариант (1.a) первого формата предложения «**while_do**» несет следующую смысловую нагрузку: для заданной *переменной цикла* (ПЦ), принимающей в качестве значений значения последовательных операндов указанного после **in**-слова *выражения*, циклически повторяется вычисление *последовательности предложений* (ПП), ограниченной скобками {**do ... end do**}, до тех пор, пока *логическое условие* (ЛУ) будет возвращает *true*-значение. В качестве *выражения*, как правило, выступают множество, список и диапазон, тогда как относительно ЛУ имеет место силу все сказанное для **if**-предложения. При этом, значения операндов, составляющих *выражение*, могут носить как *числовой*, так и *символьный* характер. Сказанное в адрес **if**-предложения относится и к типу используемого для предложения **while_do** завершающего {;|:}-разделителя.

После завершения предложения «**while_do**» управление получает следующее за ним предложение. Смысл вариантов (1.b..1.d) первого формата предложения «**while_do**» с учетом сказанного довольно прозрачен и особых пояснений не требует. Следующий достаточно простой фрагмент иллюстрирует принципы выполнения всех вариантов первого формата предложения **while_do** по организации циклических вычислений:

Первый формат «while do»-предложения:

```
> x:= 1: for k in {14, 22, 43, 48, 63, 68} while (x <= 4) do printf('%s%a, %s%a, %s%a | ',
`k=`, k, `k^2=`, k^2, `k^3=`, k^3); x:= x + 1 end do:
k=14, k^2=196, k^3=2744 | k=22, k^2=484, k^3=10648 | k=43, k^2=1849, k^3=79507 | k=48,
k^2=2304, k^3=110592 |
> for k in (h$h=1 .. 14) do printf('%a | ', k^2) end do:
1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 | 81 | 100 | 121 | 144 | 169 | 196 |
> k:= 0: in (h$h=1 .. 99) while (not type(k, 'prime')) do k:=k+1: printf('%a | ', k^2) end do:
1 | 4 |
> do V:= 68: G:= 63: S:= 43: Ar:= 22: Kr:= 14: Arn:= 48: R:= V + G + S + Ar + Kr + Arn:
printf('%s%a`, `R=`, R); break end do:
R=258
> M:= {}: N:= {}: for k in [13, 14, 22, 37, 72, 63] do if type(k/4, 'odd') then break end if;
M:= `union`(M, {k}); end do: M; => {13, 14, 22, 37, 63, 72} # (1)
> for k in [13, 14, 22, 37, 72, 63] do if type(k/2, 'odd') then next end if; N:= `union`(N, {k});
end do: N; => {13, 37, 63, 72} # (2)
> T:= time(): K:=0: t:=10: do K:=K+1: z:= time() - T: if (z >= t) then break else next end if
end do: printf('%s %a %s`, `Обработка =`, round(K/z), `оп/сек`); # (3)
Обработка = 313587 оп/сек
> AG:=array(1..4,1..4): for k in a$a=1..4 do for j in a$a=1..4 do AG[k,j]:= k^j+j^k end do
end do: AV:=copy(AG): for k in a$a=1..4 do for j in a$a=1..4 do if (k=j) then AG[k, j]:=0
else AG[k, j]:= k^j + j^k end if end do end do: print(AV, AG);

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 8 & 17 & 32 \\ 4 & 17 & 54 & 145 \\ 5 & 32 & 145 & 512 \end{bmatrix}, \begin{bmatrix} 0 & 3 & 4 & 5 \\ 3 & 0 & 17 & 32 \\ 4 & 17 & 0 & 145 \\ 5 & 32 & 145 & 0 \end{bmatrix}$$

> restart; N := {}: for h in [F(x), G(x), H(x), S(x), Art(x), Kr(x)] do N := `union`(N, {R(h)})
end do: N; => {R(F(x)), R(G(x)), R(H(x)), R(S(x)), R(Art(x)), R(Kr(x))} # (4)
> map(R, {F(x), G(x), H(x), S(x), Art(x)}); => {R(F(x)), R(G(x)), R(H(x)), R(S(x)), R(Art(x))}
> restart: for h in [F, G, H, S] do map(h, [x, y, z, t, u]) end do;
[F(x), F(y), F(z), F(t), F(u)]
[G(x), G(y), G(z), G(t), G(u)]
[H(x), H(y), H(z), H(t), H(u)]
[S(x), S(y), S(z), S(t), S(u)]
```

Второй формат «while do»-предложения:

```
> S:= {}: for k from 1 by 2 to infinity while (k <= 25) do S:= `union`(S, {k^3}) end do: S;
{1, 27, 125, 343, 729, 1331, 2197, 3375, 4913, 6859, 9261, 12167, 15625}
> S := []: for k from 1 to infinity while (k <= 18) do S := [op(S), k^3] end do: S;
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832]
> S := {}: for k while (k <= 100) do S := {op(S), nextprime(k)} end do: S;
```



```

{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, 101}
> S := []; p := 0: while (p <= 14) do p := p + 1: S := [op(S), p^3] end do: S;
    [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375]
> S := []; p := 0: do p := p + 1: if (p > 15) then break end if; S := [op(S), p^3] end do: S;
    [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375]
> h := 0: for k from 19.42 by 0.001 to 24.20 do h := h + k: end do: h; ⇒ 104273.610
> print("Прошу подготовить CD-ROM с АРМ – Вы располагаете одной мин.");
  T := time(): do if time() - T >= 10 then break end if end do: print("Продолжение
вычислений после ожидания:"); # (5)
    "Прошу подготовить CD-ROM с АРМ – Вы располагаете одной мин.!"
    "Продолжение вычислений после ожидания:"

```

Несколько пояснений следует сделать относительно варианта (1.e) первого формата, который в общем случае определяет *бесконечный* цикл, если среди *ПП* не определено программного выхода или не производится внешнего прерывания по **stop**-кнопке 3-й строки *GUI*. В общем же случае для обеспечения *выхода* из циклической конструкции служит управляющее *break*-слово, по которому производится немедленный выход из содержащей его конструкции с передачей управления предложению, следующему за ней. В случае *вложенных* циклов передача управления производится циклу, внешнему относительно *break*-содержащего цикла.

Тогда как по *next*-слову производится *переход* к вычислению *циклической* конструкции со следующим значением *переменной* цикла. Нагляднее всего различие управляющих слов *break* и *next* иллюстрируют примеры нижней части предыдущего фрагмента. Из них легко заметить, что если *break*-значение обеспечивает *немедленный выход* из цикла, в котором оно вычислено, с передачей управления следующему за ним предложению или внешнему относительно его циклу, то *next*-слово позволяет управлять счетчиком выполнения цикла, обеспечивая выбор очередного значения для *переменной* цикла без выполнения самого тела цикла, т.е. обеспечивается условное выполнение цикла. Так, в упомянутом выше фрагменте цикл, содержащий *break*-значение (1), завершается по мере встречи *первого* нечетного числа, тогда как для аналогичного цикла (2), который содержит *next*-значение, цикл выполняется полностью, но *N*-множество, в отличие от *M*-множества предыдущего цикла, формируется только из *к*-чисел списка, значения *к/2* которых являются четными. Пример (3) демонстрирует *совместное* использование для управления циклической конструкцией варианта (1.e) первого формата **while_do**-предложения управляющих слов *next* и *break* с целью оценки *производительности ПК*, работающего под управлением среды *Windows+Maple 11*, в единицах весьма простых операций. Для обеспечения возможности последующего использования результатов выполнения циклических конструкций в их телах вполне возможно предусматривать аккумуляцию и сохранение результатов, таким образом получая вполне приемлемую возможность доступа к ним после завершения (даже в целом ряде случаев *ошибочного* или *аварийного*) выполнения циклической конструкции.

Наряду с *численными* вычислениями *первый* формат **while_do**-предложения позволяет вполне успешно производить и *символьные* вычисления и обработку, так, в частности,

пример (4) предыдущего фрагмента иллюстрирует эквивалентную замену функции *tar* соответствующей циклической конструкцией наряду с рядом других достаточно полезных преобразований.

Наиболее общего вида вариант (2.а) второго формата предложения «**while_do**» несет следующую смысловую нагрузку: для заданной *переменной цикла (ПЦ)*, принимающей в качестве значений *последовательность* значений $A, A+B, A+2*B, \dots, A+n*B \leq C$ (где A, B, C – результаты вычисления соответствующих выражений), будет *циклически* повторяться *вычисление последовательности предложений (ПП)*, ограниченной скобками `{do...end do}`, до тех пор, пока *логическое условие (ЛУ)* возвращает *true*-значение. В качестве A, B, C , как правило, выступают целочисленные выражения, тогда как по отношению к *ЛУ* имеет место силу ранее сказанное для случая *if*-предложения, включая и тип завершающего разделителя, используемого для управляющего предложения второго формата.

В случае отсутствия во втором формате предложения «**while_do**» *ключевых слов* `{from, by}` для них по умолчанию полагаются *единичные значения*. Для *C*-выражения второго формата, стоящего за *to*-словом, допускается *infinity*-значение, однако в таком случае во избежание *заикливания (бесконечного цикла)* необходимо определять условия *выхода* из цикла в *ЛУ* и/или в *ПП*. При этом, минимально допустимым форматом **while_do**-предложения *Maple*-языка является конструкция `do...end do {;|:}`, которая определяет собой *пустой* бесконечный цикл, на чьей основе можно программировать различного рода программные задержки (как это показано в примере (5) последнего фрагмента), но и создавать различной сложности вычислительные конструкции вида «*do ... ПП end do*», решающие циклического характера задачи с обеспеченным определенным выходом из них. В этом смысле (*do ... end do*)-блок можно рассматривать в качестве тела любой циклической конструкции, управление режимом выполнения которой производится через управляющую оболочку типов *for_from ... to_while* и *for_in_while*.

При этом, стандартные конструкции **for_while_do** допускают лишь фиксированный уровень вложенности (*вложенные циклы*), тогда как в целом ряде приложений уровень вложенности определяется динамически в процессе выполнения алгоритма. В состав нашей *Maple*-библиотеки [45] включена процедура *FOR_DO*, обеспечивающая работу с динамически генерируемыми конструкциями *for_while_do* произвольного конечного уровня вложенности. Так, данная процедура обеспечивает динамическую генерацию «**for_do**»-конструкций следующих достаточно широко используемых типов:

- | | |
|--|------------------|
| (1) for k1 to M do Expr[k1] end do; | [k1, M] |
| (2) for k2 by step to M do Expr[k2] end do; | [k2, step, M] |
| (3) for k3 from a by step to M do Expr[k3] end do; | [k3, a, step, M] |
| (4) for k4 in M do Expr[k4] end do; | [k4, `in`, M] |

Вызов процедуры *FOR_DO* принимает следующий вид (*подробности см. в* [30,42-45]):

FOR_DO([k1,M],[k2,step,M],[k3, `in`, M],[k4,a,step,M],..., [kp,...], "Expr[k1,k2,k3,...,kp]")

```
> FOR_DO := proc() local k, T, E, N, n, p, R, f;
  `if`(nargs < 2, ERROR("quantity of actual arguments should be more than 1"), `if`({true} <>
    {seq(type(args[k], 'list'), k = 1 .. nargs - 1)} or not type(args[-1], 'string'),
```

```

ERROR("actual arguments are invalid"), seq(`if`(type(args[k], 'list') and
type(args[k][1], 'symbol') and member(nops(args[k]), {2, 3, 4}), NULL,
ERROR("cycle parameters %1 are invalid", args[k])), k = 1 .. nargs - 1));
assign(N = "", n = 0, R = [], E = cat(seq("end do; ", k = 1 .. nargs - 1)),
f = cat(currentdir(), "/$Art22_Kr15$"));
T := table([2 = [for, to], 3 = [for, by, to], 4 = [for, from, by, to]]);
for k to nargs - 1 do assign('R' = [op(R), cat("", args[k][1])]);
  if member(`in`, args[k]) then
    N := cat(N, " for ", args[k][1], " in ", convert(args[k][3], 'string'), " ")
  else for p in T[nops(args[k])] do n := n + 1; N := cat(N, p, " ", args[k][n], " ") end do
  end if;
  assign('n' = 0, 'N' = cat(N, "do "))
end do;
writebytes(f, cat(N, " ", args[nargs], " ", E)), close(f); read f;
fremove(f), unassign(op(map(convert, R, 'symbol')))
end proc;
> Kr := Array(1..3, 1..4, 1..5, 1..7, 1..14): FOR_DO([k,3], [j,2,4], [h,2,1,5], [p,2,3,7], [v,2,4,14],
  "Kr[k,j,h,p,v]:=k*j*h*p*v;"); interface(rtablesizer = infinity); eval(Kr);
      [ 1..3 x 1..4 x 1..5 x 1..7 x 1..14 5-D Array ]
      Data Type: anything
      Storage: rectangular
      Order: Fortran_order

```

Циклические структуры Mathematica. Одна из базовых *циклических* структур пакета базируется на **For**-функции, имеющей следующий общий формат кодирования:

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

4

86

643541

```
In[113]:= 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[$W[k], \{k_1, t_1, n_1\}, \{k_2, t_2, n_2\}, \dots, \{k_p, t_p, n_p\}$] – вычисляет выражение $W[k_1, k_2, \dots, k_p]$ при изменении переменных цикла k_j от t_j до n_j ($j=1..p$) с шагом единица.

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

```
In[9]:= t := 0; Do[t = t + 1, {1000000}]; t
```

```
Out[9]= 1000000
```

```
In[10]:= t := 0; Do[t = t + k^2, {k, 1000}]; t
```

```
Out[10]= 333833500
```

```
In[11]:= t := 0; Do[t = t + k^2, {k, 1000, 10000}]; t
```

```
Out[11]= 333050501500
```

```
In[12]:= t := 0; Do[t = t + k^2, {k, 1000, 10000, 20}]; t
```

```
Out[12]= 16700530000
```

```
In[13]:= t := 0; Do[t = t + k^2 + j^2 + h^2, {k, 10, 100}, {j, 10, 100}, {h, 10, 100}]; t
```

```
Out[13]= 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[48]:= t := 0; Do[t = t + k^2; If[k < 10000, Continue[], Break[]], {k, 1000, 1000000, 20}]; t
```

```
Out[48]= 16700530000
```

В отличие от пакета *Maple* пакет *Mathematica* не имеет аналога достаточно полезных циклических конструкций типа (1.b) и (1.d), позволяющих производить 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[72]:= Op[x_] := Module[{a, b}, a := {}; If[ListQ[x], a = x,
Do[a = Insert[a, Part[x][[b]], -1], {b, Length[x]}]; a]]

In[73]:= Op[Sin[x] + Cos[x]]
Out[73]= {Cos[x], Sin[x]}
In[74]:= Op[Sin[x]*Cos[x]]
Out[74]= {Cos[x], Sin[x]}
In[75]:= Op[{1, 2, 3, 4, 5, 6}]
Out[75]= {1, 2, 3, 4, 5, 6}
In[76]:= Op[Sqrt[a + b]]
Out[76]= {1 + a, 1/2}
In[77]:= Op[Sqrt[a + b] + Sin[x] - c/d]
Out[77]= {Sqrt[1 + a], -(c/d), Sin[x]}
In[78]:= Op[(x + y*Cos[x])/(y + x*Sin[y])]
Out[78]= {x + y Cos[x], 1/(y + x Sin[y])}
In[79]:= Map[Op, {Sin[x], Cos[a + b], 1/(a + b)}]
Out[79]= {{x}, {1 + a}, {1 + a, -1}}
In[80]:= Op[f[g[a, b], h[c, d, e, j, k, l]]]
Out[80]= {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 пакета *Maple*.

```
In[125]:= DO[x_, y_, k_] := Module[{a, b, c, d, R}, d := 1; R := {}; a = x; b := Op[y];
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 моделируются 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 реализовывают любой алгоритм. Но во временном отношении, как будет показано несколько ниже, это далеко не так, и в случаях использования вложенных циклических конструкций достаточно большого уровня вложенности пакет *Maple* может иметь весьма существенные преимущества.

3.3. Специальные типы циклических управляющих структур в программной среде пакетов Maple и Mathematica

Наряду с рассмотренными базовыми структурами *Maple* располагает рядом полезных специальных управляющих структур циклического типа, позволяющих существенно упрощать решение целого ряда важных задач. Такие структуры реализуются на базе ряда встроенных функций и процедур {*add*, *mul*, *seq*, *sum*, *product*, *map*, *member* и др.}, а также \$-оператора, позволяющих весьма компактно описывать алгоритмы массовых задач обработки и вычислений. При этом, обеспечивается не только вполне хорошая наглядность *Maple*-программ, но и довольно высокая эффективность их выполнения. Следующий фрагмент иллюстрирует результаты вызовов некоторых перечисленных средств с эквивалентными им конструкциями пакета, реализованными посредством базовых управляющих структур *следования*, *ветвления* и *циклических*.

```
> add(G(k), k=1..9); ⇒ G(1) + G(2) + G(3) + G(4) + G(5) + G(6) + G(7) + G(8) + G(9)
> A:=0: for k from 1 to 5 do A:=A+G(k) end do: A; ⇒ G(1) + G(2) + G(3) + G(4) + G(5)
> mul(F(k), k=1..14); ⇒ F(1) F(2) F(3) F(4) F(5) F(6) F(7) F(8) F(9) F(10) F(11) F(12) F(13) F(14)
> M:=1: for k from 1 to 8 do M:=M*F(k) end do: M; ⇒ F(1) F(2) F(3) F(4) F(5) F(6) F(7) F(8)
> seq(F(k), k=1..12); ⇒ F(1), F(2), F(3), F(4), F(5), F(6), F(7), F(8), F(9), F(10), F(11), F(12)
> S:=[]: for k from 1 to 6 do S:=[op(S), F(k)] end do: op(S); ⇒ F(1), F(2), F(3), F(4), F(5), F(6)
> map(F, [x1,x2,x3,x4,x5,x6,x7,x8]); ⇒ [F(x1), F(x2), F(x3), F(x4), F(x5), F(x6), F(x7), F(x8)]
> M:=[]: for k in [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,x11] do M:=[op(M), F(k)] end do: M;
    [F(x1), F(x2), F(x3), F(x4), F(x5), F(x6), F(x7), F(x8), F(x9), F(x10), F(x11)]
> L:= [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,x11,x12,x13,x14]: member(x10, L); ⇒ true
> for k in L do if k = x10 then R := true: break else R := false end if end do: R; ⇒ true
```

Во фрагменте каждый пример *вызова* функции из указанной группы сопровождается следующим за ним примером, представляющим эквивалентную *Maple*-конструкцию в терминах базовых управляющих структур. Читателю в качестве довольно полезного упражнения рекомендуется разобраться в представленных примерах фрагмента. При этом, указанные функции не только могут моделироваться базовыми циклическими управляющими структурами, но и сами могут моделировать некоторые типы *вторых*, а также допускают применение в точке вызова соответствующих базовых *циклических*

управляющих структур, как это иллюстрирует следующий простой фрагмент:

```
> G := [68, 63, 43, 48, 14, 22]: add(G[k]*if(type(G[k], even), 1, 0), k=1..nops(G)); => 152
> mul(G[k]*if(type(G[k], 'even'), 1, if(G[k]=0, 1, 1/G[k])), k=1..nops(G)); => 1005312
> F:=[14,GS,22]: (seq(F[k]*if(type(F[k], symbol),1,0), k=1..nops(F)))(x,y); => 0, GS(x,y), 0
> (seq(if(type(F[k], 'symbol'), true, false) and F[k], k=1..nops(F))); => false, GS, false
```

Первые два примера фрагмента иллюстрируют применение *управляющей* структуры `if` для обеспечения дифференцированного выбора слагаемых и множителей внутри функций `add` и `mul`. Тогда как 2 остальных примера формируют последовательности вызовов функций на основе результатов их тестирования. Эти приемы вполне могут оказаться довольно полезным средством в практическом программировании в пакете *Maple* различного рода циклических вычислительных конструкций, что нами не раз использовалось в процедурах нашей Библиотеки [45]. С особенностями применения указанных циклических конструкций возможно ознакомиться, в частности, в книгах [11-14,28-39,42]. Вот только некоторые из них.

Так, в циклических конструкциях «`for k in n$ n=a..b...`» не допускается отождествления идентификаторов `k`, `n`, не распознаваемого синтаксически, но приводящего к ошибке выполнения конструкции. Более того, в общем случае нельзя отождествлять в единой конструкции переменные цикла и суммирования/произведения, например:

```
> h := 0: for k to 420 do h := h + sum(1, k = 1 .. 68) end do: h; => 0
Error, (in sum) summation variable previously assigned, second argument evaluates to 1=1..68
> h := 0: for k to 420 do h := h + sum(1, 'k' = 1 .. 68) end do: h; => 28560
> h := 0: for k to 420 do h := h + product(1, k = 1 .. 68) end do: h; => 0
Error, (in product) product variable previously assigned, second argument evaluates to 1=1..64
> h := 0: for k to 420 do h := h + product(2, 'k' = 1 .. 14) end do: h; => 6881280
> h := 0: for k in ['k' $ 'k'=1..68] do h := h + k end do: h; => 2346
> h := 0: for k in [k $ 'k'=1 .. 68] do h := h + k end do: h; => 4624
> h := 0: for k in [k $ k=1 .. 68] do h := h + k end do: h; => 0
Error, invalid input: $ expects its 2nd argument, range, to be of type name, but received 68=1..68
```

Вместе с тем, как иллюстрирует фрагмент, корректность сохраняется при кодировании переменной суммирования/произведения в невычисленном формате. Более того, три последних примера фрагмента иллюстрируют как допустимость, так и корректность использования *общей* переменной внешнего цикла и циклической `$`-конструкции, но при условии использования последней в невычисленном формате. Это определяется соглашениями *Maple*-языка по использованию глобальных и локальных переменных, детально рассматриваемых в шестой главе, посвященной процедурным объектам.

Дополнительно к сказанному необходимо иметь в виду весьма существенное отличие в выполнении `seq`-функции и логически эквивалентного ей `$`-оператора, точнее: если функция `seq` носит довольно универсальный характер, то `$`-оператор несколько более ограничен, а в целом ряде случаев определяет *некорректную* операцию, что довольно наглядно иллюстрирует следующий фрагмент применения обоих средств пакета:

```
> S:="aqwertyupsgghjkzxc": R:=convert(S, 'bytes'): convert([R[k]], 'bytes') $ k=1..nops(R);
```

Error, byte list must contain only integers

```
> cat(seq(convert([R[k]], 'bytes'), k = 1 .. nops(R)));    ⇒ "aqwertyupsgghjkzxc"
> X,Y := 7, 9: seq(H(k), k= `if` (X<9, 4, 9).. `if` (Y>8, 9, 9));    ⇒ H(4), H(5), H(6), H(7), H(8), H(9)
```

Итак, оба средства формирования последовательностных структур, на первый взгляд, эквивалентные следует применять довольно осмотрительно, по возможности отдавая предпочтение *первому* средству как наиболее универсальному. В этом отношении для *seq*-функции имеют место (в ряде случаев весьма полезные) следующие соотношения:

$$\begin{aligned} seq(A(k), k = [B(x)]) &\equiv seq(A(k), k = \{B(x)\}) \equiv (A@B)(x) \equiv A(B(x)) \\ seq(A(k), k = x) &\equiv seq(A(k), k = B(x)) \equiv A(x) \end{aligned}$$

При этом, при использовании *`if`*-функции для организации выхода из циклических конструкций рекомендуется проявлять внимательность, т.к. возвращаемое функцией *break*-значение не воспринимается в качестве управляющего слова *Maple*-языка:

```
> W := 3: do W := W - 1; if W = 0 then break end if end do: W;    ⇒ 0
> W := 3: do W := W - 1; `if` (W = 0, `break`, NULL) end do: W;    ⇒ 0
Error, invalid expression for eval
```

В частности, первый пример фрагмента иллюстрирует успешный выход из **do**-цикла после достижения **W**-переменной нулевого значения и удовлетворения логического условия **if**-предложения. Тогда как второй пример показывает невозможность выхода из идентичного **do**-цикла на основе *`if`*-функции, возвращающей на значении **W = 0** *break*-значение, не воспринимаемое в качестве управляющего слова. Более того, если в версиях *Maple 7* и выше инициируется ошибочная ситуация, то в более младших во втором примере не инициируется ошибочная ситуация, а выполняется бесконечный цикл, требуя прекращения вычислений по **stop**-кнопке **GUI**. В случае использования вместо **break** предложений **done**, **quit** и **stop** также выполняется бесконечный цикл.

Циклические вычислительные конструкции можно определять и на основе функций *{select, remove}*, имеющих следующий единый формат кодирования, а именно:

$$\{select \mid remove\}(<ЛФ>, <Выражение> \{, <Параметры>\})$$

Результатом вызова *select*-функции является объект аналогичного типа, что ее *второй* фактический аргумент, но содержащий только те операнды выражения, на которых логическая функция (*ЛФ*) возвращает *true*-значение. Третий необязательный аргумент функции *select* определяет дополнительные параметры, передаваемые *ЛФ*. В качестве выражения могут выступать список, множество, сумма, произведение либо произвольная функция. Функция *remove* является обратной к *select*-функции. Простой фрагмент иллюстрирует применение функций *select* и *remove* для циклических вычислений:

```
> select(issqr, [seq(k, k = 1 .. 420)]);
      [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
> LF := x -> `if` (x >= 42 and x <= 99, true, false): select(LF, [seq(k, k = 1 .. 68)]);
      [42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68]
> remove(LF, [seq(k, k = 1 .. 68)]);    ⇒ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41]
```


Следующий достаточно простой пример иллюстрирует применение SEQ-процедуры

```
> G := Array(1..2, 1..3, 1..4): SEQ([k, 1..2], [j, {1, 2, 3}], [h, 2..3], "assign('G'[k,j,h]=k+j^6+h)");
  ArrayElems(G); => {(1, 1, 2) = 4, (1, 1, 3) = 5, (2, 1, 2) = 5, (2, 1, 3) = 6, (1, 2, 2) = 67, (1, 2, 3) =
  68, (1, 3, 2) = 732, (1, 3, 3) = 733, (2, 2, 2) = 68, (2, 2, 3) = 69, (2, 3, 2) = 733, (2, 3, 3) = 734}
```

Специальные циклические структуры пакета Mathematica. Из средств такой группы можно отметить ряд довольно интересных, некоторые из которых имеют различного уровня аналогии с подобными средствами пакета *Maple*. Так, функция **Nest**[*G*, *expr*, *n*] возвращает выражение, определяемое применением *G* к *expr* *n* раз, например:

Прямого аналога у *Maple* для данной функции нет, однако в определенной мере его заменяет наша [15,45] достаточно простая процедура $Nest(F, x, n)$, возвращающая в том числе и указанный выше первый результат, а именно:

При этом, в отличие от функции **Nest** пакета *Mathematica* процедура *Nest* позволяет использовать в качестве уровней вложенности имена функций из заданного списка *F*.

Однако в целом, средство пакета *Mathematica* более предпочтительно при генерации вложенных выражений и, прежде всего, чистых функций.

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

```
In[1023]:= NestList[S, y, 6]
Out[1023]= {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[1024]:= NestList[Tan, 6.8, 6]
Out[1024]= {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 пакета *Maple*, как это иллюстрирует следующий достаточно простой фрагмент:

```
> NestList := (F, x, n) -> [seq(Nest(F, x, k), k=0 .. n)];
> NestList(S, y, 5);      => [y, S(y), S(S(y)), S(S(S(y))), S(S(S(S(y))))]
> NestList(tan, 6.8, 6); => [6.8, 0.568340, 0.638629, 0.742415, 0.917528, 1.30655, 3.69585]
```

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 следующего вида:

```
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[1044]:= Fold[G, x, {a, b, c, d, h, g, s, w, p, q}]
Out[1044]= 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 конструкции, представленной последним примером следующего фрагмента:

```
> FoldList := proc(G, x) local a, b, k;
`if`(nargs=2, ERROR("procedure call should has at least 3 arguments"), assign(a=x, b=[x]));
seq(op([assign('a'=Nest(G, [a, k], 1)), assign('b'=[op(b), a])), k=args[3 .. -1]), b end proc;
> FoldList(G, x, a, b, c); => [x, G(x, a), G(G(x, a), b), G(G(G(x, a), b), c)]
> S := () -> `+`(args): FoldList(S, 6.8, 4.2, 4.7, 6.7); => [6.8, 11.0, 15.7, 22.4]
> FoldList(G, x, a, b, c, d, f, h, w)[-1]; => G(G(G(G(G(G(G(x, a), b), c), d), f), h), w)
```

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

```

In[1045]:= MapIndexed[G, {a, b, c, d, f, g, h}]
Out[1045]= {G[a, {1}], G[b, {2}], G[c, {3}], G[d, {4}], G[f, {5}], G[g, {6}], G[h, {7}]}
In[1046]:= MapIndexed[G, x*Sin[x] + y*Cos[y]]
Out[1046]= G[y Cos[y], {1}] + G[x Sin[x], {2}]
> MapIndexed := proc(F, e) local a, b; a:=[op(e)]; [seq(F(a[b], {b}), b=1 .. nops(a))] end proc:
> MapIndexed(G, [a,b,c,d,f,h]); => [G(a, {1}), G(b, {2}), G(c, {3}), G(d, {4}), G(f, {5}), G(h, {6})]
> MapIndexed(G, x*sin(x)+y*cos(y)); => [G(x sin(x), {1}), G(y cos(y), {2})]

```

Тогда как в *Maple* аналогом этому формату может выступать процедура *MapIndexed*, с которой можно познакомиться в последнем примере фрагмента. Подобным 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*. Функция *MapAll[F,x]* возвращает результат применения *F* к каждому подвыражению выражения *x*. Следующий фрагмент иллюстрирует сказанное:

```

In[1047]:= Map[G, {a, b, c, d, e, f, g, h}]
Out[1047]= {G[a], G[b], G[c], G[d], G[e], G[f], G[g], G[h]}
In[1048]:= Map[G, x*Cos[x] + y*Sin[y]]
Out[1048]= G[x Cos[x]] + G[y Sin[y]]
In[1049]:= MapAll[G, x*Cos[x] + y*Sin[y]]
Out[1049]= G[G[G[x] G[Cos[G[x]]]] + G[G[y] G[Sin[G[y]]]]
> map(G, [a, b, c, d, e, f, g, h]); => [G(a), G(b), G(c), G(d), G(e), G(f), G(g), G(h)]
> map(G, x*sin(x) + y*cos(y)); => G(x sin(x)) + G(y cos(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, ЛУ]* возвращает подсписок списка *L*, начиная с его начала, чьи элементы удовлетворяют заданному логическому условию (*ЛУ*), например:

```

In[1050]:= TakeWhile[{1.4, 2.2, 4.3, 4.8, 6.3, 6.8, 68, 43, 48, 72, 14, 22}, # <= 70 &]
Out[1050]= {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[1051]:= NestWhile[Tan, 13, # > 0 &]
Out[1051]= Tan[Tan[Tan[Tan[Tan[Tan[Tan[Tan[13]]]]]]]]
> NestWhile := proc(G::symbol, y, t::symbol, z) local a, b, c; assign(a = y, b = 0);
do a := evalf(Nest(G, a, 1)); b := b + 1;
if t(a, z) then return Nest(G, y, b) end if end do end proc:
> NestWhile(tan, 13, `>`, 0); => tan(tan(tan(tan(tan(tan(tan(tan(13))))))))
> NestWhile(ln, 20110000000/1942, `>`, 0); => ln(ln(ln(ln(ln(10055000000/971))))))

```

Данная функция располагает еще 4-мя форматами кодирования, обеспечивающими более сложные вычисления. Между тем все форматы вполне успешно моделируются процедурами пакета *Maple*, например, первому формату соответствует одноименная процедура *NestWhile*, приведенная в предыдущем фрагменте. Так, первый параметр процедуры *NestWhile* определяет имя функции, второй – выражение, к которому она применяется рекуррентно, пока результат этого применения не удовлетворит условие, базирующееся на третьем и четвертом аргументах процедуры, как это иллюстрирует пример предыдущего фрагмента. Наряду с отмеченными *Mathematica* располагает и другими интересными средствами данного типа, здесь не рассматриваемыми.

3.4. Оценка механизмов циклических управляющих структур программных сред пакетов Maple и Mathematica

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.

Передачи управления безусловного типа в обоих пакетах задаются *goto*-функциями, по которым управление передается в точку *Label* программы (*процедуры*), указанную соответствующей меткой (*Label*). Встроенный язык *Maple* для организации ветвления алгоритмов наряду с рассмотренным предложением *if* допускает применение также безусловных переходов на основе встроенной функции *goto*, кодируемой в формате *goto(Label)*. Между тем, по вполне понятным причинам, обусловленным структурным подходом к программированию, явно проповедуемым пакетом, данная *goto*-функция не документирована; мы обнаружили ее, главным образом, интуитивно. Между тем в целом ряде случаев использование данного средства весьма эффективно, в частности, при необходимости погружения в *Maple*-среду программы, использующей механизм *безусловных* переходов на основе *goto*-функции. Так, достаточно типичным примером являются многочисленные *Fortran*-программы, широко распространенные в научных приложениях. Из нашего опыта следует отметить, что использование *goto*-функции в ряде случаев позволило весьма существенно упростить погружение *Fortran*-программ, относящихся к инженерным приложениям и использующих *goto*-функцию, в *Maple*.

Между тем, средство *Goto*-функции *Mathematica* представляется нам несколько более предпочтительным уже по той причине, что она в отличие от одноименного средства пакета *Maple* может использоваться и вне процедур, а также тем, что повсюду ее *Label* является *локальной* относительно данной *Goto*-функции, т.е. она не переопределяема, что очень важно с точки зрения обеспечения корректности безусловных переходов. В

данном контексте *goto*-функция *Maple* может использоваться лишь в теле процедур, а ассоциированная с ней метка носит *глобальный* характер, требуя отдельных приемов для обеспечения ее сохранности на момент выполнения процедуры без нарушений в вычислительной среде вне самой процедуры. Нами такие средства представлены [45], однако в стандартной поставке пакета они отсутствуют. Конечно, это можно было бы мотивировать тем, что делается акцент на структурном программировании, однако в среде самого пакета *goto*-функция используется, переходя из версии в версию, меняя лишь свой внутренний номер встроенной функции, например, в *Maple 8, 10, 11 – 182, 219 и 226* соответственно. Уж коли сам пакет использует средство, то не стоило бы его возбранять для пользователей, среди которых немало достаточно опытных, чтобы его использовать себе на пользу, а не во вред. В этом отношении *Mathematica* располагает более внятной и естественной позицией, тогда как механизм ее *Goto*-функции более естественен, хорошо защищен от модификации меток и допускает их определение в процедурах в качестве *локальных* переменных, хотя, вообще говоря, это необязательно.

Циклические структуры также имеют *самое* непосредственное отношение к изменению последовательного порядка вычислений, т.е. становятся повторяющимися. Конечно, циклические структуры несложно преобразовывать в последовательностные, однако это не имеет особого смысла, практически, ни с какой точки зрения. Поэтому в языке пакета *Maple* имеется циклическое предложение «*while_do*», которое предназначено для многократного вычисления предложений или их последовательностей на основе указанных его переменных (*переменных цикла*), принимающим указанное множество значений. При этом, общая группа предложений «*for_while_do*» включает **10** базовых форматов кодирования, которые поддерживают довольно изощренные циклические вычисления: от простейших циклов до *вложенных* любого уровня при разнообразных условиях по прекращению цикла *любого* уровня и организации цикла при различных значениях переменных цикла, включая произвольные выражения. В частности, цикл может быть организован по всем либо соответствующим подвыражениям выражения.

В свою очередь, одна из базовых циклических структур *Mathematica* основывается на *For*-функции, имеющей достаточно простой общий формат кодирования: начиная с заданного значения *A*, циклически вычисляется собственно тело цикла, содержащее предложения *Mathematica*-языка, с циклическим наращением переменной цикла на величину *B* до тех пор, пока некоторое логическое условие не примет значения *True*. Данная конструкция допускает любое число вложений, однако при большом уровне вложенности и обеспечении выходов во внешние уровни становится малообозримой. В качестве другого достаточно широко используемого средства в пакете *Mathematica* для организации циклических вычислений является функция *Do[.]*, которая имеет **5** форматов кодирования, однако на базе данной функции возможно организовывать лишь относительно несложные циклические конструкции. Итак, из опыта работы с рассмотренными *циклическими* структурами обоих пакетов из базовых на наш взгляд следует отдать предпочтение пакету *Maple*, тогда как по специальным циклическим структурам пакет *Mathematica* выглядит предпочтительнее из-за предоставляемых ими широких возможностей. Правда и в *Maple* можно запрограммировать подобные средства, но их нужно еще запрограммировать при полной пригодности во втором.

Глава 4. Механизм типирования объектов в среде обоих пакетов

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. Так, например, в следующей весьма простой *Maple*-процедуре *Agn*:

```
> Agn := proc(a::{0, 1, 2, 3}, b::integer) a + b end proc:
```

```
> Agn(3, 72); Agn(63, 72); ⇒ 75
```

Error, invalid input: Agn expects its 1st argument, a, to be of type {0, 1, 2, 3} but received 63 для ее формальных аргументов a и b тип определяется перечислением допустимых значений и предикатной функций *integer* соответственно; нарушение типа в вызове процедуры *Agn* инициирует соответствующую *ошибочную* ситуацию. 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 в контексте сравнения рассматриваемых нами пакетов *Maple* и *Mathematica*. И в ряде наших изданий механизм типизации рассмотрен достаточно детально [8-13]. Именно с точки зрения развитости механизма типизации мы рассматриваем оба пакета *Maple* и *Mathematica*, как наиболее развитые и популярные на сегодня универсальные CAS.

4.1. Механизмы типирования объектов в среде пакета Maple

Согласно аксиоматике *Maple* под «*типом*» понимается *T*-выражение, распознаваемое *type*-функцией пакета и иницилирующее возврат логического значения $\{true|false\}$ на некотором множестве допустимых *Maple*-выражений. В общем случае *T*-выражения относятся к одной из четырех групп: (1) *системные*, определяемые идентификаторами языка $\{float, integer, list, set \text{ и др.}\}$; (2) *процедурные*, когда тип входит в качестве аргумента в собственно тестирующую функцию $\{type(T, Tin)\}$; (3) *приписанные* и (4) *структурные*, представляющие собой *Maple*-выражения, отличные от строковых и которые могут интерпретироваться как тип $\{set(Id = float)\}$. Дополнительно, к пятой группе возможно отнести типы, определяемые модульными средствами пакета, и типы, определенные пользователем. Несколько детальнее с данной классификацией типов ознакомиться можно, например, в наших книгах [9-14, 28-39, 42-44]. В качестве типа может выступать как простой тип, так и произвольно сложный структурированный тип. В некоторых случаях для проверки типов используются также выражения формата $V::T$; в первую очередь этот формат используется для автоматической проверки выражений, которые передаются процедуре в качестве фактических аргументов.

В зависимости от версии пакет распознает целый ряд простых типов, так для версий 8, 10 и 11 пакет распознает 176, 202 и 209 простых типов соответственно. Примерами таких типов, имеющих собственные имена, являются *integer, list, set, table, procedure, module, odd* и др. Более сложные типы определяются на основе более простых типов с использованием специальных грамматических правил [9]. Класс выражений, которые представляют собой собственно распознаваемые типы, тестируется типом, который называется *type*, а именно: *type(T, type)* возвращает *true* тогда и только тогда, когда *T* – допустимый тип. В частности, *type(type, type)* возвращает *true*. Классификация типов может быть проведена на основе того, как они определяются в пакете, а именно:

– *поверхностные типы* являются наиболее простыми; они определяются в ядре пакета и характеризуют самые общие свойства выражений. Например, поверхностный тип

set определяет множество. Проверка выражения на поверхностный тип производится очень быстро независимо от размера тестируемого выражения;

– произвольно сложные *структурированные типы* образуются путем комбинирования более простых типов. Они позволяют классифицировать выражения, базирываясь на их внутренней структуре. Так, тип *set(prime)* характеризует множества, членами которых являются простые числа. Тестирование выражений на данный тип требует не только проверки на поверхностный тип *set*, но также проверки каждого элемента множества на тип *prime*. В целом ряде случаев тестирование выражения на структурированный тип может потребовать полного, рекурсивного обхода выражения, посему требуемое на такую проверку время часто увеличивается с размером выражения;

– *определенный тип* – это тип, в основе тестирования на который лежит произвольный алгоритм. Для определения такого типа *T* достаточно приписать глобальному имени '*type/T*' выражение структурированного типа или соответствующую процедуру. При этом, нет каких-либо правил для определения данных типов – алгоритмы могут быть как очень простыми, так и произвольно сложными. Соответственно и требуемое для проверки выражения на такой тип *T* время может быть довольно значительным.

Уже неоднократно упоминавшееся понятие *выражения*, хорошо знакомое читателю, обладающему определенной компьютерной грамотностью, является одним из самых фундаментальных понятий пакета *Maple*. Понятие выражения аккумулирует такие конструкции языка как константы, идентификаторы, переменные, данные, структуры, а также функции, процедуры, модули и т. д. К выражениям в полной мере можно относить, в частности, и такие конструкции языка, как процедуры, ибо их определения допустимо непосредственно использовать при создании сложных выражений. Детальнее на этом вопросе внимания не будет акцентироваться, отсылая заинтересованного читателя, к таким, например, книгам как [9-14,28-39,42-44,46-60].

Для тестирования выражений на типы пакет *Maple* располагает довольно развитыми средствами, которые базируются на специальных тестирующих функциях *typematch*, *type* и процедуре *whattype*, имеющих следующие форматы кодирования, а именно:

{type | typematch}(Maple-выражение, {Тип | Множество типов})
whattype(Выражение)

где в качестве *первого* аргумента выступает произвольное допустимое *Maple-выражение*, а в качестве *второго* указывается идентификатор требуемого *Типа* либо их *множество*. Булева функция *{type | typematch}* возвращает значение *true*, если значение выражения *Maple* имеет *тип*, определяемый ее *вторым* аргументом, и *false*-значение в противном случае. При определении *вторым* аргументом множества типов функция *{typematch | type}* возвращает *true*-значение только тогда, когда *тип* значения выражения будет принадлежать данному множеству, и *false*-значение в противном случае. Более того, следует помнить, что в качестве второго аргумента функции может использоваться только *множество* (*{}*)-конструкция, а не *[]*-конструкция; данное обстоятельство может на первых порах вызывать ошибки пользователей, ранее работавших с пакетом *Mathematica*, синтаксис которого для списочной структуры использует именно первую конструкцию, да и в целом достаточно непривычен для уже имеющих опыт программирования). Для второго

аргумента `{type | typematch}`-функции допускается более 209 определяемых пакетом типов (*Maple 11*), из которых здесь упоминаются лишь наиболее употребительные на первых шагах программирования в *Maple*, а также типы, которые непосредственно связаны с рассматриваемыми нами конструкциями языка пакета: идентификаторы, текст, числовые и символьные данные, структуры данных и др. При этом, *typematch*-функция имеет более расширенные средства тестирования типов, поэтому детальнее с ней можно ознакомиться, например, в наших книгах [9-14,28-39,42-44].

Наконец, по тестирующей процедуре *whattype*(Выражение) возвращается собственно тип выражения, определяемого ее фактическим аргументом. При этом отметим, что эта процедура в некоторых случаях решает задачу тестирования более эффективно, например, для *последовательностных* структур и в случае неизвестного типа, позволяя избегать перебора подвергающихся проверке типов. В книге [44] (табл. 5) приведены некоторые *допустимые* типы, тестируемые функцией `{type | typematch}`, используемые в качестве значений ее второго аргумента вместе с их назначением. При этом, смысл большинства типов достаточно прозрачен и особого пояснения не требует. Однако, в табл. 5 отражен далеко не полный перечень типов, распознаваемых пакетом. Данный перечень значительно шире и с каждым новым релизом пакета пополняется новыми типами. Так, для *Maple 8* такой перечень содержит 176 типов, *Maple 9* – 182, *Maple 10* – 202 и *Maple 11* – 209. Пользователь также располагает возможностью расширять пакет новыми типами и нами был определен ряд новых и важных типов, отсутствующих в *Maple*. Все они описаны в наших книгах [10,28-39,42-44] и включены в Библиотеку [45]. Ниже мы представим механизм определения пользовательских типов. Следующий достаточно простой фрагмент иллюстрирует применения функций `{type, typematch}` и *whattype*-процедуры для тестирования типов выражений, а именно:

```
> [whattype(64), whattype(x*y), whattype(x+y), whattype(a..b), whattype(a:name),
whattype([]), whattype(a <= b), whattype(a^b), whattype(Z <> T), whattype(h(x)),
whattype(a[3]), whattype({}), whattype(x,y), whattype(table()), whattype(3 <> 10),
whattype(a..b), whattype(47.63), whattype(A and B), whattype(14/22), whattype(array(1
..3, [])), whattype(proc() end proc), whattype(a.b), whattype(module() end module),
whattype(hfarray(1 .. 3)), whattype("a+b"), whattype(AVZ)];
[integer, *, +, .., ::, list, <=, ^, <>, function, indexed, set, exprseq, table, <>, .., float, and, fraction,
array, procedure, function, module, hfarray, string, symbol]
> A:= -64.42: Art:= array(1..3, 1..6): S:= 67 + 32*I: V:= -57/40: L:= {5.6, 9.8, 0.2}: T:= table():
LL:=[[V, 68, 42], [G, 47, 63]]: K:= "Академия": W:= array(1 .. 100): W[57]:= 99:
> [type(A, 'alnum'), type(Art, 'array'), type(`true`, {'boolean', 'logical'}), type(S,
'complex'(integer)), type(56*Pi, 'constant'), type(56/28, 'even'), type(1.7, 'float'), type(A,
'fraction'), type(A*infinity, infinity), type(V, 'integer'), type(L, 'set'(float)), type(LL,
'listlist'), type(Art, 'matrix'), type(AVZ, 'symbol'), type(A,'negative'), type(V,'negint'),
type(S,'numeric'), type(A,'rational'), type(infinity, 'realcons'), type(K, 'string'), type(Art,
'table'), type(T, 'table'), type(real, 'type'), type(W, 'vector'), type(hfarray(1.. 3),'hfarray')];
[false, true, true, true, true, true, true, true, false, true, false, true, true, true, true, true, false, false, false,
true, true, true, true, false, true, true]
> map(whattype,[H, A, eval(Art),`true`, eval(T)]); => [symbol, float, array, symbol, table]
```

```
> map(type, [64, 47/59, 14.22, `H`, G[3], "TRG", 'literal']); => [true, true, true, false, false, true]
> map(type, [range, float, set, list, matrix, string, symbol, array, Array, matrix, `..`, `**`, 'type']);
[true, true, true, true, true, true, true, true, true, true, true, true]
```

Приведенный сводный фрагмент охватывает многие типы, тестируемые функциями `{type, typematch}` и `whattype`-процедурой, достаточно прозрачен и особых пояснений не требует. Как уже отмечалось, `whattype`-процедура дает возможность тестировать последовательностные структуры, тогда как `{type | typematch}`-функция этого сделать не в состоянии. Более того, в отличие от вторых, `whattype`-процедура ориентирована, прежде всего, на тестирование выражений, структурно более сложных, чем данные и структуры данных. При этом следует иметь в виду, что данные и их структуры также можно рассматривать как частный случай более общего понятия *выражения*.

Так как *выражение* представляет собой более широкое понятие, чем данные (значения), то для тестирования их типов *Maple*-язык располагает достаточно развитым набором средств. Прежде всего, для прямого определения типа выражения используется ранее упомянутая процедура `whattype`, которая имеет очень простой формат кодирования и возвращает тип заданного выражения, если он является одним из нижеследующих:

```
`*` `+` `.` `..` `::` `<` `<=` `<>` `=` `^` `|` `|` `and` array complex
complex(extended_numeric) exprseq extended_numeric float fraction function
hfarray implies indexed integer list module moduledefinition `not` `or`
procedure series set string symbol table uneval unknown `xor` zppoly
Array Matrix SDMPolynomial Vector[column] Vector[row]
```

Приведенный перечень идентифицируемых типов выражений, включая некоторые данные и их структуры, представлен для *Maple 10,11*, для более младших версий этот перечень несколько короче. Следует еще раз отметить, хотя тип последовательности (`exprseq`) не является типом, определяемым функциями `{type, typematch}`, вызывая на ней ошибочную ситуацию, однако он идентифицируется `whattype`-процедурой. При этом, процедура возвращает только тип высшего уровня вложенности выражения в соответствии с приоритетным порядком составляющих его операторов. Следующий простой пример иллюстрирует применение тестирующей `whattype`-процедуры:

```
> [whattype(64), whattype(x*y), whattype(x + y), whattype(x <= y), whattype(a <> b),
whattype([]), whattype(a = (b + c)), whattype(a^b), whattype(Z), whattype(h(x)),
whattype(a[22]), whattype({}), whattype({}), whattype(x, y, z, h), whattype(table()),
whattype(x..y), whattype(6.8), whattype(A and B), whattype(proc() nargs end proc),
whattype(module() end module), whattype(14/22), whattype("SVG"), whattype(a.b),
whattype(hfarray(1..14))];
[integer, *, +, <=, <>, list, =, ^, symbol, function, indexed, set, list, exprseq, table, .., float, and,
procedure, module, fraction, string, function, hfarray]
```

При этом, имеют место следующие идентификации типов `whattype`-процедурой:

```
{+ | -} → +      {/ | *} → *      {>= | <= } → <=      {> | < } → <      {** | ^ | sqrt(a)} → ^
{a | a."b" | a.b | a.`b` | `a`.b | `a`.`b`} → symbol      {"a" | "a"."b" | "a".`b` | "a".b} → string
{array | vector | matrix} → array
```

что необходимо учитывать при использовании указанной процедуры тестирования. Обусловлено это обстоятельством, что предварительно вызов процедуры *whattype(A)* вычисляет и/или упрощает выражение *A*. Тогда как для расширенного тестирования типов выражений служит вышерассмотренная в связи с данными и их структурами функция *{type | typematch}*. Приведем простой пример на применение *type*-функции для тестирования различных выражений, а именно:

```
> [type(sqrt(x)+a/b,'anything'), type(a**b, `**`), type(x*y+z^a, dependent(z)), type('a.b', `.`),
type(x^a-3*x+x^4-47 = b, 'equation'), type(x^3+4*x-56, 'expanded'), type(ifactor(1998),
'facint'), type(h!, `!`), type(F(x), 'function'), type(G[47, 96, 42, 68, 63, 89], 'indexed'),
type(5*y^4 + x^3 - 47*x, 'quartic(y)'), type(arctanh, 'mathfunc'), type('RANS', 'symbol'),
type(ln, 'mathfunc'), type(14/22, 'numeric'), type(A -> B, 'operator'), type({G = 63, V = 68},
'point')]; => [true,true,true,true,true,true,true,true,true,true,true,true,true,true,true,true]
```

Следует отметить еще одну полезную тестирующую функцию *hastype(expr, t)*, вызов которой возвращает *true*-значение, если выражение *expr* содержит подвыражение типа, заданного значением *t*, и *false*-значение в противном случае, например:

```
> map2(hastype, (a*x + 10)/(sin(x)*y + x^4), ['integer', 'symbol', 'function', `**`, `+`,
symbol^integer]); => [true, true, true, true, true, true, true]
> map2(hastype, (10^Kr + 96)/(Art^17 + 89), [even^symbol, symbol^odd]); => [true, true]
```

При этом, можно проводить тестирование выражений и относительно простых типов, и структурных, как это иллюстрирует второй пример предыдущего фрагмента.

Дополнительно к рассмотренным средствам тестирования следует отметить 6 весьма полезных процедур из класса так называемых *is*-процедур. Прежде всего, процедура *isprime(n)* осуществляет стохастическую проверку *n*-числа на предмет его простоты, тогда как процедура *issqr(n)* тестирует наличие точного квадратного корня из целого *n*-числа. Процедура *is(V, C)* тестирует наличие у *V*-выражения указанного свойства *C*, в то время как процедура *ispoly(P, {1 | 2 | 3 | 4}, x)* тестирует является ли *P*-выражение полиномом степени {1 | 2 | 3 | 4} по ведущей *x*-переменной. Процедура *iscont(W, x=a..b { 'closed' })* тестирует факт наличия непрерывности *W*-выражения на интервале *[a, b]* по *x*-переменной, включая его граничные точки, если закодирован необязательный *closed*-аргумент. Наконец, процедура *isdifferentiable(V, x, n)* тестирует *V*-выражение, содержащее кусочно-определенные функции (*abs*, *signum*, *max* и ряд др.), на предмет принадлежности его к C^n -классу по ведущей *x*-переменной. Следующий достаточно простой фрагмент иллюстрирует использование упомянутых *is*-процедур, а именно:

```
> iscont(x*sin(x) + x^2*cos(x), x=-Pi..Pi, 'closed'); => true
> isprime(1999), isprime(1984979039301700317); => true, false
> issqr(303305489096114176); => true
> x := 32: is(3*sin(x) + sqrt(10) + 0.42, 'positive'); => true
> ispoly(3*h^4 - 10*h^3 + 32*h^2 - 35*h + 99, 'quartic', h); => true
> isdifferentiable(y*(abs(y)*y + signum(y)*sin(y)), y, 2); => false
> Groebner[IsZeroDimensional]([x^2-7*x*z+14, x*y^2+y*z^3, 6*y^2-22*z^3]); => true
```

Наряду с отмеченными пакет располагает целым рядом *Is*-процедур, определения которых находятся в соответствующих пакетных модулях и чей вызов обеспечивается по конструкциям, например, формата *Модуль[процедура](...)*. Так, последний пример предыдущего фрагмента производит проверку того, будет ли множество полиномов по ведущим переменным $\{x, y\}$ иметь конечное число решений над алгебраическим замыканием поля коэффициентов. В свою очередь нами также был создан целый ряд *is*-процедур для тестирования различных свойств *Maple*-объектов [9,45]. В частности, наши процедуры *isFile* и *isDir* обеспечивают тестирование реального существования файлов и каталогов в файловой системе компьютера, как иллюстрирует пример:

```
> isFile("F:/Archive/Books/Cellular_Automata/HS_CA_Book.pdf"); ⇒ true
> isDir("F:/"), isDir("D:\Math_myLib"), isDir("C:/Temp\RANS"); ⇒ true, true, false
```

Под *структурированным* типом в *Maple* понимается выражение, отличное от имени (и не идентифицируемое отдельным словом) и которое может интерпретироваться как тип. В общем случае структурированный тип представляет собой *алгебраическое выражение* от известных пакету типов, в основных чертах наследующее структуру тестируемого выражения. На основе механизма *структурированных* типов предоставляется хорошая возможность тестировать как структуру выражения в терминах его *подвыражений*, так и их типовую принадлежность в терминах базовых типов языка. Особый смысл такая возможность приобретает в задачах, которые имеют дело с символьной обработкой и вычислениями. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> type([68, V, sqrt(Art)], [integer, name, `^`]), type(63^sin(x), integer^trig); ⇒ true, true
> type(G(x)^cos(y), function^symmfunc), type(sqrt(F(x)), sqrt(function)); ⇒ true, true
> type(68 + sin(x)*cos(y), `&+`(integer, `&*`(trig, symmfunc))); ⇒ true
> type(sqrt(Art^22+Kr^15 + 10), sqrt(`&+`(name^even, symbol^odd, integer))); ⇒ true
> type((G(x)+cos(y))^(68*I+sin(x)), (`&+`(function, trig)^`&*`(complex, trig(x)))); ⇒ true
```

Из примеров представленного фрагмента довольно прозрачно прослеживается очень тесная взаимосвязь между структурным деревом термов тестируемого выражения, а также выбранным для него структурированным типом. Более детально с вопросами структурированных типов пакета можно ознакомиться по книгам [10,28-39,42-44] или оперативно по конструкции вида *?type, {anyfunc | identical | specfunc | structure}* в среде текущего сеанса пакета. Следует иметь в виду, что по конструкции следующего вида:

type('Id', name(Tun))

предоставляется возможность тестировать *тип* значения, которое было присвоено *Id*-идентификатору, как это иллюстрирует следующий весьма простой фрагмент:

```
> GS := cos(x)*sin(y) + F(z) + 2010: AV:= 42 .. 68: type('AV', name(range)); ⇒ true
> SV := sqrt(Art + Kr): type('SV', name(sqrt(`&+`(name, symbol)))); ⇒ true
> type('GS', name(`&+`( `&*`(symmfunc, trig), function, even))); ⇒ true
```

Как следует из приведенного достаточно простого фрагмента, описанный механизм позволяет обеспечивать тестирование типов значений определенных переменных в терминах как базовых типов, так и *структурированных* типов, существенно расширяя

возможности пакета. Для автоматического тестирования типов выражений, прежде всего, в процедурных конструкциях, используется (::)-оператор типирования, детально рассматриваемый в связи с процедурными объектами. Еще 2 тестирующие функции, позволяющие проводить структурное тестирование типов как самих выражений, так и составляющих их подвыражений, вполне заслуживают специального внимания. В частности, *hastype*-функция имеет следующий формат кодирования, а именно:

hastype(Выражение, Структурированный тип)

и возвращает *true*-значение только тогда, когда *Выражение* содержит подвыражения указанного *структурированного типа*, например:

```
> Kr := 14*sin(x) + 22*cos(y)/AV + sqrt(AG^2 + AS^2)*TRG + H^3 - 63*sin(z) - Catalan:
> map2(hastype,Kr,[name^integer,constant,sqrt,fraction,`*`]); => [true, true, true, true, true]
> map2(has,Kr,[14*sin(x),1/AV, AG^2+AS^2,H^3,-63*sin(z)]); => [true, true, true, true, true]
> [hasfun(Kr, sin, z), map2(hasfun, Kr, [sqrt, sin, cos])]; => [true, [false, true, true]]
```

В контексте приведенного фрагмента уже несложно заметить, что в качестве второго аргумента функции *hastype* вполне могут выступать не просто рассмотренные выше типы, допустимые *Maple*, а также их структурированные конструкции, отвечающие структурам входящих в тестируемое выражение подвыражений. Таким образом, на основе этой функции можно не только тестировать *выражения* на типы составляющих их *подвыражений*, но также на их *структурную* типизацию в рамках общей структуры исходного выражения. В целом ряде случаев это достаточно актуальная задача.

По тестирующей функции *has*(*Выражение*, *Подвыражение*) обеспечивается проверка на наличие вхождения в выражение, заданное первым аргументом, указанного вторым аргументом подвыражения. Если в качестве второго аргумента *has*-функции указан список подвыражений, тогда в этом случае выражение подвергается тестированию по каждому из подвыражений списка. Функция *has* возвращает *true*-значение, если факт вхождения установлен, в противном случае возвращается *false*-значение. При этом, в случае списка подвыражений в качестве второго аргумента *has*-функция возвращает *true*-значение тогда, когда имеет место факт вхождения в тестируемое выражение по меньшей мере одного из *подвыражений* указанного списка. Третий пример фрагмента иллюстрирует сказанное. Данная функция представляет большой интерес в задачах структурного анализа *символьных* выражений, поступающих в качестве определенной входной информации для процедур *символьных* обработки и вычислений.

Наконец, по тестирующей функции *hasfun*(*V*, *Id-функции* {*x*}) возвращается значение *true*, если заданное первым фактическим аргументом *V*-выражение будет содержать вхождение функции, заданной своим идентификатором, и, возможно, от указанной третьим необязательным аргументом *ведущей x*-переменной. Так, последний пример предыдущего фрагмента не только иллюстрирует вышесказанное, но и указывает на то, что, в частности, *hasfun*-процедура не тестирует факт вхождения *sqrt*-функции в выражения. В этой связи нами была создана процедура *Hasfun* с набором аргументов, аналогичным стандартной *hasfun*-процедуре [45], которая избавлена от указанного недостатка *hasfun*-процедуры, как это иллюстрируют следующие примеры:

```
> Hasfun(SVG(sin(xy + ab) + sqrt(zy)) + ln(z), [ln, g, sqrt], zy);      ⇒ true
> [Hasfun(Kr, sin, z), map2(Hasfun, Kr, [sqrt, sin, cos])]; ⇒ [true, [true, true, true]]
```

Типы, определяемые пользователем. Учитывая весьма большую важность механизма типизации выражений, *Maple* предоставляет достаточно большое число встроенных типов, а также дает возможность пользователю самому создавать собственные типы, исходя как из его конкретных нужд, так и из более общих соображений. Начиная уже с 8-й версии, пакет пополнен модулем *TypeTools*, пять процедур которого служат для расширения перечня типов, стандартно поддерживаемых *type*-функцией пакета. Со своей стороны, еще ранее мы определили процедуру *usertype*, обеспечивающую ряд массовых операций по работе с типами пользователя, поддерживаемых процедурами с именами формата '*type/name*', где *name* – имя пользовательского типа. Такие типы определяются в виде процедур, имеющих следующую организацию, а именно:

```
`type/usertype` := proc(Выражение) <Тело процедуры> end proc { : ; }
```

при этом, такая процедура должна возвращать значение {*true* | *false*}. В качестве весьма простого примера приведем процедуру *`type/parse`*, возвращающую *true*, если строка *s* содержит корректное *Maple*-выражение, и *false* в противном случае. При этом, через глобальную переменную *_ResTest* возвращается само выражение в случае *true*.

```
> `type/parse` := proc(s::string) try assign('_ResTest' = parse(s, 'statement'))
                                catch : return false end try; true end proc;

> type("(y^4)^2 + x^2^2", parse); ⇒ false
> _ResTest; ⇒ _ResTest
> type("(y^4)^2 + x^(2^2)", parse); ⇒ true
> _ResTest; ⇒ y^8 + x^4
> type("a := proc() `+(args)/nargs end proc;", parse); ⇒ true
> _ResTest; ⇒ _ResTest
> eval(%); ⇒ proc() `+(args)/nargs end proc
> type(`type/parse`, boolproc); ⇒ true
```

Уже такой простой подход позволяет довольно существенно расширять стандартный набор типов, распознаваемых *Maple*. Нами было создано новых и модифицировано стандартных более 35 типов [8-14,30-38,42-45]. Наряду с представленным подходом в данном контексте может оказаться достаточно полезным пакетный модуль *TypeTools*, который используется для расширения стандартной *type*-функции пакета типами, определенными пользователем; между тем, его использование приводит к созданию 2 непересекающихся систем типизации в среде *Maple*, а именно: (1) типов пользователя, базирующихся на конструкциях вида *`type/name` := proc ... end proc*, и сохраненных в *Maple*-библиотеках, и (2) системе, базирующейся на пакетном модуле *TypeTools*.

Уже упомянутая процедура *usertype* может оказаться достаточно полезным средством при организации работы с пользовательскими типами, имена определений которых имеют формат вида *`type/name`*. Следующий фрагмент представляет исходный текст и примеры применения процедуры *usertype*; процедура находится в Библиотеке [45].

```

> usertype := proc() local a, b, c, d, k, j, h;
  if nargs < 2 then if nargs = 0 then assign67(h = [libname], d = NULL)
    elif args[1] = 'active' then assign(a = [anames('procedure')], d = []);
      for k in a do if "" || k[1..5]="type/" then d:=[op(d),"" || "" || k[6..-1]] end if end do;
      return sort(map(convert, d, 'symbol'))
    elif type(args[1], {'mlib', 'mla'}) then assign67(h = [args[1]], d = NULL)
    else error "<%1> is not a Maple library", args[1] end if;
    for j in h do assign('a' = march('list', j), 'b' = []);
      for k in a do c := Search2(k[1], {".m", "type/"});
        if c[1] = 1 and c[2] = length(k[1]) - 1 then b := [op(b), cat(`, k[1][6 .. -3])] end if
      end do;
      d := d, j, sort(b)
    end do;
    d
  elif nargs = 2 then
    if type(args[1], 'symbol') and args[2] = 'code' then
      try type(a, args[1])
      catch "type `%1` does not exist": error "type `%1` does not exist", args[1]
      end try;
      eval(`type/` || args[1])
    elif type(args[1], {'mlib', 'mla'}) and type(eval(cat(`type/`, args[2])), 'boolproc') then
      UpLib(args[1], [cat(`type/`, args[2])])
    end if
    elif type(args[1], {'mlib', 'mla'}) and type(eval(cat(`type/`, args[2])), 'boolproc') and
      args[3] = 'del' then march('delete', args[1], cat(`type/`, args[2]))
    else error "factual arguments are invalid", [args]
    end if
  end proc;
> usertype("C:/Program Files/Maple 11/LIB/UserLib");
  "C:/Program Files/Maple 8/LIB/UserLib", [Lower, Table, Type, Upper, arity, assignable1,
  binary, boolproc, byte, color, complex1, digit, dir, dirax, emptyplot, file, file1, fpath, heap, letter,
  libobj, lower, mla, mlib, mod1, nbinary, nestlist, nonsingular, package, parse, path, plot3dopt,
  plotopt, plotstruct, realnum, relation1, rlb, sequent, setset, ssign, upper]
> usertype('active'); ⇒ [boolproc, dir, file, matrix, mla, mlib, nestlist, parse, vector]

```

Необходимая функция процедуры *usertype* может быть получена использованием ее вызова на кортеже соответствующих фактических аргументов, а именно:

аргументы отсутствуют – возвращает списки всех имен типов формата *`type/name`*, находящихся во всех *Maple*-библиотеках, определяемых предопределенной *libname*-переменной. Функция позволяет получать только имена типов, зарегистрированных, используя вышеуказанный метод. В частности, процедура не может использоваться для получения встроенных типов и созданных на основе модуля *TypeTools*. Каждый

возвращаемый указанный список предваряется полным путем к *Maple*-библиотеке, содержащей определения типов, чьи имена находятся в списке;

'active' – возвращает список имен вида ``type/name``, активных в текущем сеансе;

L – возвращает список имен типов вида ``type/name``, находящихся в *Maple*-библиотеке, определенной полным путем *L* к ней. Данный список предваряется полным путем к *Maple*-библиотеке *L*, содержащей определения типов;

name, 'code' – возвращает исходный текст процедуры ``type/name``;

L, *name* – помещает определение типа ``type/name`` в *Maple*-библиотеку, определенную полным путем *L* к ней. Процедура ``type/name`` должна иметь *boolproc*-тип. В данном случае вызов процедуры `usertype(L, name)` выводит соответствующие сообщения;

L, *name*, 'del' – удаляет определение типа ``type/name`` из *Maple*-библиотеки, полный путь к которой определен аргументом *L*. Никаких сообщений не выводится.

В процессе использования с пакетом *Maple* версий 6–11 процедура `usertype` проявила себя достаточно эффективным средством при работе с пользовательскими типами. Еще на одном достаточно существенном моменте следует акцентировать внимание. В среде пакета вызов `type(J, 'type')` тестирует выражение *J* на предмет быть допустимым выражением типа, т.е. опознаваемым *встроенной type-функцией* в качестве типа. При этом, по определению под выражением типа *J* понимается выражение, для которого успешно выполняется вызов `type(Выражение, J)`; в качестве *первого* аргумента выступает произвольное *Maple*-выражение. В качестве типов допускаются *системные* типы {float, string, integer, symbol и др.}, типы, определяемые процедурами с именами ``type/name``, а также типы, определяемые присвоением либо комбинацией типов. Однако, данная проверка не столь корректна, как декларируется разработчиками *Maple*. Следующий достаточно простой фрагмент наглядно подтверждает вышесказанное, а именно:

```
> restart; with(TypeTools); ⇒ [AddType, GetType, GetTypes, RemoveType, Type]
> AddType(listodd, L → type(L, 'list'(odd)));
> type([68, 63, 14, 22, 43], 'listodd'), type([63, 43, 47], 'listodd'); ⇒ false, true
> type(listodd, 'type'); ⇒ true
> `type/listeven` := L → type(L, 'list'(even)): type(listodd, 'type'); ⇒ true
> type([68, 14, 22], 'listeven'), type([63, 22, 43], 'listeven'); ⇒ true, false
> `type/neg` := proc(x::{integer, float, fraction}) if x < 0 then true else false end if end proc:
> map(type, [68, -14, 63, -22, 43, -47], 'neg'), type(neg, 'type');
      [false, true, false, true, false, true], false
> UpLib("C:/Temp/userlib", [`type/neg`]): restart; libname:=libname, "C:/Temp/userlib":
  type(neg, 'type'); ⇒ false
> map(type, [68, -14, 63, -22, 42, -45], 'neg'); ⇒ [false, true, false, true, false, true]
> `type/Type` := proc(x::anything) local a; try type(a, x) catch "type `%1` does not exist":
      return false catch : return true end try; true end proc:
> type(neg, 'type'), type(neg, 'Type'); ⇒ false, true
> map(type, [neg, aa, bb, set, mla, dir, file], 'Type'); ⇒ [true, false, false, true, true, true, true]
```

В первой части производится тестирование типов, определенных средствами модуля *TypeTools* и процедурами с именами формы ``type/name``. Оказывается, что во втором случае вызов `type(neg, 'type')` некорректно тестирует *neg*-тип. После чего определяется ``type/Type``-процедура, устраняющая данный недостаток. В заключение фрагмента эта процедура проверяется на предмет корректности, демонстрируя, что она решает задачу тестирования типов успешнее стандартного средства, обеспечивая корректное тестирование типов, определенных любым допустимым в *Maple* способом.

Из представленных в разделе средств тестирования типов данных, структур данных и выражений в их общем понимании можно сделать вполне определенные выводы о возможностях *Maple*-языка в данном весьма важном аспекте его вычислительных как численных, так и алгебраических средств. Поэтому механизму тестирования пакета нами уделялось достаточно серьезное внимание в процессе апробации и применения *Maple*, а также подготовке на данной основе наших изданий и создания собственных средств, как расширяющих его возможности, так и устраняющих выявленные на этом пути недостатки стандартных средств. Прежде всего, это касалось массовых средств в свете существенной ориентации пакета не столько на сугубо численные вычисления, сколько на сугубо алгебраические вычисления и преобразования.

4.2. 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[122]:= FileType["D:\\Math_myLib"]
Out[122]= Directory
In[123]:= FileType["D:\\Math_myLib\\ArtKr.mx"]
Out[123]= File
In[124]:= FileExistsQ["D:\\Math_myLib\\ArtKr.mx"]
Out[124]= True
In[125]:= FileExistsQ["D:\\Math_myLib"]
Out[125]= 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[127]:= 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[127]= 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**:

```
In[224]:= IsFile[x_] := If[FileExistsQ[x], If[! DirectoryQ[x], true, false], false]
In[225]:= IsFile["D:\\Math_myLib"]
Out[225]= false
In[226]:= IsFile["D:\\Math_myLib\\ArtKr.mx"]
Out[226]= true
```

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

```
In[227]:= {FileType["D:\\Math_myLib"], FileType["D:\\Math_myLib\\ArtKr.mx"]}
Out[227]= {Directory, File}
```

Пакет *Mathematica* располагает также рядом других подобных тестирующих средств.

В отличие от 209 типов пакета *Maple 11*, тестируемых встроенной процедурой *type* (не считая значительного набора пользовательских типов, подключенных к пакету посредством библиотеки [45]), *Mathematica 7* 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*. Однако здесь имеется и одно отличие: если функция **ToExpression** корректно работает более, чем с одним выражением, то у процедуры *parse* это вызывает определенные затруднения:

```
In[118]:= ToExpression["c := a + b; m := n - p; d := a + h;"]
In[119]:= %
In[120]:= {c, m, d}
Out[120]= {a + b, n - p, a + h}
> parse("c := a + b; m := n - p; d := a + h; g := d + x", 'statement');
Warning, extra characters at end of parsed string
      c := a + b
      a + b
```

> c, m, d, g; $\Rightarrow a + b, m, d, g$

Последний пример фрагмента иллюстрирует, что процедура *parse* вычисляет только одно выражение из корректных четырех. Именно поэтому была создана процедура *Parse* [45], расширяющая возможности стандартной *parse*, как иллюстрирует пример:

> Parse("c:=a + b; m:=n -p; d:=a + h; g:=d + x"); $\Rightarrow true$

> c, m, d, g; $\Rightarrow a + b, n - p, a + h, a + h + x$

> Parse("c := a + b; m := n^p; d := a + h; g := d + x"); $\Rightarrow false$

> Parse("c := a + b; m := n^p; d := a + h; g := d + x", 't'), t; $\Rightarrow false, ["m := n^p;"]$

Процедура *Parse* в отличие от стандартной *parse* не только тестирует корректность 1 и более выражений, но и через второй необязательный аргумент позволяет получать список некорректных выражений, обнаруженных в процессе такого тестирования. В случае необходимости пользователь может создавать собственные функции *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.

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, как это иллюстрирует следующий весьма простой пример:

In[124]:= Head[x + y]

Out[124]= Plus

In[125]:= Part[x + y, 0]

Out[125]= Plus

> whattype(x+y), op(0, x+y); $\Rightarrow +, +$

In[126]:= Head[{1, 2, 3, 4, 5}]

Out[126]= List

In[127]:= Part[{1, 2, 3, 4, 5}, 0]

Out[127]= List

> whattype([1, 2, 3, 4, 5]), op(0, [1, 2, 3, 4, 5]); $\Rightarrow list, list$

In[128]:= Head[Cos[x]]

Out[128]= Cos

In[129]:= Part[Cos[x], 0]

Out[129]= Cos

> whattype(cos(x)), op(0, cos(x)); $\Rightarrow function, cos$

In[130]:= Head[Type]

Out[130]= Symbol

In[131]:= Part[Type, 0]

Out[131]= Symbol

> **whattype**(Type), **op**(0, Type); \Rightarrow *symbol, symbol*
 > **whattype**(eval(Type)), **op**(0, eval(Type)); \Rightarrow *procedure, procedure*

Как следует из приведенного примера, процедуры *whattype* и *op* пакета *Maple* имеют более широкие возможности по определению типов *Maple*-объектов, чем **Head** и **Part** по определению типов *Mathematica*-объектов, что обусловлено, по-видимому, более ограниченным набором типов, распознаваемых пакетом. Так, модули (процедуры) не тестируются на тип, что является достаточно существенным недостатком. Между тем, использование шаблонов в сочетании с типами, которые распознаются **Head** и **Part**, в определении значений переменных (аргументов) имеет свои положительные черты. В этом случае пакет позволяет представлять классы выражений. 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 шаблоны являются одними из допустимых пакетом

<i>x_h</i>	выражения общего типа <i>h</i>
<i>x_Integer</i>	выражения общего типа Integer
<i>x_List</i>	выражения общего типа List
<i>x_Symbol</i>	выражения общего типа Symbol

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

```
In[244]:= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2} /. x_Real -> 0
Out[244]= {0, 2, 3, 0, 5, 6, 7, 0, 9, 0, 11, 0, 0}
In[245]:= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2} /. x_Real -> Round[x]
Out[245]= {2, 2, 3, 5, 5, 6, 7, 9, 9, 11, 11, 12, 13}
In[246]:= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2} /. x_Real -> Rationalize[x]
Out[246]= {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2}
In[247]:= Map[Rationalize, {1.5, 2, 3, 4.6, 5, 6, 7, 8.9, 9, 10.9, 11, 12.3, 13.2}]
Out[247]= {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[351]:= G[x_Integer, y_Rational] := Sin[x] + Cos[y]
```

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

```
In[352]:= G[68.42, 19/42]
Out[352]= G[68.42, 19/42]
```

```
In[353]:= G[68, 42]
Out[353]= G[68, 42]
In[354]:= G[68.42, 42.68]
Out[354]= G[68.42, 42.68]
In[355]:= G[68, 19/42]
Out[355]= Cos[19/42] + Sin[68]
In[356]:= {MM[42, 68], MM[19.42, 68], MM[68, 19.42], MM[19.42, 19.47]}
Out[356]= {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 невозможно использовать для точного тестирования фактических аргументов, передаваемых функции/процедуре, общие типы которых отличны от допустимых. Тогда как в случае пакета *Maple* использование *типирования* формальных аргументов обеспечивает тестирование типов фактических аргументов с точностью до первого несоответствия типов с возвратом диагностики, что позволяет достаточно легко программно обрабатывать соответствующие особые ситуации, как это иллюстрирует следующий весьма простой фрагмент, а именно:

```
> G := (x::rational, y::integer) -> evalf(sin(x) + cos(y), 4): x := 19.42: G(x, 1947);
Error, invalid input: G expects its 1st argument, x, to be of type rational, but received 19.42
> try G(x, 1947) catch "invalid input:" : G(convert(x, rational), 1947) end try;  => 1.246
```

В реальности подобный механизм обработки ситуаций, вызванных несоответствиями типов формальных и фактических аргументов, достаточно широко используется при программировании самых разнообразных задач как системных, так и прикладных. В нашей Библиотеке [45] немало процедур используют именно данный механизм.

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, нижеприведенный пример представляет достаточно простой модуль **Type**, который обеспечивает тестирование *Mathematica*-объектов на *ListList*-тип, т.е. на тип *вложенного списка*, элементы которого в свою очередь также являются списками одинаковой длины, а именно:

```
In[664]:= 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[665]:= L := {a, b, c}; L1 := {{a, b}, {c, d}, {e, f}}; Type[L, ListList]
Out[665]= False
In[666]:= Type[L1, ListList]
Out[666]= 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 нижеприведенному варианту процедуры **Type** для тестирования двух типов *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 *tuna* 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. Такой подход позволяет довольно просто организовать систему *типирования* пакета, однако он требует определенной *квалификации* пользователя и его довольно затруднительно полностью интегрировать в среду пакета (*подобно случаю пакета Maple*), когда в целом ряде версий пакета можно просто сохранять пользовательские типы непосредственно в его главной библиотеке. В более старших версиях это весьма просто обеспечивается посредством *логического сцепления* библиотек пользователя, содержащих определения типов, с главной библиотекой пакета. Тогда как в *Mathematica* возможно обеспечить автоматическую загрузку в текущий сеанс пакета определения требуемых типов при каждом старте пакета. Между тем, *Mathematica* допускает создание пользовательских типов с сохранением их в отдельном пакете, скажем, "**UserTypes`**", при каждом новом старте пакета который автоматически загружается, обеспечивая содержащиеся в нем определения типов доступными (*активными*) в текущем сеансе. Именно такой подход использовался нами в период подготовки примеров для настоящей книги.

4.3. Оценка механизмов типирования Maple и Mathematica

Прежде всего следует отметить, что набор распознаваемых *Maple* типов существенно больше аналогичного набора для *Mathematica*. Тут же вполне уместно отметить, что средства тестирования типов, обеспечиваемые, например функцией `{typematch | type}` существенно более развиты, чем подобные им средства *Mathematica* [28-30,32,42,43]. В результате на основе этих средств предоставляется достаточно удобная возможность создания достаточно эффективных средств программного анализа типов выражений, данных и их структур наряду с организацией эффективной программной обработки особых 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 *процедурного* программирования весьма существенный акцент делает на вопросе тестирования недопустимости фактических аргументов в 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 недопустимых фактических аргументов уже на этапе вызова процедур;
- (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 существенно позволяет облегчить задачу отладки с концентрацией внимания только на ошибках, связанных с самой логикой программируемого алгоритма.

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.

Наиболее типичной реализацией механизма проверки аргументов *Maple*-процедуры на их допустимость является метод типизации, состоящий в присвоении формальным аргументам в определении процедуры допустимых для них типов, а именно:

$$A := \text{Proc}(x::\text{Type}_x, y::\text{Type}_y, z::\text{Type}_z, \dots) \text{ <Тело процедуры> end proc}$$

где $\{\text{Type}_x, \text{Type}_y, \text{Type}_z, \dots\}$ – типы (булевы функции), которые при вызовах процедуры $A(x, y, z, \dots)$ тестируют допустимость ее фактических аргументов $\{x, y, z, \dots\}$. В качестве данных типов могут выступать и встроенные типы (как простые, так и структурные), и типы, определяемые пользователем. Например, в нашу библиотеку [45] был включен ряд различного назначения типов, которые довольно успешно используются в задачах тестирования фактических аргументов на предмет их допустимости.

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 функции, а именно:

$$\text{Proc}[x_; T_x[x], y_; T_y[y], \dots] := \text{Module}[\{\dots\}, \text{<Тело процедуры>}]$$

где $\{T_x[x], T_y[y], \dots\}$ – некоторые булевы функции, тестирующие при вызове процедуры $\text{Proc}[x, y, \dots]$ допустимость фактических аргументов $\{x, y, \dots\}$. Итак, на функции $T_p[p]$, возвращающей при вызове процедуры True на фактическом p -аргументе, p -аргумент полагается допустимым, и недопустимым при возврате значения False; $p \in \{x, y, \dots\}$. Если вызов $\text{Proc}[x, y, \dots]$ процедуры будет содержать по крайней мере один недопустимый фактический аргумент, то на таком кортеже фактических аргументов он возвращается невычисленным. Ниже эквивалентные методы тестирования допустимых аргументов в вызовах процедур в среде пакетов *Maple* и *Mathematica* сможет достаточно наглядно проиллюстрировать следующий достаточно несложный фрагмент, а именно:

```
> Proc := proc(x::listlist({0, 1}), n::{0, 1, 2, 3}, y::symbol) [x, n, y] end proc:
> Proc([[[0, 1, 0], [1, 1, 0], [1, 1, 1]], 3, Art); => [[[0,1,0], [1,1,0], [1,1,1]], 3, Art]
> Proc([[[0, 1, 2], [1, 1, 0], [1, 1, 1]], 3, Art);
Error, invalid input: Proc expects its 1st argument, x, to be of type listlist({0, 1}), but
received [[0, 1, 2], [1, 1, 0], [1, 1, 1]]
> Proc([[[0, 1, 0], [1, 1, 0], [1, 1, 1]], 7, Art);
Error, invalid input: Proc expects its 2nd argument, n, to be of type {0, 1, 2, 3}, but
received 7
> Proc([[[0, 1, 0], [1, 1, 0], [1, 1, 1]], 3, "Art");
Error, invalid input: Proc expects its 3rd argument, y, to be of type symbol, but
received Art
> Err := () -> `if`(lasterror[1..14] = "invalid input:", [lastexception][-2..-1], lasterror):
```

```

> Err(); => [symbol, "Art"]
> `type/Args1_Proc1` := (x::anything) -> `if`(op(0, x) = 'set', op([true, assign(unassign(
    "%Args1"))]), op([false, assign("%Args1" = cat("Args1::Error: should be list,
    but was received ", whattype(x)))])):
> Proc1 := proc(L::Args1_Proc1, N::nonnegative) L[N] end proc:
> Proc1({63, 68, 43, 14, 22, 48}, 3); => 43
> Proc1("Art_Kr", 3); %Args1;
Error, invalid input: Proc1 expects its 1st argument, L, to be of type Args1_Proc1, but
received Art_Kr
    "Args1::Error: should be list, but was received string"
> Proc1({63, 68, 43, 14, 22, 48}, 3); => 43
> %Args1; => %Args1
=====
In[464]:= 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[465]:= P[72.42, {63, 68, 43, 14, 22, 48}]
Out[465]= P[72.42, {63, 68, 43, 14, 22, 48}]
In[466]:= $Args1
Out[466]= "Args1::Error: type should be Integer, but was received <Real>"
In[467]:= {P[3, {63, 68, 43, 14, 22, 48}], $Args1}
Out[467]= {43, $Args1}
In[468]:= {P[68, "Art22_Kr14"], $Args1, $Args2}
Out[468]= {P[68, "Art22_Kr14"], $Args1, "Args2::Error: type should be List, but
    was received <String>"}
In[469]:= Clear[$Args1, $Args2]
In[470]:= {P[6.8, "Art22_Kr14"], $Args1, $Args2}
Out[470]= {P[6.8, "Art22_Kr14"], "Args1::Error: type should be Integer, but was received
    <Real>", $Args2}
    
```

Первая часть данного фрагмента иллюстрирует наиболее типичный для пакета *Maple* способ проверки фактических аргументов на их допустимость в передаче процедуре при ее вызове, когда всем или нескольким формальным аргументам приписываются типы: *простые, структурные либо пользовательские*. Если при вызове такой процедуры будут обнаружены фактические аргументы, типы которых не будут соответствовать типам отвечающих им *формальных* аргументов, то инициируется ошибка с привязкой к *первому* недопустимому в кортеже фактическому аргументу. Подобные ошибочные ситуации ввода можно обрабатывать, например, **try**-предложением. Более того, такая информация по данным ошибкам помещается и в глобальные переменные *lasterror* и *lastexception Maple*. Для несколько более удобной *детализации* таких ситуаций можно

использовать *весьма* простую процедуру *Err*, вызов *Err()* которой после возникновения ошибочной ситуации возвращает 2-элементный *список*, детализирующий возникшую ошибочную ситуацию, обусловленную только несоответствием типов формальных и фактических аргументов при вызове некоторой процедуры. *Первый* элемент данного списка определяет требуемый тип, тогда как второй элемент определяет полученное при вызове процедуры значение; при этом, при возникновении ошибочных ситуаций другого типа возвращается значение переменной *lasterror*.

Тогда как второй пример первой части фрагмента иллюстрирует в ряде случаев один довольно полезный *подход* к типированию формальных аргументов процедур, в ряде случаев обеспечивающий довольно детальное тестирование *фактических* аргументов. Наглядно этот метод иллюстрируется на примере организации типирования первого *формального* аргумента процедуры *Proc1* определенным процедурой ``type/Args1_Proc1`` типом *Args1_Proc1*. Процедура в соответствии с соглашениями *Maple* определяет тип *Args1_Proc1*, обеспечивая тестирование первого фактического аргумента на предмет его *допустимости*, и при *недопустимости* данного аргумента не только иницируется стандартная ошибочная ситуация, но также и через глобальную переменную `%Args1` возвращается *детализирующая* информация. Вообще говоря, этот подход к типизации может базироваться на довольно сложных *процедурных* реализациях, обеспечивающих *тестирование* по довольно сложным алгоритмам фактических аргументов процедур. В целом же, пакет *Maple* допускает *два* способа тестирования *фактических* аргументов на их допустимость при вызове процедуры, а именно:

- (1) *посредством типизации формальных аргументов* (могут использоваться как встроенные, так и пользовательские типы);
- (2) *проверка полученных процедурой фактических аргументов в самом теле процедуры*;
- (3) *совмещение обоих перечисленных методов проверки аргументов*.

Как правило, выбор подхода к проверке фактических аргументов на их допустимость определяется как опытом программиста, так и в значительной степени реализуемым процедурой алгоритмом. Пакет *Maple* предоставляет пользователю довольно большой (и регулярно расширяемый) набор типов, позволяя многим процедурам обходиться 1-м методом тестирования фактических аргументов на их допустимость исключительно. Более того, определяемые пользователем типы (один из возможных подходов к реализации таких типов был проиллюстрирован выше) могут основываться на достаточно сложных тестирующих алгоритмах и обеспечивать достаточно *тонкую* проверку передаваемых процедуре *фактических* аргументов. Более того, как показывает наш опыт, существует немалое число достаточно *сложных* алгоритмов, реализуемых процедурами, проверка допустимости *фактических* аргументов для которых сама по себе представляет весьма непростую задачу и которую нецелесообразно возлагать на вводимые пользователем типы. В этом случае намного более эффективно использовать 2-й метод тестирования фактических аргументов на допустимость. Наконец, для ряда задач хорошо отвечает и 3-й метод тестирования фактических аргументов на допустимость. В частности, 3-й способ в качестве эффективного следует применять в процедурах с неопределенным числом аргументов различного типа. Например, в процедурах нашей Библиотеки [45] использовались все 3 способа тестирования *фактических* аргументов на *допустимость*;

более того, для целого ряда задач оказалось целесообразным определить и ряд новых типов выражений и даже файлов данных и каталогов.

Наконец, *вторая* часть предыдущего фрагмента представляет *типичную* организацию механизма тестирования *фактических* аргументов в *Mathematica*-процедурах, которая базируется на проверке шаблона *<Шаблон> /; <Тест>* на допустимость. Шаблон будет полагаться в качестве допустимого, если на нем *Тест* возвращает значение True. При этом, все переменные, используемые *тестом*, должны быть и в тестируемом *Шаблоне*. Детальнее с организацией механизма тестирования на допустимость *шаблонов* можно ознакомиться в справочной системе по пакету *Mathematica*.

Подобно случаю типирования аргументов *Maple*-процедуры для случая процедуры **P** пакета *Mathematica* использован такой же самый подход, но с учетом особенностей второго пакета. Между тем, и здесь дается довольно детальное описание ошибочных ситуаций, возникающих при несоответствии *фактических* аргументов, получаемых **P**-процедурой при ее вызове, типам соответствующих *формальных* аргументов на основе значений глобальных *\$Argsj*-переменных (*j=1..n*). Все различие, в принципе, 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 в самом заголовке процедуры, используя, в частности, т.н. чистые функции.

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[510]:= VS[N_Integer, L_ /; ListQ[L] && MemberQ[{0, 1, {0, 1}},
Sort[DeleteDuplicates[Flatten[L]]]] := Block[{}, L[[N]]]
In[511]:= VS[2, {63, 68, 14, 22, 43, 48}]
Out[511]= VS[2, {63, 68, 14, 22, 43, 48}]
In[512]:= VS[-1, {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[512]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
In[522]:= BinaryListQ[L_] := ListQ[L] &&
MemberQ[{0, 1, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]]
In[523]:= VS[N_Integer, L_ /; BinaryListQ[L]] := Block[{}, L[[N]]]
In[524]:= VS[2, {63, 68, 14, 22, 43, 48}]
Out[524]= VS[2, {63, 68, 14, 22, 43, 48}]
In[525]:= VS[-1, {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[525]= {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/функциям. Так, в предыдущем фрагменте определен тест **BinaryListQ[L]**, возвращающий True на *бинарном* списке *L* (список может быть, при этом, и вложенным), в противном случае возвращается False. В дальнейшем такой тест может быть вполне эффективно применен в тестировании *фактических* аргументов в целом ряде практически важных процедур и функций.

При этом, следует иметь ввиду, что *понятие* типа в обоих пакетах трактуется намного шире, чем это имеется ввиду 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; у *Maple* такой возможности нет.

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 задачу.

```
In[847]:= TestArgsTypes[P_] := Module[{a, b, c, d, h, p, f = "$$Art22$$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[848]:= 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[849]:= VS[6, -4, "A", "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[849]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
In[850]:= VS[6, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
Out[850]= VS[6, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]
In[851]:= TestArgsTypes[VS[9, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[851]= {True, False, True, True, True}
In[852]:= $TestArgsTypes
Out[852]= {{2, 7.2}}
In[853]:= TestArgsTypes[VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[853]= {True, False, True, False, False}
In[854]:= $TestArgsTypes
Out[854]= {{2, 7.2}, {4, vz}, {5, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}}}
In[855]:= TestArgsTypes[VS[9, 0, "A", "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[855]= {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
In[856]:= $TestArgsTypes
Out[856]= $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. Естественно, дальнейшее развитие процедуры **TestArgsTypes** вполне может завершиться созданием достаточно полезного во многих отношениях средства манипулирования с процедурными объектами, 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.

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 следующего фрагмента, глобальная *\$TestArgsTypes*-переменная пакетом *Mathematica* таковой не рассматривается.

```
In[848]:= 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[7]:= TestArgsTypes[VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[7]= {True, False, True, False, False}
In[8]:= $TestArgsTypes
Out[8]= $TestArgsTypes
=====
In[1]:= 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[3]:= LoadNameFromM["D:\\Math_myLib\\AVZ_Package.m", "Map3"]
In[4]:= Map3[LoadNameFromM, "D:\\Math_myLib\\AVZ_Package.m", {"StrStr",
          "ProcQ", "TestArgsTypes"}]
Out[4]= {Null, Null, Null}
In[5]:= TestArgsTypes[VS[9, 7.2, A, "vz", {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]]
Out[5]= {True, False, True, True, True}
In[6]:= $TestArgsTypes
Out[6]= {{2, 7.2}}
In[7]:= TestArgsTypes[VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[7]= {True, False, True, False, False}
In[8]:= $TestArgsTypes
Out[8]= {{2, 7.2}, {4, vz}, {5, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}}}
```


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 текст процедуры и пример ее применения 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*.

Резюмируя, можно рассматривать механизмы *типирования* обоих пакетов достаточно развитыми, однако с учетом *функционального* программирования и функциональных средств именно **Mathematica** более предпочтительна для создания новых типов как в виде отдельных процедур, так и в составе шаблонов для формальных аргументов.

Глава 5. Организация обработки особых и ошибочных ситуаций

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 пакетами *Maple* и *Mathematica* для решения этой задачи.

5.1. Обработка особых и ошибочных ситуаций в среде Maple

В процессе выполнения *Maple*-алгоритма возможно появление особых и аварийных ситуаций, в большинстве случаев требующих специальной обработки во избежание серьезного нарушения всего вычислительного процесса. Идентифицируемые *Maple* ситуации такого рода возвращаются в предопределенную *lasterror*-переменную, чье значение определяет последнюю обнаруженную в текущем сеансе ошибку и доступно текущему сеансу для необходимой обработки возникшей особой ситуации. Значение *lasterror*-переменной определяется лишь в момент возникновения ситуации, а сразу же после загрузки пакета переменная является *неопределенной*. К *lasterror*-переменной непосредственно примыкает и *tracelast*-функция, обеспечивающая вызов последней ошибочной ситуации из стека ошибок пакета. При этом, между ними имеется также и принципиальное различие: через *lasterror*-переменную возвращается ошибочная диагностика, обрабатываемая функциональными средствами *Maple*-языка, тогда как *tracelast*-функция инициирует вызов последней ошибочной ситуации, если она не была удалена из стека ошибок. Значение *lasterror*-переменной имеет строчный тип. Следующий простой пример иллюстрирует применение обоих указанных средств *Maple*-языка, предназначенных для обработки ошибочных ситуаций:

```
> read("D:/Academy/UserLib6789/Common/HelpBase/SveGal.mws");
Error, unable to read `D:/Academy/UserLib6789/Common/HelpBase/SveGal.mws`
> lasterror;      ⇒ "unable to read `%1`"
> tracelast;
Error, unable to read `D:/Academy/UserLib6789/Common/HelpBase/SveGal.mws`
```

Для обработки ошибочных ситуаций в ранних релизах вместе с *lasterror*-переменной использовалась встроенная функция *traperror*(*V1*, *V2*, ..., *Vn*), по которой возвращается сообщение, соответствующее *первой* встреченной ошибочной ситуации в выражениях *Vk* (*k* = 1..*n*). При этом, каждый вызов функции *traperror* отменяет старое определение предопределенной переменной *lasterror*. То же производится и по *traperror()*-вызову. Если при вызове функции *traperror* не обнаружено особых ситуаций, возвращается *упрощенные/вычисленные* выражения, входящие в состав ее фактического аргумента. В

случае указания в качестве фактического аргумента последовательности выражений только первому, вызвавшему ошибочную ситуацию, приписывается соответствующее диагностическое сообщение. Данное сообщение может использоваться и совместно с *lasterror*-информацией для организации обработки особых и ошибочных ситуаций, возникающих в процессе вычислений в документе либо в *Maple*-процедуре. Простой пример иллюстрирует вышесказанное, а именно:

```
> VS:=0: AG:=27.10: `if`(traperror(AG/VS)=lasterror, assign(T=AG/21.14),1): [T, lasterror];
[1.281929991, "numeric exception: division by zero"]
```

В этом примере на основе вызова *traperror(AG/VS)* и *lasterror* обрабатывается особая ситуация «деление на ноль», в результате чего производится устранение данной особой ситуации путем перехода к вычислению другого выражения. При этом, необходимо обратить внимание на обстоятельство, что при возникновении ошибочной ситуации в вычисляемом по *traperror*-функции выражении сообщение о ней будет поглощено функцией, не отражаясь в документе. Большинство серьезных особых и аварийных ситуаций идентифицируется *lasterror*-переменной пакета, следовательно совместное использование указанных средств может оказаться достаточно эффективным. Более того, следует иметь в виду, что ряд возникающих *особых* ситуаций не обрабатывается функцией *traperror*. В общем же случае, следующие ошибочные и особые ситуации не обрабатываются *traperror*-функцией, а именно:

- * *interrupted* (прерванное вычисление)
- * *assertion failed* (генерируется при активном ASSERT-механизме)
- * *out of memory* (недостаток памяти)
- * *stack overflow* (переполнение стека)
- * *object too large* (объект слишком велик).

Это объясняется невозможностью восстановления на момент возникновения данных ситуаций. Следует иметь в виду, что вызов *traperror(V)* не связывает с *V*-выражением особой ситуации типа «деление на ноль», если в качестве *V*-аргумента будет выступать конструкция следующего вида $\{V/0 \mid V/(a-a) \mid V/(a*0)\}$, т.е. если знаменатель выражения тождественно равен нулю, а не принимает нулевое значение в результате вычисления (присвоения) или упрощения выражения. Простой фрагмент иллюстрирует сказанное

```
> x := 68: y := 0: T := traperror(x/(a - a)): [lasterror, T];
["numeric exception: division by zero", "numeric exception: division by zero"]
> x := 68: y := 0: T := traperror(x/(a*0)): [lasterror, T];
Error, numeric exception: division by zero
["numeric exception: division by zero", "numeric exception: division by zero"]
> x := 14: y := 0: T := traperror(x/y): [lasterror, T];
["numeric exception: division by zero", "numeric exception: division by zero"]
> x:=0: if traperror(Kr*sin(x)/x) = lasterror then limit(Kr*sin(t)/t, t=x) end if;      => Kr
> x:=0: if lasterror = traperror(Kr*sin(x)/x) then G:=limit(Kr*sin(t)/t, t=x) end if; G; => G
> x:=0: lasterror = traperror(Kr*sin(x)/x) then G:=limit(Kr*sin(t)/t, t=x) end if; G;
G := Kr
Kr
```

Последний пример фрагмента иллюстрирует факт, что *порядок* следования *lasterror*-переменной и вызова *traperror*-функции в логической связке достаточно существенен в общем случае и первой следует кодировать *traperror*-функцию. Однако, повторное выполнение *if*-предложения последнего примера фрагмента возвращает корректные результаты, ибо *lasterror*-переменная сохраняет последнюю ошибочную ситуацию. Ниже мы еще вернемся к рассмотренным средствам обработки ошибочных и особых ситуаций, однако следует отметить, что *traperror*-функция в довольно значительной мере является устаревшим (*obsolete*) средством, сохранившись с целью более полного обеспечения совместимости ранее разработанного программного обеспечения, и его полностью заменяет появившееся для таких целей и более функционально широкое *try*-предложение, рассматриваемое нами несколько ниже.

Важный тип особых ситуаций определяется *временными* ограничениями, связанными с возможностью больших временных затрат на вычисления выражений. Например, вычисление факториальных или циклических конструкций. *Maple*-язык располагает рядом функций, обеспечивающих работу с таким важным фактором реальности, как время. Временной фактор можно успешно использовать в различных конструкциях по управлению вычислительным процессом как в операционной среде компьютера, так и в среде пакета. Рассмотрим основные функциональные средства данного типа.

По *time*-функции, имеющей формат кодирования следующего простого вида:

time({ | <Выражение> })

возвращается соответственно общее время {с начала текущего сеанса работы с пакетом | вычисления указанного выражения} в с. в *float*-формате. При этом, следует иметь в виду, что использование второго формата кодирования *time*-функции позволяет получать время вычисления заданного фактическим аргументом *выражения* без учета времени, затраченного на его упрощение, т.е. *чистое* время вычисления. Первый формат *time*-функции используется, как правило, в виде конструкций продемонстрированного в нижеследующем фрагменте типа, тогда как второй формат функции более удобен в случае временной оценки вычисления отдельных выражений в чистом виде. Если же требуется оценить общее время вычисления сложного выражения, включая затраты на его упрощение, следует воспользоваться первым форматом *time*-функции, как это иллюстрирует следующий фрагмент применения *time*-функции, а именно:

```
> t := time(): G := sin(14.3^8!); t1 := time(): printf(`%s%1.3f%s`, `Время вычисления
    выражения "sin(14.3^8!)" равно: `, t1 - t, `сек.`); G;           ⇒ -0.8614831869
    Время вычисления выражения "sin(14.3^8!)" равно: 3.218 сек.
> Time := () -> evalf(map2(`*`, time(`if`(nargs = 0, NULL, eval(args))), [1/60, 1/3600]), 2):
> Time(), Time('681942!');    ⇒ [0.03, 0.00043], [1.8, 0.031]
```

Фрагмент включает пример простой процедуры *Time*({*expr*}), возвращающей общее время в минутах и часах или от начала текущего сеанса, или вычисления выражения *expr*. Функция *time* используется организацией управления вычислением в контексте его временных характеристик и сможет служить в качестве весьма удобного средства управления непредсказуемыми во временном отношении вычислениями типа *циклы*, *итерации* и прочие длительного характера вычисления.

Вторым средством, обеспечивающим временной контроль вычислений, служит также встроенная функция *timelimit*(*t*, *V*). Если время в *сек.*, затребованное процессором для вычисления *V*-выражения, не превысило значения, заданного первым *t*-аргументом, возвращается результат вычисления *V*-выражения. В противном случае генерируется ошибочная ситуация с диагностикой «*Error, (in V) time expired*», обрабатываемая выше рассмотренной *traperror*-функцией или *try*-предложением, рассматриваемым ниже. Однако функция *timelimit* не используется с функциями машинной арифметики, т. к. она не обрабатывает ситуацию исчерпания отведенного временного интервала для вычислений, как это иллюстрирует следующий пример, а именно:

```
> S := proc(n) local k; for k to n do sin(14.3*n) end do end proc: S(14^5), evalhf(S(22^5)),
    timelimit(3, evalhf(S(22^7)));
    0.6501308714, -0.562136533598019517, -0.562136533598019517
> timelimit(3, S(22^7));
    Error, (in sin) time expired
```

Из фрагмента легко видно, что на *evalhf*-функции машинной арифметики *timelimit*-функция не приводит к нужному результату, не ограничивая времени вычисления.

```
> Timetest := proc(t::{float, integer}, x::anything, V::symbol)
    local a, h, z;
    assign(z = time(), a = (x -> evalf(time() - x, 6)));
    if traperror(timelimit(t, assign(h = V(x)))) = "time expired" then
        `if` (nargs = 4 and type(args[4], 'symbol'), assign(args[4]=[a(z), 'undefined']), NULL), false
    else `if` (nargs = 4 and type(args[4], 'symbol'), assign(args[4] = [a(z), h]), NULL), true
    end if
end proc:
> G := proc(n) local k; for k to n do k end do end proc:
> Timetest(0.5, 10^6, G, p);    =>  true, [0.282, 1000000]
> Timetest(0.5, 10^7, G, h);    =>  false, [0.515, undefined]
> Timetest(0.5, 10^6, G), Timetest(0.5, 10^8, G);    =>  true, false
```

В качестве простого примера применения *timelimit*-функции предыдущий фрагмент представляет *Timetest*(*t*, *x*, *V*{*p*})-процедуру, возвращающую *true*-значение лишь в том случае, когда вычисление *V*(*x*)-выражения укладывается в отведенный ему временной интервал в *t*-секунд, в противном случае ею возвращается *false*-значение. При этом, через *необязательный* аргумент *p* процедура возвращает 2-элементный список, первый элемент которого определяет время вычисления искомого *V*(*x*)-выражения, а второй - результат вычисления либо *undefined*-значение соответственно. В данном фрагменте *Timetest*-процедура применяется для временного тестирования вызова процедуры *G*. Однако следует иметь в виду, что в общем случае *совместное* использование функций *time* и *timelimit* в одной процедуре может приводить к некорректным результатам [9, 10, 13, 28]. Между тем, при получении определенного навыка в использовании средств языка существующих средств обработки особых и аварийных ситуаций оказывается

вполне достаточным для разработки в среде *Maple* достаточно эффективных средств пользователя для обработки ситуаций указанного типа. Более того, на основе знания специфики реализуемого в среде языка алгоритма решаемой задачи средства пакета дают возможность проводить ее предварительный программный анализ на предмет предупреждения целого ряда возможных особых и аварийных ситуаций. Рассмотрим вопросы обработки особых ситуаций в контексте программирования процедур.

По функции **ERROR**, имеющей формат кодирования следующего вида:

$$\text{ERROR}(\{V1, V2, \dots, Vn\})$$

производится немедленный выход из процедуры, содержащей ее, в точке ее вызова с возвратом диагностического сообщения следующего вида, а именно:

$$\text{Error, (in } \{Proc \mid unknown\}) \{<V1>, <V2>, \dots, <Vn>\}$$

Proc – имя процедуры, вызвавшей **ERROR**-ситуацию, и *<Vk>* – результат вычисления *Vk*-выражения (*k*=1 .. *n*); *unknown*-идентификатор выводится для непоименованной процедуры. Тогда как по предложению **error**, имеющему формат кодирования вида:

$$\text{error } \{Msg \{, p1, p2, \dots, pn\}\} \{ : | ; \}$$

производится немедленный выход из процедуры в точке его выполнения с возвратом диагностического сообщения следующего вида, а именно:

$$\text{Error, (in } \{Proc \mid unknown\}) \{Msg(\{p1, p2, \dots, pn\})\}$$

где *Proc* – имя процедуры, вызвавшей **error**-ситуацию, и *Msg*(*{p1, p2, ..., pn}*) – результат подстановки *pk*-параметров в *Msg*-сообщение; *unknown*-идентификатор выводится для непоименованной процедуры. *Msg* – строка текста, определяющая суть ошибочной ситуации. Строка может содержать пронумерованные параметры вида «%*n*», где *n* – целое от 0 до 9. Тогда как *pk* – один или более параметров (*Maple*-выражений), которые подставляются вместо соответствующих по номеру вхождений «%*n*», когда возникает ошибочная ситуация, как это иллюстрирует следующий весьма простой пример:

```
> error "invalid arguments <%1> and <%2>", 3, 7;
```

```
Error, invalid arguments <3> and <7>
```

```
> 68/0;
```

```
Error, numeric exception: division by zero
```

```
> error;
```

```
Error, numeric exception: division by zero
```

```
> restart; error;
```

```
Error, unspecified error
```

Если в **error** отсутствует *Msg*, выполнение **error**-предложения в качестве диагностики возвращает диагностику последней ошибочной ситуации текущего сеанса, если же и такой нет, то возвращается сообщение о неуточненной ошибке. Если параметр имеет вид «%*n*», то в возвращаемом сообщении он появляется как *n*-й параметр в нотации *lprint*, тогда как вид «%-*n*» обеспечивает появление в обычной нотации, например:

```
> error "%1 and %2 arguments are invalid", 3, 7;
```

```
Error, 3 and 7 arguments are invalid
```

> **error** "%-1 and %-2 arguments are invalid", 3, 7;

Error, 3rd and 7th arguments are invalid

Сказанное в полной мере относится и к оформлению **ERROR**-функции, начиная уже с 6-й версии *Maple*. Более детально с оформлением диагностических сообщений для **ERROR**-функции (**error-предложения**) можно ознакомиться в справке по пакету.

Более того, значения *Vk*-выражений из **ERROR**-функции (*Msg* для **error-предложения**) помещаются в глобальную **lasterror**-переменную пакета и доступны для следующей обработки возникшей процедурной ошибки, которая с точки зрения *Maple*-языка в общем случае может и не быть ошибочной, как иллюстрирует следующий фрагмент:

```
> restart; lasterror;  => lasterror
> A:=proc() local k; `if` (type(nargs,'odd'),ERROR("odd number of arguments"),`+`(args))
end proc: A(68, 63, 42, 14, 21, 44, 95, 2010);  => 2357
> A(68, 63, 42, 14, 21, 44, 2010);
Error, (in A) odd number of arguments
> lasterror;  => "odd number of arguments"
> if lasterror = "odd number of arguments" then `Introduce new arguments` end if;
      Introduce new arguments
> proc(x, y) `if` (x > y, ERROR("for args[1] & args[2]", args[1] > args[2]), y/x) end proc(17, 10);
Error, (in unknown) for args[1] & args[2], 10 < 17
> proc(x, y) if x > y then error "for args[1] & args[2]", args[1] > args[2] else y/x end if end
proc(17, 10);
Error, (in unknown) for args[1] & args[2], 10 < 17
```

В данном фрагменте простая **A**-процедура предусматривает обработку ошибочной ситуации, определяемой фактом получения процедурой при вызове нечетного числа фактических аргументов. В случае такой ситуации по **ERROR**-функции выполняется выход из процедуры с возвратом соответствующего сообщения. После чего на основе анализа значения **lasterror**-переменной обеспечивается выбор способа последующих вычислений. Так, последний пример фрагмента иллюстрирует возврат ошибочного сообщения, связанного с определенным соотношением между аргументами, в случае непоименованной процедуры пользователя. С данной целью была использована как **ERROR**-функция, так и эквивалентное ей **error-предложение**.

> **plot**(sqrt(-2));

Plotting error, empty plot

> **lasterror, lastexception;** => **lasterror, lastexception**

Следует отметить, что к сожалению пакет *Maple* не отражает в переменных **lasterror** и **lastexception** целый ряд весьма существенного типа ошибочных ситуаций, которые связаны, прежде всего, с графическими объектами (*а в общем случае со средствами GUI пакета*), как это наглядно иллюстрирует предыдущий достаточно простой фрагмент. В свою очередь это не позволяет обрабатывать такого типа ошибки средствами *Maple*. Отметим, что данная недоработка была устранена только в версии 11 пакета.

Так как механизм **ERROR**-функции (**error-предложения**) позволяет делать обработку

ситуаций, ошибочных с точки зрения внутренней логики реализуемого процедурой алгоритма, т. е. прогнозируемых ошибок, то для этих целей может быть использована и функция **RETURN** (**return**-предложение), как это иллюстрирует следующий фрагмент:

```
> WM := proc() local k; global V; for k to nargs do if whattype(args[k]) <> 'float' then
    error nargs, [args] end if end do: `End of WM-procedure` end proc: WM(6.8,420,14,21);
lasterror;    => 4, [6.8, 420, 14, 21]
Error, (in WM) 4, [6.8, 420, 14, 21]
> WM1 := proc() local k; global V; for k to nargs do if whattype(args[k]) <> 'float' then
    return nargs,[args] end if end do: `End of WM1` end proc: WM1(6.8,420,14,21);
4, [6.8, 420, 14, 21]
> if whattype(%) = exprseq then WM1(op(evalf(lasterror[2]))) end if;    => End of WM1
> if whattype(%%) = exprseq then WM1(op(evalf(%%[2]))) end if;    => End of WM1
```

В данном фрагменте проиллюстрированы эквивалентные конструкции по обработке передаваемых **WM**-процедуре фактических аргументов, которые созданны на основе предложений **error** и **return**. При этом, если результат выполнения предложения **error** можно получать через глобальную переменную *lasterror*, то по **return**-предложению он доступен непосредственно, что в целом ряде случаев бывает предпочтительнее.

Механизм возврата результата вызовов процедуры на основе **error**-предложения нам представляется довольно важным средством обработки особых ситуаций, связанных со спецификой самой процедуры, а не средств *Maple*-языка. В этой связи пользователю предоставляется возможность организации собственных алгоритмов обработки как ошибочных, так и особых ситуаций, которые связаны с аргументами, передаваемыми процедуре, или алгоритмом вычислений, как это иллюстрирует простой фрагмент:

```
> VK := proc()
    local k, a; a := NULL;
    if nargs = 0 then error "procedure call has no arguments" else
        for k to nargs do
            if whattype(args[k]) <> 'float' then a := a, k end if
        end do;
        if a <> NULL then error "arguments %1 have a type different from 'float'", [a] end if
    end if;
    "Procedure body"
end proc:
> VK(6.8, 6.3, 14, 6, 21);
Error, (in VK) arguments [3, 4, 5] have a type different from 'float'
> lasterror;
"arguments %1 have a type different from 'float'", [3, 4, 5]
> VK(6.8, 6.3, 14.3, 7.8, 21.8);
"Procedure body"
```

Процедура **VK** программно обрабатывает ситуацию получения как *NULL*-значения, так и аргументов некорректного типа на основе **error**-предложения с последующим

возвратом соответствующей диагностики, которая может обрабатываться вне самого тела процедуры программно. Некоторые особенности и ряд полезных рекомендаций по использованию функции **ERROR (error-предложения)** в организации специальных выходов из процедур довольно детально рассмотрены в наших книгах [10-13,28-44].

Наконец, по вызову *встроенной* функции **traperror(Выражение)** возвращается результат вычисления *Выражения*, указанного ее фактическим аргументом; если же в процессе вычисления возникает ошибочная ситуация, то возвращается идентифицирующее ее диагностическое сообщение, доступное для следующей обработки и не нарушающее естественный порядок вычислений. Таким образом, **traperror**-функция в отличие от функции **ERROR (error-предложения)** и результата стандартной обработки ситуации не прекращает выполнения процедуры, позволяя «блокировать» ситуацию и создавая возможность более гибкой ее обработки. Простой пример иллюстрирует сказанное:

```
> ArtKr := proc()
  local k, h, z;
  assign(z = 68 + sqrt(`+(args))), assign(h = traperror(z/`+(args)));
  if z = 0 then `actual arguments are absent`
  elif h <> "numeric exception: division by zero" then return h
  else z/(63 + `+(args))
  end if
end proc;
> evalf([ArtKr(68, -63, 14, 21), ArtKr(68, 63), ArtKr(0, 0, 0)]);
[1.858113883, 0.6064543752, 1.079365079]
```

В данном фрагменте **ArtKr**-процедура использует **traperror**-механизм для обработки особой ситуации, которая может приводить к аварийному завершению процедуры в момент ее выполнения – *нулевая сумма значений* ее аргументов инициирует ситуацию *division by zero*. Данная ситуация обрабатывается **traperror**-функцией во избежание стандартной обработки **Maple** и обеспечивает пользовательскую обработку, хорошо усматриваемую из листинга процедуры. При этом, следует иметь в виду, по **tracelast**-команде возможно получать полезную отладочную информацию, представляющую собой последнее содержимое *стэка* ошибок ядра пакета **Maple**, как это иллюстрирует следующий достаточно простой фрагмент, а именно:

```
> Svegal := proc(x) local y; y := 68; 1/(x - y) end proc: Svegal(68);
Error, (in Svegal) numeric exception: division by zero
> tracelast;
Svegal called with arguments: 68
#(Svegal, 2): 1/(x-y)
Error, (in Svegal) numeric exception: division by zero
locals defined as: y = 68
> S := proc(n::integer) local k, G; G:= 0: for k to 10^n do G:=G+1 end do end proc: S(14);
Warning, computation interrupted
> tracelast;
```

```
S called with arguments: 14
#(S, 3): G := G+1
Error, (in S) interrupted
locals defined as: k = 53079803, G = 53079802
```

Из второго примера фрагмента следует, что *tracelast*-команда позволяет определять точное местоположение прерванного вычисления и вывести основные промежуточные результаты на момент прерывания, что может оказаться весьма полезным свойством при отладке, например, рекурсивных процедур.

Механизм try-предложения Maple. Для программной обработки ошибочных и особых ситуаций, отражаемых в *lasterror*-переменной пакета, пакет располагает встроенной функцией *traperror*, рассмотренной выше, процедурами *stoperror*, *unstoperror* наряду с *try*-предложением. Первые 3 средства остались в наследство от предыдущих версий пакета (заинтересованный читатель с ними может ознакомиться, например, в [12]), тогда как *try*-предложение явилось дальнейшим развитием средств обработки ошибочных и особых ситуаций. Данное средство впервые было включено в пакет Maple 6.

Предложение *try* обеспечивает механизм для выполнения предложений Maple-языка в управляемой программной среде, когда ошибочные и особые ситуации без какой-либо серьезной причины не будут приводить к завершению выполнения процедуры без возврата соответствующих диагностических сообщений. Предложение *try* имеет нижеследующий формат кодирования, а именно:

```
try <Блок ArtKr Maple-предложений>
  {catch "Строка соответствия 1": <последовательность Maple-предложений>}
  {catch "Строка соответствия 2": <последовательность Maple-предложений>}
  =====
  {catch "Строка соответствия n": <последовательность Maple-предложений>}
  {finally <последовательность Maple-предложений>}
end try {:|;}
```

Формат предложения *try* определяет, что блоки *catch* и *finally* в нем необязательны. Более того, *catch*-блок имеет достаточно простую структуру: "Строка соответствия": <последовательность Maple-предложений>. Механизм выполнения *try*-предложения в общем случае вкратце сводится к следующему. В результате получения управления *try*-предложение инициирует выполнение блока ArtKr предложений пакета Maple, в которых мы пытаемся локализовать ошибочные (особые) ситуации. Если в процессе выполнения блока не возникло данных ситуаций, то выполнение будет продолжено, начиная с Maple-предложений *finally*-блока, если такой блок был определен. Затем выполнение продолжается с предложения, следующего за закрывающей скобкой (*end try*) или в кратком формате (*end*) *try*-предложения, которая является обязательной.

В случае обнаружения исключительной ситуации в процессе выполнения ArtKr-блока его выполнение немедленно прекращается и производится сравнение каждой из *catch*-строк на соответствие возникшей исключительной ситуации (эта ситуация отражается в переменных пакета *lasterror* и *lastexception*). Если данная *catch*-строка существует, то

выполняется последовательность *Maple*-предложений, соответствующая этому **catch**-блоку (данные предложения расположены между ключевым словом **catch** и элементами **try**-предложения, а именно: (1) следующий **catch**-блок, (2) **finally**-блок, (3) закрывающая скобка «*end try*»). В данном случае предполагается, что исключительная ситуация распознана и обработана. Иначе исключительная ситуация предполагается необработанной и ее обработка передается вне **try**-предложения. При этом, найденный **catch**-блок может содержать функцию **ERROR (error-предложение)** без аргументов, которая аналогично передает обработку исключительной ситуации вне **try**-предложения. Если ситуация передается вне **try**-предложения, то создается новый специальный объект, который повторяет имя процедуры, содержащей **try**-предложение, сам текст диагностического сообщения, а также параметры исходной исключительной ситуации.

```
> F := proc(x::numeric)
  try
    if x <= 42 then error "Branch_1"
    elif 42 < x and x <= 47 then error "Branch_2"
    elif 47 < x and x <= 67 then error "Branch_3"
    elif 67 < x and x <= 100 then error "Branch_4"
    else evalf(sin(x) + cos(x), 4)
    end if
  catch "Branch_1": Art(x)
  catch "Branch_2": Kr(x)
  catch "Branch_3": Sv(x)
  catch "Branch_4": Arn(x)
  end try
end proc;
> seq(F(x), x = [42, 45, 64, 14, 21, 72, 100, 420]);
      Art(42), Kr(45), Sv(64), Art(14), Art(21), Arn(72), Arn(100), -0.2643
```

Предложение **try** может содержать любое количество **catch**-блоков и в каждом из этих блоков выполняется соответствующая последовательность *Maple*-предложений. Для иллюстрации сказанного и механизма активации **catch**-блоков рассмотрим вкратце реализацию простой кусочно-определенной функции *F* на основе **try**-предложения (*предыдущий фрагмент*); в качестве строк совпадения **catch**-блоки используют заранее запрограммированные ошибки, сформированные предложениями **error**, в сочетании с предложением **return**. На наш взгляд, фрагмент довольно прозрачно иллюстрирует механизм инициирования **catch**-блоков, обеспечивающий довольно широкий спектр приложений **try**-предложения для обработки разного типа исключительных (*особых*) ситуаций, возникающих как естественным образом при выполнении определенного вычислительного алгоритма или работы с внешними данными и т.д., так и заранее созданных пользователем с целью обеспечения ветвления выполнения алгоритмов в зависимости от тех или иных условий, заранее предусмотренных.

При нормальных вычислениях предложения **finally**-блоков выполняются всегда, как только **try**-предложение потеряет управление. Данное поведение сохраняется даже,

если предложения **catch**-блока обеспечивают обработку *исключительной* ситуации вне **try**-предложения, сгенерированной некоей новой исключительной ситуацией, либо выполнением любого из предложений **return**, **break** или **next**. Если исключительная ситуация возникает в **catch**-блоке и она распознается отладчиком (*debugger*), то тогда пользовательская программа прекращает выполнение и предложения **finally**-блока не выполняются. Предложения **finally**-блока не выполняются также и в том случае, если возникает одна из следующих исключительных ситуаций, а именно:

- (1) вычисление превысило отведенный временной интервал (данная ситуация может быть обработана лишь в том случае, если *timelimit*-функция, которая ограничивает время вычисления, инициирует прерывание типа «time expired», которое может быть опознано и обработано);
- (2) вычисление прервано извне вычислительного процесса (клавиши *Ctrl+C*, *Break* и др.);
- (3) возникла внутренняя системная ошибка;
- (4) ошибка при выполнении *ASSERT*-функции или типизации локальной переменной;
- (5) переполнение пакетного стека.

Ошибка, возникающая в процессе упрощения **try**-предложения, не распознается, а **finally**-блок (если таковой был определен) не выполняется, поэтому **try**-предложение не может быть выполнено в данной точке. При этом, если исключительная ситуация и возникает в процессе выполнения **catch**-блока или **finally**-блока, она рассматривается как возникающая вне области **try**-предложения. При проверке на совпадение строки *соответствия* **catch**-блока с диагностикой исключительной ситуации используются следующие соглашения пакета, а именно:

- строки *соответствия* рассматриваются как префиксы текстов сообщений, генерируемых исключительными ситуациями (переменные *lastexception* и *lasterror*);
- ни специальный объект, созданный *ERROR*-функцией или **error**-предложением, ни строки *соответствия* **catch**-блока не вычисляются;
- если строка *соответствия* имеет длину *n*, то текст, сгенерированный *ERROR*-функцией либо **error**-предложением, сравнивается только в пределах его первых *n* символов;
- отсутствующая строка *соответствия* сопоставима с любой исключительной ситуацией;
- в **try**-предложении **catch**-блок с конкретной строкой *соответствия* может быть только в единственном числе, иначе инициируется ошибочная ситуация.

Результатом выполнения **try**-предложения является выполнения его последнего *Maple*-предложения. Таким образом, механизм **try**-предложения носит достаточно общий характер, обеспечивая возможность обработки, практически, любой исключительной ситуации, инициированной как *ERROR*-функцией или **error**-предложением на основе конкретного вычислительного алгоритма, так и другими средствами пакета. В нашей книге [34] представлен ряд интересных применений **try**-предложения для обработки различного типа *ошибочных* и *особых* ситуаций. Приводятся также полезные примеры применения механизма **try**-предложения для реализации нестандартных алгоритмов решения целого ряда важных прикладных и системных задач. Используемые в них приемы могут оказаться полезными для программирования приложений в *Maple*.

В качестве только одного такого примера приведем следующую практически важную процедуру *MkDir4*, обеспечивающую создание цепочки подкаталогов любого уровня вложенности. Таких важных средств для работы с файловой системой компьютера у *Maple* нет до сих пор, тогда как, например, у пакета *Mathematica* подобное средство имеется в лице функции *CreateDirectory*, как иллюстрирует следующий пример:

```
In[125]:= CreateDirectory["D:/aaa/bbb/ccd/hhh"]
Out[125]= "D:\\aaa\\bbb\\ccd\\hhh"
```

Нами был создан ряд процедур для устранения данной недоработки разработчиков *Maple* из группы т.н. *MkDir-процедур*, которые существенно используют возможности рассмотренного выше *try*-предложения, как это весьма прозрачно просматривается в подходе из нижеследующего исходного кода одной из процедур *MkDir*-группы. Так, процедуры данной группы не только решают упомянутую задачу создания цепочки каталогов, но и обладают намного более широкими функциями, которые стандартно недоступны как пакету *Maple*, так и пакету *Mathematica* [10-15,28-34,38,42,44,45].

```
> MkDir4 := proc(d::string, symbol)
  local a, b, c;
    assign(b = split(d, {"\\", "/"}), assign(c = b[1]);
    for a from 2 to nops(b) do c := cat(c, "\\ ", b[a]);
      try mkdir(c)
      catch "directory exists and is not empty": next
    end try
  end do;
  d
end proc;
> MkDir4("D:/aaa\\bbb/ccd\\ddd");
      "D:/aaa\\bbb/ccd\\ddd"
> MkDir4("D:/aaa/bbb/ccd/hhh\\sssss/wwww\\kkkkkkk");
      "D:/aaa/bbb/ccd/hhh/sssss/wwww\\kkkkkkk"
```

В заключении отметим еще одно средство для вывода результатов информационного характера, который обеспечивается процедурой *WARNING*, имеющей формат:

WARNING({*Msg* {, *p1*, *p2*, ..., *pn*}})

Msg и *pk* – полностью соответствуют описанию *error*-предложения, представленного в начале данного раздела с очевидной заменой диагностического сообщения *Msg* на информационное. В случае установки опции *warnlevel=0* (по умолчанию *warnlevel=3*) процедуры *interface* вывод сообщений не производится, тогда как вызов *WARNING* подавляется. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> H:=proc() local k; WARNING("Среди полученных значений оказалось %1 integer-
  типа", add(`if`(type(args[k], 'integer'), 1, 0), k = 1 .. nargs)); `+`(args) end proc;
> H(68, 63, 19.95, 14, 21, 20.10, 39, 44, 3.1, 95.99, 420);    => 808.14
      Warning, Среди полученных значений оказалось 7 integer-типа
> interface(warnlevel = 0): H(68, 63, 19.95, 14, 21, 20.10, 39, 44, 3.1, 95.99, 420);    => 808.14
```

Возможность вывода сообщений предоставляет удобный механизм информирования о ходе выполнения текущего документа, процедуры либо модуля. Для возможности эффективной обработки сообщений, выводимых процедурой **WARNING**, нами была создана ее полезная модификация [42], обеспечивающая через глобальную `_warning`-переменную возврат последнего в текущем сеансе сообщения, которое генерируется процедурой **WARNING**, тогда как через глобальную `_warnings`-переменную возврат всей истории сообщений текущего сеанса в виде списочной структуры, например:

```
> WARNING("the first warning");
Warning, the first warning
> WARNING("the second warning");
Warning, the second warning
> WARNING("the third warning");
Warning, the third warning
> _warning, _warnings;
"the third warning", ["the first warning", "the second warning", "the third warning"]
```

Для обеспечения полноценного функционирования процедуры **WARNING** в файл «*Maple.ini*» инициализации пакета следует поместить конструкцию «`_warnings := []`». Данный подход позволяет довольно успешно программно обрабатывать сообщения в *Maple*, начиная уже с *шестого* релиза пакета. Представленные выше средства возврата результатов вызова процедуры как в нормальном режиме ее выполнения, так и при наличии *особых* и *ошибочных* ситуаций достаточно эффективны. В дальнейшем будет представлен ряд средств пакета *Mathematica*, ориентированных на обработку особых и ошибочных ситуаций, в контексте сравнения с аналогичными средствами *Maple*.

5.2. Обработка особых и ошибочных ситуаций в Mathematica

Mathematica также располагает средствами для программной обработки *ошибочных* и *особых* ситуаций. Прежде всего, пакет также располагает механизмом для обработки сообщений, генерируемых в процессе вычислений. Большинство из функций пакета использует стандартный механизм генерации ошибочных и ряда других сообщений. Пользователь также может использовать такой механизм для генерации сообщений, связанных с выполнением его функций. Основной принцип состоит в использовании сообщений в формате '*символ::tag*', где *символ* – некоторое имя, например, функции и *tag* – сообщение в форме строки. В целом, '*символ::tag*' определяет идентификатор сообщения, к которому именно по нему и обращаются. Сообщения можно задавать как прямым определением идентификатора, так и по функции **MessageName**:

```
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 пакетной функцией сообщения в первую очередь выбирается из сообщений **Z::tag::Language**, где язык определяется значением переменной **\$Language**, если соответствующего 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 пример:

```
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. 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 **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 7-й версии предусмотрено 6 групп:

```

In[121]:= $MessageGroups[[All, 1]]
Out[121]= {"Compiler", "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, {4}]
```

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

```
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", ∞::"indet"];
```

```
    If[x/y === ComplexInfinity | | x/y === Indeterminate, On[Power::"infy", ∞::"indet"];
```

```
    Return[StringReplace["2nd or both arguments equal zero: 1", "1" -> ToString[{x, y}]]],
```

```
    On[Power::"infy", ∞::"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 за именем 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[42/47]
Out[987]= 1525801856952239

```

Для пользователей пакета *Mathematica* тексты поясняющих примеров этого раздела достаточно прозрачны и особых пояснений не требуют, будут они вполне понятны и пользователям пакета *Maple*, исходя уже из информации справки по первому пакету.

5.3. Оценка эффективности пакетов Maple и Mathematica по организации обработки особых и ошибочных ситуаций

Оба пакета располагают достаточно развитыми средствами обработки ошибочных и особых ситуаций, возникающих в процессе вычислений как пакетных, так и объектов пользователя. Однако наибольший интерес представляют такие средства именно для создания собственного робастного программного обеспечения, например, модулей и процедур. И здесь нам представляются несколько предпочтительнее средства *Maple*. Например, алгоритм обработки особых ситуаций приведенной в среде *Mathematica* функции `kr[x,y]` в среде *Maple* получает несколько более прозрачную реализацию:

```

> kr := proc(x::numeric, y::numeric)
    try x/y catch "numeric exception: division by zero": WARNING("2nd or both arguments
      equal zero: %1", [x, y]) end try end proc;
> kr(0, 0);
Warning, 2nd or both arguments equal zero: [0, 0]
> _warning, => "2nd or both arguments equal zero: [0, 0]"

```

Если же рассматривать возможности пакетов по обработке фактических аргументов процедур, чьи типы недопустимы и могут вызывать ошибочные ситуации в процессе выполнения процедуры, то средства типизации *Maple* оказываются весьма удобным фильтром при вызове процедур еще до выполнения их тела, иницилируя ошибочные ситуации, которые допускают простую программную обработку. Эту же функцию в полной мере возможно организовать и внутри процедуры на основе проверки типов получаемых фактических аргументов и их количества. Тогда как пакет *Mathematica* в значительной степени превосходит *Maple* по тестированию фактических аргументов. В целом же все возникающие в процессе выполнения особые ситуации целесообразно программно обрабатывать, для чего оба эти пакета имеют вполне достаточно средств. Поэтому в этом отношении между ними имеет место определенный *паритет*. Между тем, с учетом большей структурной и логической прозрачности наряду с простотой организации более предпочтительными для нас представляются средства обработки особых ситуаций именно пакета *Maple*. Прежде всего это можно отнести к механизму `try-предложения`, а также простоте подключения к пакету пользовательских типов.

Глава 6. Организация механизма процедур и модулей в среде программирования пакетов Maple и 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.

6.1.1. Определения процедур в среде пакета Maple и их типы

Технология модульного программирования охватывает *макроуровень* разработки *ПС* и позволяет решать важные задачи программной индустрии. В данном направлении одним из основных методов, обеспечивающих *модульность* программ, представляется механизм процедур, относительно *Maple*-языка рассматриваемый нами в настоящем разделе книги и с которым детальнее можно ознакомиться в книгах [8-14, 29-39, 42-44].

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* пакета *Maple* во многих отношениях отвечает данному решению.

Процедура в пакете *Maple* имеет следующую принципиальную организацию:

proc (<i><Последовательность формальных аргументов></i>){::Тип;}	Заголовок
local <i><Последовательность идентификаторов></i> ;	Описательная
global <i><Последовательность идентификаторов></i> ;	часть
options <i><Последовательность параметров></i> ;	определения
uses <i><Последовательность имен пакетных модулей></i> ;	
description <i><Описание></i> ;	процедуры
<i><Т Е Л О процедуры></i>	
end proc {; :}	

Заголовок процедуры содержит ключевое *proc*-слово со скобками, в которых кодируется последовательность формальных аргументов процедуры; данная последовательность может быть пустой, т.е. *минимально* допустимый заголовок процедуры имеет вид *proc()*. После заголовка процедуры допускается кодирование типа, который определяет тип возвращаемого процедурой результата. Здесь сразу же следует сделать существенное замечание. Согласно утверждению пакета данный тип является не декларацией типа, а скорее утверждением. При этом, если переменная ядра пакета *kernelopts(assertlevel=2)* установлена в 2 (*в ранних версиях пакета переменная kernelopts(ASSERT=true) установлена в true*), то проверяется тип возвращаемого результата и, если тип не отвечает данному утверждению, то инициируется *особая* ситуация. Следующий весьма простой пример иллюстрирует использование данного механизма тестирования результата:

```
> restart: kernelopts(assertlevel), kernelopts(ASSERT);    ⇒ 0, false           Maple 11
> Agn := proc(x)::float; x/63 end proc: Agn(47);             ⇒ 47/63
> kernelopts(assertlevel = 2), kernelopts(ASSERT);          ⇒ 0, true
> Agn(47);
Error, (in Agn) assertion failed: Agn expects its return value to be of type float,
but computed 47/63
```

Из данного примера следует, что упомянутая установка для переменной ядра пакета *kernelopts(assertlevel = 2)* обеспечивает возможность тестирования типа возвращаемого процедурой результата. Следовательно продекларированная возможность контроля типа результата, возвращаемого процедурой, является еще одним способом контроля возвращаемого результата; при этом, конвертации в приписанный тип не делается.

Ключевые слова **local**, **global**, **options** определяют необязательные секции процедуры, которые определяют последовательности идентификаторов локальных и глобальных переменных, параметров (*опций*) процедуры соответственно. Наконец необязательная *description*-секция содержит последовательность строк, описывающих процедуру. При этом, если секция представлена в определении процедуры, она будет присутствовать и при выводе последней на печать. Практически все библиотечные процедуры *Maple* содержат *description*-секцию, хотя стандартно она не выводится на печать. При этом, если секция содержит несколько предложений, она должна кодироваться последней, иначе инициируется ошибочная ситуация, как это иллюстрирует простой пример:

```
> P := proc() description "Сумма аргументов"; `+`(args) end proc: print(P);
      proc() description "Сумма аргументов"; `+`(args) end proc:
> P(63, 43, 68, 14, 22), op(5, eval(P));    ⇒ 210, "Сумма аргументов"
> P := proc() description "Сумма аргументов"; "Sum"; option trace; `+`(args) end proc:
Error, reserved word `option` or `options` unexpected
```

Начиная с *Maple 10* и выше дополнительно к представленным *определение* процедуры допускает использование необязательной *uses*-секции, которая определяет все имена пакетных модулей, используемых процедурой. Например, кодирование секции «*uses LinearAlgebra*» эквивалентно использованию в процедуре *use*-предложения формата:

```
use LinearAlgebra in < ... > end use;
```

Эта секция не идентифицируется при выводе или возврате определения процедуры, исчезая подобно *use*-предложению, как это иллюстрирует следующий пример:

```
> P := proc() global d; uses LinearAlgebra; description "GRSU"; BODY end proc;
      P := proc () global d; description "GRSU"; BODY end proc
```

Не возвращается содержимое этой секции вызовами *op(k, eval(P))* (*k=1..8*) аналогично случаю других секций описания произвольной процедуры *P*. Однако, на наш взгляд, секция *uses* носит непринципиальный характер, в определенной мере лишь *упрощая* (сокращая) оформление процедур, тогда как с точки зрения читабельности исходного текста использование в теле процедур *use*-предложений даже предпочтительнее. Не говоря уже о том, что данное непринципиальное новшество ведет к несовместимости процедур «сверху-вниз» относительно релизов пакета *Maple*.

Для получения информации о доступных в текущем сеансе процедурах, процедурах в библиотеках как пакета, так и пользовательских, логически сцепленных с основной *Maple*-библиотекой, созданы две полезные процедуры *ParProc* и *ParProc1*, которые в том числе позволяют получать содержимое всех секций заданной *первым* аргументом процедуры [12-14,44,45], как весьма наглядно иллюстрирует следующим пример:

```
> A := proc() local a,b,c; global x,y,z; options remember; uses linalg;
      description "Sum of actual arguments"; `+`(args) end proc: ParProc(A);
```

$$\left[\begin{array}{l} \text{locals} = (a, b, c) \\ \text{options} = (\text{remember}) \\ \text{description} = (\text{"Sum of actual arguments"}) \\ \text{globals} = (x, y, z) \end{array} \right]$$

Как отмечалось выше, не возвращается лишь содержимое **uses**-секций.

Необязательные секции **local**, **global**, **options**, **uses** и **description** образуют *описательную* часть определения *Maple*-процедуры, которая в общем случае может и отсутствовать вовсе. Управлять выводом тела процедуры на печать в общем случае возможно путем установки переменной *verboseproc interface*-процедуры *Maple*-языка. Минимальной конструкцией, распознаваемой пакетом в качестве процедуры, является конструкция следующего чрезвычайно простого вида, а именно:

```
> Proc := proc() end proc: whattype(eval(Proc));  $\Rightarrow$  procedure
```

Распознаваемый пакетом данный процедурный объект, между тем, особого смысла не имеет, т.к. вызов *Proc(args)* на произвольном кортеже фактических аргументов всегда возвращает *NULL*-значение, т.е. ничего. С целью возможности последующих вызовов процедуры ее *определение* присваивается некоторому *Id*-идентификатору, например:

```
> Id := proc({ | args}) ... end proc {:|;}
```

позволяя после его вычисления производить вызов процедуры по конструкциям вида *Id({ | args})* с соответствующей заменой ее *формальных args*-аргументов на *фактические args*-аргументы. Между тем, наряду с определением именованной процедуры *Maple*-язык допускает использование и *непоименованных* процедур, определения которых не присваиваются какому-либо идентификатору. Для вызова *непоименованных* процедур служит конструкция следующего общего формата, а именно:

```
proc({ | args}) ... end proc(args)
```

возвращающая результат вызова процедуры на *фактических* аргументах, указанных в круглых скобках, как это иллюстрирует следующий достаточно простой фрагмент:

```
> proc() [nargs, args] end proc(42, 47, 67, 89, 96, 6, 2010);  $\Rightarrow$  [7, 42, 47, 67, 89, 96, 6, 2010]
> proc(n) sum(args[k], k=2..nargs)^n end proc(3, x, y, z, a, b, c, h);  $\Rightarrow (x+y+z+a+b+c+h)^3$ 
> D(proc(y) (y^9 + 3*y^5 - 99)/6*(3*y^2 - 256) end proc)(9);  $\Rightarrow 2648753541$ 
> 68*proc(x) x^2 end proc(42) + 63*proc(y) y^2 end proc(47)/10;  $\Rightarrow 1338687/10$ 
> diff(proc(y) (y^9 + 2*y^5 - 99)/13*(3*y^2 - 256) end proc, y);
      (33*y^10-2304*y^8+42*y^6-2560*y^4-594*y)/13
```

Непоименованные процедуры можно использовать в ряде выражений и обрабатывать некоторыми функциями и операторами, как это иллюстрируют примеры фрагмента. В этом смысле наиболее часто *непоименованные* процедуры используются в *конъюнкции* с функциями $\{map, map2\}$, наряду с некоторыми другими функциями и операторами, допускающими неалгебраические выражения в качестве их фактических аргументов и операндов. Между тем, рекомендуется использовать именно *именованные процедуры*, позволяя исключить разного рода ошибки. Тогда как довольно квалифицированный пользователь может весьма эффективно использовать непоименованные процедуры при построении разного типа выражений подобно чистым функциям в *Mathematica*. Такой прием импльзовался в целом ряде процедур, составляющих Библиотеку [45].

Собственно *тело* процедуры содержит текст описания алгоритма решаемой задачи с использованием стандартных либо созданных пользователем средств *Maple*-языка, а именно: данных и их структур допустимых типов, переменных, функций модульных, пакетных и пользовательских, процедур различных типов и т. д., которые логически связаны управляющими структурами *следования*, *ветвления* и *цикла*, рассмотренными выше. Данное описание должно удовлетворять правилам синтаксиса, допускаемыми языком. Каждая процедура должна завершаться кодированием закрывающей скобки «*end proc*». Представленная конструкция является определением процедуры, которое активизируется только после его вычисления. При этом, при вычислении процедуры не производится выполнения предложений *тела* процедуры, однако интерпретатор с *Maple*-языка производит всевозможные *упрощения* тела процедуры, что представляет простой тип оптимизации. Реальное же выполнение тела процедуры производится в момент ее *вызова* с передачей ей фактических аргументов, чьи значения заменяют все вхождения в *тело* процедуры формальных аргументов.

Вторым способом определения процедур является использование функционального оператора « \rightarrow », который позволяет представлять процедуры в нотации классической операции отображения, а именно в следующем весьма простом виде:

$(args) \rightarrow$ <Выражение от формальных *args*-аргументов>

При этом, в случае единственного формального аргумента кодирование его в *скобках* не обязательно. Присвоение данной конструкции идентификатору дает возможность определить процедуру, как иллюстрирует следующий весьма простой пример:

> G := (x, y) → evalf(sqrt(x² + y²)): G(63, 68); ⇒ 92.69843580

Последовательность формальных аргументов в заданной таким способом процедуре может быть пустой, а ее тело должно быть единым выражением или *if*-предложением языка пакета, как это иллюстрирует следующий достаточно простой фрагмент:

```
> W := () → `if` (member(float, map(whattype, {args})), print([nargs, [args]]), NULL):
> W(42, 47, 19.98, 68, 96, 89, 22, 14, `TRG`); ⇒ [9, [42, 47, 19.98, 68, 96, 89, 22, 14, TRG]]
> H := x → if x < 42 then 14 elif x >= 42 and x <= 99 then 22 else infinity end if:
> map(H, [43, 49, 63, 68, 95, 98, 99, 39, 22, 14, 39]); ⇒ [22, 22, 22, 22, 22, 22, 22, 14, 14, 14, 14]
> [D(x → sin(x))(x), (x → sqrt(6.3*x2 + 420))(22)]; ⇒ [cos(x), 58.8999]
> map((n) → `if` (type(n, prime), n, NULL), [seq(k, k = 42 .. 68)]); ⇒ [43, 47, 53, 59, 61, 67]
```



```
> Z := () -> if sum(args[k], k= 1 .. nargs) <= 22 then args else false end if;
> [Z(3, 6, 8, 34, 14, 99)], [Z(3, -6, 8, -21, 6, 22)]; ⇒ [false], [3, -6, 8, -21, 6, 22]
```

При этом, следует иметь в виду, что второй способ *определения* предназначен, прежде всего, для довольно простых *однострочных* процедур и функций, т.к. не поддерживает механизма локальных и глобальных переменных, а также *опций*. Между тем, наличие определенного опыта позволяет создавать довольно сложные *однострочные* функции. Таким образом, в целом ряде случаев он оказывается достаточно полезным приемом программирования. Подобно первому способу, второй также допускает применение непоименованных процедур, используемых в тех же случаях, что и *первый* способ, что и иллюстрируют последние примеры предыдущего фрагмента.

Наконец, *третий* способ определения *однострочных* простых процедур базируется на использовании *define*-процедуры, имеющей для этого формат кодирования вида:

$$\text{define}(F, F(x1::\langle \text{Tun_1} \rangle, \dots, xn::\langle \text{Tun_n} \rangle) = \langle \text{Выражение} \rangle(x1, \dots, xn))$$

По данному формату *define*-процедура в качестве *F*-процедуры от *типируемых* ($x1, \dots, xn$)-аргументов/переменных определяет выражение от этих ведущих переменных. При этом, допускается применение не только типов, идентифицируемых функцией *type*, но и структурных типов. В то время как фактические аргументы, определяемые их типированными формальными аргументами, могут принимать любые значения, совместимые с указанными для них типами. Итак, успешный вызов процедуры *define* возвращает *NULL*-значение. Тогда вызов процедуры, определенной таким способом, производится по конструкции вида $F(x1, \dots, xn)$. Следующий фрагмент иллюстрирует применение *define*-процедуры для определения простых процедур пользователя.

```
> define(G, G(x::integer, y::anything, z::fraction) = sqrt(x^3 + y^3 + z^3));
> interface(verboseproc = 3): eval(G);
proc() local theArgs, arg, look, me, cf, term;
option `Copyright (c) 1999 Waterloo Maple Inc. All rights reserved.`;
description "a Maple procedure automatically generated by define()";
me := eval(procname, 1);
theArgs := args;
look := tablelook('procname'(theArgs), ['/POS'(1, G, 3),
`/BIND'(1, 1, `/y3`::integer),
`/BIND'(2, 1, `/y1`::anything),
`/BIND'(3, 1, `/y2`::fraction),
`/PATTERN'((`/y3`^3 + `/y1`^3 + `/y2`^3)^(1/2))');
if look <> FAIL then eval(look, `/FUNCNAME`=procname) else 'procname'(theArgs) end if
end proc
> [G(42, 68., 19/99), G(47, 63., 2010), G(67., 43, 420/72), G(`REA`, `25.11.10`)];
[623.3137309, G(47, 63., 2010), G(67., 43, 35/6), G(REA, 25.11.10)]
> define(S, S(x, y::string, z::integer) = cat(x, "", y, "", z)); S(RANS, " - 25 ноября ", 2010);
S(RANS, " - 25 ноября ", 2010)
> proc(x, y::string, z::integer) cat(x, "", y, "", z) end proc(RANS, " - 25 ноября ", 2010);
```

RANS - 25 ноября 2010

> S1 := (x, y, z) -> cat(x, "", y, "", z): S1(RANS, " - 25 ноября ", 2010);

RANS - 25 ноября 2010

Второй пример данного фрагмента иллюстрирует вид *исходного* текста процедуры G, генерируемой пакетом посредством *define*-процедуры. При передаче определенной по *define* процедуре S некорректных фактических аргументов (*несоответствие числа фактических аргументов формальным и/или их типов*) вызов процедуры S возвращается невычисленным, как это хорошо иллюстрирует третий пример фрагмента. При этом, если определенная первым способом процедура допускает корректное выполнение в случае превышения количества *фактических* аргументов над *формальными* (*игнорируя лишние*), то в случае определенной третьим способом ситуация полагается *ошибочной*, возвращая вызов невычисленным. Напоминая второй способ определения процедур, третий отличается от него довольно существенным моментом, позволяя использовать типизацию формальных аргументов. Между тем, третий способ определения имеет и определенные ограничения, не позволяя применять в теле процедуры произвольные *Maple*-функции. Например, третий пример иллюстрирует некорректность вызова S-процедуры, определенной *третьим* способом (*в ее теле была использована cat-функция*), в то время как эквивалентные процедуры, определенные первым и вторым способом, возвращают вполне корректные результаты. Следовательно, *третий* способ задания пользовательских процедур рекомендуется использовать достаточно осмотрительно.

В определенной мере подобным третьему методу определения *функции* пользователя является использование специальной *unapply*-процедуры, которая имеет следующий достаточно простой формат кодирования, а именно:

unapply(<Выражение> {, <Ведущие переменные>})

где *Выражение* определяет собственно тело самой функции, а *Ведущие переменные* – последовательность ее формальных аргументов. При этом, вызов *unapply*-процедуры возвращает рассмотренную выше функциональную конструкцию в терминах *arrow*-оператора «->», как это иллюстрирует следующий достаточно простой фрагмент:

> G := unapply((gamma*x^2 + sqrt(x*y*z) + x*sin(y))/(Pi*y^2 + ln(x+z) + tan(y*z)), x, y, z);

$$G := (x, y, z) \rightarrow \frac{\gamma x^2 + \sqrt{x y z} + x \sin(y)}{\pi y^2 + \ln(x + z) + \tan(y z)}$$

> evalf(G(6.8, 6.3, 4.3)), G(m, n, p), evalf(G(Pi/2, Pi/4, Pi));

$$0.3239528551, \frac{\gamma m^2 + \sqrt{m n p} + m \sin(n)}{\pi n^2 + \ln(m + p) + \tan(n p)}, 1.674847463$$

> W := [unapply(sin(x), x), unapply(x, x), unapply(x^3, x)], W(6.8);

$$W := [x \rightarrow \sin(x), x \rightarrow x, x \rightarrow x^3], [0.4941133511, 6.8, 314.432]$$

Реализация *unapply*-процедуры базируется на применении λ -исчисления, в то время как собственно сама процедура применяется обычно при использовании *вычисляемых* выражений для создания функциональных конструкций. Та же процедура позволяет

достаточно эффективно производить функциональные определения в рамках таких структур, как *список*, *множество*, *массив*. В наиболее же массовых случаях определения пользовательских функций оба представленных средства функциональный оператор «->» и *unapply*-процедуру можно полагать эквивалентными, хотя в общем случае они и имеются существенные различия, детально рассматриваемые, например, в [30-34].

При необходимости определения функций с неизвестным количеством формальных аргументов в общем случае второй аргумент *unapply*-процедуры (*последовательность ведущих переменных*) может не кодироваться, как иллюстрирует следующий фрагмент

```
> SV := unapply(evalf([nargs, sqrt(sum(args['k']^2, 'k' = 1..nargs))/nargs], 6));
```

$$SV := () \rightarrow \left[nargs, \frac{\sqrt{\sum_{k=1}^{nargs} args_k^2}}{nargs} \right]$$

```
> 6*SV(68,43,63,48,14,22), evalf(SV(68,43,63,48,14,22), 4); => [36, 13426^(1/2)], [6., 19.32]
```

```
> VS := () -> evalf([nargs, sqrt(sum(args['k']^2, 'k' = 1..nargs))/nargs], 6);
```

$$VS := () \rightarrow evalf \left(\left[nargs, \frac{\sqrt{\sum_{k=1}^{nargs} args_k^2}}{nargs} \right], 6 \right)$$

```
> VS(68, 43, 63, 48, 14, 22), evalf(VS(64, 39, 59, 44, 10, 17), 6); => [6., 19.3118], [6., 17.81]
```

Так, *первый* пример фрагмента представляет SV-функцию от неопределенного числа формальных аргументов, определенную на основе *unapply*-процедуры, тогда как 2-й представляет VS-функцию, определенную на базе функционального оператора «->», эквивалентную (согласно реализованного алгоритма) SV-функции. Но как иллюстрируют примеры фрагмента, между обоими определениями имеется существенное различие: игнорируются некоторые *встроенные* процедуры (*evalf*, *convert*) при определении SV-функции. Поэтому, в общем случае способ определения пользовательской функции на основе функционального оператора «->» является более предпочтительным.

Отметим, что представленный выше способ определения пользовательской функции на основе *define*-процедуры по целому ряду характеристик удобнее способа, который базируется на функциональном операторе «->» или *unapply*-процедуре, как хорошо иллюстрирует следующий весьма простой, но поучительный фрагмент, а именно:

```
> R := [68,43,63,48,14,22]: S:= [42,47,62,67,89,96]: define(V, V(x::integer) = interp(R, S, x));
```

```
> evalf(V(72)); => 43.97484029
```

```
> R := [68,43,63,48,14,22]: S:= [42,47,62,67,89,96]: define(V, V(x) = interp(R, S, x));
```

```
> evalf(V(72)); => V(72)
```

```
> R := [68,43,63,48,14,22]: S:= [42,47,62,67,89,96]: define(V, V(x::float) = interp(R, S, x));
```

```
> evalf(V(72)); => V(72)
```

```
> define(H, H(x::anything, y::integer) = sqrt(x^2 + y^2));
```

```
> H(sqrt(75),5), evala(H(sqrt(75),5)), H(sqrt(75),68.42); => 100^(1/2), 10, H(5*3^(1/2), 68.42)
```

```

> x := [a, b, c, d, e, f]: define(DD, DD(x[k]::prime$k=1..6) = (sqrt(sum(x[k]^2, k=1..6)))));
> evala(DD(3, 2010, 72, 14, 22, 7)), evala(DD(61, 67, 41, 43, 29, 13));
      DD(3, 2010, 72, 14, 22, 7), 5*510^(1/2)
> restart; define(R, R(x::even, y::odd) = sqrt(x^2 + y^2)), R(14, 21)^2;  => 637
> define(R, R(x::even, y::odd) = sqrt(x^3 + y^3)), R(14, 21);
Error, (in define) R is assigned
> R := 'R': define(R, R(x::even, y::odd) = sqrt(x^3 + y^3)), R(14, 21)^2;  => 12005
> define(AG, AG(0)=1, AG(1)=2, AG(2)=3, AG(t::integer) = AG(t-3)+2*AG(t-2)+AG(t-1));
> map(AG, [6, 8, 10, 12, 14, 22]);  => [75, 346, 1596, 7363, 33969, 15388410]
> define(G3, G3(seq(x | | k::integer, k=1 .. 6)) = sum(args[k], k=1 .. nargs)/nargs);
> 6*G3(14, 21, 43, 48, 63, 68);  => 257
> define(G4, G4()=sum(args[k], 'k'=1..nargs)/nargs): 6*G4(14, 21, 43, 48, 63, 68);  => 257 #8
> define(G4, G4()=sum(args[k], 'k'=1..nargs)/nargs): 6*G4(14, 21, 43, 48, 63, 68);  #10, 11
      6 G4(14, 21, 43, 48, 63, 68)
> define(G6, G6() = [nargs, args]): #8
Error, (in insertpattern) Wrong kind of arguments
> define(G6, G6()=[nargs, args]): G6(42,47,67,62,89,96);  => G6(42, 47, 67, 62, 89, 96) #10, 11
> G7 := () -> [{nargs}, args]: G7(42, 47, 67, 62, 89, 96);  => [{6}, 42, 47, 67, 62, 89, 96]
> define(G8, G8()=args), define(G9, G9()=nargs), G8(14,21), G9(14,21,43,48);  => 14, 4 #8
> define(G8, G8() = args), define(G9, G9() = nargs), G8(14, 21), G9(14, 21, 43, 48);  #10, 11
      G8(14, 21), G9(14, 21, 43, 48)
> define(S1, define(S2, S2()= sum(args[k], 'k'=1..nargs)), S1() = S2(V, G, S)*nargs);
> S1(14, 21, 43, 48, 63, 68), S2(14, 21, 43, 48, 63, 68);  => 6 V + 6 G + 6 S, 257 #8
> S1(14, 21, 43, 48, 63, 68), S2(14, 21, 43, 48, 63, 68);  #10, 11
      S1(14, 21, 43, 48, 63, 68), S2(14, 21, 43, 48, 63, 68)

```

Фрагмент иллюстрирует результат реакции пакета на переопределения **R**-функции, определенной на основе *define*-процедуры, а также вызовы **H**-функции пользователя от двух типированных аргументов. На примере **H**-функции проиллюстрированы два следующие весьма принципиальных момента, а именно:

- (1) вызов *define*-определенной функции на фактических аргументах, не отвечающих определенным для них типам (либо в случае определения аргументов нетипированными) возвращается невычисленным; это же демонстрирует и **R**-функция;
- (2) в результате вызова *define*-определенной функции может потребоваться в общем случае последующая обработка *evala*-функцией с целью получения окончательного приведенного результата.

Применение последовательности свойств дало возможность определить *рекуррентную* **AG**-функцию. При этом, пример определения **G3**-функции показывает возможность использования специальных переменных *args* и *nargs* в *define*-определении функции, тогда как пример **G4**-функции дополнительно иллюстрирует определение функции от неопределенного числа аргументов. Однако следует учитывать, что использование переменных *args* и *nargs* в *define*-определении носит несколько более ограниченный

характер, чем при других способах определения функций пользователя. Возникают здесь и другие проблемы, связанные с несовместимостью релизов пакета. В частности, **G4**-функция определяется в *Maple 8* корректно, тогда как в релизах *Maple 10,11* нет. В примерах определения эквивалентных функций **G6** и **G7** иллюстрируется, что в плане представимости типов выражений, используемых при определении функции, *arrow*-оператор « \rightarrow » более предпочтителен. Примеры определения функций **G8** и **G9** также иллюстрируют ограничения *define*-метода определения функций. В этом отношении следует отметить, что реализация *define*-функции более младших релизов *Maple (8-й и ниже)* во многих отношениях была более эффективной [8–14]. Так, например, **G8**, **S1** корректно определяются в *Maple 8*, не определяются в *Maple 10, 11*. Наконец, пример определения **S1**-функции иллюстрирует (в ряде случаев весьма полезную) возможность рекурсивного использования функции *define* для определения функций. К сожалению, в релизах пакета **10** и выше данный прием корректно не работает.

Применение *assign*-процедуры для присвоения определения процедуры некоторому идентификатору (имени) позволяет включать определения функций непосредственно в вычислительные конструкции, соблюдая только обязательное правило: вычисление определения процедуры должно предшествовать первому ее вызову в содержащем ее вычисляемом выражении. Более того, сказанное в полной мере относится и к любому другому способу определения функции, как иллюстрирует простой пример:

```
> assign(G1, unapply([nargs, sum(args['k'], 'k'=1..nargs)]));
> assign(G2, () -> [nargs, sum(args['k'], 'k'=1..nargs)]);
> define(G3, G3()= nargs*sum(args[k], 'k'=1..nargs)), G1(42,47,67,62,89,96),      #8
    G2(42, 47, 67, 62, 89, 96), G3(42, 47, 67, 62, 89, 96);  => [6, 403], [6, 403], 2418
> assign(Kr, unapply([nargs, sum(args[p], 'p'=1..nargs)])), Kr(14, 21);  => [2, 35]
```

При этом, третий пример фрагмента, корректно выполняясь, в *Maple 8*, некорректен в среде *Maple 10* и выше относительно определения **G3**-функции. Последний пример фрагмента иллюстрирует вычисление списочной конструкции, содержащей *unapply*-определение **Kr**-процедуры, с последующим ее вызовом. Следует при этом отметить, что механизм определения функций пользователя, поддерживаемый *Mathematica* [1,6, 7], представляется нам существенно более гибким при реализации алгоритмов.

Между тем, следует иметь в виду, что в случае определения процедуры в конструкции с *assign*, ее определение становится глобальным, если ее имя не определено в секции **local**, содержащей конструкцию процедуры, как иллюстрирует следующий фрагмент

```
> P := proc() local Proc; assign(Proc = () -> '+'(args)); Proc(args)/nargs end proc;
> P(42, 47, 67, 68, 63, 43), eval(Proc);  => 55, Proc
> restart; P := proc() assign(Proc = () -> '+'(args)); Proc(args)/nargs end proc;
> P(42, 47, 67, 68, 63, 43), eval(Proc);  => 55, () -> '+'(args)
```

Вкратце рассмотрев способы определения процедур в *Maple*-языке и их структурную организацию, обсудим несколько более детально отдельные структурные компоненты процедур, определяемых первым способом как наиболее универсальным, так и часто используемым. Это тем более актуально, что позволит вам не только хорошо уяснить реализацию пакетных процедур и процедур нашей Библиотеки [45] (некоторые из них

в иллюстрационных целях представлены в настоящей книге), но и самому непосредственно перейти к программированию собственных конкретных приложений в среде пакета Maple, реализованных в форме процедур и их пользовательских библиотек.

6.1.2. Формальные и фактические аргументы Maple-процедуры

Формальный аргумент процедуры в общем случае принимает следующий вид *Id::Tun*, т.е. *Id*-идентификатор с приписанным ему *типом*; между тем, кодирование *типа* не является обязательным и в случае его отсутствия полагается любой тип, допускаемый пакетом. В случае определения *типированного* формального аргумента при передаче процедуре фактического аргумента в момент ее вызова, такой аргумент проверяется на соответствие типу формального аргумента. И при несовпадении типов возникает ошибочная ситуация с возвратом соответствующей диагностики. Совершенно другая ситуация имеет место при несовпадении *числа* передаваемых процедуре *фактических* аргументов числу ее формальных аргументов, а именно:

(1) в случае числа фактических аргументов, меньшего определенного для процедуры числа формальных аргументов, как правило, возникает ошибочная ситуация формы «Error, (in *Proc*) Proc uses a *nth* argument *Id*, which is missing», указывающая на то, что *Proc*-процедуре было передано меньшее число фактических аргументов, чем имеется *формальных* аргументов в ее определении; *Id* – идентификатор первого недостающего *n*-го фактического аргумента в вызове процедуры;

(2) в случае числа фактических аргументов, большего определенного для процедуры числа *формальных* аргументов, ошибочной ситуации не идентифицируется, лишние аргументы игнорируются. Более того, и в *первом* случае возможен корректный вызов. Это будет лишь в том случае, когда в теле процедуры не используются формальные аргументы явно. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> Proc:=proc(a,b,c) nargs, [args] end proc: Proc(1,2,3,4,5,6), [Proc(5)], [Proc()], [Proc(5,6,7)];
6, [1, 2, 3, 4, 5, 6], [1, [5]], [0, []], [3, [5, 6, 7]]
> Proc := proc(a, b, c) a*b*c end proc: Proc(14, 21, 2), Proc(14, 21, 7, 8, 9, 2, 3); => 588, 2058
> Proc(72, 420);
Error, invalid input: Proc uses a 3rd argument, c, which is missing
> A := proc(x::integer, y, z::float) evalf(sqrt(x^3 + y^3)/(x^2 + y^2)*z) end proc:
> A(2.3, 63, 68, 43, 14, 21);
Error, invalid input: A expects its 1st argument, x, to be of type integer, but received 2.3
> A(2010, 440);
Error, invalid input: A uses a 3rd argument, z (of type float), which is missing
> A(68, 42, 19.42, 72), A(68, 42, 9.42, 420, 63); => 1.894920561, 0.9191633206
```

При вызове процедуры с передачей ей *фактических* выражений для соответствующих формальных аргументов первые вычисляются, а по-возможности и упрощаются, и их значения передаются в тело процедуры для замещения соответствующих *формальных* аргументов, после чего производится вычисление составляющих тело предложений с возвратом значения *последнего* вычисленного предложения, если не было определено

противного. В случае наличия в определении процедуры *типированных* формальных аргументов элементы последовательности передаваемых при ее вызове фактических значений проверяются на указанный тип и в случае их несовпадения инициируется *ошибочная* ситуация, в противном случае выполнение процедуры продолжается. При этом, в качестве *типов* формальных аргументов процедуры используются любые из допустимых пакетом и опознаваемых функцией *type* и процедурой *whattype*. Между тем, при использовании *нетипированного* формального аргумента рекомендуется все же указывать для него *anything*-тип, информируя других пользователей процедуры о том, что для заданного формального аргумента допускаются значения любого типа, например, кодированием заголовка процедуры в виде *proc(X::integer, Y::anything)*.

В качестве *формальных* аргументов могут выступать *NULL*, *последовательность* простых или типированных символов; в простейшей форме – это *последовательность* символов, используемых в процедуре. При этом, допустимы более сложные формы аргументов: значения по умолчанию, зависимости между аргументами наряду с ограничением на число передаваемых аргументов. Типированная переменная кодируется в формате: *Переменная::Тип*. Тип может быть простым либо сложным. При обнаружении в точке вызова процедуры *фактического* аргумента с типом, отличным от определенного для *формального* аргумента, соответствующего ему, возникает *особая* ситуация с возвратом через *lasterror*-переменную и в текущий сеанс соответствующей диагностики:

```
> AVZ := proc(a::integer, b::float) a*b end proc: AVZ(68, 63);
```

```
Error, invalid input: AVZ expects its 2nd argument, b, to be of type float, but received 63
```

```
> lasterror;
```

```
"invalid input: %1 expects its %2 argument, %3, to be of type %4, but received %5"
```

Использование *типированных* формальных аргументов позволяет контролировать на допустимость передаваемые процедуре фактические аргументы; между тем, данный подход не совсем удобен при решении вопроса робастности процедур. С этой целью рекомендуется обеспечить проверку получаемых процедурой *фактических аргументов* в теле самой процедуры и производить соответствующую программную обработку с целью устранения возможных нежелательных последствий, вызванных данного типа фактическими аргументами. В качестве довольно простого примера модифицируем предыдущий фрагмент следующим вполне очевидным образом, а именно:

```
> AVZ := proc(a::numeric, b::numeric) local a1, b1;
```

```
    assign(a1 = a, b1 = b); if not type(a, 'integer') then a1 := round(a) end if;
```

```
    if not type(b, 'float') then b1 := evalf(b) end if; a1*b1
```

```
end proc;
```

```
> AVZ(68, 19.42), 17*AVZ(21/14, 63); ⇒ 1320.56, 2142.0
```

Большинство *процедур* нашей Библиотеки [45] использует именно подобный данному программный подход к обработке получаемых процедурой фактических аргументов на предмет их допустимости и, по-возможности, производятся допустимые согласно реализуемого процедурой алгоритма корректировки или инициируются ошибочные ситуации. Такой подход существенно повышает робастность процедур относительно *некорректных* фактических аргументов, в целом ряде случаев позволяя их исправлять.

Между тем, следует еще раз отметить, что *контроль* поступающих на вход процедуры фактических аргументов на основе механизма типирования формальных аргументов обеспечивается только на самом общем уровне, в то время как для обеспечения более тонкой их фильтрации на предмет допустимости с целью обеспечения *робастности* процедуры производится только в самом теле процедуры, исходя из реализуемого ею алгоритма и возможных особых и ошибочных ситуаций, обусловленных спецификой алгоритма и его реакцией на получение некорректных фактических аргументов.

Между тем, при организации процедур роль типированных формальных аргументов не ограничивается лишь задачами проверки входной информации, но несет также и ряд других важных нагрузок. Например, использование *uneval*-типа для *формального* аргумента позволяет *вне* процедуры производить его модификацию (*т.е. обновлять на «месте» Maple-объект, определенный вне тела процедуры под этим идентификатором*) как это хорошо иллюстрирует следующий достаточно простой фрагмент, а именно:

```
> AGN := proc(L::list, a::anything) assign('L' = subs(a = NULL, L)) end proc:
> L:= [68, 63, 43, 48, 14, 21]; AGN(L, 68);  => [68, 63, 43, 48, 14, 21]
Error, (in assign) invalid arguments
> Agn := proc(L::uneval, a::anything)
    if not type(L, 'symbol') then error "1st actual argument should be symbol but had
                                   received %1", whattype(L)
    elif type(eval(L), {set, list}) then assign('L' = subs([if (not type(a, {set, list})), a = NULL,
    seq(k = NULL, k = a)]), eval(L)))
    else error "1st actual argument should has type {list, set} but had received %1-type",
    whattype(eval(L))
    end if
end proc:
> Agn(L, 68), L, Agn(L, 63), L, Agn(L, {63, 43, 48, 14, 21}), L;
[63, 43, 48, 14, 21], [43, 48, 14, 21], []
> Agn(AVZ, 68), AVZ;
Error, (in Agn) 1st argument must has type {list, set} but had received symbol-type
> Agn([63, 43, 48, 14, 21], 63);
Error, (in Agn) 1st argument must be symbol but had received list
```

Попытка определить такую операцию для стандартно типированного *L*-аргумента в *AGN*-процедуре вызывает ошибку выполнения тогда как, определив тот же аргумент *L* как аргумент *uneval*-типа и применив в дальнейшем обращение к нему через *eval*-функцию, получаем вполне корректную *Agn*-процедуру, обеспечивающую «на месте» обновление списка/множества *L* путем *удаления* его элементов, определенных вторым *a*-аргументом, в качестве которого может выступать как отдельный элемент, так и их список/множество. Проверка же на тип фактического *L*-аргумента производится уже программно в самой процедуре; при этом, проверяется не только на тип *{list, set}*, но и на получение *идентификатора* объекта, а не его значения (*т.е. в качестве фактического L-аргумента должно выступать имя списка/множества*). Данный прием может оказаться весьма полезным в практическом программировании, именно он используется рядом процедур из нашей Библиотеки [10-14,28-33,41-45].

Уже начиная с *Maple 10*, кроме типирования *формальных* аргументов пакет допускает определение для позиционных аргументов значений по умолчанию. Кодируется это посредством оператора присвоения в формате *<аргумент> := <значение>*, например:

```
> P := proc(a, b, c := sin(x), d) (a+b)/(c+d) end proc;
Error, optional positional parameters must follow any required positional parameters
> P := proc(a, b, c := sin(x), d := cos(x)) (a+b)/(c+d) end proc;
> Proc(a, b), Proc(14, 21, 72), Proc(14, 21, 72, 420);
(a + b)/(sin(x) + cos(x)), 35/(72 + cos(x)), 35/492
> proc(a,b,c,d) (a+b)/if(nargs=2, (sin(x)+cos(x)), if(nargs=3, (c+cos(x)), (c+d))) end proc;
> %(a, b), %(14, 21, 72), %(14, 21, 72, 420);
(a + b)/(sin(x) + cos(x)), 35/(72 + cos(x)), 35/492
```

Определенные таким способом позиционные аргументы должны быть последними в списке формальных аргументов, в противном случае возникает ошибочная ситуация, как это иллюстрирует первый пример фрагмента. Этого можно избежать, кодируя их в фигурных скобках, как хорошо иллюстрирует следующий весьма простой пример:

```
> Pro := proc(f::symbol, x::symbol, {b:= evalf}, c::range(posint)) b(int(f(x), x=c)) end proc;
> Pro(sin, y, 1..5, b=G), Pro(sin, y, 1..5, G), Pro(sin, y, 1..5, evalf), Pro(sin, y, 1..5, b);
G(cos(1) - cos(6)), -0.4198679808, -0.4198679808, true(cos(1) - cos(6))
```

Из него, в частности, следует, использование в вызове процедуры такого аргумента в виде уравнения либо значения по умолчанию приводит к требуемым результатам, в то время как вызов только на *левой* части вместо фактического аргумента производит подстановку вместо фактического аргумента *true*-значения. В определенной степени такой механизм и его расширения в ряде случаев оказываются довольно полезными, однако вызывают несовместимость с более ранними релизами *Maple*. Между тем, этот механизм в целом ряде случаев весьма несложно программно реализуется прежними средствами пакета. Так, второй пример предпоследнего фрагмента может быть легко реализован способом, представленным *последним* примером фрагмента. Естественно, при большем количестве позиционных параметров программирование усложняется, а представленный механизм более эффективен; однако, имеется одно ограничение – его действие ограничивается последними формальными аргументами.

В общем случае *последовательность формальных* аргументов можно представить в виде 2 подпоследовательностей – *позиционные* и *необязательные* и/или *ключевые* аргументы; при этом, последние могут произвольно чередоваться только среди себе подобных и не пересекаться с позиционными. Выше представлен один из механизмов поддержки *необязательных* и *ключевых* аргументов с применением для них значений по умолчанию.

Между тем, сам пользователь может разрабатывать достаточно эффективные системы обработки аргументов процедуры, как позиционных, так и ключевых со значениями для них по умолчанию; *один* из подобных достаточно простых подходов представляет следующий весьма прозрачный фрагмент, а именно:

```
> Proc := (x, y, z) -> if(type(eval(y), symbol), procname(x, 2010, z), '+'(x, y, z));
> Proc(a, b, c), Proc(a, x - y, c), Proc(sin(x), h, cos(y));
```

$$a + 2010 + c, a + x - y - c, \sin(x) + 2010 + \cos(y)$$

Читателю рекомендуется рассмотреть примененный практически полезный прием.

Для создания процедуры наряду с предложениями, описывающими непосредственный алгоритм решаемой задачи (*а в ряде случаев и для обеспечения самого алгоритма*), **Maple**-язык предоставляет ряд весьма важных средств, обеспечивающих функции, которые управляют выполнением процедуры. Прежде всего к ним можно отнести переменные *args* и *nargs*, возвращающие последовательность переданных процедуре фактических аргументов и их количество соответственно. Данные средства имеют смысл только в рамках процедуры, а по конструкциям вида *args*{ | [*n*] | [*n* .. *m*]} можно соответственно получать {последовательность фактических аргументов | *n*-й аргумент | аргументы с *n*-го по *m*-й включительно}. Тогда как *nargs*-переменная возвращает количество полученных процедурой фактических аргументов. Назначение этих средств довольно прозрачно и обуславливает целый ряд полезных приложений при разработке пользовательских процедур. Прежде всего, это относится к обработке фактических аргументов, которые получает процедура. Например, *nargs*-переменная необходима с целью обеспечения определенности выполнения вычислений при передаче процедуре неопределенного числа аргументов. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> SV := proc() product(args[k], k=1 .. nargs)/sum(args[k], k=1 .. nargs) end proc:
> 317*SV(42, 47, 68, 63, 89, 96, 420, 43, 48, 14, 21); => 6138226491591720960
> GN:=proc() [nargs, [args]] end proc: GN(V, G, S, A, Art, Kr); => [6, [V, G, S, A, Art, Kr]]
> map(whattype, [68, 21/14, ln(x), 7.2, "RANS"]); => [integer, fraction, function, float, string]
> Arg_Type := proc() map(whattype, [seq(args[k], k= 1 .. nargs)]) end proc:
> Arg_Type(63, 21/14, ln(x), 7.2, "RANS"); => [integer, fraction, function, float, string]
> Arg_Type := proc() map(whattype, [args[k]$k= 1 .. nargs]) end proc:
> Arg_Type(63, 21/14, ln(x), 7.2, "RANS"); => [integer, fraction, function, float, string]
```

Приведенный фрагмент достаточно прозрачен и особых пояснений не требует. При этом, как иллюстрирует *первый* пример, число передаваемых процедуре фактических аргументов не обязательно должно соответствовать числу ее формальных аргументов. Этот пример иллюстрирует факт, что в общем случае процедуру можно определять, не привязываясь к конкретному списку ее формальных аргументов, а привязываясь к формальной функциональной конструкции следующего общего вида, а именно:

proc() <ТЕЛО> {Ψ(args[1], args[2], ..., args[n]) | n = nargs} end proc {;|:}

что оказывается довольно удобным механизмом при организации процедур, в первую очередь, ориентированных на задачи символьных вычислений и обработки [28-33,44].

Дополнительно к переменным *args* и *nargs* можно отметить еще одну весьма важную переменную *procname*, возвращающую имя процедуры, ее содержащей. В целом ряде случаев эта переменная оказывается весьма полезной, в частности, при возвращении вызова процедуры невычисленным. Для этой цели используется конструкция весьма простого формата '**procname(args)**'. Следует отметить, что многие процедуры пакета возвращают результат именно в таком виде, если не могут решить задачу. Целый ряд процедур и нашей Библиотеки [44,45] поступают аналогичным образом. Между тем,

переменная *procname* может весьма успешно применяться и в ряде других полезных приложений. Следующий фрагмент иллюстрирует применение данной переменной как для организации возврата вызова процедуры невычисленным, так и для вывода в этой связи соответствующего диагностического сообщения:

```
> Avz := () -> `if`(nargs <= 6, WARNING("%1(%2) = %3", procname, seqstr(args),
                                     `+`(args)/nargs), 'procname(args));
> Avz(68, 63, 43, 48, 14, 21);
Warning, Avz(68, 63, 43, 48, 14, 21) = 257/6
> Avz(68, 63, 43, 48, 14, 21, 6); => Avz(68, 63, 43, 48, 14, 21, 6)
```

Процедура *Avz* при получении при вызове не более шести фактических аргументов выводит сообщение, описывающее вызов процедуры и его результат, в противном же случае вызов процедуры *Avz* возвращается невычисленным.

Для проверки допустимости передаваемых процедуре необязательных фактических аргументов служит процедура *DefOpt(args)*, где *args* – передаваемые аргументы [45]:

```
> DefOpt := proc() local `0`, `1`, `2`, `3`;
[assign(`0` = [unassign(`2`, `3`)], `1` = [],
`3` = () -> {seq(lhs([args][1][`2`]), `2` = 1 .. nops(args[1]))}),
seq(`if`(type(args[`2`], 'equation') and type(lhs(args[`2`]), symbol), assign(`0` = [op(`0`),
lhs(args[`2`]) = [op(rhs(args[`2`])], 'makelist')], `if`(type(args[`2`], 'equation') and
type(lhs(args[`2`]), 'posint'), assign(`1` = [op(`1`), args[`2`]]),
ERROR("invalid arguments %1", [args])), `2` = 1 .. nargs),
`if`(nops(`1`) < `3`(`1`)[-1] or nops(`1`) <> nops(`3`(`1`))),
ERROR("invalid positional values %1", `1`), `if`(nops(`0`) <> nops(`3`(`0`))),
ERROR("invalid options values %1", `0`),
TABLE(['OptionParms' = TABLE(`0`), 'PositionalParms' = TABLE(`1`)))]
end proc;
```

Исходные тексты процедур из Библиотеки [45] предоставляют достаточно неплохой иллюстративный материал по применению аргументов, переменных процедуры *args* и *nargs*, а также *procname* для разработки различных приложений в среде пакета.

Дополнительно к двум рассмотренным процедурным переменным *args* и *nargs* с 10-й версией пакета введен ряд дополнительных процедурных переменных, а именно:

_passed – алиас для процедурной переменной *args*

_params – последовательность декларированных позиционных аргументов, которые были переданы процедуре

_options – последовательность опций в процедуре

_rest – последовательность недеklarированных аргументов, переданных процедуре

_npassed – алиас для процедурной переменной *nargs*

_nparams – число декларированных позиционных аргументов, переданных процедуре

_noptions – число опций в процедуре

_nrest – число недеklarированных аргументов, переданных процедуре

_nresults – число предполагаемых выходов из процедуры

Смысл их достаточно прозрачен уже из представленного описания, тогда как с более детальным их описанием можно ознакомиться по конструкции *?args*; примеры же их использования представляет нижеследующий достаточно простой фрагмент:

```
> GS:= proc(x, y, z) option remember; `if`(x = 1, RETURN(x+y), `if`(y = 2, RETURN(z+y),
`if`(z = 3, RETURN(x + z), NULL))); [[_params[1 .. -1]], _options, _rest, _nparams,
_noptions, _nrest, _nresults] end proc: P(68, 63, 43); => [[68, 63, 43], 3, 0, 0, undefined]
> GS(68, 63, 43, 14, 21); => [[68, 63, 43], 14, 21, 3, 0, 2, undefined]
> G := proc(x, y, z)::integer; if x = 1 then return x + y elif y = 2 then return z + y elif z = 3
then return x + z end if; [[_params[1 .. -1]], _options, _rest, _nparams, _noptions,
_nrest, _nresults] end proc:
> G(68, 63, 43); => [[68, 63, 43], 3, 0, 0, undefined]
> G(68, 63, 43, 14, 21); => [[68, 63, 43], 14, 21, 3, 0, 2, undefined]
```

При программировании процедур вышеперечисленные переменные в ряде случаев могут упрощать программирование, однако порождают несовместимость процедур «сверху-вниз» ввиду отсутствия их поддержки в более ранних релизах пакета. Между тем, и в более ранних версиях возможно создавать средства, эквивалентные указанным процедурным переменным. В частности, следующая простая процедура возвращает число возможных выходов произвольной *P*-процедуры, исключая встроенные:

```
> Nresults := (P::procedure) -> `if`(P = Nresults, 1, nops(Search2(convert(eval(P), string),
{"::evaln", " return ", "RETURN(", " error ", "ERROR(")) + 1):
> Nresults(GS), Nresults(MkDir), Nresults(sin), Nresults(Nresults); => 4, 9, 3, 1
A := proc(x,y,z) assign('_Rest' = convert('procname(args)', string)), args[nops({rhs(ParProc
(parse(_Rest[1.. `if`(search(_Rest, "(", 't'), t, 0) - 1]))[1, 1])) + 1 .. -1] end proc:
> A(a, b, c, 14, 21, 43, 48, 63, 68); => 14, 21, 43, 48, 63, 68
```

Тогда как конструкция, определяющая *_Rest*-переменную и включенная в процедуру *A*, эквивалентна процедурной *_rest*-переменной, определяющей последовательность *недекларированных* аргументов. Подобный подход допустим и для других переменных.

В старших версиях пакета определен механизм *модификации* аргументов процедуры, позволяющий определять специальные режимы обработки фактических аргументов при вызове процедур. В зависимости от релиза пакета набор данных модификаторов изменяется в сторону их расширения. Так в *Maple 11* определены 5 модификаторов, а именно: модификаторы *expects*, *depends*, *uneval*, *evaln* и *seq*, описания которых можно получить, например, по конструкции *?parameter_modifiers*. Так, модификатор *expects* изменяет необязательный аргумент на ожидаемый, определяя, что данный аргумент при вызове процедуры может быть опущен лишь тогда, когда опущены все остальные ее аргументы. Если же аргумент присутствует, его тип должен соответствовать типу в его определении, иначе возникает ошибочная ситуация. Недопустимо использование после (*но не до*) ожидаемого аргумента обязательного аргумента, вызывая ошибочную ситуацию. Следующий довольно простой фрагмент иллюстрирует сказанное.

```

> A := proc(x::expects(integer) := 75.42, y) x+y end proc;
Error, required parameter `y` cannot appear after optional or expected
parameters in procedure A
> A := proc(x::expects(integer) := 75.42) x end proc:
> A(), A(420), A(420, 75, 67, 15, 22);  $\Rightarrow$  75.42, 420, 420
> A(42.7);
Error, invalid input: A expects its 1st argument, x, to be of type integer, but received 42.7
> G := proc(x, y::expects(integer) := 75.42) x+y end proc:
> G(42), G(420); G(15, 4.7);  $\Rightarrow$  117.42, 495.42
Error, invalid input: G expects its 2nd argument, y, to be of type integer, but received 4.7

```

Как правило, типы формальных аргументов задаются при определении процедуры и при вычислении ее определения не тестируются, проходя проверку только в момент ее вызова на фактических аргументах. Между тем, в ряде случаев этого недостаточно и модификатор *depends* позволяет определять условие, расширяющее тип аргумента на некоторое *внешнее* относительно него условие, как иллюстрирует простой пример:

```

> S := proc(p::depends(polynom(float, t)), t::symbol) p end proc: S(t^2 + 7.5*t + 4.2, t);
 $t^2 + 7.5 t + 4.2$ 
> S := proc(p::depends(polynom(float, t)), t::symbol) p end proc: S(t^2 + 7.5*t + 4.2, y);
Error, invalid input: S expects its 1st argument, p, to be of type polynom(float, y),
but received t^2+7.5*t+4.2
> S := proc(p::polynom(float, t), t::symbol) p end proc: S(t^2 + 7.5*t + 4.2, y);
 $t^2 + 7.5 t + 4.2$ 
In[745]:= S[p_/_; PolynomialQ[p, t] && DeleteDuplicates[Map[IntegerQ,
CoefficientList[p, t]]] == {True}] := Module[{}, p]
In[746]:= {S[t^2 + 7.5*t + 4.2], S[t^2 + 75*t + 420]}
Out[746]= {S[4.2 + 7.5 t + t^2], 420 + 75 t + t^2}

```

Между тем, тип в *depends* должен сводиться к существующему типу *Maple*, не позволяя в отличие от *Mathematica* использовать более сложных тестов, довольно существенно ограничивая возможности такого механизма. Последний пример представляет более сложное тестирование в *Mathematica* фактического аргумента, подобное *depends*. Но в отличие от инициации ошибочной ситуации в пакете *Maple*, в пакете *Mathematica* вызов процедуры, чьи фактические аргументы не удовлетворяют тест, возвращается невычисленным, как и иллюстрирует предыдущий фрагмент.

Следующий модификатор может использоваться в 2 различных форматах: *evaln* либо *evaln(mun)*. В целом, тип *evaln* формального аргумента предназначен для наделения его свойством присваемого объекта. Данный механизм позволяет такие формальные аргументы использовать для организации возврата через них *дополнительных* выходов из процедуры и достаточно широко используется, в частности, в целом ряде процедур из нашей библиотеки [45]. Если же использован модификатор формата *evaln(mun)*, то результат, возвращаемый через него, должен иметь указанный *mun*. При этом, *evaln-*модификатор может использоваться лишь для обязательных *позиционных* аргументов.

Следующий достаточно простой фрагмент хорошо иллюстрирует сказанное.

```
> G := proc(x::evaln(integer)) sin(x) + cos(x) end proc;
> h := 420: G(h);  => sin(a) + cos(x)
> h := 7.5: G(h);
Error, invalid input: G expects its 1st argument, x, to be of type evaln(integer),
but received h := 7.5
In[579]:= G[x_ /; SymbolQ[x] || IntegerQ[x], y_ /; SymbolQ[y] || IntegerQ[y]] :=
Module[{a = 75, b = 420}, Quiet[y = a; x = b]; x^2 + y^2]
In[580]:= {G[h, t], {h, t}}
Out[580]= {182025, {420, 75}}
In[581]:= G[42, 69]
Out[581]= 6525
```

В *Mathematica* для процедур в качестве формальных аргументов можно определять и обычные аргументы, через которые процедуре передаются *фактические* аргументы, и в терминологии *Maple evaln*-аргументы, через которые процедура может возвращать дополнительные выходы помимо главного – последнего предложения либо функции **Return**. Второй пример фрагмента иллюстрирует обе данные возможности, заданные одновременно при определении процедуры. И в этом отношении пакет *Mathematica*, на наш взгляд, обладает определенным универсализмом. Ознакомиться с остальными двумя модификаторами аргументов процедуры можно по *?parameter_modifiers*.

Между тем, в связи с изложенным целесообразно еще раз акцентировать внимание на таком важном вопросе, как совместимость программного обеспечения «сверху-вниз». В общепринятом понимании именно совместимость «снизу-вверх» в значительной мере определяет важную компоненту качества программного обеспечения и здесь именно пакет *Mathematica* относительно *Maple* имеет существенные преимущества. Однако, в целом ряде случаев востребована и определенная степень совместимости основных конструкций «сверху-вниз». И в этом отношении описанные новшества *Maple* в лице введенного с 11-го релиза механизма *модификации* аргументов процедуры делает его несовместимым уже с 10-м релизом. Естественно развитию должен быть подвержен и *Maple*, но изменения базовых конструкций говорят о изначальной непродуманности.

Отметим еще ряд новшеств в определении процедурных объектов. Если до релиза 10 заголовок процедуры имел лишь один тип формальных аргументов – *обязательные* позиционные с синтаксисом *Arg{::min}*, то в последующих релизах был определен ряд новых типов формальных аргументов. Так, в релизе 10 были введены *необязательные* позиционные параметры, определяемые синтаксисом, а именно: *Arg::min := значение*. Если такой аргумент опущен при вызове процедуры либо получил значение, которое не отвечает приписанному типу, тогда соответствующий ему фактический аргумент по умолчанию получает приписанное ему *значение*, например:

```
> G := proc(x::integer := 42, y::integer := 69) x + y end proc;
> [G(), G(75, 400), G(a, b), G(a, 147.5), G(420, b)];  => [111, 475, 111, 111, 489]
```

В *Maple* релиза 11 был дополнительно определен новый тип аргументов – *ожидаемый*

позиционный аргумент, использующий синтаксис: *Arg::expects(min) := значение*. Этот тип аргумента подобен *предыдущему* за тем лишь исключением, что соответствующий аргумент при вызове процедуры может быть опущен лишь тогда, если все остальные аргументы *опущены* также. Тип фактического аргумента должен соответствовать типу формального аргумента, иначе возникает ошибочная ситуация, например:

```
> S := proc(x::integer := 42, y::expects(integer) := 75) x + y end proc:
```

```
> S(420, a);
```

Error, invalid input: S expects its 2nd argument, y, to be of type integer, but received a

```
> S(), S(420, 15), S(420); ⇒ 117, 435, 495
```

Ключевой аргумент непозиционен, кодируясь в виде {*ключевое слово::тип := значение*} и получая значение, если при вызове процедуры аргумент появляется в виде *ключевое слово = значение*. Левая часть аргумента определяет сам *ключевой аргумент*, который должен получить значение, и *правая* часть определяет получаемое *значение*. При этом, если закодировано лишь *ключевое слово*, полагается *ключевое слово = true*, например:

```
> G := proc(x::integer, {y::integer := 75}, z::integer) x + y + z end proc:
```

```
> G(42, 47), G(y=420, 42, 47), G(42, 100, y=420); ⇒ 164, 509, 562
```

```
> G(y=42.69, 42, 47);
```

Error, invalid input: G expects value for keyword parameter y to be of type integer, but received 42.69

Как правило, если в вызове процедуры указано больше аргументов, чем формальных, процедуре передаются *все* полученные фактические аргументы, которые доступны в теле процедуры через специальные переменные *_rest* и *_passed*, позволяя процедурам использовать произвольное число аргументов. Между тем, при кодировании маркера \$ обеспечивается ограничение на количество передаваемых фактических аргументов:

```
> W := proc(x::integer := 75, $) x^3 end proc: W(420), W(); ⇒ 74088000, 421875
```

```
> W(42, 69);
```

Error, invalid input: too many and/or wrong type of arguments passed to W; first unused argument is 69

Детальнее с организацией механизма формальных аргументов и всеми допустимыми их типами можно ознакомиться, в частности, по конструкции *?parameter_modifiers*.

6.1.3. Локальные и глобальные переменные процедур в Maple

Используемые в теле процедуры переменные по области определения делятся на две группы: *глобальные (global)* и *локальные (local)*. Глобальные переменные определены в рамках всего текущего сеанса и их значения доступны как для использования в теле процедуры, так и вне процедуры в любой момент и в любой *Maple*-конструкции, где их применение корректно. Для указания переменной глобальной ее идентификатор кодируется в *global*-секции определения процедуры, обеспечивая процедуре доступ к данной переменной. По этой причине во избежание возможной рассинхронизации вычислений и возникновения ошибочных и особых ситуаций нами рекомендуется в

качестве глобальных использовать в процедурах только те переменные, чьи значения ими не модифицируются, а лишь считываются. Иначе не исключено возникновение отмеченных ситуаций, включая и *непредсказуемые*. Следующий пример иллюстрирует некорректность определения в процедуре глобальной k -переменной цикла:

```
> restart; proc() sum(args[k], k=1..nargs) end proc(66, 63, 43, 14, 21); ⇒ 207
> k := 42: proc() sum(args[k], k=1..nargs) end proc(66, 63, 43, 14, 21);
Error, (in unknown) invalid subscript selector
```

вызывающей в дальнейшем ошибочную ситуацию, обусловленную тем, что заданная k -переменная суммирования не определена в процедуре локальной переменной, то она принимается *глобальной*, т.к. ей в процедуре не присваивается значения. А так как вне процедуры k -переменная получила конкретное значение, то и в теле процедуры k -переменная не может служить *переменной* суммирования, инициируя ошибочную ситуацию. При этом, следует иметь в виду, что вычисление определения процедуры не изменяет значений содержащихся в ней глобальных переменных, а вычисляются они только в момент реального вызова процедуры.

Локальные переменные также можно *типировать* подобно аргументам, но результат данного типирования имеет смысл только при установке *kernelopts(assertlevel=2)*, как достаточно наглядно иллюстрирует следующий весьма простой пример:

```
> Proc := proc(a, b) local c::integer; c := a + b end proc: Proc(68.63, 6); ⇒ 74.63
> kernelopts(assertlevel = 2): Proc(68.63, 6);
Error, (in Proc) assertion failed in assignment, expected integer, got 74.63
```

Типизация локальных переменных особого смысла не имеет, другое дело возможность присвоения им начальных значений, как это поддерживает пакет *Mathematica*, тогда как в *Maple* это можно делать заданием начальных значений в начале тела процедур.

Если для переменных, используемых в определении процедуры, не была определена область их действия {**local**, **global**}, то пакет классифицирует их следующим образом. Каждая переменная, которая получает в *теле* процедуры определение по оператору «:=» или переменная цикла, определяемая функциями {*seq*, *add*, *mul*}, рассматривается *локальной (local)*, остальные полагаются *глобальными (global)* переменными. При этом, присваивание значений переменным по *assign*-процедуре не полагает их в качестве локальных, как наглядно иллюстрирует следующий весьма простой пример:

```
> proc() assign(a = '+'(args)) end proc(68, 63, 43, 48, 14, 21), a; ⇒ 257
```

При этом, если переменные *for*-цикла не определены локальными явно, то выводится предупреждающее сообщение следующего вида, а именно:

"Warning, `k` is implicitly declared local to procedure `Proc`"

где k и *Proc* – переменная цикла и имя процедуры соответственно, как иллюстрирует следующий весьма простой пример, а именно:

```
> Proc := proc() a := 0; for k from 1 to nargs do a := a + args[k] end do; a end proc:
Warning, `a` is implicitly declared local to procedure `Proc`
Warning, `k` is implicitly declared local to procedure `Proc`
> Proc(68, 63, 43, 48, 14, 21); ⇒ 257
```


Между тем, в случае встроенной функции *seq* ее переменная цикла рассматривается в качестве локальной без необходимости указания в *local*-секции, например:

```
> Proc := proc() `+`(seq(args[k], k=1 .. nargs)) end proc:
> k := 72: Proc(68, 63, 43, 48, 14, 21); ⇒ 257
```

Эта возможность играет немаловажную роль в определении функций *однострочными* экстракодами, позволяя организовывать циклические конструкции, чьи переменные являются локальными в области действия цикла. Целый ряд средств Библиотеки [45] широко использовал указанную возможность. Тогда как в случае функций *product* и *sum* переменные цикла полагаются глобальными, не выводя каких-либо сообщений, что предполагает их *явное* определение в *local*-секции. При этом, использование этих функций в процедуре выполняется корректно до тех пор, пока извне тела процедуры не было произведено определения их переменных цикла, например:

```
> Proc := proc() sum(args[k], k=1 .. nargs) end proc:
> Proc(68, 63, 43, 48, 14, 21), k; ⇒ 257, k
> Proc(68, 63, 43, 48, 14, 21), k; ⇒ 257, k
> k := 21: Proc(68, 63, 43, 48, 14, 21);
Error, (in Proc) invalid subscript selector
```

Между тем, *вне* зависимости от наличия предупреждающих сообщений настоятельно рекомендуется явно указывать *локальные* и *глобальные* переменные, позволяя не только избегать ошибок выполнения, но и намного более *ясно* воспринимать исходный текст процедур. Следующий сводный фрагмент иллюстрирует вышесказанное:

```
> g:=2: A:=proc(n) v:=68: [args, assign('g', 5), assign('v', 9), assign(cat(H, n), `h`)] end proc:
Warning, `v` is implicitly declared local to procedure `A`
> [A(72), g, v, A(14), whattype(H9), H9]; ⇒ [[72], 5, v, [14], symbol, H9]
> k := 68: H := proc() product(args[k], k=1 .. nargs)/sum(args[k], k=1 .. nargs) end proc:
> [k, H(42, 47, 63, 68, 96, 89, 14, 21, 4), k];
Error, (in H) invalid subscript selector
> k := 68: P := () -> [seq(args[k], k=1..nargs)]: P(72, 420, 2010), k; ⇒ [72, 420, 2010], 68
> k := 68: P := () -> [sum(args[k], k=1..nargs)]: P(72, 420, 2010), k;
Error, (in P) invalid subscript selector
> k := 68: P := () -> [product(args[k], k=1..nargs)]: P(72, 420, 2010), k;
Error, (in P) invalid subscript selector
> k := 68: P := proc() for k to nargs do end do end proc: P(72, 420, 2010), k; ⇒ 68
Warning, `k` is implicitly declared local to procedure `P`
```

Итак, в вышеуказанных случаях соответствующие переменные процедуры в процессе ее вычисления неявно декларируются пакетом *Maple* локальными с выводом или без предупреждающих сообщений. С другой стороны, глобальные переменные даже без их явного декларирования в *global*-секции можно генерировать в рамках процедуры, как это иллюстрирует *первый* пример предыдущего фрагмента. Делать это позволяет процедура *assign*. Однако работа с такими *глобальными* переменными чревата весьма

непредсказуемыми последствиями. Таким образом, практическое программирование в среде *Maple*-языка рекомендует следовать следующим двум правилам определения области действия переменных, используемых в процедурах, а именно:

- (1) *глобальными определять переменные, используемые только в режиме «чтения»;*
- (2) *локальные переменные определять явно в local-секции процедуры.*

Использование этих правил позволит избегать многих ошибок, возникающих лишь в момент выполнения *Maple*-программ, синтаксически и семантически корректных, но не учитывающих специфики механизма использования пакетом как *глобальных*, так и *локальных* переменных. А именно: если *глобальная* переменная областью определения имеет весь текущий сеанс, включая *тело* процедуры (*глобально переопределять ее можно внутри любой Maple-конструкции*), *локальная* переменная областью определения имеет только *тело* самой процедуры и вне процедуры она полагается неопределенной, если до того не была определена вне процедуры переменная с тем же идентификатором. Такой механизм имеет довольно глубокий смысл, позволяя локализовывать действия переменных рамками процедуры (*в общем случае «черного ящика»*), не влияя на общий вычислительный процесс текущего сеанса. Примеры предыдущего фрагмента весьма наглядно иллюстрируют практическую реализацию данного механизма *локализации* переменных в *Maple*-процедурах. При этом, весьма частный случай, когда значения *локальных* переменных совпадают со значениями одноименных *глобальных*, например:

```
> restart; P := proc(a, b) local x, y; x, y := 42, 68; x, y end proc;  
> x, y := 42, 68; P(a, b), x, y;  => 42, 68, 42, 68  
> restart; P := proc(a, b) local x, y; x, y := a, b; x, y end proc;  
> x, y := 42, 68; P(a, b), x, y;  => a, b, 42, 68
```

особого смысла не имеет. Вопрос определения в *local*-секции локальных переменных находится полностью в компетенции пользователя, исходя из сущности реализуемого процедурой вычислительного алгоритма.

По *assign*-процедуре в теле процедур возможно назначать выражения как *локальным* (*заданным явно*), так и *глобальным* (*заданным явно или неявно*) переменным. Однако здесь существует одно достаточно существенное отличие. Как известно, пакет не допускает *динамического* генерирования имен в левой части оператора присваивания «:=», тогда как на базе *assign*-процедуры такая возможность реально имеется. Это действительно существенная возможность [28], весьма актуальная в целом ряде задач практического программирования. Однако, если по процедуре *assign* в теле процедуры присваивать выражения локальным переменным и сгенерированным одноименным с ними, то во втором случае присвоения производятся именно *глобальным* переменным, не касаясь *локальных* переменных процедур. Следующий достаточно простой пример наглядно иллюстрирует сказанное, а именно:

```
> restart; V42, G47 := 14, 21: proc() local V42, G47; assign(V42=68, G47=63); assign(cat(V,  
42)=420, cat(G, 47)=72); [V42, G47] end proc(), [V42, G47];  => [68, 63], [420, 72]
```

Таким образом, данное обстоятельство следует учитывать при работе с динамически генерируемыми переменными в *теле* процедур.

Следует еще раз подчеркнуть, для предложений присваивания в процедурах в целом ряде случаев использование *assign*-процедуры является единственно возможным. Но при данном подходе в общем случае требуется, чтобы левая часть уравнения $x = a$ для *assign*($x = a$) была неопределенным именем, т.е. для нее должно выполняться условие *type*(x , 'symbol') = true. И здесь вполне допустимо использование конструкций общего формата кодирования, имеющего следующий вид, а именно:

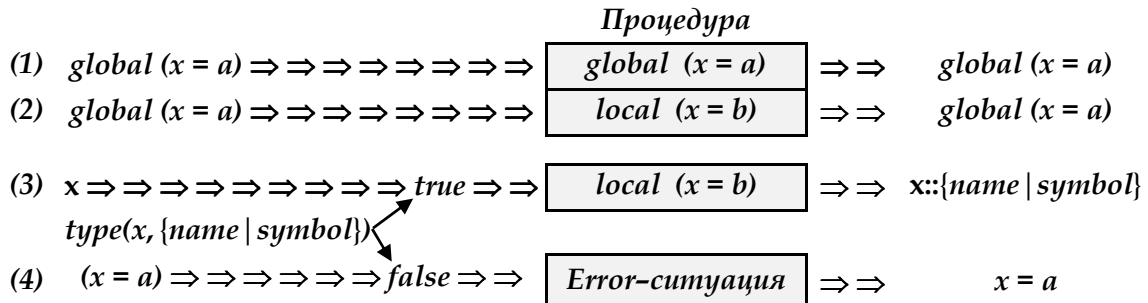
assign(op([unassign('Имя'), Имя]) = <Выражение>)

При этом, для таких объектов как процедуры, модули, таблицы и массивы (включая и векторы, и матрицы в смысле *Maple*, но не *NAG*) кодирование их имен в невычисленном формате *необязательно*, что может весьма существенно облегчить программирование. Следующий весьма простой фрагмент иллюстрирует вышесказанное:

```
> x := 68: assign(op([unassign(x), x]) = 63); x; => 68
Error, (in unassign) cannot unassign '68' (argument must be assignable)
> P := proc() end proc: M := module() end module: T := table(): A := array():
> map(whattype, map(eval, [P, M, T, A])); => [procedure, module, table, array]
> seq(assign(op([unassign(k), k])=63), k=[P,M,T,A]); [P,M,T,A], map(type, [P,M,T,A], odd);
[63, 63, 63, 63], [true, true, true, true]
```

Например, этот прием оказывается весьма удобным при необходимости присвоений выражений *глобальным переменным* или *фактическим* аргументам процедуры, которые передаются при вызове процедуры через формальный *uneval*-аргумент. Такой прием используется достаточно широко и процедурами нашей Библиотеки [45].

В связи с вышесказанным необходимо сделать одно *довольно* существенное замечание, поясняющее необходимость *явного* определения локальных переменных в процедуре. Первые два случая следующей схемы отражают классическое определение *глобальных* и *локальных* переменных, заключающееся в их явном декларировании.



В случае (1) явно либо неявно определенная глобальная x -переменная процедуры на всем протяжении текущего сеанса сохраняет свое значение до его переопределения в самой процедуре или вне ее. В случае (2) определенная локально в теле процедуры x -переменная в теле процедуры может принимать значения, отличные от ее *глобальных* значений вне ее, т.е. в процедуре временно подавляется действие одноименной с ней глобальной x или переменной. При этом, здесь имеют место и особые случаи (3, 4), не охватываемые стандартным механизмом. Для целого ряда функций, использующих ранжированные переменные (например, *product* и *sum*), возможны две ситуации, если

эти переменные не декларировались в процедуре явно. Прежде всего, как отмечалось выше, не выводится предупреждающих сообщений, что они полагаются *локальными*. Следовательно, они согласно трактовке *Maple*-языка должны будут рассматриваться *глобальными*. Однако, если на момент вызова процедуры, содержащей такие функции, ранжированная *x*-переменная была неопределенной (случай 3), то получая значения в процессе выполнения процедуры, после выхода из процедуры она вновь становится неопределенной, т.е. имеет место глобальное поведение переменной. Если на момент вызова процедуры *x*-переменная все же имела значение, то выполнение процедуры инициирует ошибочную ситуацию, а значение *x*-переменной остается неизменным (случай 4). Рассмотренные выше ситуации еще раз со всей определенностью говорят в пользу необходимости явного определения входящих в процедуру переменных.

Наряду со сказанным, *локальные* переменные могут использоваться в теле процедур в качестве ведущих переменных с неопределенными для них значениями, например:

```
> y := 68: GS := proc(p) local y, F; F := y -> sum(y^k, k = 0 .. n); diff(F(y), y$p) end proc:
> [y, simplify(GS(2))];
```

$$\left[68, \frac{y^{(n+1)} n^2 - 2 y^n n^2 + y^{(n-1)} n^2 - y^{(n+1)} n + y^{(n-1)} n + 2 y^n - 2}{(y-1)^3} \right]$$

Данный фрагмент иллюстрирует использование локальной *y*-переменной в качестве ведущей переменной *F(y)*-полинома от одной переменной. Тогда как вне процедуры глобальная *y*-переменная имеет конкретное числовое значение (68), т.е. их области не пересекаются.

Следующий простой фрагмент иллюстрирует взаимосвязь глобальных и локальных переменных, когда *первые* определяются как явно в *global*-секции, так и неявно через *assign*-процедуру. В свою очередь второй пример фрагмента иллюстрирует механизм действия *uneval*-типированного формального аргумента, когда (в отличие от *Maple* стандарта) ему могут присваиваться выражения на уровне глобальных переменных. Эффективность применения подобного весьма полезного приема иллюстрировалось на целом ряде процедур из нашей Библиотеки [45].

```
> Proc:= proc() local a; global b; x:=68; assign('m' = 67); a:= proc(x) m:= 47; assign('n' = 89);
x end proc; x*a(14) end proc:
Warning, `x` is implicitly declared local to procedure `Proc`
Warning, `m` is implicitly declared local to procedure `a`
> n, m := 100, 100: Proc(), n, m; ⇒ 952, 89, 67
> P1 := proc(m::uneval) local a, x; global b; a, x := 63, 43; b := 68; assign('m' = 42) end proc:
> m := 68: P1(m), a, x, b, m; ⇒ a, x, 68, 42
```

Еще на одном существенном моменте механизма *глобальных* и *локальных* переменных необходимо акцентировать внимание, предварительно пояснив понятие *уровневого* вычисления. Правила вычислений в пакете полагают в качестве нормальных полные вычисления для глобальных переменных и *1*-уровневые для локальных переменных. Поясним сказанное первым примером следующего весьма простого фрагмента:

```

> W := y^4;  ⇒ W := y^4 (1)
> y := z^3;  ⇒ y := z^3
> z := h^2;  ⇒ z := h^2
> h := 3;    ⇒ h := 3
> W;         ⇒ 282429536481
> [eval(W, 1), eval(W, 2), eval(W, 3), eval(W, 4)]; ⇒ [y^4, z^12, h^24, 282429536481] (2)
> G := proc() local W, y, z, h; W := y^4; y:= z^3; z:= h^2; h:= 2; W end proc: (3)
> [G(), eval(G()), evala(G()), evalf(G())]; ⇒ [y^4, 16777216, y^4, y^4]

```

в котором представлена очень простая *рекуррентная* цепочка выражений, вычисление которой выполняет полностью рекуррентную подстановку с обеспечением возврата конечного числового значения, т.е. производится полное вычисление *W*-выражения, идентификатор которого полагается глобальным. С другой стороны, вызов функции *eval(B, n)* обеспечивает *n*-уровневое вычисление заданного ее *первым* фактическим *B*-аргументом выражения, что наглядно иллюстрирует второй пример фрагмента.

Для полного вычисления произвольного *W*-выражения используется вызов функции *eval(W)*. Однако в первом примере фрагмента *W*-переменная является *глобальной*, что определяет ее полное вычисление, если (как это иллюстрирует второй пример) не было определено *противного*. Наконец, третий пример фрагмента иллюстрирует результат вычисления той же рекуррентной цепочки выражений, но теперь составляющих тело процедуры и идентификаторы которых определены в ней локальными. Из примера следует, что если не определено *противного*, то процедура возвращает только *первый* уровень вычисления *W*-выражения, поэтому для его полного вычисления требуется использование функции *eval*, как это и иллюстрирует последний пример фрагмента. Данное обстоятельство всегда следует иметь в виду, поскольку оно не имеет *аналогов* в традиционных языках программирования и наряду с требованием особого внимания обеспечивает целый ряд довольно интересных возможностей для программирования в различного рода приложениях.

Еще на одном довольно важном моменте стоит остановиться отдельно. Конструкции формата *op(2, eval(P))* и *op(6, eval(P))* возвращают соответственно последовательности *локальных* и *глобальных* переменных процедуры *P*. Между тем, если в *первом* случае это действительно так, то во втором случае ситуация совершенно иная, а именно: Если в процедуре производятся *присвоения* переменным по *assign*-процедуре и переменные явно не определены глобальными, то второй вызов их не распознает, например:

```

> restart; P := proc() local a, b, c; assign(x = 68, y = 63) end proc: P(): seq([op(k, eval(P))],
                                     k = [2, 6]), x, y; ⇒ [a, b, c], [], 68, 63

```

Это требует особого внимания при использовании в процедуре глобального уровня переменных во избежание нарушения вычислений в текущем сеансе при ее вызовах.

```

> Globals := proc(P::procedure) local a, b, c, f, k, p;
  assign(a = {op(6, eval(P))}, p = {op(2, eval(P))}, b = [], c = {anames('user')},
  f = cat(CDM(), "\$Art21_Kr15$.m"));

```

```

c := c minus {seq(`if`(cat("", k, " ") [1 .. 3] = "CM:", k, NULL), k = c)};
for k in [Assign, assign, assign67, assign6, assign7] do
  try b := [op(b), op(extrcalls(P, k))]
  catch "call of the form <%1> does not exist": next
end try
end do;
(proc() save args, f end proc)(op(c)), Unassign(op(c));
for k in b do parse(cat("try eval(parse(", k, ")) catch : NULL end try;"), 'statement')
end do;
c := a, ({anames('user')} minus p) minus a;
read f;
c, remove(f)
end proc:

> P := proc() local a; global x, y; a:= `(args); 5*assign(v=68), assign(g=63), assign(s=43);
a/(v + g + s) end proc: Globals(P);  => {y, x}, {v, s, g}
> Globals(mwsname);  => {}, {VGS_vanaduspension_14062005}
> Globals(holdof);  => {_avzagnartkrarn63}, {}
> Globals(`type/file`);  => {_datafilestate}, {_warning}
> Proc:= proc() local a, b; global c; assign('x'=68, 'c'=2010), assign67('y'=63, 43); a:=proc()
local z, h; assign67('z'=14, 21), assign(h=63); [z], h end proc; a() end proc: [Proc()],
Globals(Proc);  => [[14, 21], 63], {c}, {x, y, h, z}
> restart; Proc := proc() local a, b; global c; assign('x'=68, 'c'=2010), assign67('y'=63, 43);
a:=proc() local z, h; assign67('z'=14, 21), assign(h=63); [z], h end proc; a() end proc:
[Proc()]: c, x, [y], [z], h;  => 2010, 68, [63, 43], [z], h
> restart; P1 := proc() local a; assign('a'=68); a end proc: P1(), a;  => 68, a
> restart; P1 := proc() local a; assign67('a'=68); a end proc: P1(), a;  => 68, a

```

Для этой цели нами была определена процедура *Globals* [45], обеспечивающая более точное тестирование глобальных переменных процедур. Вызов процедуры *Globals(P)* возвращает 2-элементную последовательность множеств глобальных переменных для процедуры, имя которой определяется фактическим аргументом *P*. Первое множество возвращаемой последовательности содержит имена глобальных переменных, которые определяемых *global*-секцией процедуры *P*, в то время как второе множество содержит имена глобальных переменных, которые определяются вызовами процедур: пакетной *assign* и наших процедур *assign6*, *assign7*, *assign67* и *Assign* [9,45]. Процедура *Globals* функционирует в среде пакета *Maple* релиза 9 и выше. При этом, необходимо иметь в виду следующее достаточно важное обстоятельство, а именно.

Для простой процедуры *P* (не содержащей внутри себя других процедур) вызов *Globals(P)* возвращает и определенные в *global*-секции процедуры *P* переменные, и переменные значения которым присваивались посредством следующих процедур: *assign*, *assign67*, *assign6*, *assign7* и *Assign*. Совсем иным образом ситуация обстоит в случае вложенных процедур, когда вложенная процедура (подпроцедура) также будет содержать вызовы

указанных процедур, но некоторые из вычисляемых ими переменных определены в *local*-секции подпроцедуры. В данном случае такие переменные, идентифицируясь процедурой *Globals* глобальными, на самом деле носят локальный характер. Сказанное хорошо иллюстрирует пример *Proc*-процедуры предыдущего фрагмента (переменные *z* и *h*). Данное обстоятельство необходимо учитывать при использовании процедуры *Globals* относительно вложенных процедур, содержащих *assign*-вызовы.

В целях повышения надежности при операциях с глобальными переменными может быть использована процедура *uglobal* [45], чей вызов *uglobal('x','y','z',...)* обеспечивает отмену значений глобальных переменных *x, y, z, ...* на период выполнения процедуры, в которой данный вызов был сделан. После чего процедура может вполне произвольно их использовать с поддержкой возможности последующего восстановления значений переменных перед выходом из процедуры, не влияя на их состояния вне процедуры.

```
> uglobal := proc() local a, b;
  assign(a:=cat(CDM(),"/$ArtKr2$"),b:=interface(warnlevel)), null(interface(warnlevel=0));
  if type(a, 'file') then read3(a); fremove(a), null(interface(warnlevel = b))
  else save3(args, a); unassign(args), null(interface(warnlevel = b))
  end if
end proc;
> V:=68: P:=proc() global V; uglobal('V'); V:=`*(args); V,uglobal() end proc: P(14,21,43),V;
12642, 68
```

При этом, процедура *uglobal* в *m*-файле сохраняет значения *всех* переменных *x, y, z, ...*, обеспечивая возможность их последующего восстановления на основе вызова *uglobal()*. Предыдущий фрагмент представляет исходный текст *uglobal*-процедуры и пример конкретного ее применения при работе с глобальной *V*-переменной в процедуре *P*.

Начиная с *Maple 9*, для встроенной функции *anames* был определен новый аргумент *user*, по которому вызов *anames('user')* возвращает последовательность имен, значения для которых в текущем сеансе были определены пользователем. Данная возможность довольно важна для целого ряда приложений. С целью получения некоторого аналога данного средства для *Maple* релиза 8 и ниже нами была определена процедура *pusers*, чей исходный текст и пример применения представлены следующим фрагментом:

```
> pusers := proc() local a, b, c, d, k, p, h;
  assign(a = {}, b = interface(warnlevel), c = "procedure ", d = " has been activated in the
  current session"), interface(warnlevel = 0);
  p := ({seq(`if` (cat("", k, " ") [1 .. 3] = "CM:", NULL, k),
  k={anames('procedure')})} minus {anames('builtin')}) minus {`index`/`fill`, Testzero, pusers};
  p := {seq(`if` (cat("", p[k]) [1..9] = "unassign/", NULL, p[k]), k=1..nops(p))};
  for k in p do unassign('h');
    try ParProc1(k, h); if type(h, 'symbol') then a := {k, op(a)} end if
    catch "invalid input: %1 expects": next
  end try
end do;
```

```

null(interface(warnlevel = b)), unassign('_warning'), a
end proc:
> Proc := () -> `(args): Proc1:=() -> `(args): Proc2:=() -> `(args): Proc3:=() -> `(args):
> Proc4 := () -> `(args): pusers(); => {Proc1, Proc2, Proc3, Proc4, Proc}

```

Вызов процедуры *pusers()* возвращает множество имен процедур, которые в текущем сеансе определялись пользователем. При этом, в него не включаются процедуры, чьи определения находятся в *Maple*-библиотеках пользователя, логически сцепленных с главной библиотекой пакета. Данная процедура функционирует в среде *Maple 6 - 11*.

```

> elib := proc(e::symbol, L::{mla, mlib}) local a, k;
  for k in march('list', L) do
    if k[1] = "" || e || ".m" then
      a := SUB_S(["", "="], `(seqstr(op(k[2][1 .. 3])))`);
      return true, `if` (2 < nargs, assign(args[3] = a), NULL)
    end if
  end do;
false
end proc:
> elib(pusers, "C:/program files/maple 11/lib/userlib", 'h'), h; => true, 2007/1/21
> elib(MkDir, "C:/program files/maple 11/lib/userlib", 'h'), h; => true, 2006/10/31

```

Вызов представленной выше процедуры *elib(e,L)* возвращает *true*, если *Maple*-объект *e* находится в *Maple*-библиотеке *L*, иначе возвращается *false*. Если указан 3-й аргумент, то через него возвращается дата сохранения *e*-объекта в библиотеке. Представленные выше четыре процедуры *Globals*, *uglobal*, *pusers* и *elib* не только представляют весьма важные средства для продвинутой работы с процедурными объектами *Maple*, а также наглядно иллюстрируют достаточно веские преимущества пакета, предоставляющие в случае такой необходимости дополнять базовые средства пакета с помещением их в главную библиотеку либо в пользовательские, логически сцепленные с первой.

Еще на одном вопросе целесообразно остановиться. Уже в релизе 8 пакета обеспечена возможность типирования *локальных* переменных процедур, по формату аналогично глобальным переменным. Между тем, такая возможность не представляется довольно существенной, ибо тип локальных переменных (*в отличие от получаемых фактических аргументов*) легко определяется уже на стадии программирования процедур, не имея серьезной возможности контроля корректности выполнения. Между тем, излишество имеет место быть и может быть задействовано по установке *kernelopts(assertlevel = 2)*:

```

> S := proc(x::integer, {y::integer := 75}, z:=42) local a::integer; a:=100.5; x+y+z+a end proc:
> S(420); => 637.5
> kernelopts(assertlevel = 2): S(420);
Error, (in S) assertion failed in assignment, expected integer, got 100.5

```

Между тем, присваивания локальным переменным начальных значений невозможно, тогда как пакет *Mathematica* допускает как тестирование локальных переменных, так и присвоение им начальных значений, как иллюстрирует простой пример, а именно:


```

In[620]:= G[x_, y_] := Module[{a = If[IntegerQ[x], 75, 42.69], b = If[ListQ[y], Length[y],
    420]}, a*x + b*y]; S[x_, y_] := Module[{a = If[IntegerQ[x], 75, Return[$Failed]]},
    If[a == $Failed, Return[Defer[S[x, y]]], a]; a*(x + y)]
In[621]:= {G[42, {15, 22, 69}], G[4.2, {15, 22, 69}], G[42, xyz]}
Out[621]= {{3195, 3216, 3357}, {224.298, 245.298, 386.298}, 3150 + 420 xyz}
In[622]:= {S[42.69, 420], S[22, 420]}
Out[622]= {S[42.69, 420], 33150}

```

Еще на одном существенном моменте необходимо обратить внимание, иллюстрируя его нижеследующим весьма простым фрагментом, а именно:

```

> Av := proc(x, y) local a, b; b := 75; x+y; unassign('b'); map(type, [a, b], symbol) end proc:
> Av(15, 22); => [true, true]
In[659]:= Av[x_, y_] := Module[{a, b = 75}, x + y; Clear[b]; Map[HowAct, {a, b}]]
In[660]:= Av[42, 69]
Out[660]= {True, True}

```

Как следует из приведенного фрагмента, в *Maple*-процедуре локальные переменные (если не было определено противного) являются неопределенными символами; более того, в случае присвоения такой переменной значения вновь сделать ее неопределенной весьма просто, что и иллюстрирует первый пример предыдущего фрагмента. Тогда как 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. Поэтому в *Mathematica*-процедуре невозможно использовать локальные переменные в качестве формальных аргументов подпроцедур, через которые возвращаются дополнительные результаты. Для этой цели следует использовать глобальные переменные, обеспечив в основной процедуре сохранение их значений, очистку от значений и восстановление их значений перед любым возможным выходом из основной процедуры, как показано, в частности, в процедуре **ProcsAct** и в ряде других, рассмотренных в данной книге.

6.1.4. Определяющие параметры и описания Maple-процедур

Прежде всего, представим секцию описания (*description*), завершающую описательную часть определения процедуры и при ее наличии располагающуюся между секциями {*local, global, uses, options*} и непосредственно телом процедуры. При отсутствии этих секций *description*-секция располагается непосредственно за заголовком процедуры и кодируется в следующем формате, а именно:

description <Строчная конструкция> { : | ; }

Определенная в *description*-секции строчная конструкция не влияет на выполнение процедуры и используется в качестве комментирующей компоненты – она содержит текстовую информацию, предназначенную, вообще говоря, для документирования процедуры. Более того, в отличие от обычного комментария, который игнорируется

при чтении процедуры, описание ассоциируется с процедурой при ее выводе даже в случае, когда *ее* тело не выводится по причине использования рассматриваемой ниже опции *Copyright*. Более того, определяемый *description*-секцией комментарий может быть одного из типов {string, symbol}, как это иллюстрирует следующий фрагмент:

```
> RAEN := proc() description `Average`; sum(args[k], k= 1 .. nargs)/nargs end proc;
> RAEN(19.42, 19.47, 19.62, 63, 68, 14, 89, 20.10), eval(RAEN);
39.07625000, proc () description Average; sum(args[k], k= 1 .. nargs)/nargs end proc
> RAEN := proc() option Copyright; description "Average of real arguments";
sum(args[k], k= 1 .. nargs)/nargs end proc;
> eval(RAEN); ⇒ proc() description "Average of real arguments"; ... end proc
```

Данный фрагмент иллюстрирует результат использования *description*-секции *RAEN*-процедуры в случаях как отсутствия, так и наличия в ней дополнительно и *Copyright*-опции. В примерах приведенного фрагмента в *description*-секциях были использованы комментарии типов {string, symbol}. В связи со сказанным, механизм *description*-секций достаточно прозрачен и особых пояснений не требует. Между тем, следует отметить, подавляющее большинство пакетных процедур и процедур нашей Библиотеки [9,45] не содержат *description*-секций. Тогда как в ряде случаев информация из *description*-секций может оказаться достаточно полезной при проведении анализа процедур.

Рассмотрев секции *local*, *global* и *description*, мы несколько детальнее остановимся на секции {*options* | *option*}, которая должна кодироваться сразу же за двумя первыми (или быть первой при их отсутствии) в описательной части определения процедуры. Более того, в качестве параметров (*опций*) данной секции допускаются следующие: *operator*, *builtin*, *arrow*, *Copyright*, *trace*, *remember*, *autocompile*, *hfloat*, *inline*, *load=memberName*, *system*, *unload=memberName* и *call_external*. Приведенный перечень опций относится к *Maple* релиза 11 и, вообще говоря, зависит от используемого релиза пакета *Maple*.

Пакет располагает 3 типами процедур: встроенными непосредственно в ядро пакета, библиотечными и определяемыми самим пользователем. Параметр *builtin* определяет *встроенную* функцию пакета и при наличии он должен кодироваться *первым* в списке параметров *option*-секции. Этот параметр визуализируется при полном вычислении процедуры посредством *eval*-функции или при печати по *print*-функции, например

> print(eval), eval(map);	proc() option builtin; 169 end proc proc() option builtin; 209 end proc	# 8
> print(eval), eval(map);	proc() option builtin; 117 end proc proc() option builtin; 246 end proc	# 10
> print(eval), eval(map);	proc() option builtin = eval; end proc proc() option builtin = map; end proc	# 11
> [ParProc(eval)], [ParProc(map)]; [builtin function, 124], [builtin function, 253]		

Каждая встроенная функция идентифицируется уникальным номером (зависящим от номера релиза пакета); в то время как сам пользователь не имеет прямой возможности

определять собственные *встроенные* функции. В приведенном выше примере первым выводится результат вызова *print*-функции, тогда как вторым – *eval*-функции, из чего следует, что встроенные функции *eval* и *map* имеют соответственно номера 169 и 209 (*Maple 8*), в то время как для *Maple 10* и *Maple 11* эти номера будут 117, 246 и 124, 253, соответственно. Более того, из этого же фрагмента следует, что и вид возвращаемой и выводимой информации по встроенным функциям/процедурам также определяется текущим релизом пакета, что вовсе не добавляет рекламы его разработчикам.

Для проверки процедур могут быть полезны и *три* наши процедуры *ParProc*, *Sproc* и *ParProc1* [45], обеспечивающие возврат как основных параметров процедур, модулей и пакетов, так и их местоположение в библиотеках пакета *Maple* и пользовательских, как это иллюстрирует следующий достаточно простой фрагмент, а именно:

```
> ParProc(MkDir, ParProc(came)); map(ParProc, ['add', march, goto, iostatus, seq]); #11
      Arguments = (F::{string, symbol})
      locals = (cd, r, k, h, z, K, L, A, t, d, o, u, f, s, g, v)
      Arguments = (E::anything)
      locals = (f, h)
      globals = (_Art_Kr_)
[builtin function,121, iolib function,31, builtin function,226, iolib function, 3, builtin function, 132]
> ParProc(DIRAX); => DIRAX is module with exports, [new, replace, extract, empty, size, reverse,
insert, delete, sortd, printd, conv]
> ParProc(process); => process is module with exports, [popen, pclose, pipe, fork, exec, wait, kill,
block, launch]
> ParProc(Int); => inert_function
Warning, <Int> is inert version of procedure/function <int>
> ParProc1(ParProc1, 'h'), h;
Warning, procedure ParProc1 is in library [Proc, User, {"C:/Program Files/
Maple 11/LIB/UserLib"}]
      Arguments = (M::{module, procedure})
      locals = (a, b, c, d, p, h, t, z, cs, L, R, N, o, v)
      globals = (_62, ParProc, Sproc)
[Proc, User, {"C:/Program Files/Maple 11/LIB/UserLib" }]
> ParProc1(Sockets, 't'), t;
Warning, module Sockets is in library [package, Maple, {"C:\Program Files\Maple 11/lib"}]
[exports = (socketID, Open, Close, Peek, Read, Write, ReadLine, ReadBinary, WriteBinary, Server,
Accept, Serve, Address, ParseURL, LookupService, GetHostName, GetLocalHost, GetLocalPort,
GetPeerHost, GetPeerPort, GetProcessID, HostInfo, Status, Configure, _pexports)],
[locals = (defun, trampoline, soPath, solib, passign, setup, finalise)],
[options = (package, noimplicit, unload = finalise, load = setup)],
[description = ("package for connection oriented TCP/IP sockets")],
[package, Maple, {"C:\Program Files\Maple 11/lib"}]
> Sproc(MkDir, 'h'), h; => true, [Proc, User, {"C:/Program Files/Maple 11/LIB/UserLib"}]
> Sproc('type/package', 'h'), h; => true, [Proc, Maple&User, {"c:\Program Files\Maple 11/lib",
"C:/Program Files/Maple 11/LIB/UserLib"}]
```

С описанием приведенных процедур можно ознакомиться в [34,37,44], тогда как сами они включены в Библиотеку [45]. Там же можно получить и их исходные тексты.

Параметр *Copyright* определяет авторские права процедур, ограничивая в некоторой степени возможности вывода их определения на печать. В качестве такого параметра пакет полагает любую конструкцию *option*-секции, начинающуюся с *Copyright*-слова. Все библиотечные *Maple*-процедуры определены с параметром *Copyright*, требуя для обеспечения возможности вывода на печать определений процедур установки опции *verboseproc = n* ($n = \{2 | 3\}$) в *interface*-процедуре. Типичное содержимое *option*-секции в библиотечных процедурах пакета *Maple* имеет следующий общий вид, а именно:

option `Copyright (c) 1997 Waterloo Maple Inc. All rights reserved.`;

option {system} `Copyright (c) 1992 by the University of Waterloo. All rights reserved.`;

в зависимости от релиза пакета; при этом, каждый релиз пакета содержит процедуры и более ранних релизов. Текст любой пакетной процедуры (включая также их таблицы *remember*), кроме встроенных, можно получать по конструкции следующего вида:

interface(verboseproc = 3): {print | eval}(<Имя процедуры>);

как это весьма наглядно иллюстрирует следующий достаточно простой пример:

```
> interface(verboseproc = 3): eval(sin);
proc(x::algebraic)
local res, n, t, pull_out, keep_in, tmp;
option builtin = HFloat_sin, automap(1), `Copyright (c) 1992 by the University of
Waterloo. All rights reserved.`;
Тело процедуры
end proc
# [permanent]( $\pi/6$ ) = 1/2
# [permanent]( $\infty$ ) = undefined
# [permanent]( $\pi/4$ ) = 2^(1/2)/2
# [permanent]( $\pi$ ) = 0
# [permanent](I) = sinh(1)*I
# [permanent]( $-\infty$ ) = undefined
# [permanent]( $\pi/2$ ) = 1
# [permanent](0) = 0
# [permanent]( $\pi/3$ ) = 3^(1/2)/2
```

Такая возможность представляется нам весьма полезной не только для начинающего программиста в среде *Maple*-языка, но и даже довольно искушенному пользователю.

Отметим, что параметр *package* определяет принадлежность процедуры *внутреннему модулю* пакета. Тогда как по *trace*-параметру определяется режим трассировки вызова процедуры независимо от значения глобальной *printlevel*-переменной пакета, которая устанавливает режим вывода служебной информации на экран монитора.

Совместное использование 2 параметров *operator* и *arrow* информирует пакет о том, что процедура должна рассматриваться как функциональный оператор «->» во всех ее вызовах, а также при выводе ее определения на печать, например:

> $G := \text{proc}() \text{ option operator, arrow; evalf}(\sqrt{\sum_{k=1}^{\text{nargs}} \text{args}[k]^2}) \text{ end proc;}$

$$G := () \rightarrow \text{evalf} \left(\sqrt{\sum_{k=1}^{\text{nargs}} \text{args}_k^2} \right)$$

> $S := \text{proc}() \text{ local } k, p; \text{ option operator, arrow; for } k \text{ to nargs do } p := \text{args}[k];$
 if $\text{type}(p, \text{'numeric'})$ then next else $\text{print}(p, \text{whattype}(p))$ end if end do end proc:

Между тем, если определение процедуры использует параметры *operator* и *arrow*, но располагает секциями *global* и/или *local*, то такая процедура не возвращается в виде функционального оператора «->». Использование данных параметров предполагает также упрощение тела процедуры. С другой стороны, как иллюстрируют примеры G и S процедур, далеко не каждая процедура допускает представление в нотации «->»-функционального оператора. Данную возможность допускают, как правило, простые процедуры, реализованные однострочным экстракодом, что и иллюстрирует простая G -процедура первого примера предыдущего фрагмента.

Прежде, чем переходить к рассмотрению довольно важного *remember*-параметра, мы представим общие средства доступа к процедурным объектам *Maple*-языка. С целью идентификации процедурного типа *Proc*-объекта служат два тестирующих средства:

$\text{type}(P, \text{'procedure'})$ и $\text{whattype}(\text{eval}(P))$

возвращающие соответственно значения *true* и *procedure* при условии, если P -объект представляет собой процедуру *Maple*, определенную любым из трех рассмотренных выше способов, и, кроме того, определение которой является *вычисленным* в текущем сеансе. Следующий простой фрагмент иллюстрирует вышесказанное, а именно:

```
> O1 := proc() end proc: [type(O1, procedure), whattype(eval(O1))]; => [true, procedure]
> O2 := () -> sum(args[k], k=1..nargs): [type(O2, 'procedure'), whattype(eval(O2))];
                                     [true, procedure]
> define(O3, O3(x::anything, y) = x*y): [type(O3, 'procedure'), whattype(eval(O3))];
                                     [true, procedure]
> Type_Proc := proc(Id) `if` ((type(Id, 'symbol') = true) and (whattype(eval(Id)) =
`procedure`), true, false) end proc: map(Type_Proc, [O1, O2, O3]); => [true, true, true]
```

Например, последний пример фрагмента представляет очень простую тестирующую процедуру *Type_Proc*, возвращающую *true*-значение лишь в том случае, когда объект, приписанный *Id*-идентификатору, является *Maple*-процедурой. Более того, еще раз следует обратить внимание на то важное обстоятельство, что *корректное* применение тестирующей процедуры *whattype* предполагает полное вычисление процедурного объекта, что обеспечивается использованием *eval*-функции, рассмотренной выше.

Так как программная структура процедуры вычисляется по специальным табличным правилам, для полного доступа к ее элементам следует использовать *eval*-функцию, производящую полное вычисление процедуры, подобно тому, как она это делает для других структур, например, массивов. По *eval(Proc)*-вызову возвращается *определение* процедуры, приписанное *Proc*-переменной, в качестве которой возможно указывать любой допустимый идентификатор. В связи со сказанным по *whattype(Proc)*-вызову

возвращается *symbol*-значение, а по вызову *whattype(eval(Proc))* – *procedure*-значение. Поэтому по вызову *op(n, eval(Proc))* возвращаются значения 8 компонент определения процедуры, приписанного *Proc*-переменной. Ниже представлены все *n*-значения для указанной конструкции и их смысловая нагрузка, а именно:

По конструкции op(n, eval(Proc)) возвращается:

- 1 – последовательность *формальных* аргументов *Proc*-процедуры
- 2 – последовательность *локальных* переменных *Proc*-процедуры
- 3 – последовательность параметров (*опций*) *Proc*-процедуры
- 4 – содержимое *remember*-таблицы *Proc*-процедуры
- 5 – содержимое *description*-секции *Proc*-процедуры
- 6 – последовательность *глобальных* переменных *Proc*-процедуры
- 7 – *лексическая таблица*; данная таблица представляет собой внутреннюю структуру, которая используется для *записи* соответствия между *необъявленными* переменными и *локальными, глобальными* переменными или параметрами *среды* окружения самой процедуры. Данная таблица не соответствует никакой части процедуры
- 8 – тип возвращаемого процедурой результата (*если он был предварительно определен*)

В случае отсутствия в определении процедуры какой-либо из рассмотренных секций на соответствующем *n*-значении конструкция *op(n, eval(Proc))* возвращает значение *NULL*. По *nops(eval(Proc))*-вызову всегда возвращается значение 8 (*начиная с Maple 7*) – *максимально возможное* число составляющих компонент определения *Proc*-процедуры. Вместе с тем, в число указанных компонентов не входит само тело процедуры, в целях возможности доступа к нему используется прием, рассматриваемый несколько ниже. Следующий достаточно прозрачный фрагмент иллюстрирует вышесказанное:

```
> IAN := proc(x::float, y::integer, z::numeric) local k, h; global G,V,S; option `Copyright
  Tallinn Research Group * 30.11.10`, remember; description "G-average of arguments";
  V*evalf(sum(args[k]^G, k= 1 .. 3)^(1/G))/S end proc;
IAN:=proc(x::float, y::integer, z::numeric) description "G-average of arguments";... end proc
> G := 63: V := 68: S := 43: [IAN(19.95, 63, 20.10), IAN(0.1, 21, 1.14)];
                               [99.62790698, 33.20930233]
> seq(print(op(k, eval(IAN))), k = 1 .. 8);
                               x::float, y::integer, z::numeric
                               k, h
                               remember, Copyright Tallinn Research Group * 30.11.10
                               table([(19.95, 63, 20.10) = 99.62790698, (0.1, 21, 1.14) = 33.20930233])
                               "G-average of arguments"
                               G, V, S
> proc() local k; option `Copyright * 30.11.10`; sum(args[k], k=1..nargs) end proc,
  % (68, 63, 43, 48, 21, 14);  => proc() ... end proc, 257
```

В приведенном фрагменте определяется весьма простая *IAN*-процедура, содержащая все допустимые компоненты определения *Maple*-процедуры, затем производится ее вычисление, выводящее на экран только частичный текст определения процедуры, т. к. при ее определении был задействован *Copyright*-параметр *option*-секции. Однако, возвращаемое в результате вычисления определение по сути является полным и его

последующий вызов возвращает корректные результаты, как наглядно иллюстрирует последний пример фрагмента с непоименованной процедурой. После вычисления ее определения производится двукратный вызов процедуры; последующее *циклическое* применение конструкции *op(k, eval(IAN))* на базе встроенной *seq*-функции выводит содержимое всех восьми компонент *IAN*-процедуры.

Как следует из вышесказанного, тело процедуры *IAN* рассмотренными средствами не идентифицируется в качестве ее компоненты и доступ к нему возможен на базе иных средств, рассматриваемых несколько ниже. Здесь лишь уместно отметить *dismantle(P)* процедуру, обеспечивающую вывод *структуры данных*, определяемой *P*-выражением. Процедура *dismantle* выводит структуру *P*-выражения (*которое для случая процедуры должно быть полностью вычисленным*) в разрезе составляющих его компонент, их длины и относительных адресов в {десятичном | 16-ричном | 8-ричном} представлении. Простой фрагмент иллюстрирует вывод структуры данных, отвечающей *IAN*-процедуре:

```
> dismantl(eval(IAN));
PROC(10) #[remember, `Copyright\n  Tallinn Research Group * 30.08.11`]
EXPSEQ(4)
  DCOLON(3)
    NAME(4): x
    NAME(5): float #[protected]
  DCOLON(3)
    NAME(4): y
    NAME(5): integer #[protected]
  DCOLON(3)
    NAME(4): z
    NAME(5): numeric #[protected]
EXPSEQ(3)
  NAME(4): k
  NAME(4): h
EXPSEQ(3)
  NAME(6): remember
  NAME(15): `Copyright\n  Tallinn Research Group * 30.08.11`
EXPSEQ(1)
PROD(7)
  NAME(4): V
  INTPOS(2): 1
  FUNCTION(3)
    NAME(5): evalf #[protected]
  EXPSEQ(2)
    POWER(3)
      FUNCTION(3)
        NAME(4): sum #[protected, _syslib]
        EXPSEQ(3)
          POWER(3)
```

```

    TABLEREF(3)
    PARAM(2): [-1]
    EXPSEQ(2)
    LOCAL(2): [1]
    NAME(4): G
    EQUATION(3)
    LOCAL(2): [1]
    RANGE(3)
    INTPOS(2): 1
    INTPOS(2): 3
    PROD(3)
    NAME(4): G
    INTNEG(2): -1
    INTPOS(2): 1
    NAME(4): S
    INTNEG(2): -1
    EXPSEQ(2)
    STRING(9): "G-average of arguments"
    EXPSEQ(4)
    NAME(4): G
    NAME(4): V
    NAME(4): S
    EXPSEQ(1)
    BINARY(2)
    3

```

Данный фрагмент представляет внутреннюю структуру *IAN*-процедуры и на ее базе искушенный пользователь вполне может решать целый ряд весьма интересных задач. Однако, в нашу задачу рассмотрение данной проблематики не входит. Наконец, уже вполне можно вновь возвратиться к рассмотрению *remember*-параметра *option*-секции, предварительно *приведя* дополнительную полезную информацию. Набор из четырех функций *assemble*, *addressof*, *disassemble* и *pointto*, совместно с процедурой *dismantle* известен как «*хакерский*» пакет в *Maple*. Первые четыре функции обеспечивают доступ к внутренним представлениям объектов *Maple* и к указывающим на них адресам. Но перед использованием этого набора средств пользователь должен быть *хорошо* знаком с внутренним представлением объектов *Maple*. Для этого рекомендуется обратиться к соответствующей литературе и фирменным руководствам по пакету.

Для целого ряда типов процедур (и в первую очередь для рекурсивных), характеризующихся многократными вызовами на одних и тех же наборах фактических аргументов, очень важную задачу приобретает вопрос повышения эффективности их выполнения. Эта задача решается на основе сохранения истории вызовов процедуры в текущем сеансе в специальной *remember*-таблице. Применение такого механизма требует некоторых пространственных издержек, которые в ряде случаев могут быть катастрофическими, однако позволяют получать существенный *временной* выигрыш при различного рода

циклических вычислениях на одинаковых значениях фактических аргументов. Наш опыт подтверждает достаточно высокую эффективность данного механизма.

А именно, по *remember*-параметру с *Proc*-процедурой будет ассоциирована таблица *remember*, которую можно получать по рассмотренной конструкции *op(4, eval(Proc))*. Данная таблица аналогична обычной *table*-структуре, допуская те же самые средства обработки, что и последняя. Эта таблица содержит все вызовы процедуры, включая и рекурсивные, в разрезе передаваемых процедуре значений фактических аргументов и возвращаемых на них процедурой значений (*результатов вызовов*). Данная таблица для произвольной *Proc*-процедуры имеет следующий простой вид, из которого при необходимости несложно извлекать ранее полученные значения вызовов процедуры:

> *op(4, eval(Proc));*

```
table([( <Фактические аргументы_1>) = <Возвращаемое значение_1>
=====
(<Фактические аргументы_h>) = <Возвращаемое значение_h>])
```

Таким образом, *remember*-таблица в качестве входов использует последовательности фактических аргументов каждого ее вызова, а выходов – значения, возвращаемые на соответствующих фактических аргументах. Такая организация обеспечивает весьма эффективную работу с часто используемыми либо рекурсивными процедурами, ибо при вызове процедуры в случае установления наличия в ее *remember*-таблице вызова, аналогичного текущему, сразу же производится возврат соответствующего результата без повторного выполнения тела процедуры, т.е. входы в таблицу не дублируются. В главной библиотеке пакета и в нашей Библиотеке [45] целый ряд средств определены с *remember*-параметром, например, функции *diff*, *readlib*, процедура *ParProc* и др.

Однако, использование *remember*-механизма наряду с повышением эффективности выполнения процедуры может очень существенно использовать ресурсы оперативной памяти ЭВМ, требуемые для размещения *remember*-таблицы. В этом случае требуется нахождение оптимального консенсуса, обеспечиваемого, в частности, возможностью как полного обнуления таблицы, так и отдельных ее входов. В частности, совместное применение с *remember*-параметром *system*-параметра обеспечивает автоматическую очистку (без удаления) *remember*-таблицы в процессе выполнения ядром *периодических* операций по очистке от «мусора» памяти ПК, занимаемой текущим сеансом. Именно поэтому не рекомендуется использовать *system*-параметр для тех процедур, текущий результат выполнения которых зависит от истории вычислений, как это имеет место для рекурсивных процедур. Ряд других средств *Maple*-языка позволяют обеспечивать модификацию *remember*-таблицы более дифференцированно.

Для обнуления *remember*-таблицы для *Proc*-процедуры применяется вызов процедуры *forget(Proc)*. Вызов *forget(Proc{, A}{, 0})* позволяет удалять из *remember*-таблицы входы для определяемой первым фактическим аргументом *Proc*-процедуры. При этом, для *Proc*-процедуры допускается передавать фактические *A*-аргументы и использовать 2 необязательные *O*-опции (*reinitialize*, *subfunction*), управляющие непосредственным выполнением *forget*-процедуры. Детальнее с ними возможно ознакомиться в справке по пакету. Посредством передачи *forget* фактических *A*-аргументов обеспечивается

возможность удаления из *remember*-таблицы указанной *Proc*-процедуры ее конкретных *Proc(A)*-вызовов. Тогда как в случае отсутствия фактических *A*-аргументов *remember*-таблица полностью *обнуляется*, оставаясь, между тем, ассоциированной с процедурой *Proc*. Наконец, посредством подстановки *subsop(4 = NULL, eval(Proc))* производится удаление таблицы *remember* для *Proc*-процедуры. Однако, *forget*-процедуру следует по ряду причин использовать с определенной осмотрительностью [10-14,30-34,44].

Так, *forget*-процедуру не рекомендуется использовать для расширенного управления *remember*-таблицей, т.к. процедура, например, не работает с целым рядом *модульных* процедур пакета, а также со многими процедурами, определенными в модулях *Share*. Ряд достаточно простых примеров следующего фрагмента иллюстрирует сказанное. Для модификации *процедурной* таблицы можно использовать также нашу процедуру *Remember_T* от неопределенного числа формальных аргументов, которая выполняет редактирование входов и выходов *remember*-таблицы. Следующий простой фрагмент иллюстрирует исходный код процедуры *Remember_T* [45] и примеры ее применения:

```
> Remember_T := proc(NP::{name, symbol}) local k, p, h, G, F, S; global __T;
  if not type(NP, procedure) then error "<%1> is not a procedure", NP
  elif nargs < 2 then return op(4, eval(NP))
  else assign('__T' = op(4, eval(NP)), F = cat([libname][1][1 .. 2], "\$$$vgs"))
  end if;
  for k from 2 to nargs do assign('G' = "", 'S' = "");
    if whattype(args[k]) = '=' then __T[op(lhs(args[k]))] := rhs(args[k]) else
      for p to nops(args[k]) - 1 do G := cat(G, convert(args[k][p], 'string'), ",") end do;
      G := cat("__T[", cat(G, convert(args[k][nops(args[k])], 'string'), "];");
      assign('S' = convert(__T[args[k]], 'string')), assign('S' = cat(S[1 .. 4], S[6 .. -2]));
      assign('G' = cat(cat(S, "!="), G)), writebytes(F, convert(G, 'bytes')), close(F);
      read F;
      __T[seq(args[k][p], p = 1 .. nops(args[k]))] := %, fremove(F)
    end if
  end do;
  unassign('__T'), op(4, eval(NP))
end proc;

> P3 := proc() options operator, arrow, remember; evalf(sum(args[k], k=1..nargs)/nargs)
end proc: [P3(14,21,43,48,63,68),P3(96,89,67,62,47,42),P3(30,3,99,420,72,6)]: op(4,eval(P3));
table([(96, 89, 67, 62, 47, 42)=67.1666, (14, 21, 43, 48, 63, 68)=42.8333, (30, 3, 99, 420, 72, 6)=105.])

> Remember_T(P3, [14,21,43,48,63,68] = 2010, [96,89,67,62,47,42], [31, 3, 99, 42] = [14, 21]):
op(4, eval(P3));
table([(30, 3, 99, 420, 72, 6) = 105., (14, 21, 43, 48, 63, 68) = 2010, (31, 3, 99, 42) = [14, 21]])
```

В приведенном фрагменте предварительно определяется *Remember_T*-процедура от неопределенного числа формальных аргументов, из которых первым фактическим аргументом *должен* выступать идентификатор процедуры, *remember*-таблица которой модифицируется. Остальные фактические аргументы кодируются в формате:

$[x_1, x_2, \dots, x_n] = \langle \text{Значение} \rangle$ либо $[x_1, x_2, \dots, x_n]$

где первый формат определяет необходимость замены выхода на (x_1, x_2, \dots, x_n) -входе таблицы *remember* на заданное значение либо помещение в таблицу *remember* нового входа, если указанный отсутствует, или удаления заданного входа из таблицы. Затем определяется P3-процедура с опцией *remember*, согласно которой для процедуры P3 создается *remember*-таблица. После 3 вызовов процедуры содержимое этой таблицы выводится. Последующий вызов *Remember_T*-процедуры иллюстрирует результат по модификации *remember*-таблицы процедуры P3 в разрезе вышеперечисленных трех операций. Организация процедуры *Remember_T* здесь не рассматривается, оставаясь читателю в качестве достаточно полезного практического упражнения.

В ряде случаев *Remember_T*-процедура оказывается достаточно полезным средством, прежде всего, при необходимости более эффективного использования памяти ЭВМ в работе с рекурсивными и часто используемыми процедурами в циклических структурах. Пример в нижеследующем фрагменте иллюстрирует строго линейную зависимость количества входов в таблицу *remember* при вызове процедуры *Proc_30* в циклической конструкции, однако такая зависимость может носить также и нелинейный характер, весьма существенно увеличивая размер *remember*-таблицы и требуемое под нее место в оперативной памяти. В связи с этим при работе с рекурсивными процедурами либо при наличии вызовов процедур в теле циклических конструкций рекомендуется все же оценивать целесообразность использования как *remember*-таблицы, так и режима ее возможной последующей модификации.

```
> Proc := proc() options package; evalf(sum(args[k], k= 1 .. nargs)/nargs) end proc:
> Proc(14, 21, 43, 48, 63, 68);  => 42.83333333
> Proc1 := proc() options trace; evalf(sum(args[k], k= 1 .. nargs)/nargs) end proc:
> Proc(14, 21, 43, 48, 63, 68);
    {--> enter Proc1, args = 14, 21, 43, 48, 63, 68
      42.83333333
    <-- exit Proc1 (now at top level) = 42.83333333}
    42.83333333
> Proc2:=proc() options operator, arrow; evalf(sum(args[k], k= 1 .. nargs)/nargs) end proc:
> [eval(Proc2), Proc2(14, 21, 43, 48, 63, 68)];
    [ ( ) -> evalf(  $\frac{\sum_{k=1}^{\text{nargs}} \text{args}_k}{\text{nargs}}$  ), 42.83333333 ]
> Proc3:= proc() options operator, arrow, remember; evalf(sum(args[k],k=1..nargs)/nargs)
  end proc: [Proc3(14, 21, 43, 48, 63, 68), Proc3(96, 89, 67, 62, 47, 42), Proc3(72, 6, 99, 420, 492)]:
  op(4, eval(Proc3));
table([(72,6,99,420,492) = 217.800, (96,89,67,62,47,42) = 67.1667, (14,21,43,48,63,68) = 42.8333])
> forget(Proc3, 14, 21, 43, 48, 63, 68): op(4, eval(Proc3));
  table([(72, 6, 99, 420, 492) = 217.800, (96, 89, 67, 62, 47, 42) = 67.1667])
> Proc_30 := proc() options remember; sum(args[n], n= 1 .. nargs) end proc:
> h := 0: for k to 420 do h:= h + Proc_30(a$a = 1 .. k) end do: h;  => 12436340
```

```
> nops([indices(op(4, eval(Proc_30)))]); ⇒ 420
> Fib := proc(n::integer) if n = 0 or n = 1 then n else Fib(n - 1) + Fib(n - 2) end if end proc:
> Fib1 := proc(n::integer) option remember; if n = 0 or n = 1 then n else Fib1(n - 1) +
    Fib1(n - 2) end if end proc: T := time(): [Fib(38), time() - T]; ⇒ [39088169, 169.154]
> T := time(): [Fib1(38), time() - T], nops([indices(op(4, eval(Fib1)))]); ⇒ [39088169, 0.], 39
```

Между тем, наиболее важной особенностью *remember*-параметра является возможность на основе *remember*-таблицы определять весьма эффективные во *временном* отношении рекурсивные процедуры. Итак, обеспечивая возможность определения рекурсивных процедур, механизм ядра, вместе с тем, определяет эффективность их выполнения в зависимости от использования таблицы *remember*. Так, в конце последнего фрагмента приведены две простые функционально эквивалентные рекурсивные процедуры *Fib* и *Fib1*, вычисляющие числа *Фибоначчи*; при этом, описательная часть *Fib1*-процедуры включает *remember*-параметр. Временные результаты вызова обоих процедур говорят сами за себя. По экспертным оценкам *ассоциирование* с процедурами *remember*-таблиц позволяет на целом ряде задач получать экспоненциальный временной выигрыш за счет определенных ресурсов памяти в виде требуемого места под *remember*-таблицу. В частности, для нашего примера *remember*-таблица *Fib1*-процедуры содержит лишь *39 входов*, что не соизмеримо с полученным *временным* выигрышем. Однако возможно и наоборот, например, при использовании процедур в циклических конструкциях (*в качестве примера можно рассмотреть Proc_30-процедуру последнего фрагмента*).

Для модификации *remember*-таблицы произвольной *Pr*-процедуры, чье определение содержало *remember*-параметр, наряду с приведенной выше *Remember_T*-процедурой можно использовать конструкции следующего весьма простого вида, а именно:

$$Pr(<\text{Фактические аргументы } (\Phi A)>) := <\text{Требуемый результат}> \{ : | ; \}$$

$$T := op(4, eval(Pr)): \quad T[(\langle \Phi A \rangle)] := evaln(T[(\langle \Phi A \rangle)]) \{ : | ; \}$$

По первой конструкции производится добавление в *remember*-таблицу *Pr*-процедуры входа, соответствующего указанным фактическим аргументам (*ΦА*), которому будет соответствовать *выход (требуемый результат)*, возвращаемый процедурой на заданных фактических аргументах. По второй конструкции производится *удаление* из таблицы *remember* процедуры *Pr* *входа*, соответствующего заданным фактическим аргументам. Следующий достаточно прозрачный фрагмент иллюстрирует вышесказанное:

```
> P := proc() local k; options remember; sum(args[k], k= 1 .. nargs) end proc: # (1)
> [P(14, 21, 43, 48, 63, 68), P(96, 89, 67, 62, 47, 42), P(14, 6, 98, 420, 2010)]: op(4, eval(P));
    table([(14, 21, 43, 48, 63, 68) = 257, (96, 89, 67, 62, 47, 42) = 403, (14, 6, 98, 420, 2010) = 2548])
> P(42, 47, 67, 89, 96):= 2010: P(56, 51, 31, 2, 10):= 2500: op(4, eval(P)); # (2)
    table([(14, 21, 43, 48, 63, 68) = 257, (96, 89, 67, 62, 47, 42) = 403, (14, 6, 98, 420, 2010) = 2548,
    (56, 51, 31, 2, 10) = 2500, (42, 47, 67, 89, 96) = 2010])
> T:=op(4, eval(P)): T[(42, 47, 67, 89, 96)]:= evaln(T[(42, 47, 67, 89, 96)]): op(4, eval(P)); # (3)
    table([(14, 21, 43, 48, 63, 68) = 257, (96, 89, 67, 62, 47, 42) = 403, (14, 6, 98, 420, 2010) = 2548,
    (56, 51, 31, 2, 10) = 2500])
```

В первом примере фрагмента определяется *P*-процедура с *remember*-параметром, но

затем производится ее вычисление и после *трех* вызовов процедуры *P* возвращается содержимое ассоциированной с ней *remember*-таблицы. Во втором примере на основе присвоения данная таблица расширяется на *два* новых входа и возвращается ее новое состояние. Наконец, в третьем примере производится удаление из *remember*-таблицы ее конкретного входа с возвратом нового состояния таблицы.

Следует отметить, *remember*-механизм может быть очень успешно использован также для решения других важных задач, например, имеющих дело с функциями, которые имеют особенности (точки разрыва, сингулярные точки). Идея здесь состоит в том, чтобы использовать приоритетность поиска возвращаемого результата сначала в *remember*-таблице (если она имеется) и только затем реального выполнения тела процедуры при его отсутствии. С этой целью выражение с особенностями оформляется процедурой с параметром *remember* в ее секции *option*, процедура вычисляется и сразу же способом, описанным выше, производится дополнение в ее *remember*-таблицу входов, которые соответствуют особым точкам выражения, как это иллюстрирует следующий пример:

```
> G:=proc(x) evalf(1/x,3) end proc: G(0):=infinity: map(G, [14,0,21]); => [0.0714, ∞, 0.0476]
> op(4, eval(G)); => table([0 = ∞])
```

Этот фрагмент иллюстрирует еще один важный момент, связанный с процедурами, а именно. С каждой процедурой независимо от присутствия в ее *определении* параметра *remember* ассоциируется *remember*-таблица, сохраняющая историю всех *присвоений* по конструкциям *Pr(ΦA) := <Значение>* и *assign('Pr(ΦA)', <Значение>)*, где *Pr* – процедура, определенная любым допустимым способом, и *ΦA* – ее фактические аргументы, на которых процедура принимает *указанное* значение. Таким образом, *remember*-таблица для *Proc*-процедуры может создаваться следующими двумя способами, а именно:

- (1) *Proc := proc(...) option remember; ... end proc { : | ; }*
- (2) *Proc := proc(...) ... end proc: Proc(ΦA) := <Значение> { : | ; }*

В обоих случаях с *Proc*-процедурой *Maple* ассоциирует *remember*-таблицу, однако по разному организует работу с ней. В *первом* случае *remember*-таблица будет содержать лишь определенные по вышеуказанным конструкциям входы и вовсе не обновляться вызовами *Proc*-процедуры, тогда как во *втором* случае каждый вызов *Proc*-процедуры соответствующим образом *модифицирует* таблицу, сохраняя историю всех *уникальных* вызовов процедуры. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> G := proc(x) x end proc: [assign('G(42)', 68), assign('G(47)', 63), assign('G(67)', 43)]:
> G1 := x -> x: [assign('G1(42)', 68), assign('G1(47)', 63), assign('G1(67)', 43)]:
> define(G2,G2(x::numeric)=x):[assign('G2(42)',68),assign('G2(47)',63),assign('G2(67)',43)]:
  op(4, eval(G)), op(4, eval(G1)), op(4, eval(G2));
table([67 = 43, 42 = 68, 47 = 63]),table([67 = 43, 42 = 68, 47 = 63]),table([67 = 43, 42 = 68, 47 = 63])
> P := proc(x) option remember; x^10 end proc: for k to 3 do P(k) end do:
> P1 := proc(x) x^10 end proc: P1(42) := 68: P1(47) := 63: for k to 5 do P1(k) end do:
  op(4, eval(P)), op(4, eval(P1)); => table([1 = 1, 2 = 1024, 3 = 59049]), table([42 = 68, 47 = 63])
```

Первые *три* примера фрагмента иллюстрируют *нестандартной* прием ассоциирования с процедурами всех допустимых типов *remember*-таблицы, в то время как последние 2

примера поясняют принципиальные различия *remember*-таблиц процедур, созданных как на основе *remember*-параметра, так и нестандартно. Из приведенного фрагмента следует, что появляется простая возможность наделять *remember*-механизмом (пусть в усеченном варианте) любой тип процедуры, включая и функции. Это тем более важно, что в ряде случаев простая *remember*-таблица даже на несколько входов может весьма существенно повышать эффективность выполнения процедуры, с которой она будет ассоциирована. В [14,30-34] можно найти ряд примеров применения такого подхода.

6.1.5. Механизмы возврата процедурой результатов ее вызова

В результате вызова стандартным является возврат результата *выполнения* процедуры через значение последнего предложения ее тела. Однако, *Maple*-язык поддерживает еще 3 основных механизма возврата результата вызова процедуры: через фактический *evaln*-аргумент, встроенные функции **RETURN** (*return-предложение*) и **ERROR** (*error-предложение*). При возникновении ошибочной ситуации при выполнении процедуры производится аварийный выход из процедуры с возвратом соответствующего *abend*-сообщения. В случае отсутствия ошибочных ситуаций предварительно вычисленная процедура с *Proc*-именем (идентификатор, которому присвоено определение процедуры) вызывается подобно функции по конструкции следующего формата кодирования:

Proc(*<Фактические аргументы (ФА)>*)

возвращая на полученных фактических аргументах (*ФА*) соответствующее значение, которое определяется одним из вышеуказанных способов. Из указанных механизмов возврата результатов вызовов наиболее часто используемым является использование функции **RETURN** либо *return-предложения* следующих форматов кодирования:

RETURN(*<Последовательность выражений>*)

return *<Последовательность выражений>* {*:* | *;*}

которые вызывают *немедленный* выход из процедуры с возвратом последовательности значений выражений. Следующий простой фрагмент иллюстрирует использование стандартного механизма наряду с **RETURN**-функцией и с *return-предложением* для организации возврата результата выполнения процедуры, а именно:

```
> AG := proc(x::integer, y::float,R) `if`(x*y > R, x, y) end proc: AG(68, 72.5, 420); => 68
> SV := proc(x, L::list) member(x, {op(L)}) end proc: SV(14, [68, 63, 14, 43, 21]); => true
> MV := proc(x, L::list) local k, H; assign(H=[]), seq(`if`(x=L[k], assign('H'=[op(H), k]),
  NULL), k=1..nops(L)); H end proc: MV(95, [57, 95, 52, 95, 3, 98, 32, 14, 95, 43, 95, 34, 23, 95]);
  [2, 4, 9, 11, 14]
> MM:=(x,L::list) -> op([seq(`if`(x=L[k], RETURN([k,{L[k]}]), NULL), k=1..nops(L)),false]):
> k:=68: MM(2010, [68, 63, 3, 2012, 32, 14, 1942, 37, 420, 23, 2.2214, 2010]), k; => [12, {2010}], 12
> MM(1942, [68, 63, 3, 2012, 32, 14, 1942, 37, 420, 23, 2.2214, 2010]); => [7, {1942}]
> TP := proc() local m, n, k; (m, n) &ma 0; for k to nargs do `if`(type(args[k], 'numeric'),
  assign('m', m+1), assign('n', n + 1)) end do; return [ `Аргументы:`, [m, `Числовые`,
  [n, `Нечисловые`]]; end proc: TP(68, 63., 14/21, "RANS", Pi, TRG);
```

[Аргументы, [3, Числовые], [3, Нечисловые]]
 > W := (x, y) -> proc() local g,v,h; g,v:=63,68; h:=g*args[1]+v*args[2]; if h > 420 then return
 x*y end if; h end proc(x, y): W(42, 47), W(3, 2); \Rightarrow 1974, 325
 > VS:=proc() local k, T; assign(T=array(1..nargs+1, 1..2)), assign(T[1,1]=Argument,T[1,2]=
 Type); for k to nargs do T[k+1, 1]:= args[k]; T[k+1, 2]:= whattype(args[k]) end do:
 return eval(T) end proc: VS(RANS, "IAN", 2010, F(x), n..p, array(1..3, [14, 21, 43]));

Argument	Type
RANS	symbol
"IAN"	string
2010	integer
F (x)	function
n .. p	..
[14 , 21 , 43]	array

Первые *три* примера фрагмента представляют простые процедуры *AG*, *SV* и *MV*, из которых назначение первой легко усматривается из ее определения, тогда как другие *две* возвращают соответственно результат тестирования и список номеров позиций по вхождению *x*-элемента в заданный *L*-список. Все три данные процедуры возвращают результат стандартным способом через *последнее* предложение тела процедуры. Тогда как остальные примеры фрагмента иллюстрируют возврат результата процедуры по *RETURN*-функции либо по *return*-предложению *Maple*-языка.

Процедура *MM* возвращает список с номером позиции *первого* вхождения *x*-элемента в *L*-список и сам элемент, в противном случае возвращается *false*-значение. Данная процедура для возврата результата своего вызова использует как стандартный метод, так и *RETURN*-функцию. При этом, рекомендуется обратить внимание на *реализацию* процедуры однострочным экстракодом. Процедура *TP* возвращает результат анализа получаемых при ее вызове фактических аргументов в разрезе числовых и нечисловых с идентификацией числа аргументов обоих типов. Процедура *W* реализована в виде *однострочного* экстракода, используя для возврата результата как *return*-предложение, так и последнее предложение. Наконец, процедура *VS* возвращает таблицу, первый столбец которой содержит передаваемые процедуре фактические аргументы, тогда как второй – их типы. Отметим здесь, одним из основных достоинств использования функции *RETURN* является возможность эффективного возврата значений *локальных* переменных из целого ряда довольно важных вычислительных конструкций, а также обеспечение гибкой системы избирательности возврата результатов вызова процедур пользователя. Тогда как предложение *return* имеет значительно менее ограниченные выразительные возможности по представлению вычислительных алгоритмов в среде *Maple*-языка. Следует отметить, что, начиная с *Maple 6*, разработчики объявили, что *RETURN*-функция является устаревшим средством (*obsolete*) и сохранена только для обеспечения обратной совместимости процедур, разработанных в младших релизах пакета. При этом, настоятельно рекомендую использование *return*-предложения. Наш опыт работы с пакетом *Maple* различных релизов, между тем, говорит совершенно об обратном. Использование функции *RETURN* во многих случаях предпочтительнее и

позволяет создавать эффективные выходы из вычислительных конструкций, прежде всего, в однострочных *экстракодах*. Ниже на данном моменте акцентируем несколько большее внимание, подтверждая ту истину, что старое далеко не всегда хуже нового.

В общем случае функция **RETURN(V1, ..., Vn)**, где в качестве фактических аргументов могут использоваться произвольные *Maple-выражения*, предварительно вычисляемые, может не только возвращать конкретные результаты *вычисления V_k-выражений (k=1..n)*, связанных со спецификой реализуемого процедурой алгоритма, но в определенных условиях и возвращать вызов процедуры *невывычисленным*. Для этих целей, как правило, используется конструкция вида **RETURN('Proc(args)')**, где *Proc* – имя (*идентификатор*) процедуры. Между тем, библиотечные процедуры *Maple* в случае некорректного или прерванного вызова возвращают *FAIL*-значение, если нецелесообразно возвращать вызов процедуры невычисленным. Следующий фрагмент иллюстрирует сказанное:

```
> LO := proc() local k; for k to nargs do if whattype(args[k]) <> float then return
  'LO(args)' end if end do end proc: LO(68, 63, 43.67, 19.81, 19.83, 14, 21, G, S, Art, Kr);
  LO(68, 63, 43.67, 19.81, 19.83, 14, 21, G, S, Art, Kr)

> [sqrt(15), sin(22), ln(gamma), exp(Pi)]; => [√15, sin(22), ln(γ), eπ]
```

В этом примере *LO*-процедура при обнаружении среди переданных ей фактических аргументов выражения типа, отличного от *float*, осуществляет выход по предложению **return** с возвращением своего вызова невычисленным. Второй пример иллюстрирует возврат невычисленных вызовов базовых функций *Maple*-языка пакета.

В целом ряде случаев возвращать результат вызова процедуры представляется весьма удобным через ее формальный *h*-аргумент, кодируемый в *заголовке* процедуры среди последовательности ее формальных аргументов в виде *h{::evaln}*. Соответствующие же им фактические аргументы кодируются в виде *{'p' | p}*, т.е. процедуре сообщается, что ей передается не значение соответствующего ему *фактического* аргумента, а лишь его идентификатор. В этом случае в теле процедуры производится присвоение данному фактическому аргументу-идентификатору требуемого значения, которое доступно сразу же после вызова процедуры. Следующий фрагмент иллюстрирует сказанное:

```
> G := proc(L::list(numeric), x::evaln, y::evaln, z::evaln) local a, b, c;
  `&ma`(a, b, c, [], seq(`if` (type(k, integer), assign('a' = [op(a), k]), `if` (type(k, fraction),
  assign('b' = [op(b), k]), assign('c' = [op(c), k]))), k = L), assign(x = a, y = b, z = c),
  evalf(sqrt(`+` (op(L))))
end proc:

> G1 := proc(L::list, x::evaln, y::evaln, z::evaln) local a, b, c, d;
  `&ma`(a, b, c, d, [], seq(`if` (type(k, integer), assign('a' = [op(a), k]), `if` (type(k, fraction),
  assign('b' = [op(b), k]), `if` (type(k, float), assign('c' = [op(c), k]),
  assign('d' = {whattype(eval(k)), op(d)}))), k = L),
  assign(x = a, y = b, z = c, `if` (4 < nargs, assign(args[5] = d), NULL)), evalf(sqrt(`+` (op(L))))
end proc:

> G([2010, 68, 14/21, 19.42, 63/1947, 420, 72.5, 11.50], a, v, z), a, v, z;
  51.01096964, [2010, 68, 420], [2/3, 21/649], [19.42, 72.5, 11.50]
```



```
> G1([2010, 68, 14/21, 19.42, 63/1947, 420, 72.5, 11.50, 10+17*I, 63-47*I], a, v, z, 'h'), a, v, z, h;
51.7224 - 0.29 I, [2010, 68, 420], [2/3, 21/649], [19.42, 72.5, 11.50], {complex(extended_numeric)}
```

Первый пример фрагмента представляет *G*-процедуру, которая допускает при своем вызове четыре фактических аргумента, из которых три последних имеют *evaln*-тип и через которые передаются списки фактических аргументов, имеющих тип *float*, *integer* и *fraction* соответственно. Тогда как в качестве основного возврата процедуры является результат вычисления корня квадратного из суммы фактических аргументов. Пример вызова процедуры *G* иллюстрирует сказанное. Тогда как второй пример фрагмента представляет *G1*-процедуру, являющуюся модификацией предыдущей процедуры и допускающей при вызове более четырех фактических аргумента, из которых четыре первых аналогичны случаю процедуры *G*, в то время как через пятый необязательный аргумент передается множество типов фактических аргументов процедуры, которые отличны от типов {*float*, *integer*, *fraction*}. При этом, основной возврат процедуры будет аналогичным случаю *G*-процедуры. Вызов процедуры *G1* иллюстрирует сказанное. В обоих процедурах рекомендуется обратить внимание на использование процедуры/оператора *&ma* [41–45], обеспечивающего присвоение того же самого выражения для последовательности переменных, и *assign*-процедуры для присвоения значений для фактических аргументов, через которые производятся вторичные возвраты вызовов.

Однако, возвращать вторичные результаты вычислений процедуры возможно и через их фактические аргументы, не определяемые *evaln*-типом и имеющие *symbol*-тип; в таком случае их рекомендуется кодировать в невычисленном формате, т.е. в прямых верхних кавычках `{}`. Следующий простой фрагмент процедурой *Kris* иллюстрирует данный подход, а именно:

```
> Kris := proc(x, y, z) local a, b, k;
  b := [args];
  for k in b do
    if type(k, 'assignable') then a := k; assign(a = map(whattype, map(eval, b)));
      WARNING("2nd result had been returned via <%1>", a); return b
    end if
  end do;
  b
end proc;
> Kris(63, 68.42, z, [2, 12, 2010], x);
Warning, 2nd result had been returned via <z>
[63, 68.42, z, [2, 12, 2010], x]
> z; ⇒ [integer, float, symbol, list, symbol]
> Kris(63, 68.42, 43, [2, 12, 2010]); ⇒ [63, 68.42, 43, [2, 12, 2010]]
```

Процедура *Kris* при обнаружении в точке вызова среди фактических аргументы типа *symbol* через первый такой аргумент возвращает вторичный результат наряду с выводом соответствующего сообщения и возвратом основного результата. Тогда как, если оный отсутствует, то возвращается только основной результат вызова процедуры.

Так как пакет *Maple* вычисляет формальные аргументы единожды, то их не следует в теле процедуры использовать аналогично *локальным* переменным. Для формального аргумента, через который будет возвращаться результат вызова процедуры, в ее теле должно быть выполнено только одно присвоение, которое определяет возвращаемый результат, отличный от основного. При этом, в целом ряде случаев процедура может вполне обходиться и без основного возврата, ограничиваясь только вторичными, а то даже и вовсе без них. В обоих случаях вызов данной процедуры возвращает значение *NULL*, т.е. ничего. Механизм возврата результатов вызова *Maple*-процедуры через ее фактические аргументы дает возможность наряду со стандартным подходом либо на основе *RETURN*-функции организовывать специальные множества дополнительных (*вторичных*) возвратов, определяемых наличием/отсутствием при вызове процедуры соответствующих аргументов либо выполнением определенных условий. Подобная организация возврата результатов процедуры достаточно широко практикуется для библиотечных процедур как собственно самого пакета *Maple*, так и Библиотеки [45]. Наряду с представленными выше механизмами возврата результатов *вызова* процедур можно предложить еще один механизм, полезный в целом ряде приложений и в ряде процедур Библиотеки [45] применяемый неоднократно. Суть состоит в следующем.

Результаты вызова процедуры возвращаются через ее ключевые аргументы, которые представляют собой некоторые идентификаторы. Данные переменные относительно процедуры выступают на уровне глобальных переменных и их определение делается в теле процедуры, *реализуясь* лишь при указании такого аргумента при вызове. Схема такой процедуры *Proc* может быть представлена следующим фрагментом, а именно:

```
> Proc := proc({Id1} {, Id2} ... {, Idp}) local t;
    if member('Id1', [args], 't') then assign(args[t] = expr1) end if;
    =====
    if (member('Idp', [args], 't'), assign(args[t] = exprp), NULL)
end proc;
```

где *expr_k* – *Maple*-выражение (*k=1..p*). При этом, следует иметь в виду, здесь требуется использование *assign*-процедуры, а не оператора присваивания «:=», т.е. *assign(args[t] = expr)*, а не *args[t]:=expr*. В противном случае выводится предупреждение о недопустимом использовании переменной *args* в качестве *локальной* переменной с инициированием ошибочной ситуации с диагностикой «*Error, args cannot be declared as a local*». Это еще один пример принципиальных отличий оператора «:=» от *assign*-процедуры.

Таким образом, необязательное ключевое слово *Idk* является глобальной переменной, получающей значение только при вызове процедуры *Proc(..., Idk, ...)*, т.е. процедура через него возвращает выражение, присвоенное в теле процедуры. При этом, наряду с описанным в качестве механизмов возврата результатов *вызова* процедуры возможно использовать и другие, описанные выше. Более того, наряду с *ключевыми аргументами* в качестве формальных аргументов процедуры могут использоваться и другие типы.

Данная организация механизма возврата результатов *вызова* процедуры в целом ряде случаев оказывается довольно эффективной, позволяя: (1) легко варьировать возврат требуемого кортежа *глобальных* переменных, и (2) достаточно легко расширять набор

функций, поддерживаемых процедурой, при относительно простой модификации ее исходного текста. В подкаталоге *Mws6789* файловой системы нашей Библиотеки [45] можно найти целый ряд интересных примеров процедур, использующих механизмы возврата результатов процедуры, как стандартные, так и полезные нестандартные.

Наконец, достаточно часто в качестве как *основного*, так и дополнительного возвратов процедур используют *глобальные* переменные или *пользовательские*, или *пакетные*. Как правило, глобальные пакетные переменные начинаются префиксом «_», в частности, *_SolutionsMayBeLost* и *_EnvAllSolutions*, ассоциированные с процедурой *solve Maple*, или *глобальная* переменная *_warning*, ассоциированная с *процедурой WARNING* нашей Библиотеки [45], которая возвращает текст выводимого сообщения для возможности его последующей программной обработки. Между тем, такие *глобальные* переменные могут быть и без указанного *префикса* «_», например, предопределенные переменные пакета *lasterror*, *libname* и др., а также ряд переменных из нашей Библиотеки [45].

Как уже отмечалось выше, определение процедуры допускает следующий формат:

Proc := proc(args)::type; ... end proc

который допустим лишь в релизах выше 7-го. Кодирование за заголовком процедуры типа не является в полном смысле слова типированием возвращаемого процедурой результата, а скорее утверждением (*assertion*). При установке *kernelopts(assertlevel = 2)* производится проверка типа возвращаемого результата вызова процедуры. Если тип не соответствует утверждению, то возникает ошибочная ситуация "assertion failed: %1 expects its return value to be of type %2, but computed %3". При остальных установках *assertlevel*-опции утверждение игнорируется. Особого смысла в данном формате мы не видим и вот почему. Если необходимо типировать результат возврата, то намного удобнее и эффективнее это делать в самой процедуре на основе как реализуемого ею алгоритма, так и получаемых типов фактических аргументов. При этом, сохраняется непрерывность вычислений и производится обработка *ошибочных* и *особых* ситуаций, связанных с типом возвращаемого результата. При этом, при *качественной* разработке процедуры в ней уже должна быть предусмотрена (*в случае необходимости*) проверка типов получаемых ею фактических аргументов. Так что и здесь имеется «*фильтр*» на допустимость фактических аргументов. К тому же, если данного формата процедура находится в библиотеке и используется в программировании, то при возникновении ошибки указанного выше типа пользователю будет достаточно не просто обнаружить причину такой ошибки, привязанной лишь к результату вызова процедуры, которая может иметь и несколько *разнотипных* точек возврата. В одном лишь, пожалуй, можно согласиться, что такого рода «*типизация*» в некотором отношении *подобна* типизации формальных аргументов, но относится к результатам ее вызова, и может в целом ряде случаев представить определенный интерес, в частности, в случае множественности точек возврата процедуры, упрощая обработку типов результатов вызова процедуры. Следующий достаточно простой фрагмент иллюстрирует результаты использования указанного формата кодирования процедуры:

```
> P := proc():float; `+`(args)/nargs end proc: P(68, 63, 43, 14, 21, 48); ⇒ 257/6
> kernelopts(assertlevel = 2): P(68, 63, 43, 14, 21, 48);
```

```
Error, (in P) assertion failed: P expects its return value to be of type float, but computed
257/6
> lasterror;
      "assertion failed: %1 expects its return value to be of type %2, but computed %3"
> P1 := proc()::float; local conv; conv:= proc(a::anything, b::anything) if a = NULL then b
      else convert(b,a) end if end proc; conv(op(8,eval(procname)), `+`(args)/nargs) end proc:
> P1(68, 63, 43, 14, 21, 48);  => 42.83333333
> P2 := proc()::+`; op(8, eval(procname))(args)/nargs end proc:
> kernelopts(assertlevel = 1): P2(68, 63, 43, 14, 21, 48, 23);  => 40
> P3 := proc()::proc() `if`(type(nargs, odd), `+`, `*`) end proc; op(8, eval('procname'))(args)
      (args)/nargs end proc:
> P3(42, 47, 67, 89, 105), P3(68, 63, 43, 14, 21, 48);  => 70, 433266624
```

Процедура *P1* содержит подпроцедуру *conv*, результат вызова которой обеспечивает конвертацию возвращаемого процедурой *P1* выражения в *тип*, определенный после заголовка процедуры. Возможно, в некоторых случаях такой прием может оказаться достаточно полезным. В частности, кодировать после заголовка процедуры *W* можно любое корректное выражение, доступное внутри процедуры по вызову *op(8, eval(W))*, что предоставляет целый ряд дополнительных *возможностей* при программировании приложений, как это иллюстрируют две последние процедуры *P2* и *P3* фрагмента. С этой целью следует применять установку *kernelopts(assertlevel = 1)* для того, чтобы не возникало ошибочных ситуаций типа "assertion failed: %1 expects its return value to be of type %2, but computed %3".

Последний механизм возврата результатов вызова процедуры связан с ошибочными и особыми ситуациями, возникающими в процессе ее выполнения. Каждая *ошибочная* ситуация, возникшая в момент передачи процедуре фактических аргументов либо в процессе ее выполнения, вызывает прекращение выполнения процедуры с возвратом соответствующего диагностического сообщения, которое в целом ряде случаев может недостаточно адекватно отражать возникшую ситуацию (см. [28-34]). Однако, наряду с данного типа ситуациями, обрабатываемыми пакетом автоматически, пользователь имеет возможность как производить *обработку* ситуаций, определяемых спецификой вызываемой процедуры, так и в определенной степени *перехватывать* обработку этих ошибочных ситуаций, стандартно обрабатываемых пакетом. Ранее рассматривались средства обработки ошибочных и особых ситуаций, имеющих особый смысл именно для процедурных и модульных программных объектов.

6.1.6. Расширенные средства Maple для работы с процедурами

Рассмотрим теперь вопросы использования *механизма* процедур несколько детальнее, учитывая прикладную значимость процедур для разработки различных приложений в пакете *Maple*. Прежде всего, рассмотрим вопрос создания вложенных процедур, т.е. таких процедур, определения которых, в свою очередь, содержат определения других процедур. В целом ряде случаев данная возможность может оказаться эффективным

средством при программировании разнообразных прикладных задач. В общем случае поименованная процедура наряду с общепринятым допускает также определение на основе следующей (довольно часто весьма удобной) вычислительной конструкции:

assign(<Id-процедуры> {, | =} proc() ... end proc)

эквивалентной рассмотренной выше стандартной конструкции ***Id:=proc() ... end proc.*** Данное обстоятельство позволяет использовать определение процедуры не только в теле другой процедуры, но и внутри вычислительных конструкций, например:

```
> assign(A, proc() local k; sum(args[k], k=1..nargs) end proc); A(63, 68, 43, 21); ⇒ 195
> (assign('t' = 3) - assign(G = proc(x, y) evalf(sqrt(x^2 + y^2)) end proc) + 6)*G(63,68)^t;
4779345.953
```

При этом, как будет показано ниже, при использовании таких подходов к определению подпроцедур имеется достаточно принципиальное различие. Более того, если *первый* пример фрагмента иллюстрирует определение *A*-процедуры описанным способом с последующими ее вычислением и вызовом, *второй* пример фрагмента иллюстрирует использование определенной вышеуказанным способом *G*-процедуры уже в простой вычислительной конструкции. Так как *assign*-процедура возвращает *NULL*-значение, то следует учитывать следующие соотношения для *NULL*-значения: *NULL - NULL = 0* и *NULL + NULL = NULL*. С учетом этого возможно строить корректные выражения.

```
> GS := proc(x::numeric, y::numeric, z::{14, 21})
  `if`(z = 14, assign(AG, proc(x, y) [x, y, evalf(sqrt(x*y), 6)] end proc),
    assign(AV, proc(x, y) [x, y, evalf(sqrt(x + y), 6)] end proc)); AG(x, y)
end proc:
> {GS(68, 63, 14), AG(1995, 2010)}; ⇒ {[68, 63, 65.4522], [1995, 2010, 2002.48]}
> GS1 := proc(x::numeric, y::numeric, z::{14, 21})
  `if`(z = 14, op([assign(AG, proc(x, y) [x, y, evalf(sqrt(x*y), 6)] end proc), AG(x, y)]),
    op([(assign(AV, proc(x, y) [x, y, evalf(sqrt(x + y), 6)] end proc), (AV(x, y))]))
end proc:
> {GS1(68, 63, 14), AG(68, 42)}; ⇒ {[68, 63, 65.4522], [68, 42, 53.4416]}
> {GS1(68, 63, 21), AG(68, 42)}; ⇒ {[68, 42, 53.4416], [68, 63, 11.4455]}
> GS2 := proc(x::numeric, y::numeric, z::{14, 21}) local A, V, G; global AG, AV;
  `if`(z = 14, assign(AG, proc(x, y) [x, y, evalf(sqrt(x*y), 6)] end proc),
    assign(AV, proc(x, y) [x, y, evalf(sqrt(x + y), 6)] end proc));
  eval(parse(" " || A || ("if(z = 14, G, V)) || "(" || x || "," || y || ")"))
end proc:
> GS2(59, 68, 14), AV(68, 42), AG(63, 47); ⇒ [59, 68, 63.3404], AV(68, 42), [63, 47, 54.4152]
```

Приведенный выше способ определения процедуры оказывается весьма полезным в целом ряде приложений. Между тем, вложенные процедуры возможно создавать и на базе стандартного определения. Предыдущий фрагмент иллюстрирует применение обоих способов для определения двух вложенных процедур, генерируемых основной процедурой в зависимости от конкретных условий применения.

Так, *первый* пример фрагмента представляет определение вложенной GS-процедуры, возвращающей результат вызова AG-подпроцедуры, чье тело определяется на основе assign-процедуры в зависимости от значения третьего z-аргумента, который передается внешней GS-процедуре. При этом, в текущем сеансе доступной является также и AG-подпроцедура, если ее идентификатор не декларируется во внешней процедуре как *локальная* переменная.

Второй пример представляет GS1-процедуру, возвращающую результат вызова одной из подпроцедур {AG, AV} в зависимости от значения третьего фактического аргумента z, передаваемого внешней процедуре. При этом, в текущем сеансе обе подпроцедуры носят локальный характер, т.е. доступны лишь в рамках содержащей их *внешней* GS1-процедуры. Наконец *третий* пример представляет GS2-процедуру, эквивалентную предыдущей GS1-процедуре, но только с той разницей, что входящие в нее AG и AV подпроцедуры определены глобальными. Результат последующих вызовов процедур GS2, AG и AV весьма наглядно иллюстрирует вышесказанное.

Из сказанного следует вынести следующий *довольно* важный вывод, иллюстрируемый представленной ниже таблицей. В случае явного декларирования идентификаторов подпроцедур область их действия соответствует декларации *независимо* от способа их определения, тогда как в случае отсутствия декларации определенная стандартно и по assign-процедуре подпроцедура становится соответственно *локальной* и *глобальной*.

	<i>Proc := proc() ... end proc</i>	<i>assign(Proc, proc()) ... end proc</i>
<i>global</i>	<i>global</i>	<i>global</i>
<i>local</i>	<i>local</i>	<i>local</i>
<i>не продекларирована</i>	<i>local</i>	<i>global</i>

Представленный в таблице *принцип* декларирования идентификаторов подпроцедур в полной мере распространяется и на идентификаторы вообще, что весьма наглядно иллюстрирует следующий достаточно прозрачный фрагмент, а именно:

```
> restart: H := proc(x) local h; h := x^2 end proc: [H(68), h];    => [4624, h]
> restart: H := proc(x) global h; h := x^2 end proc: [H(68), h]; => [4624, 4624]
> restart: H := proc(x) h := x^2 end proc: [H(68), h];    => [4624, h]
Warning, `h` is implicitly declared local to procedure `H`
> restart: H := proc(x) local h; assign(h = x^2); h end proc: [H(68), h];    => [4624, h]
> restart: H := proc(x) global h; assign(h = x^2); h end proc: [H(68), h]; => [4624, 4624]
> restart: H := proc(x) assign(h = x^2); h end proc: [H(68), h];    => [4624, 4624]
```

Данное обстоятельство необходимо учитывать при *практическом* программировании, иначе в целом ряде случаев его *игнорирование* может быть причиной весьма серьезных ошибок, как отслеживаемых ядром пакета Maple, так и семантических ошибок.

Таким образом, Maple-язык позволяет создавать вложенные процедуры, для которых поддерживается не только возможность определения *одной* процедуры в теле другой, но и возврат процедурой другой процедуры в качестве ее выхода. Поддерживаемый Maple-языком механизм вложенных процедур (*lexical scoping*) обеспечивает доступ *вложенных* процедур к переменным, которые находятся в *окружающих* их процедурах.

Данный аспект лежит в основе обеспечения механизма инкапсуляции, на котором и базируется современное *объектно-ориентированное* программирование. При этом, для глобальных переменных поддерживается только режим их разделения вложенными процедурами, т.е. процедурами, расположенными в теле других процедур.

Процедуры допускают любой уровень вложенности, определяемый только объемом памяти, доступной рабочей области пакета *Maple*. Однако, доступность внутренних процедур в текущем сеансе определяется следующими двумя моментами, а именно: (1) типом *внутренней* процедуры (*local*, *global*) и (2) ее режимом использования. Если внутренняя процедура определена локальной (*local*), то прямой доступ к процедуре невозможен извне содержащей ее главной процедуры, тогда как при определении ее глобальной (*global*) ситуация несколько иная, а именно. Сразу же после вычисления определения главной процедуры все ее *внутренние* процедуры, включая и *глобальные*, остаются неопределенными, т.е. недоступными извне процедуры, содержащей ее. И лишь после *первого* вызова главной процедуры все ее *внутренние* процедуры, которые определены тем либо иным способом *глобальными*, становятся доступными в текущем сеансе вне содержащей их процедуры. Следующий достаточно несложный фрагмент весьма наглядно иллюстрирует вышесказанное.

```
> P := proc() local P1; P1 := () -> '+'(args); P1(args) end proc;
> map(type, [P, P1], 'procedure'); => [true, false]
> P(68, 63, 43, 14, 21): map(type, [P, P1], 'procedure'); => [true, false]
> restart; P := proc() global P1; P1 := () -> '+'(args); P1(args) end proc;
> map(type, [P, P1], 'procedure'); => [true, false]
> P(68, 63, 43, 14, 21): map(type, [P, P1], 'procedure'); => [true, true]
> restart; P := proc() assign('P1' = (() -> '+'(args))); P1(args) end proc;
> map(type, [P, P1], 'procedure'); => [true, false]
> P(68, 63, 43, 14, 21): map(type, [P, P1], 'procedure'); => [true, true]
> restart; P := proc() global P1, P2; P1 := () -> '+'(args); P2 := () -> '*'(args); if nargs=3 then
  P1(args) else P2(args) end if end proc: map(type, [P, P1, P2], 'procedure');
                                     [true, false, false]
> P(68, 63, 43, 14, 21): map(type, [P, P1, P2], 'procedure'); => [true, true, true]
> restart; P := proc() local P1; P1 := proc() global P2; P2 := () -> '+'(args); [args] end proc;
  '+'(args) end proc: map(type, [P, P1, P2], 'procedure'); => [true, false, false]
> P(68, 63, 43, 14, 21): map(type, [P, P1, P2], 'procedure'); => [true, false, false]
> restart; P := proc() local P1; P1 := proc() global P2; P2 := () -> '+'(args); [args] end proc;
  P1(args) end proc: map(type, [P, P1, P2], 'procedure'); => [true, false, false]
> P(68, 63, 43, 14, 21): map(type, [P, P1, P2], 'procedure'); => [true, false, true]
```

Таким образом, вложенные процедуры (*глобальные относительно содержащей ее главной процедуры*) становятся *доступными* в текущем сеансе пакета лишь после *первого* вызова главной процедуры. При этом, как иллюстрируют последние примеры фрагмента, в случае более *одного* уровня вложенности подпроцедуры, определенные глобальными, становятся доступными в текущем сеансе только после реального вызова содержащих их подпроцедур. При этом, кажущаяся «*вложенность*» следующего типа, а именно:

P := proc() global P; P := proc() end proc; [args] end proc:

в качестве вложенности рассматриваться не может и представляет собой своего рода исключительный случай, как это наглядно иллюстрирует следующий фрагмент:

```
> restart; P := proc() global P; P := () -> `+`(args); P(args); [args] end proc: eval(P);
      proc() global P; P := () -> `+`(args); P(args); [args] end proc
```

```
> P(68, 63, 43, 14, 21), eval(P), P(68, 63, 43, 14, 21); => [68, 63, 43, 14, 21], () -> `+`(args), 209
```

Таким образом, во избежание недоразумений не рекомендуется использовать имена процедур, в которых они определяются, в качестве имен глобальных переменных.

В свете вышесказанного довольно полезной представляется процедура *intproc(P)* [45], обеспечивающая проверку процедуры *P* быть главной или вложенной/внутренней. Успешный вызов процедуры возвращает последовательность из двух списков, *первый* из которых определяет процедуры *текущего* сеанса, идентичные исходной процедуре *P*, тогда как второй список определяет процедуры, содержащие процедуру *P* (либо ей идентичную) в качестве внутренних/вложенных процедур. Первый элемент данных списков, а именно – *analogous* и *innet* – определяет тип содержащихся в них элементов: *аналогичная* (главная либо внутренняя/вложенная, но глобальная) и *внутренняя/вложенная* соответственно. Нижеследующий фрагмент представляет исходный текст процедуры *intproc* и некоторые достаточно типичные примеры ее использования.

```
> intproc := proc(P::symbol) local a, b, k, x, y;
  if type(eval(P), 'symbol') then return FAIL elif type(P, 'procedure') then
    try P() catch: NULL end try;
    assign(a = {anames('procedure')}, x = ['analogous'], y = ['innet']);
    b := {seq(`if`(" " || a[k][1 .. 3] = "CM:", NULL, a[k]), k = 1 .. nops(a))}
  else error "<%1> has `%2`-type but should be procedure", P, whattype(eval(P))
  end if;
  if member(P, b) then
    for k in b minus {P} do
      if Search1(convert(eval(k), 'string'),
        convert(eval(P), 'string'), 't') then
        if t = ['coincidence'] then x := [op(x), k] else y := [op(y), k] end if
      else next
      end if
    end do; x, y
  else FAIL
  end if
end proc:
> P := proc() [args] end proc: P1 := proc() [args]; end proc: T := table():
> P2 := proc() local P1; P1 := proc() [args] end proc; P1(args) end proc:
> P3 := proc() global P1; P1 := proc() [args] end proc; P1(args) end proc: intproc(P);
      [analogous, P1], [innet, P2, P3]
> {intproc(P1)}, {intproc(P2)}; => {[analogous, P], [innet, P2, P3]}, {[analogous], [innet]}
```



```
> {intproc(P3)}, intproc(AGN); ⇒ {[analogous], [innet]}, FAIL
> intproc(T);
Error, (in intproc) <T> has `table`-type but should be procedure
```

Если в качестве фактического аргумента *P* выступает *символьное* выражение, то вызов процедуры *intproc* возвращает *FAIL*-значение по причине невозможности установить истинное определение символа (*возможно, процедура находится в библиотеке, логически не сцепленной с главной Maple-библиотекой*). В случае типа аргумента *P*, отличного от *procedure*, возникает ошибочная ситуация. Процедура *intproc* имеет ряд достаточно интересных приложений при процедурном программировании в *Maple* [8-14,28-34].

Дальнейшее рассмотрение целесообразно начать с *CompSeq*-функции, позволяющей в определенной мере автоматизировать процесс создания процедур на базе некоторой последовательности *Maple*-предложений. Для организации разных вычислительных последовательностей (*ВП*) в формате *невычисляемых* конструкций (*своего рода макетов вычислительных блоков*) пакет располагает специальной *CompSeq*-функцией, имеющей следующий довольно простой формат кодирования, а именно:

CompSeq(*locals* = *L1*, *globals* = *L2*, *params* = *L3*, <*Список ВК*>)

где в качестве первых трех необязательных аргументов выступают списки локальных (*L1*), глобальных (*L2*) и переменных *ВП* соответственно, а также параметров (*L3*). Если локальные переменные областью определения имеют лишь тело *ВП*, то глобальные – текущий сеанс, а *параметры* могут передаваться в *ВП* извне ее, как и воспринимаемые извне значения ее глобальных переменных. Последний обязательный аргумент функции *CompSeq* представляет собой *список вычисляемых конструкций (ВК)* следующей формы <*Id-переменной*>=<*Выражение*>; при этом, последняя конструкция последовательности *ВК* и возвращает окончательный результат ее вычисления. При этом, *ВП* может быть упрощена, оптимизирована, а также конвертирована в форму *процедуры*, и наоборот. Следующий простой фрагмент иллюстрирует использование *CompSeq*-функции для создания *ВП*, а затем конвертацию полученной конструкции в *Maple*-процедуру с ее последующим выполнением в текущем сеансе пакета:

```
> GS := CompSeq(locals = [x, y, z], globals = [X, Y, Z], params = [a, b, c], [x:=a*sqrt(X^2 +
    Y^2 + Z^2), y = (x+b)*exp(Z)/(ln(X) + sin(Y)), z = x*y/(a*x + b*y), x*y/(x + z)]);
> SG := convert(GS, 'procedure');
SG := proc(a, b, c) local x, y, z;
global X, Y, Z;
    x := a*(X^2 + Y^2 + Z^2)^(1/2); y := (x + b)*exp(Z)/(ln(X) + sin(Y));
    z := x*y/(a*x + b*y); x*y/(x + z)
end proc
> X := 6.8: Y := 6.3: Z := 4.3: evalf(SG(42, 47, 67), 12); ⇒ 17778.1074831
```

Между тем, возможности *CompSeq*-функции позволяют описывать довольно простой вычислительный алгоритм в виде последовательностей простых *ВК*, в последующем конвертируемых в процедуры, определяющие законченные вычислительные блоки, вызовы которых можно производить на заданных списках фактических значений их

формальных аргументов. Это позволяет существенно упростить решение целого ряда прикладных, но достаточно простых задач. К *CompSeq*-функции в некоторой степени примыкает и процедура *makeproc* пакетного модуля **codegen**, служащая для создания *Maple*-процедур из выражений типа формула, список или массив формул, например

```
> S := [z:=ln(x) + 6.8, a = cos(x), (y - 6*x - a*x - z^2*x + 6.3*x)/sqrt(x^2 + y^2)]:
H := codegen[makeproc](S, [x::numeric, y::numeric]); H(6.3, 4.3);
H := proc(x::numeric, y::numeric) local z, a; z := ln(x) + 6.8; a := cos(x); (y + 0.3*x - a*x -
z^2*x)/(x^2 + y^2)^(1/2) end proc
-61.67897708

> G := codegen[optimize](H); G(6.3, 4.3);
G := proc(x::numeric, y::numeric) local a, t1, t4, t7, t8, t10;
t1 := ln(x); a := cos(x); t4 := (t1 + 6.8)^2; t7 := x^2; t8 := y^2; t10 := (t7 + t8)^(1/2);
(y + 0.3*x - a*x - t4*x)/t10
end proc
-61.67897708
```

Наконец, к процессу создания процедуры относится и процедура *optimize* пакетного модуля **codegen**, которая ориентирована на оптимизацию *Maple*-выражений в целом, включая и процедуры. Вызов **codegen[optimize](C)** возвращает оптимизированную *C*-процедуру. В текущих последних релизах пакета процедура *optimize* может проводить оптимизацию лишь простых процедур, тело которых представляет собой некоторую последовательность простых предложений присвоения, но не содержит предложений цикла или *if*-предложений. Последний пример предыдущего фрагмента достаточно иллюстрирует применение процедуры *optimize* для оптимизации *H*-процедуры.

Для обеспечения вывода из процедур полезной для пользователя информации в теле процедуры можно помещать специальную встроенную функцию *userinfo*, имеющую следующий достаточно простой формат кодирования, а именно:

userinfo(<Уровень>, <Id>, <Выражение_1> {, ..., <Выражение_n>})

и позволяющую согласно указанному уровню выводить информацию, представляемую ее аргументами, начиная с третьего (обязательного), для процедур, идентификаторы которых задаются вторым аргументом функции, допускающим либо отдельное имя, либо множество имен. Но вывод информации, получаемой в результате вычисления выражений *к*, производится только в случае, если определенный первым аргументом функции уровень не превышает установленного в *infolevel*-таблице для ее *all*-входа. Исходным состоянием данной таблицы является: **print(infolevel); ⇒ table([hints = 1])**. Ее единственный *hints*-вход определяет вывод результата вызова функции/процедуры невычисленным, если нельзя получить его точное значение. Вычисленные выражения *userinfo*-функции выводятся в *lprint*-формате, разделенные тремя пробелами.

Предложение **infolevel[all]:=n** определяет общий уровень вывода для всех последующих процедур, содержащих *userinfo*-функцию. Соответствующая информация выводится только, если определяемый их первым аргументом уровень не превышает указанного в предложении *infolevel* уровня *n*, как это иллюстрирует следующий фрагмент:

```

> infolevel[all] := 10;  ⇒ infolevelall := 10
> P1:=proc() userinfo(5, P1, `Суммирование` | | nargs | | `аргументов`); `+(args) end proc:
> P2:=proc() userinfo(12, P2, `Суммирование` | | nargs | | `аргументов`); `+(args) end proc:
> P1(14, 21, 43, 48, 63, 68); restart:  ⇒ 257
P1: Суммирование 6 аргументов
> P1 := proc() global infolevel; infolevel[P1] := 6: userinfo(5, P1, `Суммирование` | |
nargs | | `аргументов`); `+(args) end proc:
> P2 := proc() global infolevel; infolevel[P2] := 9: userinfo(12, P2, `Суммирование` | |
nargs | | `аргументов`); `+(args) end proc:
> P1(14, 21, 43, 48, 63, 68);  ⇒ 257
P1: Суммирование 6 аргументов
> P2(14, 21, 43, 48, 63, 68);  ⇒ 257
> print(infolevel);  ⇒ table([P2 = 9, P1 = 6, hints = 1])

```

В частности, библиотечные процедуры пакета используют следующие *уровни* вывода информации: (1) – обязательная информация; (2, 3) – общая информация, включая и метод решения проблемы, и (4,5) – детальная информация по процедуре. Между тем, при отсутствии для процедуры общего уровня вывода информации он определяется на базе индивидуальных *infolevel*-предложений, кодируемых в теле самих процедур либо вне их в виде *infolevel*[<Имя процедуры>]:= *Уровень*. По данному предложению соответствующая информация заносится в *infolevel*-таблицу, с которой уже работают функции *userinfo* процедур. Последний пример предыдущего фрагмента достаточно наглядно иллюстрирует сказанное. Средство *userinfo*-функции довольно полезно для обеспечения пользователя документированными процедурами.

При этом, при определении *возврата* результата процедуры через *RETURN*-функцию (*return*-предложение) следует кодировать предложения *infolevel* и *userinfo* перед ними, ибо в противном случае действие последних подавляется, как это довольно наглядно иллюстрирует следующий достаточно простой фрагмент, а именно:

```

> infolevel[all] := 10;  ⇒ infolevelall := 10
> P1 := proc() local infolevel; userinfo(5, P1, `Суммирование` | | nargs | | `аргументов`);
`+(args); end proc: P2 := proc() local infolevel; return `+(args); userinfo(5, P2,
`Суммирование` | | nargs | | `аргументов`) end proc:
> P1(14, 21, 43, 48, 63, 68);  ⇒ 257
P1: Суммирование 6 аргументов
> P2(14, 21, 43, 48, 63, 68); restart:  ⇒ 257
> P1 := proc() local infolevel; infolevel[P1] := 10: userinfo(2, P1, `Суммирование` | |
nargs | | `аргументов`); `+(args); end proc: P2 := proc() global infolevel;
infolevel[P2] := 10: userinfo(2, P2, `Суммирование` | | nargs | | `аргументов`);
`+(args) end proc: P1(14, 21, 43, 48, 63, 68); print(infolevel);  ⇒ 257  table([hints = 1])
> P2(14, 21, 43, 48, 63, 68); print(infolevel);  ⇒ 257  table([hints = 1, P2 = 10])
P2: Суммирование 6 аргументов

```

При этом, последние 2 примера фрагмента иллюстрируют необходимость *глобального*

определения *infolevel*-переменной, в противном случае она не редактирует *infolevel*-таблицу пакета и не влияет на вывод *userinfo*-информации.

Для обеспечения мониторинга основных вычислительных ресурсов, затребованных при выполнении процедур/функций, служит группа *profile*-процедур, позволяющих получать оперативную информацию по выполнению заданных процедур/функций в разрезе: количество вызовов, временные издержки, требуемая оперативная память и др. Информация выводится в табличном виде, смысл которой особых пояснений не требует. Для начала режима мониторинга процедур/функций используется *profile(P1, ..., Pn)*-процедура которая в случае успешного вызова возвращает значение *NULL* и по которой устанавливается режим мониторинга вызовов *Pj*-процедур, определяемых ее фактическими аргументами. Между тем следует иметь в виду, попытка вызова *profile*-процедуры для уже находящейся в режиме мониторинга процедуры либо функции вызывает ошибочную ситуацию, как иллюстрирует пример следующего фрагмента. На основе вызова *profile()* выполняется мониторинг вызовов всех процедур/функций текущего сеанса. Однако, в таком объеме *profile*-средство рекомендуется применять с большой осторожностью во избежание существенного увеличения вычислительных ресурсов ПК: замедление вычислений и увеличение используемой памяти, вплоть до критического, блокирующего весь вычислительный процесс.

Результаты мониторинга носят кумулятивный характер и их возможно периодически выводить по процедуре *showprofile({ | P1, ..., Pn })* в разрезе или всех профилируемых процедур, или только относительно *Pj*-указанных в качестве фактических аргументов процедуры. По вызову процедуры *unprofile({ | P1, ..., Pn })* производится прекращение режима мониторинга в разрезах, аналогичных предыдущей процедуры. Результатом успешного вызова *unprofile*-процедуры является возврат *NULL*-значения, удаление профильной информации и прекращение режима мониторинга по соответствующим процедурам либо функциям. Повторное применение *unprofile*-процедуры вызывает ошибочную ситуацию. Простой фрагмент иллюстрирует применение рассмотренных средств для мониторинга пользовательской *VSV*-процедуры и некоторых встроенных функций пакета *Maple*.

```
> VSV := proc() local k; product(args[k], k=1 .. nargs) end proc: profile(VSV);
> [VSV(14, 21, 43, 48, 63, 68), VSV(96, 89, 67, 62, 47, 42), VSV(k$k=1..9)];
[2599599744, 70060765824, 362880]
> showprofile(VSV);
function      depth  calls  time   time%    bytes   bytes%
-----
VSV           1       3   0.016  100.0    22940   100.0
-----
total:        1       3   0.016  100.0    22940   100.0
> profile(VSV);
Error, (in profile) VSV is already being profiled.
> profile(sin, exp); [sin(6.8), sin(4.2), exp(0.63), sin(21)*exp(14)]: showprofile();
function      depth  calls  time   time%    bytes   bytes%
```

VSV	1	3	0.016	25.40	23940	49.07
sin	1	3	0.032	50.79	15792	33.78
exp	1	2	0.015	23.81	8016	17.15
total:	3	8	0.063	100.00	46748	100.00
> unprofile(): showprofile();						
function	depth	calls	time	time%	bytes	bytes%
total:						
	0	0	0.000	0.00	0	0.00

С учетом сказанного особых пояснений примеры фрагмента не требуют. Близкой по назначению к *profile*-процедуре является *exprofile*-процедура, которая обеспечивает мониторинг всех вызовов пакетных процедур и функций. С рядом некоторых других типов мониторинга различных аспектов выполнения процедур или функций в среде пакета можно ознакомиться в [9-14,28-37,42-45]. Рассмотренные средства мониторинга предоставляют полезную информацию, в частности, для оптимизации вычислений.

Создав собственную достаточно серьезную процедуру с использованием собственных и пакетных процедур и функций, поместив ее в свою библиотеку, естественно встает проблема ее *оптимизации*, в частности, установление частоты использования средств, содержащихся в ней, и основных компьютерных ресурсов, используемых ими. В этом контексте проблема оптимизации пользовательских процедур весьма актуальна. Для данных целей достаточно полезной представляется процедура *StatLib(L)* [45], которая обеспечивает полный сбор *основной* статистики по заданной *L*-библиотеке и возврату статистики для последующего анализа. Естественно, в процессе своего выполнения *StatLib*-процедура требует некоторых дополнительных ресурсов памяти и времени, однако эти затраты с лихвой окупаются той информацией, которая позволяет более эффективно организовать пользовательскую библиотеку процедур и функций. Этот вопрос будет рассматриваться нами в книге несколько ниже.

6.1.7. Расширение функциональных средств Maple-языка пакета

Многие встроенные и библиотечные процедуры *Maple* допускают пользовательские *расширения*, увеличивающие область применения данных средств. Например, можно определять *новые* типы и операторы, расширять диапазон математических функций, обрабатываемых функциями *evalf*, *diff*, расширяя тем самым средства главной *Maple*-библиотеки. Каждое *уникальное* расширение средства типа *type* или *diff* *ассоциируется* с именем. Такое имя определяет имя расширяемого средства, например, для *type* это *type*. Существуют 2 механизма расширения: (1) классический механизм расширения, используемый в большинстве случаев, и (2) *новейший* механизм расширения, который используется новыми средствами, в частности, пакетный модуль *TypeTools* (начиная с *Maple 8*), содержащий 5 процедур для расширения множества типов, распознаваемых функцией *type* пакета, играющей важную роль при типизации объектов пакета.

Классический механизм расширения имеет *единственную глобальную область* для имен расширения, тогда как современный механизм обеспечивает *именование расширения* в ряде областей *имен* (иными словами, использование современного механизма предоставляет возможность присваивать имена расширениям, которые будут локальными для процедур или модулей). С вопросами использования современного механизма расширения средства пакета можно познакомиться по запросу *?extension*, здесь же мы вкратце рассмотрим классический механизм расширений средств пакета, носящий глобальный характер.

Все встроенные процедуры пакета и большинство библиотечных процедур, которые допускают пользовательские расширения, используют именно *классический механизм*, основанный на конкатенации имен, а именно. Расширяемой процедуре *wname* дают новые функциональные возможности, определяя выражение (как правило, процедура) с именем формата *`wname/new`*, где *new* – имя расширения. Например, новый тип для бинарных выражений, распознаваемый *type*-функцией, вполне может быть определен процедурой с именем *`type/binary`*. Но определяемые расширения для стандартного средства *wname* должны удовлетворять основным правилам, которым удовлетворяет данное средство *wname*, и которые отражены в справке по этому средству. Например, при расширении стандартной *type*-функции процедурой с именем *`type/abc`* нужно, чтобы она возвращала только логические значения *{true, false}*, иначе инициируется ошибка с диагностикой "result from type `%1` must be true or false", например:

```
> `type/abc` := proc(x) if x < 0 then true elif x > 0 then false else FAIL end if end proc;
> type(0, 'abc');
Error, result from type `abc` must be true or false
> lasterror; => "result from type `%1` must be true or false"
```

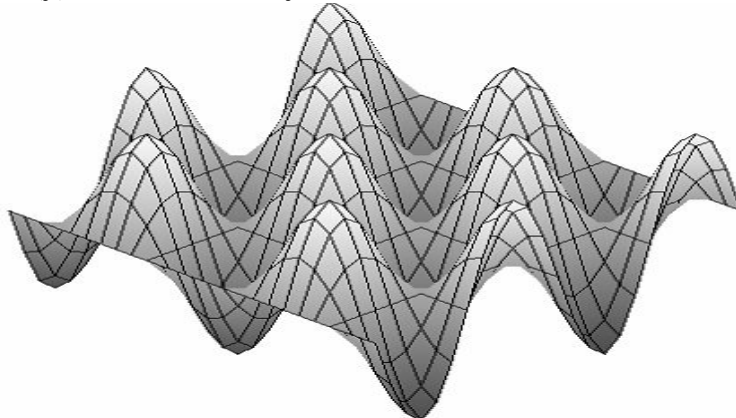
В качестве простого примера расширения стандартной *type*-функции зададим новый тип *color*, отсутствующий в пакете текущих релизов и весьма полезный для большого числа задач, имеющих дело с *графическими объектами*. Вызов процедуры *type(x, color)* возвращает значение *true*, если *x* есть имя цвета, процедура либо выражение, которые возможно рассматривать в качестве *цвета* при формировании графического объекта; в противном случае возвращается *false*-значение. В свою очередь, через глобальную переменную *_ColorType* возвращается двухэлементный бинарный список, если вызов процедуры *type(x, color)* возвращает *true*-значение. Более того, его первый и второй элементы определяют допустимость *x*-выражения в качестве цвета (*0* – нет, *1* – да) при оформлении графического объекта с опцией *color=x* при *размерности* соответственно *2* и *3*. Если же вызов процедуры возвращает *false*-значение, то переменная *_ColorType* возвращается неопределенной. Процедура *`type/color`* имеет немало весьма полезных приложений, однако ее применение предполагает дополнительную проверку через глобальную переменную *_ColorType* из-за различий механизмов раскраски для 2- и 3-мерных графических объектов. Ниже даются исходный текст процедуры *`type/color`* и примеры ее использования с графическими объектами пакета *Maple*.

```
> `type/color` := proc(C::anything) global _ColorType;
_ColorType := [0, 0];
try plot(1, x = 0 .. 1, color = C); _ColorType := _ColorType + [1, 0]
```

```

catch: NULL
end try;
try plot3d(1, x = 0 .. 1, y = 0 .. 1, color = C); _ColorType := _ColorType + [0, 1]
catch: NULL
end try;
if _ColorType = [0, 0] then unassign('_ColorType'); false else true end if
end proc;
> type(COLOR(0.68, 0.63, 0.43), 'color'), _ColorType; ⇒ true, [1, 1]
> type(0.68, 0.63, 0.43, 'color'), _ColorType; ⇒ true, [0, 1]
> type([sin(x*y), cos(x*y), tan(x*y)], 'color'), _ColorType; ⇒ true, [0, 1]
> type(AGN(0.68, 0.63, 0.43), 'color'), _ColorType; ⇒ true, [0, 1]
> type([1.68, 0.63, 0.43], 'color'), _ColorType; ⇒ true, [0, 1]
> type(red, color), _ColorType, type(blue, color), _ColorType; ⇒ true, [1, 1], true, [1, 1]
> plot3d(sin(x)*cos(y), x = -2*Pi..2*Pi, y = -2*Pi..2*Pi, color = [1.68, 0.63, 0.43]);

```



Читателю рекомендуется рассмотреть использованный процедурой прием, который положен в основу алгоритмов тестирования и который может быть довольно полезен как для создания средств тестирования других *графических* опций, так и в целом ряде других задач. Ниже представлены два примера применения *классического* механизма расширения средств *Maple*, которые рекомендуется рассмотреть в качестве полезного *упражнения*, учитывая, что указанные расширения имеют практические приложения.

```

> `type/complex1` := proc(Z::anything)
  local `1`, `2`, a, b, c, d, h, t, k, rm, im, sv, gv, tn;
  option `Copyright (C) 2004 by the International Academy of Noosphere. All rights reserved.`;
  assign(`1` = cat("", convert(interface('imaginaryunit'), 'string'))),
  assign(`2` = [cat("i", `1`), cat(`1`, "i"), `1`]); gv := simplify(expand(Z));
  if map2(search, convert(gv, 'string'), {op(`2`)} = {false} then false else
    assign(h = interface(warnlevel), a = normal(evalf(gv))), null(interface(warnlevel = 0));
    tn := proc(x) local a;
      a := convert(x, 'string');

```

```

    `if`(a[-1] = ".", came(a[1 .. -2]), x)
  end proc;
sv := s -> `if`(4 < length(s) and member(s[1 .. 4], {"-1.*", "+1.*"}), s[5 .. -1], s);
d := denom(a);
if d = 1 or Search2(convert(d, 'string'), {op(`2`)} = [] then
  a := convert(normal(evalf(a)), 'string')
else a := convert(expand(normal(evalf(conjugate(d)*numer(a)))), 'string')
end if;
a := `if`(member(a[1], {"-", "+"}), a, cat("+", a));
assign67(b = seqstr('procname(args)'), t = NULL);
b := b[length(cat("`type/convert1", "(", convert(gv, 'string')) + 1 .. -2)];
if b = "" then t := 'realnum'
else t := came(sub_1([seq(k = "realnum", k = ["fraction", "rational"])], b[2 .. -1]))
end if;
c := Search2(a, {"-", "+"});
if nops(c) = 1 then a := sub_1([`2`[1] = NULL, `2`[2] = NULL], a);
null(interface(warnlevel = h), type(tn(came(sv(a))), t)
else assign(rm="", im="", 'b'=[seq(a[c[k] .. c[k + 1] - 1], k=1 .. nops(c) - 1), a[c[-1] .. -1]]);
  for k to nops(b) do
    if Search2(b[k], {`2`[1], `2`[2]}) = [] then rm := cat(rm, b[k])
    else im := cat(im, sub_1([`2`[1] = NULL, `2`[2] = NULL], b[k]))
    end if
  end do;
try `if`(im = "", RETURN(false), assign('rm' = came(sv(rm)), 'im' = came(sv(im))))
catch: RETURN(null(interface(warnlevel = h), false)
end try;
null(interface(warnlevel = h), `if`(map(type, {tn(rm), tn(im)}, t) = {true}, true, false)
end if
end if
end proc;

> type((a + I)*(a - a*I + 3*I), complex1(algebraic));    => true
> type(a + sqrt(8)*I, complex1({symbol, realnum}));    => true
> map(type, [-6.7, 42072, 2010, -1995, 14/21], complex1);  => [false, false, false, false, false]
> type(2010, complex1(numeric));    => false
> type([a, b] - 3*I, complex1({realnum, list}));    => true
> type(-42072, 'complex'), type(-42072, 'complex1');    => true, false
> assume(a, 'integer'); type((a - I)*(a + I), complex(algebraic)), type((a - I)*(a + I),
  complex1(algebraic));    => true, false
> type((14 - I)*(14 + I), complex), type((14 - I)*(14 + I), complex1);    => true, false
> type(21 + I^2, complex(algebraic)), type(21 + I^2, complex1);    => true, false

```



```

> type(68,complex),type(68,complex1),type((3+4*I)/(5+6*I),complex1); => true, false, true
> `&ma` := proc () op(map(assign, [seq(args[k], k = 1 .. nargs - 1)], `if` (args[-1] = _NULL,
    NULL, args[-1]))) end proc:
> x, y, z, v, g, s, art, kr := 68;
    Error, cannot split rhs for multiple assignment
> &ma(h(x), g(y), v(z), r(g), w(h), (a+b)/(c-d)); h(x), g(y), v(z), r(g), w(h);
     $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ 
> &ma('x', 'y', 'z', 'g', 'h', "(a+b)/(c-d)"); x, y, z, g, h;
    "(a+b)/(c-d)", "(a+b)/(c-d)", "(a+b)/(c-d)", "(a+b)/(c-d)", "(a+b)/(c-d)"
> &ma('x', 'y', 'z', 'g', 'h', _NULL); x, y, z, g, h;
> &ma('x', 'y', 'z', 'g', 'h', 2010); x, y, z, g, h; => 2010, 2010, 2010, 2010, 2010
> ('x', 'y', 'z', 'g', 'h') &ma _NULL; x, y, z, g, h;
> ('x', 'y', 'z', 'g', 'h') &ma 2010; x, y, z, g, h; => 2010, 2010, 2010, 2010, 2010
> ('x', 'y', 'z', 'g', 'h') &ma (sin(a)*cos(b)); x, y, z, g, h;
    sin(a) cos(b), sin(a) cos(b), sin(a) cos(b), sin(a) cos(b), sin(a) cos(b)
> ('x', 'y', 'z', 'g', 'h') &ma ((a+b)/(c-d)); x, y, z, g, h;
     $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ ,  $\frac{a+b}{c-d}$ 

```

В первом примере определяется модификация стандартного пакетного *complex*-типа. Вызов стандартной функции *type(Z, complex)* возвращает *true*-значение в том случае, если *Z* - выражение формы $x + y*I$, где *x* (если существует) и *y* (если существует), есть конечные значения типа 'realcons'. Между тем, вызов процедуры *type(X, complex(W))* возвращает *true*-значение, если *Re(X)* (если существует) и *Im(X)* (если существует) оба типа *W*. К сожалению, стандартная процедура не обеспечивает, по нашему мнению, корректного тестирования *Maple*-выражений упомянутого *complex*-типа, включая в него и действительные значения, что в целом ряде случаев недопустимо. Следующий простой пример вполне убедительно подтверждает вышесказанное, а именно:

```

> map(type, [-72.420, 68, 420, -2011, 14/22], 'complex'); => [true, true, true, true, true]

```

К сожалению, такая серьезная ошибка имеет место для всех релизов *Maple*, начиная с шестого. Процедура '*type/complex1*' имеет те же самые формальные аргументы как и стандартная процедура и устраняет ошибки, свойственные второй. Кроме того, она обеспечивает более широкую проверку *Maple*-объектов *complex*-типа. Между тем, эта процедура не различает числовые типы {*fraction*, *rational*}, идентифицируя их более общим *numeric*-типом. В отличие от стандартной процедура '*type/complex1*' позволяет обеспечить более корректное тестирование выражений на *complex*-тип. Приведенные примеры иллюстрируют применение и стандартной, и '*type/complex1*'-процедуры.

В конструкции *lhs := rhs* оператор назначения `:=` присваивает *lhs* выражение *rhs*. При этом, оператор назначения допускает многократные назначения. Между тем, в таком случае количество элементов последовательности *lhs* должно строго соответствовать количеству элементов последовательности *rhs*, в противном случае возникает ошибка

с диагностикой "Error, ambiguous multiple assignment". Между тем, в целом ряде случаев возникает необходимость назначения идентичного выражения достаточно длинной последовательности имен либо вызовов функций.

Эта проблема решается оператором **&ma**, который имеет идентичный с оператором **:=** приоритет. Оператор **&ma** имеет 2 формата кодирования, а именно: *процедурный* и *операторный*. Вообще говоря, в обоих случаях элементы *lhs* должны кодироваться в невычисленном формате. Исключение – только самое *первое* назначение. Кроме того, в операторном формате, левая часть *lhs* должна быть закодирована в скобках. Кроме того, если правая часть *rhs* удовлетворяет условию *type(rhs, {`..`, `<`, `<=`, `:=`, `*`, `^`, `+`, `=`, `}) = true*, то правая часть должна быть также закодирована в скобках. Наконец, если необходимо присвоить *NULL*-значение элементам *левой* части *lhs*, тогда в качестве *rhs* кодируется *_NULL*-значение. Успешный вызов процедуры **&ma** либо использования оператора **&ma** возвращает *NULL*-значение с выполнением указанных назначений. В целом ряде приложений оператор **&ma** оказывается достаточно полезным средством.

Наконец, последний пример фрагмента иллюстрирует *определение* пользовательского оператора **&ma**, обеспечивающего многократные присвоения *одинакового* выражения переменным либо вызовам функций. На данном вопросе имеет смысл акцентировать внимание отдельно, учитывая его важность для практического программирования.

Наряду со стандартно определяемыми, пакет допускает *пользовательские* операторы (в терминологии пакета называемые *нейтральными операторами*), идентифицируемые ключевыми словами вида **&Символ**, при этом *Символ* кодируется без верхних кавычек и не должен содержать нижеследующих символов, а именно:

& | () [] { } ; : ' ` # % \ пробел перевод строки

Длина **&**-цепочки *символов* не должна превышать **495**. Сам же *Maple*-язык использует такого типа оператор **&*** для представления некоммутативного произведения, тогда как все *другие* идентификаторы **&Символ** вышеописанного формата рассматриваются языком в качестве ключевых слов для идентификации пользовательских операторов.

Пользовательский оператор можно использовать в качестве унарного префиксного, бинарного инфиксного операторов или вызова процедуры/функции. В *любом* из этих указанных случаев производится вызов процедуры, чье определение имеет формат:

``&Символ` := proc(x, y, z, ...) ... end proc { : | ; }`

при этом, в случае одного формального аргумента получаем унарный префиксный оператор, двух аргументов – бинарный инфиксный оператор и более двух – *n*-арный (*n* ≥ 3) префиксный оператор; в *любом* из этих случаев определен вызов **&Символ(x, y, z, ...)**. При этом, *Maple*-язык не накладывает на пользовательские операторы *специальной* семантики и рассматривает идентификатор оператора в качестве имени процедуры, соответствующей ему. В общем случае, пользовательский оператор является *n*-арным *префиксным* оператором или *вызовом n-арной* функции, т.е. следующие 2 конструкции эквивалентны в среде *Maple*-языка пакета, а именно:

`&Символ (x1, x2, ..., xn) ≡ &Символ(x1, x2, ..., xn)` **(n ≥ 1)**

Простой фрагмент иллюстрирует рассмотренные типы *пользовательских* операторов:

```

> `&Cs`:=proc(x) subs(I=-I, evalc(x)) end proc: z:=(a+b*I)/(c-d*I)*I+h*I: [&Cs z, &Cs(z)]:
> collect(simplify(expand(%)), distributed);

$$\left[ -\frac{b c + a d + a c I - b d I + h c^2 I + h d^2 I}{c^2 + d^2}, -\frac{b c + a d + a c I - b d I + h c^2 I + h d^2 I}{c^2 + d^2} \right]$$

> `&Kr` := proc(x::numeric, y::numeric) evalf(sqrt(x*y)/(x + y) + sin(x*y)) end proc:
> [1942 &Kr 2010, &Kr(1942, 2010)];  $\Rightarrow$  [-0.250548964, -0.250548964]
> `&Art`:=proc() sqrt(product(args['k'], 'k'=1..nargs))/sum(args['k'], 'k'=1..nargs) end proc:
> 257*&Art (63, 68, 43, 14, 21, 48), 257*&Art(63, 68, 43, 14, 21, 48);  $\Rightarrow$  504  $\sqrt{10234}$ , 504  $\sqrt{10234}$ 

```

Первый пример представляет унарный префиксный **&Cs**-оператор, чье применение к *комплексному* числу возвращает *сопряженное* ему число. Второй пример представляет бинарный инфиксный **&Kr**-оператор, определенный над 2 числовыми значениями и возвращающий значение, вычисляемое по заданной формуле. Между тем как третий пример иллюстрирует *n*-арный префиксный **&Art**-оператор, обеспечивающий над выражениями вычислительную процедуру также *формульного* характера. Каждый из приведенных примеров иллюстрирует применение соответствующего оператора как в *традиционной* для него нотации, так и в форме вызова соответствующей процедуры. Итак, в среде **Maple**-языка *пользовательский (нейтральный) &-оператор* представляется вызовом соответствующей процедуры. При этом, *инфиксная* нотация допустима лишь для случая двух операндов, тогда как *префиксная* допустима для произвольного числа операндов. Описанный метод определения пользовательских **&-операторов** с учетом механизма процедур **Maple**-языка достаточно прозрачен и имеет важное прикладное значение, позволяя вводить собственные **&-операторы** для специальных операций. В качестве одного из таких приложений рассмотренного метода был приведен пример **&ma** оператора, приведем еще несколько достаточно полезных примеров.

```

> &Shift := proc() local k;
  `if`(nargs < 3, ERROR("incorrect quantity <%1> of actual arguments", nargs),
  `if`(not(type(args[1], 'symbol') and type(args[2], 'list')),
  ERROR("incorrect type of the first argument %1 and/or the second argument %2",
  args[1], args[2]), `if`(nops(args[2]) <> nargs - 2, ERROR("incorrect quantity of shifts
  <%1> and/or leading variables of function <%2>", nops(args[2]), nargs - 2),
  `if`(member(false, {op(map(type, [args['k'] $ ('k' = 3 .. nargs)], name)))} = true,
  ERROR("incorrect types of actual arguments %1", 3 .. nargs), NULL)))));
  args[1]((args['k' + 2] + args[2]['k']) $ ('k' = 1 .. nops(args[2])))
end proc:
> &Shift (Ar, [a, b, c, d, g, s, q], x, y, z, t, u, r, h);  $\Rightarrow$  Ar(x+a, y+b, z+c, t+d, u+g, r+s, h+q)
> &Shift (Ar, k, x, y, 68, z);
Error, (in &Shift) incorrect type of the first argument Ar and/or the second argument k
> lop3 := proc(O::symbol, x::{0, 1, 2}, y::{0, 1, 2})
  if O = 'Inv' then if x = 0 then 2 else x mod 2 end if
  elif O = 'Dis' then if [x, y] = [0, 0] then 0 elif type([x, y], list([0, 1])) then 1 else 2 end if

```

```

elif O = 'Con' then if [x, y] = [2, 2] then 2 elif x*y = 0 then 0 else 1 end if
elif O = 'Webb' then if [x, y] = [0, 0] then 1 elif type([x, y], list({0, 1})) then 2 else 0 end if
else error "Operation <%1> is not stipulated", O
end if
end proc:

```

```
> lop3(Inv, 0), lop3(Dis, 0, 2), lop3(Con, 1, 2), lop3(Webb, 1, 1); ⇒ 2, 2, 1, 2
```

```
> lop3(Art, 0, 2);
```

```
Error, (in lop3) Operation <Art> is not stipulated
```

Вышепредставленный фрагмент иллюстрирует применение описанного способа для реализации пользовательского оператора сдвига **&Shift** для функции от нескольких переменных, определяемого следующим соотношением, а именно:

$$\&Shift\ (G, [h_1, h_2, \dots, h_n], x_1, x_2, \dots, x_n) \Rightarrow G(x_1 + h_1, x_2 + h_2, \dots, x_n + h_n)$$

Первый операнд определяет имя функции, второй определяет список величин сдвигов по ведущим переменным функции и третий – последовательность ведущих переменных. Между элементами второго и третьего операндов предполагается взаимно-однозначное соответствие. Наряду с рядом полезных приемов, использованных при определении этого пользовательского **&**-оператора, иллюстрируется и пример анализа операндов, над которыми определен оператор, на корректность. Предыдущий пример приводит определение оператора и некоторые результаты его использования для реализации указанного выше оператора функционального сдвига **&Shift**.

Второй пример фрагмента определяет процедуру **lop3(O,x,y)**, которая поддерживает следующие 4 операции (O) 3-значной логики над переменными (x,y) из множества {0, 1, 2}, а именно: **Inv** – инверсия, **Dis** – дизъюнкция, **Con** – конъюнкция и **Webb** – Вебба функция. Данная процедура может быть довольно несложно расширена и на другие операции 3-значной логики (а в общем случае и k-значной), что позволит использовать ее для представления любых многозначных функций алгебры логики. Читателю же в качестве весьма полезного упражнения рекомендуется, используя описанный подход, запрограммировать несколько **&**-операторов и процедур, определяющих какие-либо интересные нестандартные операции над данными и/или структурами данных, как рассмотренных, так и других практически полезных типов.

Как стандартная функция **member** пакета **Maple**, так и наша обобщающая процедура **belong** [45] не позволяют корректно тестировать любой вложенный список на предмет наличия в нем искомого элемента. Вызов следующей процедуры **member1(L, x {, y})** не только корректно решает вопрос принадлежности x-выражения произвольному списку L, но и через третий необязательный аргумент y возвращает число вхождений x в L.

```
> L := [[V, G, S], [68, 63, 72], [Art, Kr, 2], [42, 47, 72, 89, 96]]: belong(72, L), member(72, L);
false, false
```

```
> member1 := proc(L:: {list, Vector, vector}, x::anything) local a, b, k;
a, b := 0, subs(list = NULL, [OP(convert(L, list))]);
seq('if (x = b[k], assign('a' = a + 1), NULL), k = 1 .. nops(b));
```

```

    if a = 0 then false else `if`(nargs = 3 and type(args[3], symbol),
        assign(args[3] = a), NULL), true
end if
end proc:
> member1(L, 72), [member1(L, 72, 'h'), h]; ⇒ true, [true, 2]

```

Наряду с вложенными списками процедура **member1** обеспечивает также корректное тестирование объектов L типа $\{Vector, vector\}$. Более того, если для вложенных списков и объектов $Vector$ -типа можно использовать встроенную функцию *numboccur*, то уже для объектов $vector$ -типа она не работает, требуя использования процедуры *member1*.

При работе со вложенными списками/множествами достаточно полезными могут быть и две следующие процедуры. Вызов процедуры *DecLevLS(LS, n)* возвращает n -уровень вложенности списка/множества LS , если n – целое положительное, и результат снятия всех уровней вложенности для списка/множества LS , если $n = 0$; если n – символ (имя), то вызов процедуры возвращает элемент максимального уровня вложенности для LS , тогда как через n -аргумент возвращается максимальный уровень вложенности. Более того, невложенный список/множество имеет нулевой уровень вложенности, тогда как первый уровень вложенности определяет исходный список/множество LS .

```

> DecLevLS := proc(LS::{set, list}, n::{nonnegint, symbol})
    local a, b, c, k, p;
    if n = 0 then return convert([parse([Sub_st(`if`(whattype(LS) = set, ["[" = "", "]" = ""],
        ["[" = "", "]" = ""]), convert(LS, string), d)][1]), whattype(LS))
    else a, p, b, c := LS, 0, `if`(type(LS, list), [], {}), whattype(LS)
    end if;
    while type(a, `if`(c = list, nestlist, nestset)) do p := p + 1;
        if type(n, posint) and p = n then return a end if;
        for k to nops(a) do
            if type(a[k], c) then
                if c = list then b := [op(b), op(a[k])] else b := {op(b), op(a[k])} end if
            end if
        end do;
        assign('a' = b, 'b' = [])
    end do;
    a, `if`(type(n, symbol), assign(args[2] = p + 1), NULL)
end proc:
> L := [1,2,5,6,8,4, [a,b,8], 9,6,78, [a, [b, [a,b,8], 8, [a,b, [a,b,8], 8], 8], 8, 8,12,8];
> DecLevLS(L, 'y'), y; ⇒ [a, b, 8], 5
> DecLevLS(L, 2); ⇒ [a, b, 8, a, [b, [a, b, 8], 8, [a, b, [a, b, 8], 8], 8], 8]
> DecLevLS(L, 0); ⇒ [1, 2, 5, 6, 8, 4, a, b, 8, 9, 6, 78, a, b, a, b, 8, 8, a, b, a, b, 8, 8, 8, 8, 12, 8]
> member2 := proc(LS::{list, set}, x::anything)
    local a, b, c, k, j, T;

```

```

if not (type(LS, nestlist) or type(LS, nestset)) then a := numboccur(LS, x);
  if a <> 0 then return true, `if`(nargs = 3 and type(args[3], symbol),
    assign(args[3] = a), NULL)
  else return false
  end if
else a, T := [DecLevLS(LS, b), b][2], table([])
end if;
for k to a do c := DecLevLS(LS, k); T[k] := 0; for j to nops(c) do
  if c[j] = x then T[k] := T[k] + 1 end if end do end do;
if map(op, {entries(T)}) = {0} then false else true, `if`(nargs = 3 and type(args[3], symbol),
  assign(args[3] = eval(T)), NULL)
end if
end proc:
> L := [1,72,5,6,72,4, [a,b,72,72,8], 9,6,78, [a, [b, [a,72,8], 8, [a,b, [a,b,8], 8], 72], 8], 8,72,18]:
> member2(L, 72), [member2(L, 72, 'h'), eval(h)], numboccur(L, 72);
      true, [true, table([1 = 3, 2 = 2, 3 = 1, 4 = 1, 5 = 0])], 7
> S := {1, 4, 5, 6, 7, 8, 9, 18, 63, 68, 72, {8,68,72,a,b}, {a, {b, {8,72,a}}}, {8, {7,72, {8,a,b, {8,72,a}}}}}:
> member2(S, 72), [member2(S, 72, 'h'), eval(h)], numboccur(S, 72);
      true, [true, table([1 = 1, 2 = 1, 3 = 1, 4 = 1, 5 = 1])], 5

```

Тогда как вызов процедуры *member2(LS, x)*, использующей предыдущую процедуру, возвращает *true*, если *x*-выражение принадлежит любому из уровней вложенности *LS* и *false* в противном случае; при этом, в случае кодирования третьего необязательного аргумента *symbol*-типа через него возвращается количество вхождений *x*-выражения в каждый из уровней вложенности списка/множества *LS*. При этом, невложенный список/множество имеет нулевой уровень вложенности, тогда как первый уровень вложенности определяет исходный список/множество *LS* в вызове процедуры.

При этом, еще раз следует акцентировать внимание на том моменте, что большинство представленных в данной книге примеров *Maple*-процедур, впрочем, как и процедур *Mathematica*, используют не только стандартные средства, но и средства, созданные в процессе использования и апробации обоих пакетов, которые можно найти в [45,90].

С большим числом других достаточно интересных и полезных процедур различного назначения заинтересованный пользователь может в Библиотеке [45]. Большинство из них создавалось в целях расширения возможностей соответствующих релизов пакета и устранения некоторых их недостатков, поэтому использованные для этого приемы и методика во многом релизо-независимы, позволяя использовать их и для хорошего освоения программной среды пакета *Maple* безотносительно его релиза. К тому же, как показывает опыт, целый ряд из них хорошо дополняют аналогичные средства пакета, что дает возможность использовать их по прямому назначению при программировании собственных программных средств в среде пакета. В целом весьма значительная часть из них сохраняет свою актуальность и для последних *Maple*-релизов. На наш взгляд, наиболее ценным в настоящей Библиотеке является даже не столько сам набор пусть

и достаточно полезных дополнительных средств, который постоянно расширяется и обновляется, а именно используемые представленными в ней средствами способы и методика программирования, чье действие является достаточно пролонгированным. Именно это обстоятельство наиболее существенно для практикующих в среде *Maple* программистов, ибо представленные в нашей библиотеке средства могут устаревать, появляться в том или ином виде в стандартных поставках, либо не в совсем полной мере удовлетворять запросы пользователя, тогда как используемые в них методика и приемы носят *общий* характер и могут оказаться весьма полезными и эффективными при практическом программировании в *Maple*. В настоящее время они используются нами для разработки разнообразных *Maple*-приложений в целом ряде стратегически важных естественно-научных областей (*математика, физика, кибернетика, технология, информатика и др.*). Тогда как пользователь данной Библиотеки может использовать ее средства не только по их прямому назначению, но и на основе анализа их *исходных* текстов (*прилагаются к Библиотеке*) освоить целый ряд достаточно полезных приемов и методов программирования в среде математического пакета *Maple*.

В завершении раздела целесообразно сделать *одно* довольно полезное в практическом отношении замечание. Определения процедур можно объединять в *модули*, помещая их в качестве элементов (*имя процедуры – вход, ее определение – выход*) *модульных* таблиц, как это весьма наглядно иллюстрирует следующий достаточно простой фрагмент:

```
> PT[Sr] := () -> `+`(args)/nargs: PT[H] := () -> sqrt(`*(args)): PT[G] := () -> (`+`(args))^2:
> PT[Ds] := proc() local k; sum((args[k] - Sr(args))^2, k = 1 .. nargs)/nargs end proc:
> type(PT, table), whattype(eval(PT)), eval(PT);  => true, table
table([G = (( ) -> `+`(args)^2), Ds = (proc() local k; sum((args[k] - Sr(args))^2, k = 1 .. nargs) / nargs end proc),
  H = (( ) -> sqrt(`*(args))),
  Sr = (( ) -> `+`(args)/nargs)
])
> with(PT); PT[G](68,63,43,14,21,48), 6*Sr(68,63,43,14,21,48), PT[H](68,63,43,14,21,48)^2,
36*PT[Ds](68,63,43,14,21,48);  => [Ds, G, H, Sr]      66049, 257, 2599599744, 14249
```

Такой простой прием можно применять как для создания пользовательских модулей, так и для организации отложенных определений процедур, что в целом ряде случаев позволяет создавать более эффективные программные *Maple*-средства. Именно такой метод был использован в ряде средств нашей Библиотеки [45], а также при разработке некоторых важных приложений физико-технического характера [8-14,28-38,44].

Между тем, своей политикой обновления пакета, выражающейся в частом появлении новых релизов, которые по целому ряду показателей не совместимы «*сверху-вниз*», его разработчики *весьма* нередко ставят в тупик создателей средств как специального, так и широкого назначения. Более того, обновления обусловлены не только улучшением алгоритмов процедур, но и изменением статуса средств, не имеющими обоснованных объяснений модификациями и т.д. И что самое неприятное для пользователя, так это то, что большинство их вообще не анонсируется. Вот лишь один из многочисленных примеров результата такого типа модификаций, и это далеко не худший вариант.

```

> ParProc(cos);                                     # Maple 8
[
    Arguments = (x::algebraic)
    locals = (n, t)
    options = (Copyright (c) 1992 by the University of Waterloo. All rights reserved. )
    rem_tab = table([0 = 1,  $\frac{\pi}{3} = \frac{1}{2}$ ,  $\infty = \text{undefined}$ ,  $-\infty = \text{undefined}$ ,  $\frac{\pi}{6} = \frac{\sqrt{3}}{2}$ ,  $\frac{\pi}{2} = 0$ ,  $I = \cosh(1)$ ,  $\pi = -1$ ,  $\frac{\pi}{4} = \frac{\sqrt{2}}{2}$ ])
]

> ParProc(cos);                                     # Maple 10
[
    Arguments = (x::algebraic)
    locals = (res, n, t)
    options = (automap(1), Copyright (c) 1992 by the University of Waterloo. All rights reserved. )
    rem_tab = Cache(512, 'permanent' = [0 = 1,  $\frac{\pi}{3} = \frac{1}{2}$ ,  $I = \cosh(1)$ ,  $-\infty = \text{undefined}$ ,  $\frac{\pi}{2} = 0$ ,  $\frac{\pi}{4} = \frac{\sqrt{2}}{2}$ ,  $\infty = \text{undefined}$ ,  $\frac{\pi}{6} = \frac{\sqrt{3}}{2}$ ,  $\pi = -1$ ])
]

> ParProc(cos);                                     # Maple 11
builtin function, 427119921110011011211212211111012112010120010112121111211221

```

Из представленного фрагмента со всей очевидностью вытекает, что наша процедура **ParProc(W1)**, предназначенная для проверки параметров процедуры, программного либо пакетного модуля **W1**, на трех релизах 8 – 11 пакета дает различные результаты, маркируясь, между тем, одним и тем же 1992-м годом. Из данного фрагмента следует, что процедура **cos** изменялась не только на внутреннем уровне, но затем и вовсе была переведена на уровень *встроенной* функции с весьма внушительным номером. Такого типа примеров немало. Естественно, рядовому пользователю это не является помехой при использовании пакета для него главное требование – корректность процедуры и ее реактивность, тогда как для разработчиков средств, базирующихся на внутренней кухне пакета такие недокументированные изменения чреваты неприятностями. При этом, описанные новшества пакета (*от релиза к релизу*) далеко не самые худшие, в ряде случаев они значительно существеннее, предполагая переработку (*в целом ряде случаев нетривиальную*) средств, созданных с использованием скрытых возможностей пакета. Между тем, нами в целях расширения функциональных возможностей **Maple** наряду с устранением ошибок ряда его средств была создана довольно обширная Библиотека процедур, ряд из средств которой предназначен именно для расширенной работы с процедурными и модульными объектами пакета. Достаточно детально можно с ними ознакомиться в наших работах [8-14,28-38,42-45]. Некоторые из них рассматриваются в следующем разделе в качестве иллюстративных примеров оформления процедур.

6.1.8. Иллюстративные примеры оформления Maple-процедур

Представленное описание структурной организации **Maple**-процедур иллюстрируется нижеследующими примерами, отражающими основные ее элементы и принципы, что позволяет с учетом вышерассмотренного материала уже непосредственно приступать к созданию, на первых порах, относительно несложных процедур пользователя разного назначения. В качестве иллюстративных примеров был взят ряд процедур из нашей Библиотеки [45], содержащих практически полезную методику программирования в

среде *Maple*-языка пакета. Между тем, примеры представляют процедуры, которые имеют непосредственный прикладной интерес для программирования приложений.

В качестве первого примера приведем процедуру ``convert/proc``, иллюстрирующую и подход к расширению стандартных средств, и достаточно полезную в практическом отношении. Процедура обеспечивает конвертирование равенства или их множества/списка в процедуру или множество/список. При этом, левые части равенств должны быть функциями вида $f(x, y, z, \dots)$, тогда как правые части – любыми алгебраическими выражениями. В случае *одного равенства* вызов процедуры `convert(x, `proc`)` возвращает *тело* процедуры, в остальных случаях возвращается *NULL*-значение с обеспечением требуемой конвертации. Ниже следующий фрагмент иллюстрирует исходный текст процедуры и некоторые типичные примеры ее использования.

```
> `convert/proc` := proc(x::{equation, list(equation), set(equation)}) local a, b;
  a := proc(x) if not type(lhs(x), 'function'('symbol')) then return x
    else parse(cat("", op(0, lhs(x)), ":=", proc(" convert([op(lhs(x)), 'string')[2 .. -2], ") ",
      convert(rhs(x), 'string'), " end proc;"), 'statement')
    end if
  end proc;
  if type(x, 'equation') then a(x) else for b in x do a(b); NULL end do end if
end proc;

> convert(y(x, y) = sqrt(x^2 + y^2), `proc`);  => proc(x, y) (x^2 + y^2)^(1/2) end proc
> evalf(y(68, 63)), convert(a = b, `proc`);    => 92.69843580, a = b
> convert({v(x) = sin(x), z(h) = cos(h)}, `proc`); eval(v), eval(z);
      proc(x) sin(x) end proc, proc(h) cos(h) end proc
```

Представленная процедура ``convert/proc`` весьма полезна в целом ряде приложений, в частности, при работе с дифференциальными уравнениями. В качестве достаточно полезного упражнения рекомендуется рассмотреть организацию данной небольшой процедуры. Ниже будет представлен ряд других процедур, иллюстрирующих те или иные аспекты *Maple*-языка. Большое же количество самых разнообразных процедур в исходном виде, использующих немало достаточно полезных, эффективных (*а в целом ряде случаев и нестандартных*) приемов программирования в среде *Maple*, может быть найдено в архиве, поставляемом с уже упомянутой нашей Библиотекой [45].

В ряде случаев возникает потребность определения имен, которым в текущем сеансе были присвоены заданные выражения. Данная задача решается процедурой *nvalue*.

```
> nvalue := proc(A::anything) local a, b, c, d, f, k, t, h;
  assign(a = (b -> `if`(search(cat("", b), "RTABLE_SAVE/"), NULL, b)), d = {},
  f = cat(currentdir(), "/$Art22_Kr15$"));
  if nargs = 0 then {seq(`if`(eval(cat(`, k)) = NULL, cat(`, k), NULL),
    k = map(convert, {anames()}, 'string'))}
  elif whattype(args) = 'exprseq' then a := map(convert, {anames()}, 'string');
    {seq(`if`([eval(cat(`, k)]) = [args], cat(`, k), NULL), k = a)}
```

```

elif type(A, `module`) then assign('a' = sstr(["\n" = NULL], convert(A, 'string')),
    'b' = {anames(`module`)});
for k to nops(b) do (proc(x) save args, f end proc)(b[k]);
    assign('c' = sstr(["\n" = NULL], readbytes(f, infinity, 'TEXT')[1 .. -3])),
        search(c, ":", 't'), fremove(f);
    if c[t + 3 .. -1] = a then d := {op(d), cat(`, c[1 .. t - 2])} end if
end do;
d
elif type(eval(A), {function, Matrix, Vector, matrix, vector, Array, array, table, `module`, range})
then map(a, {seq(`if` (convert(eval(k), list) = convert(eval(A), 'list'), k, NULL),
    k = {anames(whattype(eval(A))))})
else {seq(`if` (evalb(eval(k) = A), k, NULL), k = {anames(whattype(eval(A))))})
end if
end proc:
> a:=68: b:=63: c:="ransian68": c9:= "ransian68": d:= table([1=68, 2=63, 3=43]): L:= [x, y, z]:
t47:= 19.42: assign('h' = `svegal`): t47:= 19.42: t42:= 19.42: R:= a+b*I: Z:= (a1+b1)/(c1+d1):
B:= a+b*I: g:=proc() `+(args)/nargs end proc: r:= x..y: m:= x..y: n:=x..y: Grodno:=NULL:
Lasnamae := NULL: Tallinn := NULL: Vilnius := NULL: Moscow := NULL: f := cos(x):
> map(nvalue, [68, "ransian68", table([1=68, 2=63, 3=43]), svegal, 19.42, [x, y, z], a + b*I,
(a1 + b1)/(c1 + d1), cos(x), proc () `+(args)/nargs end proc, x..y]), nvalue());
[{a}, {c, c9}, {d}, {h}, {t47, t42}, {L}, {R, B}, {Z}, {f}, {g}, {r, n, m}], {Lasnamae, Tallinn, Grodno,
Vilnius, Moscow}

```

Эта задача решается процедурой, базирующейся на стандартной процедуре *anames* с использованием некоторых особенностей *Maple* [11,42-44]. Вызов процедуры *nvalue(t)* возвращает список глобальных имен, которым в текущем сеансе пакета было присвоено выражение *t*, тогда как вызов *nvalue()* возвращает список глобальных имен, которым в текущем сеансе присваивалось *NULL*-значение. Вызов процедуры возвращает пустое множество, т.е. {}, если в текущем сеансе никаких назначений не производилось.

В ряде случаев возникает задача определения наличия вызовов заданной процедуры/функции *N* в некоторой процедуре *P*. Точнее нас будут интересовать форматы таких вызовов. Данная задача решается процедурой *extrcalls*, представленной ниже.

```

> extrcalls := proc(P::procedure, N::symbol) local a, b, c, d, k, p, Res;
    assign67(a = "" || N || "(", Res = NULL, b = convert(eval(P), 'string'));
    unassign('_Lab'), assign(d = length(a), c = Search2(b, {a}));
    if c = [] then error "call of the form <%1> does not exist", "" || a || "...)" else
        for k in c do p := d;
            _Lab; try parse(cat(b[k .. k + p]))
            catch: p := p + 1; goto(_Lab)
            end try;
            Res := Res, b[k .. k + p]
        end do;
    end do;

```

```

[Res]
end if
end proc:
> Proc := proc() local a, b, c, d, y, x, v, x1, y1, w, g; a := 68; assign(x = 68, y = (a+b)/(c+d));
sin(sqrt(x+y)*(a+b)/(c-d)); writeline(f, "GRSU"); assign(v=42, 'h'=(a+b)^2*(x1+y1)/(c+d));
Close(f); assign('w'=63, 'g'=(a+b)^2*(x1+y1)/(c+d)); sin(a1+b1); a*x*y*sin(x+y) end proc:
> extrcalls(Proc,assign), extrcalls(Proc,sin), extrcalls(Proc,writeline), extrcalls(Proc,Close);
extrcalls(Proc, cos);
["assign(x = 68, y = (a+b)/(c+d))", "assign(v = 42, ('h') = (a+b)^2*(x1+y1)/(c+d))",
"assign(('w') = 63, ('g') = (a+b)^2*(x1+y1)/(c+d))", ["sin(sqrt(y+x)*(a+b)/(c-d))",
"sin(a1+b1)", "sin(y+x)", ["writeline(f, "GRSU")"], ["Close(f)"]
Error, (in extrcalls) call of the form <cos(...)> does not exist

```

Вызов процедуры *extrcalls(P,N)* возвращает список вызовов функции *N*, используемых процедурой *P*. Вызовы *N* возвращаются в строчном формате, чтобы не инициировать реальных вызовов вне тела процедуры. В отсутствие искоемых вызовов инициируется ошибочная ситуация с диагностикой "call of the form <%1> does not exist". Процедура имеет целый ряд достаточно полезных применений при разработке приложений.

Как уже отмечалось, *Maple*-язык допускает использование встроенной *goto*-функции безусловных переходов. Поэтому в ряде случаев возникает задача проверки на предмет наличия в процедуре данного типа вызовов. Вызов процедуры *isplabel(P)* возвращает значение *true*, если процедура *P* использует вызов функции *goto*; в противном случае возвращается *false*-значение. Тогда как вызов процедуры *isplabel(P,T)* с *T*-аргументом обеспечивает возврат через него множества имен всех меток, обусловленных вызовами *goto* в процедуре *P*. При этом, процедура обеспечивает проверку меток процедуры *P* на допустимость. В случае обнаружения недопустимости выводится соответствующее сообщение. Для обеспечения корректности использования в процедуре меток может быть использована процедура *Lprot* [45], которая обеспечивает присвоение *protected*-атрибута всем меткам указанной процедуры. Ниже приведен исходный код *isplabel*-процедуры и пример ее конкретного применения для тестирования *goto*-переходов.

```

> isplabel := proc(P::procedure) local a, b, c, d, k, p, h, t;
assign(d = {}, a = interface(verboseproc), interface(verboseproc = 3);
assign(b = convert(eval(P), 'string'), p = {}), assign(c = Search2(b, {" goto(")},
null(interface(verboseproc = a)));
if c = [] then false else
if search(b[1..c[1]], "unassign(", 't') then h:={parse(b[t + 8 .. nexts(b, t, ")] [2])} end if;
for k in c do d := {op(d), cat(`, b[k + 6 .. nexts(b, k, ")] [2] - 1)} end do;
true, seq(`if` (type(eval(d[k]), 'symbol'), NULL, assign('p'={d[k], op(p)})), k=1..nops(d)),
`if` (p = {}, NULL, `if` (map(eval, map(eval, h)) = map(eval, d), NULL,
WARNING("procedure <%1> contains invalid labels %2; error is possible at the
procedure call", P, `if` (type(h, 'symbol'), p, p minus h))),
`if` (1 < nargs and type(args[2], 'symbol'), assign(args[2] = d), NULL)

```

```
end if
end proc:
```

```
> isplabel(MkDir), isplabel(mwsname,R),R; ⇒ false, true, {VGS_vanaduspension_14062005}
```

Немало задач включают различные операции над строчными данными. Пакет *Maple* для этих целей включает ряд средств пакетного модуля **StringTools**. Наша процедура *swmpat(Str, pat, p, d {, h})* возвращает *true*-значение тогда и только тогда, когда строка *Str* содержит вхождения подстрок, соответствующих шаблону *pat* с группирующими символами, определенными четвертым фактическим *d*-аргументом процедуры.

```
> swmpat := proc(S::{string, symbol}, m::{string, symbol}, p::list(posint), d::{string, symbol})
local a, b, c, C, j, k, h, s, s1, m1, d1, v, r, res, nu, n, omega, t, epsilon, x, y;
  assign67(c = {args} minus {S, m, insensitive, p, d}, s = convert([7], 'bytes'), y = args);
  C := (x, y) -> `if`(member(Insensitive, {args}), Case(x), x);
  if not search(m, d) then h := Search2(C(S, args), {C(m, args)});
    if h <> [] then RETURN(true, `if`(c = {}, NULL, `if`(type(c[1], 'assignable1'), assign(c[1] = h),
      WARNING("argument %1 should be symbol but has received %2", c[1],
        whattype(eval(c[1])))))
    else RETURN(false)
    end if
  else
    assign(nu = ((x, n) -> cat(x $ (b = 1 .. n))), omega = (t -> `if`(t = 0, 0, 1)));
    epsilon := proc(x, y) local k; [seq(`if`(x[k] = y[k], 0, 1), k = 1 .. nops(x))] end proc
  end if;
  assign(s1 = cat("", S), m1 = cat("", m), d1 = cat("", d)[1], v = [], h = "", r = [], res = false, a = 0);
  for k to length(m1) do
    try
      if m1[k] <> d1 then h := cat(h, m1[k]); v := [op(v), 0]
      else a := a + 1; h := cat(h, nu(s, p[a])); v := [op(v), 1 $ (j = 1 .. p[a])]
      end if
    catch "invalid subscript selector": h := cat(h, s); v := [op(v), 1]; next
    end try
  end do;
  assign('h' = convert(C(h, args), 'list1'), 's1' = convert(C(s1, args), 'list1'), assign(t = nops(h));
  for k to nops(s1) - t + 1 do
    if epsilon(s1[k .. k + t - 1], h) = v then
      res := true; r := [op(r), [k, k + t - 1]]; k := k + t + 1
    end if
  end do;
  res, `if`(c = {}, NULL, `if`(type(c[1], 'assignable1'), assign(c[1] = r),
    WARNING("argument %1 should be symbol but has received %2", c[1],
      whattype(eval(c[1])))))
end proc:
```

```
> S := "avz1942agn1947art1986kr1996svet1967art1986kr1996svet": m := "a*1986*svet":
p:= [2,6]: [swmpat(S, m, p, "", z), z], [swmpat(S, "art1986kr", [7, 14], "", r), r];
[true, [[15, 31], [36, 52]]], [true, [15, 36]]
```

Тогда как третий *p*-аргумент задает список кратностей соответствующих вхождений группирующих символов в *m*-шаблон. Если же вызов процедуры *swmpat*(*S, m, p, d, h*) использует пятый необязательный *h*-аргумент и возвращает *true*-значение, то через *h* возвращается вложенный список, чьи 2-элементные подписки определяют первую и последнюю позиции непересекающихся подстрок *S*, соответствующих *m*-шаблону. В случае отсутствия в *m*-шаблоне каких-либо *группирующих* символов через *h*-аргумент возвращается целочисленный список, чьи элементы определяют первые позиции для непересекающихся подстрок строки *S*, соответствующих *m*-шаблону. Более детально с процедурой *swmpat* и ее модификацией *swmpat1* можно ознакомиться в [42-45].

Вызов процедуры *Red_n*(*W, G, N*) возвращает результат сведения кратных вхождений в строку/символ *W* символов/строк, задаваемых *G*-аргументом, в качестве которого могут выступать строка, символ или список строк/символов, к кратности не большей, чем *N*, в качестве которого могут выступать положительное целое либо их список.

```
> Red_n := proc(S::{string, symbol}, G::{string, symbol, list({string, symbol})},
N::{posint, list(posint)})
local k, h, Lambda, z, g, n;
if type(G, {'symbol', 'string'}) then g := G; n := `if`(type(N, 'posint'), N, N[1])
else h := S;
for k to nops(G) do
try n := N[k]; h := procname(h, G[k], n)
catch "invalid subscript selector": h := procname(h, G[k], `if`(type(N, 'list'), 2, N))
catch "invalid input: %1 expects": h := procname(h, G[k], `if`(type(N, 'list'), 2, N))
end try
end do;
return h
end if;
`if`(length(g) < 1, ERROR("length of <%1> should be more than 1", g),
assign(z = convert([2], 'bytes')));
Lambda := proc(S, g, n) local a, b, h, k, p, t;
`if`(search(S, g), assign(t = cat(convert([1], 'bytes') $(k = 1 .. n - 1))),
RETURN(S, WARNING("substring <%1> does not exist in string <%2>", g, S)));
assign(h = "", a = cat("", S, t), b = cat("", g $(k = 1 .. n)), p = 0);
do
seq(assign('h'=cat(h, `if`(a[k..k+n-1]=b, assign('p'=p+1), a[k])), k=1..length(a) - n+1);
if p = 0 then break else p := 0; a := cat(h, t); h := "" end if
end do;
h
end proc;
```

```

if length(g) = 1 then h := Lambda(S, g, n, g)
else h := Subs_All(z = g, Lambda(Subs_All(g = z, S, 2), z, n, g), 2)
end if;
convert(h, whattype(S))
end proc:
> Red_n('aaaaaaaa14aaaaaagnaaaaa21aaaaaaaaahhhaaaaaartaakr', `a`, 6);
          aaaaa14aaaaaagnaaaaa21aaaaahhhhaaaaaartaakr
> Red_n("aaccbcbccccbccccccccckcccccccc", "cccc", 1);  => "aabcbbkk"

```

В частности, в случае $N = \{1 \mid 2\}$ строка/символ G удаляется из строки/символа S либо сохраняется с кратностью 1 соответственно. При этом, тип результата, возвращаемого процедурой *Red_n*, соответствует типу S -аргумента. Процедура выполняет регистрозависимый поиск. Если G не входит в S , то вызов процедуры *Red_n(S, G, N)* возвращает *первый* аргумент S без обработки с выводом соответствующего сообщения. Процедура *Red_n* обрабатывает возникающие основные особые и ошибочные ситуации.

Если 2-й и 3-й аргументы определяют списки с взаимно однозначным соответствием их элементов, то сведение кратностей производится по всем элементам списка G для соответствующих им кратностям из списка N . Более того, если $nops(G) > nops(N)$, тогда последние $nops(G) - nops(N)$ элементов из строки/символа G получают кратность *один* в возвращаемом результате. Если же G – список и N – положительное целое число, то все элементы G получают одинаковую кратность N . Наконец, если G – символ/строка и N – список, тогда G получает кратность $N[1]$ в возвращаемом результате. Процедура *Red_n* является достаточно полезным средством при работе со строками/символами, используется она и целым рядом средств из нашей Библиотеки [45].

Вызов процедуры *deltab(Tab, r, a)* возвращает *NULL*-значение, т.е. ничего, обеспечивая удаление из таблицы *Tab* всех элементов, которые имеют соответствие по отношению к выражению a , в зависимости от данного режима удаления r : $r=0$ – равенство входов, $r=1$ – равенство выходов и $r=2$ – наличие любого из указанных равенств. Процедура – достаточно полезное средство в целом ряде приложений, имеющих дело с *таблицами*.

```

> deltab := proc(T::table, r::{0, 1, 2}, a::anything) local b, c, d, k, p, n;
  assign(b = op(2, eval(T)), d = op(1, eval(T)), c = ((x, y) -> `if`(x(y) = args[3..-1], NULL, y)));
  assign67(n = nops(b), p = args[3 .. -1]), assign("T" = table(d, [seq(`if`(r = 0, c(lhs, b[k], p),
  `if`(r = 1, c(rhs, b[k], p), `if`(c(rhs, b[k], p) = NULL or c(lhs, b[k], p) = NULL, NULL, b[k]))),
  k = 1 .. n)));
end proc:
> T := table([x=a, y=b, z=Art, Art=21, Kr=14]): deltab(T, 0, Art), eval(T), deltab(T, 1, a),
  eval(T), deltab(T, 2, Art), eval(T);
  table([y = b, x = a, z = Art, Kr = 14]), table([y = b, z = Art, Kr = 14]), table([y = b, Kr = 14])

```

В качестве замены для стандартной *assign*-процедуры для пакета *Maple 6 - 11* может вполне выступить процедура *assign67*. Своими возможностями процедура *assign67* [9] достаточно существенно расширяет процедуру *assign*, выполняя даже те назначения, на которых даже *Maple 11* иницирует ошибочные ситуации. Следующий фрагмент

представляет исходный текст процедуры и некоторые примеры ее применения.

```

> assign67 := proc(X) local Art, Kr, k;
  `Copyright (C) 2002 by the International Academy of Noosphere. All rights reserved.`;
  Art, Kr := [], [];
  try
    for k to nargs do
      if type(args[k], 'equation') then Kr := [op(Kr), k] else NULL end if
    end do;
    if nargs = 0 then NULL elif Kr = [] and type(X, {'symbol', 'function', 'name', '::'}) then
      X := args[2 .. -1]
    elif nargs = 1 and type(X, 'equation') and nops([lhs(X)]) = nops([rhs(X)]) then
      seq(procname(op(k, [lhs(X)]), op(k, [rhs(X)])), k = 1 .. nops([lhs(X)]))
    elif nargs = 1 and type(X, {'set', 'list'}) then seq(procname(op(k, X)), k = 1 .. nops(X))
    elif type(X, '=') and Kr = [1] or type(X, {'set', 'list'}) then procname(op(X), args[2 .. -1])
    elif Kr <> [] and type(X, {'equation', 'symbol', 'function', 'name', '::'}) then
      for k to Kr[1] - 1 do procname(args[k]) end do;
      for k to nops(Kr) - 1 do procname(args[Kr[k] .. Kr[k + 1] - 1]) end do;
      procname(args[Kr[-1] .. -1])
    else procname('Art' = [seq('if (type(args[k], 'name') or type(args[k], 'equation') and
      type(lhs(args[k]), 'name'), k, NULL), k = 1 .. nargs]),
      `if (Art = [], NULL, procname(args[Art[1] .. -1]))
    end if
  catch "wrong number (or type) of parameters": seq(procname(args[Kr[k]]), k=1..nops(Kr))
end try;
NULL
end proc;

> assign(x = 68, y = 63, 43, z = 21, 14); # Maple 6 - 11
Error, (in assign) invalid arguments
> assign67(x = 68, y = 63, 43, z = 21, 14); x, [y], [z]; => 68, [63, 43], [21, 14]
> assign67(('q', 'w', 'e') = (42, 47)); [q], [w], [e]; => [w, e, 42, 47], [w], [e]
> assign67(1,2,3); assign67(); assign(NULL); assign(Z), Z, assign67(ArtKr=NULL), ArtKr;
> assign67(('q', 'w', 'e') = (14, 21, 6)); assign67(('x', 'y', 'z') = (68, 63, 43)), [x, y, z], [q], [w], [e];
[68, 63, 43], [14], [21], [6]
> assign67(V=42,68, G=47,63, S=67,43, Ar=89,21, K=96,14, A=62,48),[V],[G],[S],[Ar],[K],[A];
[42, 68], [47, 63], [67, 43], [89, 21], [96, 14], [62, 48]

```

Процедура *assign67*, расширяя возможности стандартной *assign*-процедуры релизов 6 - 11, обеспечивает дополнительно такие операции как: множественные присвоения последовательностей значений переменным, *множественные* присвоения переменным значений *NULL*, и др. Более того, вызов процедуры *assign67* является корректным на *любом* кортеже фактических аргументов, обеспечивая достаточно широкий диапазон типов и вариантов присвоений наряду с непрерывностью вычислений. Лучше всего

возможности процедуры *assign67* иллюстрируются наиболее типичными примерами ее применения [8,28-38,42-45]. Например, вызов процедуры *assign67(x1, x2, x3,...)*, если фактическими аргументами являются символы, выполняет присвоение $x1 := x2, x3, \dots$. Тогда как вызов *assign67(x1, x2,..., xk, y=a,...)* при том же предположении производит присвоения $x1 := NULL, \dots, xk := NULL$. Уже в текущей реализации процедура *assign67* во многих случаях упрощает программирование в среде *Maple*, однако она допускает и дальнейшие достаточно интересные и полезные расширения.

Вызов процедуры *type(P,fpath)* возвращает *true*-значение, если аргумент *P* определяет допустимый полный путь, и *false*-значение в противном случае. При этом, необходимо иметь в виду, возврат *true*-значения говорит только, что тестируемый путь *P*, вообще говоря, может быть потенциальным элементом файловой системы компьютера.

```
> `type/fpath` := proc(P::{string, symbol}) local a, b, f, f1, k, p, w, z, dir, df;
assign(a = CF2(P), w = interface(warnlevel), z = (x -> null(interface(warnlevel = x))), z(0);
`if` (Red_n(a, " ", 1) = "" or length(a) < 3 or a[2 .. 3] <> ":\\", [z(w), RETURN(false)],
assign(p = 97 .. 122));
z(w), `if` (member(convert(a[1], 'bytes'), {[k] $ (k = p)}), `if` (member(a[1], map(CF2,
{Adrive()})), NULL, [WARNING("<%1> is idle disk drive", Case(a[1], 'upper')),
RETURN(false)]), ERROR("<%1> - illegal logical drive name", Case(a[1], 'upper')));
assign(f = cat(WT(), "\_$$$14_06_2005$$$"), b = currentdir());
f1 := cat(f, a[3 .. -1]);
df := proc(x) for k to nops(x) do try rmdir(x[k]) catch: next end try end do end proc;
try Mkdir(f1)
catch: dir := [op(InvL(DirF(f))), f];
df(dir); currentdir(b); RETURN(false)
end try;
dir := [op(InvL(DirF(f))), f];
df(dir);
true
end proc;
> type("D:/MathMaple\\MapleMathem.doc", 'fpath'); => true
> type("@:/aaa/bbb/cc/ddvb/hhh", 'fpath');
Error, (in type/fpath) <@> - illegal logical drive name
> type("C:\\*aaa\\bbb\\cc*c\\ddvb\\hhh", 'fpath'); => false
> type("C:/aaa\\bbb/cc\\ddd/G/RansIan.txt", 'fpath'); => true
```

Данная процедура хорошо дополняет процедуру *type/path`* [42-45] и представляется достаточно полезной в целом ряде задач, имеющих дело с обработкой файлов. Она использует наряду со стандартными восемь процедур из нашей Библиотеки [45].

Для целого ряда задач весьма полезным тестирующим средством является процедура *type(P, boolproc)*, чей вызов возвращает *true*-значение, если *P* – булева процедура (*m.e.* возвращает только значения *true* или *false*), и *false*-значение в противном случае. Данная процедура производит тестирование с высокой степенью достоверности. В случае же

необходимости процедура **type(P, boolproc)** легко адаптируется на специальные типы тестируемых процедур посредством подключения дополнительных **catch**-блоков.

```
> `type/boolproc` := proc(P::anything) local _Art21_Kr14_, _avz68_, omega, nu, z;
option remember, `Copyright International Academy of Noosphere - Tallinn, November, 2003`;
`if` (P = lexorder, RETURN(true), `if` (type(eval(P), 'procedure'),
    assign(`type/_avz68_` = eval(P)), RETURN(false)));
assign(omega = (x -> interface(warnlevel = x)), z = (() -> unassign(`type/_avz68_`)));
try nu := interface(warnlevel); omega(0); type(_Art21_Kr14_, '_avz68_');
    true, null(omega(nu)), z()
catch "result from type `%1` must be true or false": RETURN(false, null(omega(nu)), z())
catch "%1 is not a procedure": RETURN(false, null(omega(nu)), z())
catch "%1 uses ": RETURN(false, null(omega(nu)), z())
catch "invalid input: %1 expects": RETURN(false, null(omega(nu)), z())
catch "quantity of actual arguments": RETURN(false, null(omega(nu)), z())
catch: RETURN(true, null(omega(nu)), z())
end try
end proc;

> map(type, [IsOpen, `type/file`, PP, isDir, IsFempty, Mkdir, lexorder, Empty, save2,
    `type/boolproc`, `type/package`, `type/dir`, 'boolproc'];
    [true, true, false, true, true, false, true, true, false, true, true, true])
```

Вызов процедуры **type(F, file)** возвращает *true*-значение при условии, что *F* есть реально существующий файл, и *false*-значение в противном случае. При этом, следует иметь в виду, что возврат *false* говорит только о том, что тестируемый файл не: (1) последний элемент в цепочке каталогов *F*, (2) в текущем каталоге и (3) среди открытых файлов. В целом ряде задач, связанных с обработкой файлов, это средство оказывается довольно востребованным. Между тем, для более детального тестирования *Maple*-объектов для установления факта принадлежности к *file*-типу существует процедура **type(F, 'file1')**, которая представлена исходным текстом и примером ее использования. Более того, в тексте процедуры используется еще девять процедур из нашей Библиотеки [45].

```
> `type/file1` := proc(F::anything) local a, b, k, p, t, t1, t2, t3, h, u, omega;
`if` (type(eval(F), {'string', 'symbol', 'name'}), assign(b = iostatus()), RETURN(false));
omega, u := () -> ERROR("<%1> is invalid path to a directory or a file", F),
    interface(warnlevel);
`if` (Red_n(cat(" ", F), " ", 2) = " " or member(F, {'`, ""})), omega(),
    null(interface(warnlevel = 0)));
assign(a=Red_n(Subs_All("/"="\\", Case(cat("", F)), 1), "\\ ", 2)), null(interface(warnlevel=u));
`if` (3 < length(a) and a[-1] = "\\ ", assign('a' = a[1 .. -2]), `if` (length(a) = 2 and a[-1] = ":",
    assign('a' = cat(a, "\\ ")), NULL));
`if` (nops(b)=3, NULL, seq(`if` (CF(a)=CF(b[k][2]), RETURN(true), NULL), k=4 .. nops(b)));
`if` (type(holdof('hold'), 'set'('integer')), NULL, assign(t = 7));
```

```

if a[2 .. 3] <> ":\\\" then
  try assign67(h = cat(CCM(), "\\\", \"maplesys.ini\"), Close(h));
  do t1 := readline(h);
    if t1[1 .. 5] = \"UserD\" then t1 := t1[15 .. -1]; Close(h); break end if
  end do;
  t1, t2, t3 := cat(t1, "\\\", a), cat(CDM(), "\\\", a), cat(CCM(), "\\\", a);
  if {op(map(procname, [t1, t2, t3]))} <> {false} then RETURN(holdof('restore', 7), true)
  end if;
  `if` (nops(b) = 3, NULL, seq(`if` (Search1(CF(b[k][2]), a, 'p') and p = ['right'],
    RETURN(holdof('restore', 7), true), 7), k = 4 .. nops(b)))
  catch \"file or directory does not exist\": RETURN(false)
  catch \"file or directory, %1, does not exist\": RETURN(false)
  end try
end if;
try Close(fopen(a, 'READ'))
catch \"file already open\": `if` (t = 7, holdof('restore', 7), NULL), RETURN(true)
catch \"file or directory does not exist\": `if` (t=7, holdof('restore', 7), NULL), RETURN(false)
catch \"file or directory, %1, does not exist\": `if` (t = 7, holdof('restore', 7), NULL),
  RETURN(false)
catch \"file I/O error\": `if` (t = 7, holdof('restore', 7), NULL), RETURN(false)
catch \"permission denied\": `if` (t = 7, holdof('restore', 7), NULL), RETURN(false)
end try;
`if` (t = 7, holdof('restore', 7), NULL), true
end proc:
> type(\"F:/UserLib6789/CopyRight_Info/Miscellaneous/Aladjev1.jpg\", 'file1'); ⇒ true

```

Вызов процедуры *type(F, 'file1')* возвращает *true*-значение, если *F* определяет *реальный* файл данных и *false*-значение в противном случае. При этом, *необходимо* иметь в виду, возврат *false*-значения говорит только о том, что тестируемый файл не: (1) последний элемент в цепочке каталогов *F*, (2) в главном каталоге *Maple*, (3) в каталоге с пакетным ядром, (4) в каталоге *Users* либо (5) среди открытых файлов. Между тем, тестируемый файл, вообще говоря, может быть элементом файловой системы *ПК*. При этом, вызов этой процедуры не изменяет *текущих* атрибутов и состояния {*open* | *close*} тестируемых файлов. Процедура достаточно полезна при разработке приложений, имеющих дело с обработкой файлов данных, и неоднократно успешно использовалась в этом плане.

Встроенные стандартные функции *system* и *ssystem* позволяют выполнять целый ряд достаточно важных процедур, обеспечивая более эффективное программирование. В некоторых операционных системах, между тем, не все системные команды доступны через такой механизм. Как правило, данные *Maple*-функции имеют дело с файлами типов {'.exe', '.com'}, стандартно поставляемых с операционной системой. Между тем, множество таких файлов довольно обширно и содержит целый ряд весьма полезных средств, представляющих несомненный интерес при разработке приложений в среде

Maple. Процедуры *com_exe1* и *com_exe2* в полной мере решают вопросы пополнения операционной среды требуемыми системными командами. С описанием алгоритмов, реализуемых данными процедурами, можно ознакомиться в [9,42-45]. Тогда как ниже представлены лишь исходный текст процедуры *com_exe1* и примеры ее применения.

```

> com_exe1 := proc(P::{0, 1, path}, L::path) local a, b, c, f; global _TABprog, __ABprog;
  if P = 1 then
    if type(_TABprog, 'table') then return map(op, [indices(_TABprog)])
    else error "_TABprog is inactive in the current Maple session"
    end if
  end if;
  if type(L, 'dir') and type(L, {'mlib', 'mla'}) then
    assign(f = "$Art21Kr14$.m", c = interface(warnlevel)), null(interface(warnlevel = 0))
  else error "<%1> is not a Maple library", L
  end if;
  if P = 0 then march('extract', L, `_TABprog.m`, f);
    if type(f, 'file') then
      (proc(f) filepos(f, 8), writebytes(f, [95]), close(f); read f; delf(f) end proc)(f);
      return null(interface(warnlevel = c), map(op, [indices(__ABprog)]),
        assign('__ABprog' = '_ABprog'))
    else error "_TABprog is absent in library <%1>", L
    end if
  elif type(P, 'file') then assign(a = Ftype(P), b = CFF(P)[-1])
  else null(interface(warnlevel = 0));
    error "the first factual argument must be zero or file, but has received - %1", P
  end if;
  if member(a, {".exe", ".com"}) then _TABprog[cat(`, b)] := readbytes(P, infinity); close(P)
  else error "filetype must be {`.com`, `.exe`}, but has received - %1", a
  end if;
  UpLib(L, [_TABprog]), null(interface(warnlevel = c));
  WARNING("_TABprog of library <%1> has been updated by program <%2>", L, b)
  end proc:
> com_exe1("C:\\Windows\\system32\\cmd.exe", "C:\\Maple_Lib\\UserLib");
Warning, _TABprog of library <C:/Maple_Lib/UserLib> has been updated
by program <cmd.exe>
> com_exe1(0, "C:\\Maple_Lib\\UserLib"); ⇒ [tasklist.exe, taskkill.exe, tlist.exe, cmd.exe]

```

Первым *P*-аргументом вызова *com_exe1(P, L)* может быть полный путь к файлу типа {'.exe', '.com'} с системной командой либо {0|1}-значение, которое определяет режим проверки глобальной переменной *_TABprog*, созданной/обновленной процедурой. Вызов процедуры *com_exe1(0, L)* возвращает список полных имен {com, exe}-программ, сохраненных в таблице *_TABprog*, находящейся в *Maple*-библиотеке, адрес которой указан в *L*. Тогда как вызов *com_exe1(1, L)* возвращает список полных имен программ,

находящихся в *_TABprog* и доступных в текущем сеансе. Наконец, вызов процедуры *com_exe1(P, L)* обновляет *Maple*-библиотеку *L {com, exe}*-программой, полный путь к которой определяется *P*-аргументом. Процедура *com_exe2*, дополняя предыдущую, предназначена для дозагрузки требуемых программ и в главный каталог *Windows*. С данными полезными средствами подробнее можно ознакомиться в [8-14,28-38,42-45].

Вызов процедуры *E_mail(F, H)* выполняет выбор корректных *email*-адресов из файла *txt*-формата, указанного *первым* аргументом *F*, с размещением их в выходном файле, указанным вторым аргументом *H*. При этом, при отсутствии пути к принимающему файлу *H* он создается с выводом сообщения о вновь созданном пути к *H*-файлу.

```

> E_mail := proc(InF::file, OutF::{string, symbol})
local In, Out, Art, S, G, k, p, T, V, Kr, R, F, Y, Z, E, L, t, r, omega, nu;
`if` (not type(InF, 'rlb'), ERROR("<%1> is not a text datafile", InF),
`if` (not type(eval(OutF), {'symbol', 'string'}), ERROR("<%1> is not a datafile path", OutF),
`if` (type(OutF, 'file'), assign(omega = OutF, L = [], t = 0),
[assign(nu = interface(warnlevel), L = [], t = 0), null(interface(warnlevel = 0)),
assign(omega = Mkdir(OutF, 1)), null(interface(warnlevel = nu))]);
T := {45, 46, (48 + k) $ (k = 0 .. 9), (64 + k) $ (k = 0 .. 58)} minus {64, 91, 92, 93, 94, 96};
[assign(Art = time(), S = readbytes(InF, 'TEXT', infinity), R = 0), close(InF),
assign(G = cat(" ", S, " ")), `if` (search(S, "@"), NULL,
ERROR("datafile <%1> does not contain email addresses", InF)),
assign(V = [seq(`if` (G[k] = "@", k, NULL), k = 1 .. length(G))]);
for k in V do assign('Kr' = "@", 'p' = k);
do p := p - 1;
if member(op(convert(G[p], 'bytes')), T) then Kr := cat(G[p], Kr) else p := k; break
end if
end do;
do p := p + 1;
if member(op(convert(G[p], 'bytes')), T) then Kr := cat(Kr, G[p]) else break end if
end do;
`if` (Kr[1] = "@" or searchtext(".", Kr, searchtext("@", Kr) .. length(Kr)) = 0,
assign('R' = R + 1, 'F' = 61, 'Y' = 4), NULL);
if F = 61 or Kr[-1] = "." or not search(Kr[-4 .. -2], ".") then
writebytes(cat(substring(InF, 1 .. -4), "$$$"), Kr),
writebytes(cat(substring(InF, 1 .. -4), "$$$"),
`if` (2 < nargs, [10], [44, 32])), assign('E' = 3), unassign('F')
else assign('L' = [op(L), Kr]), assign('Z' = 12)
end if
end do;
[assign('t' = nops(L)), assign('L' = map(Case, {op(L)}), assign('r' = nops(L))];
for k to nops(L) do
writebytes(omega, L[k]), writebytes(omega, `if` (2 < nargs, [10], [44, 32]))

```

```

end do;
close(`if` (Z = 12, omega, NULL), `if` (Y = 4, cat(substring(InF, 1 .. -4), "$$$"), NULL));
WARNING("%1 - total of constructions `@` in datafile <%2>
    %3 - quantity of correct email addresses in datafile <%4>
    %5 - quantity of multiple email addresses
    %6 - quantity of suspicious email addresses in datafile <%7>
    run time of the procedure = %8 minute(s)", nops(V),
    InF, r, omega, t - r, R, cat(substring(InF, 1 .. -4), "$$$"), round(1/60*time() - 1/60*Art))
end proc:

```

```

> E_mail("C:\\RANS\\My_email.txt", "C:\\RANS\\Email\\Email.txt", 14);
Warning, 420 - total of constructions `@` in file <C:\\RANS\\My_email.txt>
405 - quantity of correct email addresses in file <C:\\RANS\\Email\\Email.txt>
10 - quantity of multiple email addresses
5 - quantity of suspicious email addresses in file <C:\\RANS\\My_email.$$$>
run time of the procedure = 0 minute(s)

```

Путь к файлу *H* создается процедурой *MkDir* [9,45]. Результатом выполнения данной процедуры является не только создание текстового файла с искомыми *email*-адресами, пригодными для немедленного использования средствами *email*, но и получение всей основной статистики, чье назначение хорошо проясняет фрагмент, представленный выше. При этом, если вызов процедуры *E_mail(F,H,h)* использует необязательный 3-й *h*-аргумент (любое *Maple*-выражение), то каждый корректный *email*-адрес выводится в принимающий *H*-файл отдельной строкой; в противном случае, адреса разделяются запятой. Все подозрительные адреса (если такие существуют) помещаются в отдельный текстовый \$\$\$-файл, главное имя которого идентично имени исходного *F*-файла. При этом, процедура *E_mail* выводит другую полезную статистику о результатах работы. Процедура *email(F, H)* базируется на предыдущей процедуре, наследуя два ее первых аргумента и расширяя действие первой на файлы данных типов {*doc*, *rtf*, *htm*}, в более общем случае на файлы данных определенного нами *rlb*-типа [8-14,28-38,42-45].

В целом ряде задач возникает необходимость извлечения из выражений корректных подвыражений, идентифицируемых их началом, задаваемым либо позицией, либо строкой. В этом отношении достаточно полезной может оказаться процедура *extexp*, обеспечивающая вполне удовлетворительное решение данной задачи. Приведенный фрагмент представляет исходный текст процедуры и примеры ее применения.

```

> extexp := proc(S::{string, symbol}, n::list({posint, string})) local a, b, c, d, k, j, p, h, t;
assign(a = "" | | S, b = length(S), d = [], h = interface(warnlevel), interface(warnlevel = 0);
for j to nops(n) do
    if type(n[j], 'posint') then
        if n[j] <= b then assign('c' = a[n[j]], 't' = n[j] + 1) else
            error "%-1 element of 2nd argument should be no more than %2
                but has received %3", j, b, n[j]
        end if
    end if
end if

```

```

else
  if not search(a, n[j], 'p') then
    error "%-1 element of 2nd argument not belong <%2>", j, S
  else assign('t' = p + length(n[j]), 'c' = n[j])
  end if
end if;
for k from t to b do c := cat(c, a[k]);
  try parse(c, 'statement') catch: next end try;
  d := [op(d), c];
  break
end do
end do;
map(parse, d, 'statement'), null(interface(warnlevel = h))
end proc:
> S := `(a+b)/(c+d)+x*(x+y)/(a-b)`: extexp(S, [1, 12, "(x+y)", "(x)"]);
      [a + b, x, (x + y)/(a - b), x + y]
> P := "proc () x := proc () end proc end proc": extexp(P, ["proc ()", "x := proc ()"];
      [proc() local x; x := proc() end proc end proc, proc() end proc]
> A := (x*cos(x)+y*sin(y))/(x^2+a*x+c): extexp(convert(A, 'string'), ["(x*cos(x))", "(x^2)"]);
      [x*cos(x) + y*sin(y), x^2 + a*x + c]

```

Вызов процедуры *extexp(S, n)* возвращает список подвыражений в строчном формате, которые были извлечены из строчного/символьного формата *S* исходного выражения на основе их *начальных* позиций, определяемых списком *n*. Список содержит позиции *начала* подвыражений в *S*-строке и/или их префиксы в строчном формате. Процедура обрабатывает основные особые и ошибочные ситуации. Прием, использованный для извлечения *корректных* подвыражений, вполне может оказаться достаточно полезным при программировании других интересных приложений в среде пакета *Maple*.

Стандартная процедура *subs* имеет целый ряд дефектов, поэтому была создана *Subs*-процедура, свободная от ряда дефектов первой. *Первый* формат *Subs(x=a, expr)* вызова процедуры заменяет все вхождения подвыражения *x* в выражении *expr* на выражение *a*. Тогда как *второй* формат *Subs(S, expr)*, где *S* – список или множество уравнений *x=a*, вызова процедуры делает аналогичные замены, между тем, подстановки выполняются последовательно, начиная с первого элемента *S* и до последнего. Более того, в случае, если *второй* аргумент процедуры *Subs* имеет тип *{string | symbol}*, то подстановка будет выполнена с возвращением результата типа, отвечающего типу второго аргумента. В следующем фрагменте приведен исходный текст процедуры и некоторые наиболее типичные примеры применения данной процедуры.

```

> Subs := proc(s::{`=`, list(`=`), set(`=`)}, Z::{set, algebraic, procedure, list, `module`, string})
  local k, G, Kr, Res;
  G := proc(s, Z)
    local k, Kr, Res;

```

```

Kr := proc(s, Z)
  local a, b, c, d, h, Res;
  a, b := map(convert, [lhs(s), rhs(s)], string), cat("##", convert(eval(Z), string), "##");
  c := Search2(b, {a[1]});
  if c = [] then return convert(eval(Z), 'string') end if;
  h := [seq([c[k], c[k] + length(a[1]) - 1], k = 1 .. nops(c))];
  d := [1 .. h[1][1] - 1, seq(h[k][2] + 1 .. h[k + 1][1] - 1,
    k = 1 .. nops(h) - 1), h[-1][2] + 1 .. length(b)];
  Res := cat(seq(op([b[d[k]], `if` (member(op(0, parse(a[2])), {`*`, `^`, `+`}),
    cat("(", convert(a[2], 'string'), ")"), convert(a[2], 'string'))]),
    k = 1 .. nops(d) - 1), b[d[-1]])[3 .. -3]
end proc;
Res := Kr(s, Z);
  if member(whattype(Z), {'symbol', 'string'}) then convert(Res, whattype(Z))
  else simplify(parse(Res)) end if;
  StringTools[SubstituteAll](StringTools[SubstituteAll](Res, "1/1/", "1/"), "1*1/", "1/");
  if type(Z, 'string') then Res else simplify(parse(Res)) end if
end proc;
try if type(s, `=`) then G(s, Z) else Res := Z; for k in s do Res := G(k, Res) end do;
  eval(Res) end if catch : 'procname(args)' end try
end if
end proc:
> Subs({t = 1/x^2, a=x, b=x^2}, (b + a + t + t^2)*t); => (x^6 + x^5 + x^2 + 1)/x^6
> Subs(x^2 = t, 1 + 1/(1 + x^2 + 1/x^2)); => (2*t + t^2 + 1)/(t + t^2 + 1)
> subs(x^2 = t, 1 + 1/(1 + x^2 + 1/x^2)); => 1 + 1/(1 + t + 1/(x^2))
> subs(a + b = y, (a + b + c)^2), Subs(a + b = y, (a + b + c)^2); => (a + b + c)^2, (y + c)^2
> Subs([ln(x)^2 = t, t = 68], (a + ln(x)^2)/(b + c*ln(x)^2)); => (a + 68)/(b + 68 c)
> subs(x^2 = t, (a + x^2)/x^2), Subs(x^2 = t, (a + x^2)/x^2); => (a + t)/x^2, (a + t)/t

```

С процедурой *Subs* оказывается тесно связанной реализация процедуры *diff*, которая представляет собой существенное расширение стандартной процедуры *diff(A, x, y, ...)*, обеспечивающей дифференцирование выражения *A* по переменным *x, y, ...*. В целом ряде случаев стандартная процедура *diff(A, x)* пакета не дает возможности корректно вычислять обычные либо частные производные алгебраического выражения *A*, когда выражения либо функции используются в качестве переменной дифференцирования *x*. Тогда как процедура *diff(A, x)* во многих случаях позволяет корректно решать такую задачу, допуская два или более фактических аргументов при двух формальных. При этом, если *первый* аргумент определяет дифференцируемое выражение, то остальные определяют *обобщенные* переменные дифференцирования, в качестве которых могут быть как простые переменные, так и выражения, в частности, функции и выражения от функций. На переменных дифференцирования данного типа вызовы процедуры *diff(A, x)* инициируют ошибочную ситуацию с диагностикой «invalid input: %1 received %2, which is not valid for its %3 argument», что для целого ряда задач неприемлемо.

Вызов процедуры *diff*(A, {x, y, z, ...}) эквивалентен вызову *diff*(A, {x, y, z, ...}) стандартной *diff*-процедуры, где {x,y,z,...} – обычные переменные, для которых имеет место следующее соотношение, а именно: *map(type, {x, y, z, ...}, symbol) -> {true}*; вызов вычисляет частные производные выражения A по обобщенным переменным x, y, z, ..., соответственно. При этом, переменные дифференцирования x, x, x, ... в количестве m при вызове процедуры *diff* можно кодировать и в форме x\$m. Более того, при вызове процедуры *diff*(A, {x, y, z, ...}) не является исключением использование в качестве переменных дифференцирования выражений, содержащих радикалы, как это весьма наглядно иллюстрируют примеры нижеследующего фрагмента. В целом, процедура достаточно существенно расширяет стандартную встроенную функцию *diff* пакета.

```
> diff := proc(x::algebraic, y::algebraic) local h, Res;
  if type(x, 'numeric') or type(y, 'numeric') then return x else Res := x end if;
  for h in [args[2..-1]] do
    if type(h, 'symbol') then Res := diff(Res, h)
    else Res := Subs(k = h, diff(Subs(h = k, Res), k))
    end if
  end do;
  simplify(Res)
end proc;
```

```
> diff((a + b/x^2)/x^2, x^2);
Error, invalid input: diff received x^2, which is not valid for its 2nd argument
```

```
> diff((a + b/x^2)/x^2, x^2);
```

$$-\frac{2bx + a^2x^2}{x^6}$$

```
> diff(x*cos(x), 42), diff(2010, x^2), diff(420, 72); => x*cos(x), 2010, 420
```

```
> diff((a + b*f(t)/g(y))/(c + d*f(t)/g(y)), f(t)/g(y));
```

$$\frac{g(y)^2(bcy - da)}{(cg(y) + df(t))^2}$$

```
> diff(1/sqrt(x^3 + a), sqrt(x^3 + a), x^3); => 1/(x^3 + a)^2
```

```
> diff((a + b*f(t)/h(y))/(c + d*f(t)/h(y)^2), (f(t)/h(y)));
```

$$\frac{bh(y)^2}{ch(y)^2 + df(t)^2}$$

Представленные примеры применения *diff* хорошо иллюстрируют ее возможности.

Табличные объекты могут при уникальных входах иметь кратные выходы, что в ряде случаев требует выяснения вопроса обратимости данного типа объектов. Достаточно простая процедура *revtabl(T)* возвращает таблицу, входами которой являются выходы таблицы T, тогда как ее выходами списки входов таблицы T. Базирующаяся на основе этой процедуры процедура *istablerev(T)* возвращает true, если каждый вход таблицы T имеет уникальный выход, и false в противном случае. Данные процедуры в целом ряде случаев оказываются достаточно полезными при работе с табличными объектами.


```

> revtabl := proc(T::table)
  local b, k;
  b := table([seq(k = {}, k = map(op, [entries(T)]))]);
  seq(assign('b[T[k]]' = {k, op(b[T[k]])), k = map(op, [indices(T)])); eval(b)
end proc;
> revtabl(T);  => table([a = {x, h, t}, c = {s, z}, b = {g, y, r}])
> istablerev := (T::table) -> type(map(nops, map(op, {entries(revtabl(T))})), set(1)):
> T:=table([x=a, y=b, y=c, h=d, v=42, g=47]): T1:=table([x=a, y=b, y=c, h=42, v=42, g=47]):
> map(istablerev, [T, T1]);  => [true, false]

```

В качестве аналога функции *tuples* пакета *Mathematica*, довольно полезной во многих приложениях, может служить *Maple*-процедура *tuples(J,n)*, возвращающая список или множество всевозможных кортежей длины *n*, составленных из элементов множества/списка *J*. Между тем, тип возвращаемого результата соответствует типу фактического аргумента *J* процедуры и кортежи представляются в строчном формате.

```

> tuples := proc(A::{set, list}, n::posint)
  local a, b, c, d, h, k;
  global Res_75_420;
  assign('Res_75_420' = {}, c = map(convert, A, 'string'),
  h = eval([Res_75_420, seq(cat(k, b), b = 1 .. n)]));
  d := parse(cat(seq(cat("for k", convert(b, 'string'), " in ",
  convert(c, 'string'), " do "), b = 1 .. n),
  cat("Res_75_420 := {op(Res_75_420), ", "cat(", seq(cat("k", convert(b, 'string'),
  `if` (b = n, NULL, ","), b = 1 .. n), ")", " }"),
  cat(seq("end do ", b = 1 .. n, " :"), 'statement'));
  assign('Res_75_420' = h[1], seq(cat(k, b - 1) = h[b], b = 2 .. n + 1)),
  convert(d, whattype(A))
end proc;
> k1, k3, Res_75_420 := 56, 75, 2011: tuples(["1", "2", "3", "4", "5", "6"], 3);
["133", "134", "135", "136", "141", "142", "143", "144", "145", "146", "151", "152", "153", "154",
"155", "156", "161", "162", "163", "164", "165", "166", "211", "212", "213", "214", "215", "216",
"221", "222", "223", "224", "225", "226", "231", "232", "233", "234", "235", "236", "241", "242",
"243", "244", "245", "246", "251", "252", "253", "111", "112", "113", "114", "115", "116", "121",
"122", "123", "124", "125", "126", "131", "132", "256", "261", "262", "263", "264", "265", "266",
"311", "312", "313", "314", "315", "316", "321", "322", "323", "324", "325", "254", "255", "326",
"331", "332", "333", "334", "335", "336", "341", "342", "343", "344", "345", "346", "351", "352",
"353", "354", "355", "356", "361", "362", "363", "364", "365", "366", "411", "412", "413", "414",
"415", "416", "421", "422", "423", "425", "426", "431", "432", "424", "433", "434", "435", "436",
"441", "442", "443", "444", "445", "446", "451", "452", "453", "454", "455", "456", "461", "462",
"463", "464", "465", "466", "511", "512", "513", "514", "515", "516", "521", "522", "523", "524",
"525", "526", "531", "532", "533", "534", "535", "536", "541", "542", "543", "544", "545", "546",

```

```
"551", "552", "553", "554", "555", "556", "561", "562", "563", "564", "565", "566", "611", "612",
"613", "614", "615", "616", "621", "622", "623", "643", "644", "645", "646", "651", "652", "653",
"654", "655", "656", "661", "662", "663", "664", "665", "624", "625", "626", "631", "632", "633",
"634", "635", "641", "642", "636", "666"]
> k1, k2, k3, Res_75_420;  ⇒ 56, k2, 75, 2011
```

При этом следует отметить, алгоритм реализации данной процедуры является весьма хорошим примером подхода к реализации произвольно генерируемых выполняемых *Maple*-кодов. Его суть состоит в конструкции в строчном формате требуемого кода с последующим применением к нему встроенной *parse*-функции. При этом, процедура *tuples* в целом ряде случаев представляет собой достаточно эффективное средство. В частности, она может быть использована для весьма простой реализации процедуры, которая вычисляет *все* прямоугольные матрицы с элементами из заданного алфавита:

```
> AllMatrices := proc(m::posint, n::posint, A::{set(string), list(string)}, G::evaln)
  local a, b, k;
    b := [];
    for k in tuples(A, m*n) do b := [op(b), map(parse,
      matrix([seq([seq(k[j], j = (p - 1)*m + 1 .. p*m]), p = 1 .. n])))]
    end do;
    G := b; G
  end proc;
> AllMatrices(2, 3, {"0", "1"}, GS);  ⇒ GS
> a := {}: for k to 64 do a := {op(a), GS[k]} end do: a;
[[ [0 0], [0 0], [0 0], [0 0], [0 1], [0 1], [0 1], [0 1], [1 0], [1 0], [1 0], [1 0],
  [0 0], [0 1], [1 0], [1 1], [0 0], [0 1], [1 0], [1 1], [0 0], [0 1], [1 0], [1 1],
  [1 1], [1 1], [1 1], [1 1]]]
```

Процедура *AllMatrices(m, n, A, G)* через символ *G* возвращает множество всех матриц размерности *m*х*n* (*n* – количество строк, *m* – количество столбцов), элементами которых являются элементы в строчном формате из алфавита *A*. Фрагмент представляет текст процедуры и пример ее применения для генерации всех бинарных (2х2)-матриц.

Между тем, в целом ряде случаев требуется генерация не *сплошных* кортежей в форме строк, а в форме всевозможных списков длины *n*, чьи элементы принадлежат некоему алфавиту *A*, определяемому списком либо множеством. В данной постановке задача в полной мере соответствует возможностям функции *Tuples* из пакета *Mathematica*.

```
> Tuples := proc(A::{list, set}, n::posint)
  local a, b, c, d, h, k;
  global Res_15_22;
  a := eval([Res_15_22, seq(cat(k, b), b = 1 .. n)]);
  assign('Res_15_22' = [], c = A, h = whatype(A));
  d := parse(cat(seq(cat("for k", convert(b, string),
    " in ", convert(c, 'string'), " do "), b = 1 .. n), cat("Res_15_22 := [op(Res_15_22), ", "["
```

```

seq(cat("k", convert(b, string), `if`(b = n, NULL, ",")), b = 1 .. n), "]", " ",
cat(seq("end do ", b = 1 .. n, ":")), 'statement');
assign('Res_15_22' = a[1], seq(cat(k, b - 1) = a[b], b = 2 .. n + 1)), convert(d, h)
end proc;
> Res_15_22, k1, k3, k5 := 2011, 75, 420, 6: Tuples({0, 1}, 5);
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 1], [0, 0, 0, 1, 0], [0, 0, 0, 1, 1], [0, 0, 1, 0, 0], [0, 0, 1, 0, 1], [0, 0, 1, 1, 0],
[0, 0, 1, 1, 1], [0, 1, 0, 0, 0], [0, 1, 0, 0, 1], [0, 1, 0, 1, 0], [0, 1, 0, 1, 1], [0, 1, 1, 0, 0], [0, 1, 1, 0, 1],
[0, 1, 1, 1, 0], [0, 1, 1, 1, 1], [1, 0, 0, 0, 0], [1, 0, 0, 0, 1], [1, 0, 0, 1, 0], [1, 0, 0, 1, 1], [1, 0, 1, 0, 0],
[1, 0, 1, 0, 1], [1, 0, 1, 1, 0], [1, 0, 1, 1, 1], [1, 1, 0, 0, 0], [1, 1, 0, 0, 1], [1, 1, 0, 1, 0], [1, 1, 0, 1, 1],
[1, 1, 1, 0, 0], [1, 1, 1, 0, 1], [1, 1, 1, 1, 0], [1, 1, 1, 1, 1]]
> nops(%), k1, k3, k5, Res_15_22;  => 32, 75, 420, 6, 2011
> Tuples([1, 1/2, a], 3);
[[1, 1, 1], [1, 1, 1/2], [1, 1, a], [1, 1/2, 1], [1, 1/2, 1/2], [1, 1/2, a], [1, a, 1], [1, a, 1/2], [1, a, a], [1/2, 1, 1], [1/2, 1, 1/2], [1/2, 1, a], [1/2, 1/2, 1], [1/2, 1/2, 1/2],
[1/2, 1/2, a], [1/2, a, 1], [1/2, a, 1/2], [1/2, a, a], [a, 1, 1], [a, 1, 1/2], [a, 1, a], [a, 1/2, 1], [a, 1/2, 1/2], [a, 1/2, a], [a, a, 1], [a, a, 1/2], [a, a, a]]
> t := time(): Tuples([0, 1], 10): time() - t;  => 0.078 # Maple 8
> t := time(): Tuples([0, 1], 10): time() - t;  => 0.124 # Maple 11
In[244]:= t = TimeUsed[]; Tuples([0, 1], 20); TimeUsed[] - t
Out[244]= 0.078

```

Данную задачу решает процедура, чей вызов *Tuples(A, n)* возвращает множество либо список всевозможных списков длины *n*, составленных из элементов множества/списка *A*. При этом, тип возвращаемого результата соответствует типу первого фактического аргумента. В предыдущем фрагменте представлен исходный код процедуры *Tuples* и примеры ее применения. С целью сравнения проведена оценка времени выполнения этой процедуры в различных релизах *Maple* и одноименной функции в *Mathematica*. Результат весьма показателен: если в релизах *8, 10, 11* получаем время *0.078, 0.093, 0.124* соответственно, то в *Mathematica 8* реактивность функции *Tuples* существенно выше. Обусловлена такая ситуация тем, что алгоритмы *Maple* зачастую с новыми релизами ухудшают для ряда средств временные характеристики. Тогда как система управления памятью в пакете *Mathematica* существенно эффективнее, алгоритмы оптимальнее и используют специальные механизмы сжатия обрабатываемых данных.

Между тем, между одного назначения процедурами *tuples* и *Tuples* имеются и весьма существенные различия. Ниже представлены соотношения *G* и *V* между результатами вызовов обоих процедур, из которых следует, что при использовании алфавита *A* при вхождении в него только однобуквенных символов представленные соотношения *G* и *V* являются эквивалентными, тогда как присутствие в алфавите *A* в качестве элементов более сложных выражений нарушает данную эквивалентность. В этом контексте ряд практически полезных соображений отдает предпочтение именно процедуре *Tuples*, эквивалентной функции *Tuples* пакета *Mathematica*. С другой стороны, и процедура *tuples* в определенных случаях может оказаться достаточно полезным средством, что и было установлено в ряде приложений, имеющих дело с такого типа кортежами.

```

> A, B, n := {0, 1, 2}, {0, a+b, 1/2}, 3:

> [seq(cat(op(map(convert, k, string))), k = Tuples(A, n))] = [op(tuples(A, n))];      (G)
["000", "001", "002", "010", "011", "012", "020", "021", "022", "100", "101", "102", "110", "111", "112",
"120", "121", "122", "200", "201", "202", "210", "211", "212", "220", "221", "222"] = ["000", "001",
"002", "010", "011", "012", "020", "021", "022", "100", "101", "102", "110", "111", "112", "120", "121",
"122", "200", "201", "202", "210", "211", "212", "220", "221", "222"]

> [seq(map(parse, map(convert, k, list)), k = tuples(A, n))] = [op(Tuples(A, n))];      (V)
[[0, 0, 0], [0, 0, 1], [0, 0, 2], [0, 1, 0], [0, 1, 1], [0, 1, 2], [0, 2, 0], [0, 2, 1], [0, 2, 2], [1, 0, 0], [1, 0, 1],
[1, 0, 2], [1, 1, 0], [1, 1, 1], [1, 1, 2], [1, 2, 0], [1, 2, 1], [1, 2, 2], [2, 0, 0], [2, 0, 1], [2, 0, 2], [2, 1, 0],
[2, 1, 1], [2, 1, 2], [2, 2, 0], [2, 2, 1], [2, 2, 2]] = [[0, 0, 0], [0, 0, 1], [0, 0, 2], [0, 1, 0], [0, 1, 1], [0, 1, 2],
[0, 2, 0], [0, 2, 1], [0, 2, 2], [1, 0, 0], [1, 0, 1], [1, 0, 2], [1, 1, 0], [1, 1, 1], [1, 1, 2], [1, 2, 0], [1, 2, 1],
[1, 2, 2], [2, 0, 0], [2, 0, 1], [2, 0, 2], [2, 1, 0], [2, 1, 1], [2, 1, 2], [2, 2, 0], [2, 2, 1], [2, 2, 2]]

> [seq(cat(op(map(convert, k, string))), k = Tuples(B, n))] = [op(tuples(B, n))];
["000", "001/2", "01/20", "01/21/2", "1/200", "1/201/2", "1/21/20", "1/21/21/2", "00a+b",
"01/2a+b", "0a+b0", "0a+b1/2", "0a+ba+b", "1/20a+b", "1/21/2a+b", "1/2a+b0", "1/2a+b1/2",
"1/2a+ba+b", "a+b00", "a+b01/2", "a+b0a+b", "a+b1/20", "a+b1/21/2", "a+b1/2a+b",
"a+ba+b0", "a+ba+b1/2", "a+ba+ba+b"] = ["000", "001/2", "01/20", "01/21/2", "1/200",
"1/201/2", "1/21/20", "1/21/21/2", "00a+b", "01/2a+b", "0a+b0", "0a+b1/2", "0a+ba+b",
"1/20a+b", "1/2a+b0", "a+b00", "a+b01/2", "a+b0a+b", "a+b1/20", "a+ba+b0", "1/21/2a+b",
"1/2a+b1/2", "1/2a+ba+b", "a+b1/21/2", "a+b1/2a+b", "a+ba+b1/2", "a+ba+ba+b"]

> [seq(map(parse, map(convert, k, list)), k = tuples(B, n))] = [op(Tuples(B, n))];
Error, incorrect syntax in parse: `/' unexpected (2)

> Tuples({1, a, c^2}, 2);  => {[1, 1], [1, c^2], [c^2, 1], [c^2, c^2], [1, a], [a, 1], [a, a], [a, c^2], [c^2, a]}
In[634]:= Tuples[{1, a, c^2}, 2]
Out[634]= {[1, 1], {1, a}, {1, c^2}, {a, 1}, {a, a}, {a, c^2}, {c^2, 1}, {c^2, a}, {c^2, c^2}}

```

Между тем, говоря об эквивалентности одноименных процедуры *Tuples* пакета *Maple* и функции *Tuples* пакета *Mathematica*, понимается функциональная эквивалентность, тогда как во временном отношении первая достаточно существенно уступает второй.

Проверка наличия атрибута «protected» символов в целом ряде случаев представляется достаточно актуальной задачей. *Maple* для этой цели располагает встроенным типом «protected». Решают данную задачу и представленные ниже две процедуры *ProtectedQ* и *ProtectedQ1*; первая процедура базируется на прямом алгоритме проверки данного атрибута, исходя из определения его сути, тогда как вторая базируется на встроенной функции *attributes*. Весьма простой фрагмент иллюстрирует сказанное, а именно:

```

> type(sin, protected);  => true
> ProtectedQ := proc(x::symbol)
  local a, b;
  b := cat("", x, ":", convert(eval(x), 'string'), ":");
  try assign(x = b) catch "attempting to assign to": return true end try;

```

```

[unassign('x'), parse(b, 'statement'), false][-1]
end proc:
> ProtectedQ1 := (x::symbol) -> member(protected, {attributes(x)}):
> {ProtectedQ(sin), sin(Pi)}; => {true, 0}
> {ProtectedQ(Agn), Agn}; => {false, Agn}
> H := x -> x^2: {ProtectedQ1(H), eval(H)}; => {false, x -> x^2}
> StrOfSymbQ := (S::string, A::set) -> `if` (belong({S[k]$k=(1..length(S))}, map(convert,
A, string)), true, false):
> StrOfSymbQ("123a58g43n1", {1,2,3,4,5,6,7,8}); => false
> StrOfSymbQ("123a58g43n1", {1,2,3,4,5,8,a,g,n}); => true
In[1535]:= ProtectedQ[x_] := If[MemberQ[Attributes[x], Protected], True, False]
In[1536]:= HS=75; Ptotect[HS]; Map[ProtectedQ, {Sin, Protect, AVS, HowAct, Map, "HS"}]
Out[1536]= {True, True, False, False, True, True}
In[1537]:= ProtectedQ1[x_] := Module[{h}, Quiet[Check[Clear[x], h = "er", Clear::wrsym]];
If[h === "er", 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}
In[1568]:= StrOfSymbQ[x_String, A_List] := Module[{}, If[DeleteDuplicates[Map3[
MemberQ, Map[ToString, A], Characters[x]]] == {True}, True, False]]
In[1569]:= StrOfSymbQ["134325678", {1, 2, 4, 5}]
Out[1569]= False
In[1570]:= StrOfSymbQ["4134382556728", {1, 3, 2, 4, 5, 6, 7, 8}]
Out[1570]= True

```

Простая процедура *StrOfSymbQ(S, A)* возвращает *true*, если строка *S* содержит только символы из множества *A*, и *false* в противном случае. Тогда как пакет *Mathematica* для этих же целей предоставляет средства создания одноименных простых аналогов для 3 указанных процедур *Maple*, используя для алгоритмов подходы идентичного уровня.

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. С целью упрощения реализации алгоритма определяется довольно простая процедура *Pref(x, y)*, определяющая факт вхождения подстроки *y* в качестве конца для строки *x*. Простой фрагмент представляет исходные коды процедур с примерами применения.

```

> Pref := (x::string, y::string) -> `if` (length(y) = 0, true, `if` (length(y) > length(x), false,
`if` (x[-length(y) .. -1] = y, true, false)))

```

```

> Pref("123456abcdefh", "abcdefh"); => true
> Predecessors := proc(ltf::table, Co::string, n::posint)
    local a, b, c, d, k, i, j;
    b := revtabl(ltf); a := b[Co[1]];
    for k from 2 to length(Co) do
        assign('c' = b[Co[k]], 'd' = {});
        for i to nops(a) do for j to nops(c) do
            if Pref(a[i], c[j][1 .. n - 1]) then d := {op(d), cat(a[i], c[j][-1])} end if
        end do
    end do;
    a := d
end do;
a, `if`(nops(a) <> (n-1)^nops([indices(b)]), "Structure possesses the nonconstructability
of NCF-type", NULL)
end proc;
> f:=table(["000"="0", "001"="1", "010"="1", "011"="0", "100"="1", "101"="0", "110"="0", "111"="1"]);
f1:=table(["000"="0", "001"="1", "010"="1", "011"="0", "100"="0", "101"="0", "110"="0", "111"="1"]);
> Predecessors(f, "11001110110101", 3);
{"0010111110011100", "0100001000101010", "1001100101000111", "1111010011110001"}
> Predecessors(f1, "1100", 3); => {"111101", "111100", "001000", "001011"}
> Predecessors(f1, "11001110110101", 3);
{"", "Structure possesses the nonconstructability of NCF-type"}
> Predecessors(f1, "11001110", 3);
{"0010111110", "Structure possesses the nonconstructability of NCF-type"}
> Predecessors(f1, "110011010", 3);
{"00100010011", "00100010100", "00100010101", "11110010011", "11110010100",
"11110010101"}, "Structure possesses the nonconstructability of NCF-type"
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"}
> AtomicQ := proc(x) local F; if nops(map(F, x)) = 1 then true else false end if end proc:
> map(AtomicQ, [sin, a + b, 75, sqrt(c + d), a/b, a^x]); => [true, false, true, false, false, false]
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}
> StrExprQ := proc(x::string) try parse(x) catch : return false end try; true end proc:
> map(StrExprQ, ["/a/b", "a^^b", "a(a + b)", "Avz=72=420", "(a + b/)"]);
      [false, false, true, false, false]
In[2671]:= StrExprQ[x_String] := Module[{a}, Quiet[Check[ToExpression[x], a = 420]];
      If[a === 420, False, True]]
In[2672]:= Map[StrExprQ, {"/a/b", "a^^b", "a (a + b)", "Avz=72=420", "(a + b/)"}]
Out[2672]= {False, False, True, False, False}

```

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 также аналог процедуры *Predecessors* в среде пакета *Mathematica*. Алгоритм такого аналога не базируется на табличных структурах, а параллельные подстановки $x_1x_2 \dots x_n \rightarrow x'_1$, 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.

Завершают фрагмент 2 процедуры *AtomicQ(x)*, реализованные в среде обоих пакетов, и возвращающие *true*, если *x* – выражение, не делимое на подвыражения, и *false* иначе. Отметим, что данные процедуры носят сугубо иллюстративный характер, тогда как их функциональными аналогами являются тип *`atomic`* в *Maple* и функция *AtomQ* в *Mathematica*. В основе алгоритмов этих средств лежит использование функции *{map |*

Map}, обеспечивающей тестирование произвольного выражения x на наличие в нем выделяемых указанными функциями элементарных подвыражений. В то время как 2 процедуры *StrExprQ(x)*, также реализованные в среде обоих пакетов, возвращают *true*, если x – строка, содержащая корректное выражение, и *false* в противном случае. Как и две предыдущих, данные процедуры носят сугубо иллюстративный характер, однако вполне могут представить и определенный самостоятельный интерес.

```

> ExpLocals := proc(P::procedure, L::set(symbol))
  local a, b, c, d, h, xa, xl, xg, k;
  try type(ParProc(P), array) catch: end try;
  if %<> true then error "Object <%1> can't be processed", P
  else assign('a' = convert(ParProc(P), listlist),
    b = convert(eval(P), 'string'), xa = {}, xl = {}, xg = {}, d = {})
  end if;
  for k in a do `if` (lhs(op(k)) = `Arguments`, assign('xa' = {rhs(op(k))}),
    `if` (lhs(op(k)) = `locals`, assign('xl' = {rhs(op(k))}),
    `if` (lhs(op(k)) = `globals`, assign('xg' = {rhs(op(k))}), NULL)))
  end do;
  c := map(op, {xg, xl, xa});
  if c = {} then return WARNING("Procedure %1 have no arguments,
    global and local variables", P)
  else for k in c do
    try d := {op(d), lhs(k)} catch: d := {op(d), k} end try
  end do
  end if;
  d := L `minus` d;
  if d = {} then return WARNING("Set %1 contain no admissible variables", L)
  else WARNING("Procedure %1 has been expanded by local variables %2", P, d)
  end if;
  assign('a' = cat("", P, " := "), 'c' = cat("proc (", convert(xa, 'string')[2 .. -2], ") "));
  if xl = {} then Suffix(b, c, h);
    parse(cat(a, c, " local ", convert(d, 'string')[2 .. -2], "; ", h, ":"), 'statement')
  else
    parse(cat(a, Insert(b, " local ", cat(" local ", convert(d, string)[2 .. -2], " ")), ":"), 'statement')
  end if
  end proc:
> Agn := proc(x, y) global z; (x + y)/z end proc:
> ExpLocals(Agn, {x, y, z, a, b, c});
Warning, Procedure Agn has been expanded by local variables {b, c, a}
proc(x, y) local b, c, a; global z; (x + y)/z end proc
> Avz := proc(x, y) local a, b, g; global z; (x+y)/z end proc:
> ExpLocals(Avz, {x, y, z, a, b, c});

```



```
Warning, Procedure Avz has been expanded by local variables {c}
      proc(x, y) local c, a, b, g; global z; (x + y)/z end proc
> ExpLocals(Avz, {x, y, z, a, b});
Warning, Set {b, x, y, z, a} contain no admissible variables
> ExpLocals(sin, {x, y, z, a, b});
Error, (in ExpLocals) Object <sin> can't be processed
```

В целом ряде случаев появляется необходимость динамического расширения списка локальных переменных у процедуры, определенной *proc*-конструкцией и доступной в текущем сеансе пакета (*определение было вычислено в текущем сеансе либо расположено в Maple-библиотеке, логически сцепленной с главной библиотекой пакета*), без изменения ее исходного кода. Данную задачу решает процедура *ExpLocals*, которая представлена в предыдущем фрагменте. Успешный вызов *ExpLocals(P, L)* возвращает исходный текст модифицированной процедуры *P*, активизированный в текущем сеансе; более того, с выводом сообщения о списке переменных из множества *L*, на которые был расширен список *локальных* переменных исходной процедуры *P*. При этом, в общем случае этот список может быть меньше заданного при вызове множества *L* (*или вообще пустым*), так как из него исключаются все переменные, которые имеются в процедуре *P* в качестве формальных аргументов, глобальных или локальных переменных. Если в множестве *L* нет переменных, отличных от перечисленных, обработка процедуры *P* не делается с выводом соответствующего сообщения. Если же в качестве фактического аргумента *P* при вызове процедуры указан недопустимый объект, то инициируется ошибочная ситуация. Завершает фрагмент применение *ExpLocals* к весьма простым процедурам *Avz* и *Agn* с целью расширения их списков *локальных* переменных. Следует отметить, процедуру *ExpLocals* можно достаточно несложно распространить и на возможность расширения списков глобальных переменных для процедур. Оставляем это читателю в качестве достаточно интересного и полезного упражнения.

Процедура *ExpLocals* для *рекомбинации* и формирования строк требуемой структуры использовала ряд как стандартных, так и нестандартных средств. Между тем, для этой цели вполне может быть использована и простая процедура *insert(S, s, n)*, обеспечивая вставку в строку *S* подстроки *s* после ее *n*-й позиции. При этом, при $n = \{0, \text{length}(S)\}$ *s* помещается перед началом строки *S* и в ее конец соответственно. Фрагмент приводит исходный текст процедуры *insert* наряду с примерами ее применения, а также более общую процедуру *insertN*, базирующуюся на предыдущей процедуре *insert* и вызов которой *insertN(S, s, n)* возвращает результат вставки в строку или символ *S* символов и/или строк, определенных списком *s*, после позиций *S*, определенных множеством *n*. Предполагается, фактические аргументы *s* и *n* содержат одинаковое число элементов, иначе инициируется ошибочная ситуация; процедура очень легко модифицируется таким образом, что лишние элементы множества *n* просто игнорируются. Сказанное относительно процедуры *insert* в полной мере относится и к *insertN*. Более того, для примеров применения процедуры *insert* используется простая процедура *map7*, суть которой хорошо усматривается из примеров ее применения к процедуре *insert*; сама процедура *map7* оказывается достаточно полезной в целом ряде приложений [45]. В частности, она полезна при необходимости некоторых расширений функции *map*.

```

> insert := proc(S::{string, symbol}, s::{string, symbol}, n::nonnegative)
  local a, b;
  assign(a = map(convert, [S, s], string), b = ((x, y) -> convert(x, whattype(y))));
  if n < 1 then b(cat(a[2], a[1]), S) elif length(S) <= n then b(cat(a[1], a[2]), S)
  else b(cat(a[1][1 .. n], a[2], a[1][n + 1 .. -1]), S)
  end if
end proc:

> map7 := (F, x, y, L::{list, set}) -> [seq(F(x, y, k), k = L)]:
> map7(F, a, b, [x, y, z]); => [F(a, b, x), F(a, b, y), F(a, b, z)]
> map7(F, a, b, {x, y, z}); => {F(a, b, x), F(a, b, y), F(a, b, z)}
> map7(insert, "123456789", RANS, [0, 5, 9]);
      ["RANS123456789", "12345RANS6789", "123456789RANS"]

> insertN := proc(S::{string, symbol}, s::list({string, symbol}), n::set(nonnegative))
  local a, b, k;
  if nops(s) = nops(n) then assign(a = S, b = n)
  else error "2nd and 3rd arguments differ in number of elements"
  end if;
  for k to nops(b) do a := insert(a, s[k], b[k]); b := map(`+`, b, length(s[k])) end do; a
end proc:

> insertN("123456789", [x, y, z, h], {0, 3, 15, 6}); => "x123y456z789h"
> insertN("123456789", [x, y, z, h], {0, 3, 15, 6, 5});
Error, (in insertN) 2nd and 3rd arguments differ in number of elements

```

В противоположность предыдущей процедуре следующая процедура *DelSubStr(S, N)*, в свою очередь, обеспечивает *удаление* из строки *S* подстрок или отдельных символов, позиции которых заданы множеством либо вложенным множеством *N*, т.е. множество *N* может быть как вложенности *0*, так и *1*, например, $\{\{3, 4\}, 10, \{6, 9\}\}$ либо $\{1, 3, 5, 7, 15\}$. Процедура позволяет *удалять* из строк как их подстроки, определенные множествами их граничных позиций, так и отдельные символы, определенные номерами позиций. При этом, тип возвращаемого процедурой результата определяется типом ее первого фактического аргумента. Следующий фрагмент представляет как сам *исходный* текст процедуры, так и некоторые типичные примеры ее применения.

```

> DelSubStr := proc(S::{string, symbol}, N::{set(nonnegative), nestset(nonnegative)})
  local a, b, c, dp, p, ins, k;
  dp := (x::set) -> {seq(`if` (not type(p, set), p, eval(parse(cat("k$k=",
    sstr(["", " = ".."], convert(p, string))[2 .. -2], ":"))))), p = x)};
  assign(a = convert(S, string), b = dp(N));
  assign(c = nops(b), ins = ((x::string, h::list(positive)) -> cat(x[1 .. h[1] - 1], x[h[1] + 1 .. -1])));
  for k to c do a := ins(a, [b[k]]); b := {seq(map(`-`, b[p], 1), p = 1 .. c)} end do;
  convert(a, whattype(S))
end proc:

```

```

> DelSubStr(`123456789abcdefgh`, {5, {7, 9}, 11, 2, {13, 16}});  ⇒ 1346ach
> DelSubStr("123456789", {5, 7, 8, 9, 2, 1, 3, 4, 6});  ⇒ ""
> DelSubStr("123456789abcdefghzrw", {{5, 15}});  ⇒ "1234ghzrw"

```

Аналогично рассмотренной процедуре *ExpLocals*, следующая процедура *ExpGlobals* обеспечивает динамическое расширение списка глобальных переменных у процедуры, определенной *proc*-конструкцией и доступной в текущем сеансе, без модификации ее исходного кода. Следующий фрагмент представляет исходный код процедуры наряду с некоторыми примерами ее наиболее типичного применения.

```

> ExpGlobals := proc(P::procedure, G::list(symbol), set(symbol))
  local a, arg, b, c, h, l, g, g1, k, z, m, n, p;
  assign67(a = interface(verboseproc), interface(verboseproc = 3), h = "", l = {}, g = {});
  assign(b = convert(eval(P), string), c = convert(ParProc(P), listlist));
  for k in c do z := op(k);
    if lhs(z) = `Arguments` then arg := {rhs(z)}; h := cat(h, "proc (", seqstr(rhs(z)), ") ")
    elif lhs(z) = `locals` then l := {rhs(z)}
    elif lhs(z) = `globals` then g := {rhs(z)}
    end if
  end do;
  assign(m = l intersect {op(G)}, n = g intersect {op(G)},
    p = arg intersect {op(G)}), interface(verboseproc = a);
  g1 := {op(G)} minus `union`(m, n, p);
  if g1 = {} then error "%1 is fully absorbed by sets of formal arguments and-or global and
    local variables", G else WARNING("instead of variables %1 variables %2 have
    been used", G, g1)
  end if;
  if g = {} then parse(cat("", P, ":", sub_1(h = cat(h, " global ",
    seqstr(op(g1)), "; ", b), ";"), 'statement')
  else parse(cat("", P, ":", sub_1(cat("global ", seqstr(op(g))) =
    cat("global ", seqstr(op(g union g1))), b), ";"), 'statement')
  end if
end proc;

> G := proc(x, y) local a, b, c; x^2 + y^2 end proc: V := proc(x, y) local a, b, c; global h, k;
  x^2 + y^2 end proc: W := proc(x, y) local a, b; global c, d; x^2 + y^2 end proc;
> ExpGlobals(V, {x, y, z, t}); ⇒ proc(x, y) local a, b, c; global t, h, k, z; x^2 + y^2 end proc
  Warning, instead of variables {x, y, z, t} variables {t, z} have been used
> ExpGlobals(G, {x, y, z, t, p}); ⇒ proc(x, y) local a, b, c; global t, z; x^2+y^2 end proc
  Warning, instead of variables {x, y, t, p, z} variables {t, p, z} have been used
> ExpGlobals(W, {x, y, a, b, c, d});
  Error, (in ExpGlobals) {x, y, d, c, b, a} is fully absorbed by sets of formal arguments
  and-or global and local variables

```

Вызов процедуры *ExpGlobals(W, G)* возвращает определение модифицированной *W* процедуры, активизированное в текущем сеансе, с выводом сообщения о переменных из *G*, на которые был расширен список глобальных переменных исходной процедуры *W*. Оставляем читателю рассмотреть исходный код в качестве полезного упражнения.

Следующий фрагмент представляет *Maple*-процедуру *Flatten(L, n)*, которая является некоторым аналогом функции *Flatten* пакета *Mathematica*, и примеры ее применения.

```
> Flatten := proc(L::{set, list}, n::nonnegative)
  local a, b, c, d, h, g;
    assign(a = L, h = (x -> convert(x, whattype(L))), g = whattype(L)),
    assign(d = (x -> `if`(type(x, g), mlsnest(x)[2][2] + 1, 0)), b = h([]));
  while n < d(a) do
    for c in a do
      if type(L, set) then if n - 1 < d(c) then b := {op(b), op(c)} else b := {op(b), c} end if
      else if n - 1 < d(c) then b := [op(b), op(c)] else b := [op(b), c] end if
      end if
    end do;
    a := b; b := h([])
  end do; convert(a, g)
end proc;

> Flatten([6, 78, [[75, 42]], d, [[[1, 2, 3, [5, [a, b, c], 6]]], s], 3);
[6, 78, [[75, 42]], d, 1, 2, 3, [5, [a, b, c], 6], s]
> Flatten([6, 78, [[75, 42]], d, [[[1, 2, 3, [5, [a, b, c], 6]]], s], 1);
[6, 78, 75, 42, d, 1, 2, 3, 5, a, b, c, 6, s]
> Flatten({6, 78, [[75, 42]], d, {{{1, 2, 3, {5, {a, b, c}, 6}}}}, s), 2);
{1, 2, 3, 5, 6, 78, d, s, [[75, 42]], {a, b, c}}
> Flatten({6, 78, [[75, 42]], d, {{{1, 2, 3, {5, {a, b, c}, 6}}}}, s), 1);
{1, 2, 3, 5, 6, 78, a, b, c, d, s, [[75, 42]]}
```

Вызов процедуры *Flatten(L, n)* возвращает результат сведения вложенного множества или списка *L* к уровню вложенности, определенному вторым фактическим аргументом *n*; при этом, уровень невложенного списка или множества полагается равным единице. В целом ряде случаев при работе с данными объектами процедура *Flatten* оказывается достаточно эффективным и полезным средством.

Между тем, в отличие от пакета *Mathematica*, имеющего в качестве базовой структуры список «{}», *Maple* в качестве базовых использует как список «[]», так и множество «{}». В данном контексте второй пакет имеет целый ряд преимуществ в программировании конструкций, базирующихся на этих структурах; прежде всего, довольно существенно упрощается программирование целого ряда теоретико-множественных задач. В этой связи представляет интерес вопрос о максимальном уровне вложенности множества/списка безотносительно типов {список, множество} входящих в него элементов. Далее, интерес представляет и вопрос понижения уровней вложенности множества/списка безотносительно типов входящих в него элементов. Эти задачи решают 2 достаточно простые процедуры *MlsNest* и *FlattenFull*, представленные фрагментом ниже.

```

> MlsNest := (L::{set, list}) -> mlsnest(parse(`if`(type(L, list), sstr(["[" = "[", "]" = "]",
      convert(L, string)), sstr(["[" = "{", "]" = "}"], convert(L, string))))) [2][2] + 1:
> MlsNest([a, b, {c, d}, c, [1, 2, {4, [75, {42, 69}, 420], 5}, 3]]); => 5
> MlsNest([a, b, {c, d}, c, [1, 2, {4, [75, {42, 69}, 420], 5}, 3]]); => 5
> FlattenFull := proc(L::{set, list}, n::posint)
  local a, b, c, k;
    assign(a = [], c = L);
  for b to MlsNest(L) - n do
    for k in c do a := [op(a), `if`(type(k, {set, list}), op(k), k)] end do;
    c := a; a := []
  end do; convert(c, whattype(L))
end proc:
> FlattenFull([a, b, {c, d}, c, [1, 2, {4, [75, {42, 69}, 420], 5}, 3]], 2);
      [a, b, c, d, c, 1, 2, 4, 5, 75, {42, 69}, 420, 3]
> FlattenFull([a, b, {c, d}, c, [1, 2, {4, [75, {42, 69}, 420], 5}, 3]], 2);
      {1, 2, 3, 4, 5, 75, 420, {42, 69}, b, c, d, a}
> FlattenFull([a, b, {c, d}, c, [1, 2, {4, [75, {42, 69}, 420], 5}, 3]], 2);
      {1, 2, 3, 4, 5, 42, 69, 75, 420, b, c, d, a}

```

Вызов процедуры *MlsNest(L)* возвращает максимальный уровень вложенности списка/множества *L* безотносительно типов {список, множество} входящих в него элементов, в то время как вызов процедуры *FlattenFull(L, n)* возвращает результат сведения уровня вложенности списка/множества *L* безотносительно типов {список, множество} входящих в него элементов к уровню, определяемому вторым фактическим аргументом *n*. При этом, уровень невлóженного (*простого*) списка/множества *L* полагается равным 1.

По области применения к предыдущим процедурам примыкает и довольно полезная процедура *DifNestLevels(L)*, чей вызов возвращает 2-элементную последовательность множеств, *первое* из которых содержит уровни вложенности списков, входящих в список/множество *L*, тогда как второе – содержит уровни вложенности множеств, входящих в список/множество *L*. При этом, на пустом списке/множестве *L* вызов *DifNestLevels(L)* возвращает последовательность {}, {} пустых множеств. Фрагмент приводит исходный текст процедуры *DifNestLevels* наряду с некоторыми примерами ее применения.

```

> DifNestLevels := proc(L::{set, list})
  local a, b, c, d, k, l, s, h, t, p;
    assign(a = {}, b = {}, l = {}, s = {}, d = [op(L)]);
    `if`(type(L, nestlist), assign('l' = {1}), `if`(type(L, nestset), assign('s' = {1}), NULL));
  while d <> [] do t, p := 1, 1;
    for k in d do
      if type(k, list) then
        if t = 1 then l := {op(l), (proc() try l[-1] catch : 0 end try end proc)() + 1} end if;
        t := 0; h := [op(h), op(k)]

```

```

    elif type(k, set) then
    if p = 1 then s := {op(s), (proc() try s[-1] catch : 0 end try end proc)() + 1} end if;
    p := 0; h := [op(h), op(k)]
    end if
    end do; d := h; h := []
end do; l, s
end proc:

```

```

> DifNestLevels([1, 2, {3, 4, {4, 5}, {5, 6, {6, 15}}}, [c, d]]); ⇒ {1, 2}, {1, 2, 3}
> DifNestLevels([1, 2, {3, 4, {4, 5}, {5, 6, {6, 15}}}, [c, d]]); ⇒ {1}, {1, 2, 3, 4}
> DifNestLevels([1, 2, [c, g], s, t, {6, 15}, 22, {k, h}]); ⇒ {1}, {1, 2}
> DifNestLevels([[], {}, [], {}, {{75, 420, {42, 47}}}); ⇒ {1}, {1, 2, 3, 4}
> map(DifNestLevels, [[], {}]); ⇒ [{}, {}, {}, {}]
> DifNestLevels([[[[[]]]], [[[[[[]]]]]]); ⇒ {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}

```

Наряду с *типами* оба рассматриваемых пакета допускают создание различного рода тестирующих процедур **TestQ**{(W) | [W]}, вызов которых возвращает *true*-значение на *W*-объектах, удовлетворяющих заданным свойствам и/или атрибутам, или значение *false* в противном случае. Данный подход позволяет довольно существенно обобщать понятие «*типа*», классифицируя объекты по весьма широкому спектру их свойств. В таком понимании распознаваемые классы объектов в среде обоих пакетов могут быть произвольно определяемыми их свойствами и распознаваться соответствующими *Q*-тестами наподобие приведенных в предыдущем фрагменте простых примеров.

```

> StringTake := proc(x::string, y::anything)
    local a, b, c, d, k, f;
    d := `if`(type(y, {set, list}), map(convert, y, string), convert(y, string));
    f := x -> [seq(`if`(nops(k) < 2, NULL, k), k = x)];
    try a := SLj([seq([k[1], k[1] + k[2] - 1], k = f([seq([op(Search2(x, {b})), length(b)],
        b = `if`(type(d, string), {d}, d)))]), 1)
    catch : return x
    end try;
    c := [substring(x, 1 .. a[1][1] - 1)];
    for k to nops(a) - 1 do c := [op(c), substring(x, a[k][2] + 1 .. a[k + 1][1] - 1)] end do;
    [seq(`if`(k = "", NULL, k), k = [op(c), substring(x, a[k][2] + 1 .. -1)])]
end proc:

> StringTake("ransianavzagnvsvartkr", [ian, art, agn]); ⇒ ["rans", "avz", "vsv", "kr"]
> StringTake("ransianavzagnvsvartkr", {"ian", "420", "art"}); ⇒ ["rans", "avzagnvsv", "kr"]
> StringTake("ransianavzagnvsvartkr", {"ian", "agn", "art"}); ⇒ ["rans", "avz", "vsv", "kr"]
> StringTake("ransianavzagnvsvartkr", {ran, ian, agn, art, kr}); ⇒ ["s", "avz", "vsv"]
> StringTake("ransianavzagnvsvartkr", avz); ⇒ ["ransian", "agnvsvartkr"]
> StringTake("ransianavzagnvsvartkr", 75420); ⇒ "ransianavzagnvsvartkr"

```

Для получения подстрок строки, заданных их позициями *конца* и *начала*, пакет *Maple* располагает встроенной функцией *substring* и модульной процедурой *SubString*, но в целом ряде случаев более *удобным* оказывается получение *подстрок*, ограниченных не позициями, а заданными подстроками. Для этой цели служит процедура *StringTake*, чей исходный текст с примерами применения приведены в предыдущем фрагменте. Вызов процедуры *StringTake(x,y)* возвращает *список* подстрок строки *x*, ограниченных ее подстроками *y*; в качестве 2-го аргумента может выступать допустимое выражение либо их список. В случае отсутствия вхождений *y* в *x* возвращается исходная строка *x*.

Список является одной из базовых структур пакета *Maple*, используемой достаточно в широких пределах в организации структур других типов (*векторы, таблицы, матрицы и др.*); правда, его роль в определенной мере ниже, чем *списка* в пакете *Mathematica*. В программной среде *Maple* имеется целый ряд стандартных средств и встроенных, и в составе пакетного модуля *ListTools*, поддерживающих работу со списками, целый ряд средств данного типа представлен и нашей библиотекой [45]; однако, по отношению к пакету *Mathematica* эти средства менее представительны. Итак, если в *Mathematica* атрибут *Listable* во многом определяет возможность применения функций/процедур *отдельно* как к *каждому* элементу списка-аргумента, так и к *кортежам* соответствующих элементов каждого из списков в случае их множества, в пакете *Maple* такой механизм отсутствует, что существенно снижает возможности обработки *списочных* выражений. Так, если в *Mathematica* над списками одинаковой длины допустимы *арифметические* операции, то в пакете *Maple* из них допустима лишь операция *сложения*, как хорошо иллюстрируют первые пять примеров следующего фрагмента. Более того, операнды операции $\{+|- \}$ должны быть идентичной длины, иначе возникает ошибка. Шестой пример нижеследующего фрагмента хорошо иллюстрирует сказанное, а именно:

```
> [x, y, z] + [a, b, c];  => [a + x, b + y, c + z]
> [x, y, z] - [a, b, c];  => [-a + x, -b + y, -c + z]
> [x, y, z] * [a, b, c];  => [x, y, z] [a, b, c]
> [x, y, z] / [a, b, c];  => [x, y, z] / [a, b, c]
> [x, y, z] ^ [a, b, c];  => [x, y, z] ^ [a, b, c]
> `+`([x, y, z, h, g], [2, 3, 4]);
Error, adding lists of different length
> ListOp := proc(x::list, y::list, z::procedure)
  local a, b, c, k;
    assign(a = nops(x), b = nops(y)), assign(c = min(a, b));
    ListTools[Flatten]([seq(z(x[k], y[k]), k = 1 .. c), x[c + 1 .. -1], y[c + 1 .. -1]], 1)
  end proc;
> ListOp([x, y, z, h, g], [2, 3, 4, 5, 6], `*`);  => [2 x, 3 y, 4 z, 5 h, 6 g]
> ListOp([x, y, z, h, g], [2, 3, 4], `+`);  => [x + 2, y + 3, z + 4, h, g]
> Z := proc(x, y) `or`(x, y) end proc;
> ListOp([true, false, true, false], [true, true, false, true], Z);  => [true, true, true, true]
```

С целью устранения данного недостатка предлагается процедура *ListOp(x, y, z)*, чей вызов возвращает список, элементами которого являются результаты применения *z-*

процедуры/функции к соответствующим элементам списков x и y ; при этом, в случае различных длин данных списков описанная процедура применяется к обоим спискам в рамках минимальной длины списков, оставляя остальные элементы большего списка без изменения. Фрагмент представляет исходный текст процедуры *ListOp* и примеры ее применения, достаточно наглядно иллюстрирующие сказанное. Ниже представлен ее аналог для *Mathematica*, расширяющий класс функций в качестве 3-го аргумента.

Для манипулирования списками пакет *Mathematica* располагает такими функциями, как **Append**, **Delete**, **Insert**, **Prepend**, **ReplacePart**, **Riffle**, обеспечивающими операции различного типа добавления, удаления и замены элементов списка. Ниже приведена процедура *InsDelList(L,O)* от двух до четырех аргументов, обеспечивающая операции *Append*, *Delete*, *Insert*, *Prepend*, аналогичные одноименным операциям в *Mathematica*. В качестве первого аргумента L выступает обрабатываемый список, тогда как второй аргумент O принимает значения из множества $\{\text{'app'}, \text{'del'}, \text{'ins'}, \text{'prep'}\}$ и определяет операции *Append*, *Delete*, *Insert*, *Prepend* соответственно. Вызов *InsDelList(L, 'prep', E)* возвращает результат дополнения списка L слева выражением E , в то время как вызов *InsDelList(L, 'app', E)* возвращает результат дополнения списка L справа выражением E ; вызов *InsDelList(L, 'del', N)* возвращает результат удаления из списка L элементов с позициями N , тогда как вызов *InsDelList(L, 'ins', E, N)* возвращает результат вставки в список L выражения E в позиции, заданные N ; при этом, в обоих случаях в качестве N аргумента может выступать или положительное целое, или их множество. Следующий фрагмент представляет исходный код процедуры *InsDelList* и примеры применения.

```
> InsDelList := proc(L::list, O::anything)
  local a, b, k;
  if not member(O, {'app', 'del', 'ins', 'prep'}) then error "2nd argument should be from
    {'app', 'del', 'ins', 'prep'} but had received %1", O
  elif nargs = 4 then a := `if`(type(args[4], posint), {args[4]}, `if`(type(args[4], set(posint)),
    args[4], ERROR("4th argument should be posint or set(posint) but
    had received %1", args[4])))
  end if;
  if O = `prep` and nargs = 3 then [args[3], op(L)]
  elif O = `app` and nargs = 3 then [op(L), args[3]]
  elif O = `ins` and nargs = 4 then [seq(`if`(member(k, a), op([args[3], L[k]]), L[k]),
    k = 1 .. nops(L))]
  elif O = `del` and nargs = 3 then b := `if`(type(args[3], posint), {args[3]},
    `if`(type(args[3], set(posint)), args[3], ERROR("3rd argument should
    be posint or set(posint) but had received %1", args[3])));
    [seq(`if`(member(k, b), NULL, L[k]), k = 1 .. nops(L))]
  end if
end proc;

> InsDelList([a, b, x, c, d, f, g, h], `del`, {3, 5, 6});    => [a, b, c, g, h]
> InsDelList([a, b, x, c, d, f, g, h], `ins`, G, {3, 5, 6}); => [a, b, G, x, c, G, d, G, f, g, h]
```



```

> InsDelList([a, b, x, c, d, f, g, h], `prep`, agn);      ⇒ [agn, a, b, x, c, d, f, g, h]
> InsDelList([a, b, x, c, d, f, g, h], `app`, avz);      ⇒ [a, b, x, c, d, f, g, h, avz]
> InsDelList([a, b, c, d, f, g, h], `ins`, {75, 420}, 6); ⇒ [[a, b, c, d, f, {75, 420}, g, h]]
> InsDelList([a, b, x, c, d, f, g, h], `Ins`, G, {3, 5, 6, 15});
Error, (in InsDelList) 2nd argument should be from {`app`, `del`, `ins`, `prep`} but had
received `Ins`

```

Процедура обрабатывает *основные* ошибочные ситуации. В качестве весьма полезного упражнения читателю рекомендуется расширить процедуру *InsDelList* упомянутыми операциями **ReplacePart** и **Riffle**, что не должно представить особых затруднений.

В целом ряде случаев при работе с *индексными* выражениями, т.е. выражениями типа *`indexed`*, требуется вычислять *индексы*. Вызов следующей простой процедуры *Index(x)* возвращает индекс произвольного выражения *x* *indexed*-типа, а именно:

```

> Index := proc(x::indexed)
  local a, b;
    a := convert(x, string); search(a, "[", b); parse(a[b + 1 .. -2])
  end proc;
> map(Index, [a[a + b], cd[75], agn[x + 420], vsv[Art*Kr]]); ⇒ [a + b, 75, x + 420, Art Kr]
> Commonest := proc(L::list)
  local a, k;
  for k in L do a[k] := a[k] + 1 end do;
    a := SLj(map(convert, convert(a, list), list), 2);
    map(Index, [seq(`if` (k[2] = a[-1][2], k[1], NULL), k = a)], a[-1][2])
  end proc;
> Commonest([1, 2, a+b, 4, 11, 6, 2, 5, 4, 6, 75, 75, a+b, 6, a+b, 75, 11]); ⇒ [6, a + b, 75], 3

```

Так, данная процедура существенно используется процедурой, приведенной в том же фрагменте, чей вызов *Commonest(L)*, возвращает 2-элементную последовательность, в которой 1-й элемент определяет список элементов списка *L*, имеющих максимальное число вхождений, и 2-й элемент определяет их кратность вхождений. Эта процедура в определенной мере является аналогом одноименной функции в *Mathematica*, имея достаточно хорошие предпосылки для расширения своей функциональности.

```

> PadList := proc(L::list, R::{-1, 0, 1, set(posint)}, x::anything, n::posint)
  local a, b, k;
  assign(a = [seq(x, k = 1 .. n)], b = nops(L));
    `if` (R = -1, map(op, [L, a]), `if` (R = 0, map(op, [a, L]),
    `if` (R = 1, [seq(op([k, op(a)]), k = L[1 .. -2]), L[-1]],
    [seq(`if` (member(k, R), op([L[k], op(a)]), L[k]), k = 1 .. b)])))
  end proc;
> PadList([69, 64, 44, 15, 22], -1, h, 6); ⇒ [69, 64, 44, 15, 22, h, h, h, h, h, h]
> PadList([69, 64, 44, 15, 22], 0, h, 6); ⇒ [h, h, h, h, h, h, 69, 64, 44, 15, 22]
> PadList([69, 64, 44, 15, 22], 1, h, 3); ⇒ [69, h, h, h, 64, h, h, h, 44, h, h, h, 15, h, h, h, 22]

```

В качестве достаточно полезного аналога *Math*-функций **PadLeft**, **PadRight** обработки списков выступает процедура **PadList**, чьи исходный текст и примеры использования представлены в предыдущем фрагменте. Вызов **PadList(L, R, x, n)** возвращает список, являющийся результатом *вставки* кортежа длины *n* из выражений *x* *перед* (**R = 0**), *после* (**R = -1**) списка *L*, и *разделения* (**R = 1**) элементов списка *L* данным кортежем. Более того, если в качестве аргумента **R** выступает множество положительных целых, то данный кортеж размещается после элементов списка *L* с позициями, определенными **R**.

Вызов стандартной процедуры **unwith(P)** пакета *Maple* предназначен для удаления из текущего сеанса всех экспортов пакетного модуля *P*, возвращая **NULL**, т.е. ничего, и в этом отношении данная процедура по назначению противоположна процедуре **with**. Между тем, вызов **unwith(P)** имеет смысл лишь для пакетных модулей, реализованных именно в форме *модулей*, и не применим к пакетным модулям, реализованным либо таблицей, либо процедурой. При попытке вызова на такого типа объектах возникает ошибочная ситуация с возвратом соответствующей диагностики. Для пакета *Maple 11* в качестве таких исключительных пакетных модулей выступают следующие:

algcures, combstruct, DEtools, diffalg, diffforms, intrtrans, liesymm, LRETools, networks, simplex, Slode, stats, student, tensor

Вызов процедуры **Unwith(P)** расширяет вызовы стандартной процедуры **unwith(P)** на такие типы пакетных модулей, как *таблицы* и *процедуры*. Нижеследующий фрагмент представляет исходный код процедуры и некоторые примеры ее применения.

```
> restart; GS := with(Slode): unwith(Slode);
Error, (in unwith) the procedure 'unwith' is currently disabled due to the presence of
table-based packages
> GS;
[DEdetermine, FPseries, FTseries, candidate_mpoints, candidate_points, dAlembertian_formal_sol,
hypergeom_formal_sol, hypergeom_series_sol, mhypergeom_formal_sol, mhypergeom_series_sol,
mparse_series_sol, polynomial_series_sol, rational_series_sol, series_by_leastsquare]
> eval(DEdetermine); => proc() `Slode/DEdetermine`(args) end proc
> Unwith := proc(P::package)
    local a, k;
    assign(a = With6(P)), unprotect(op(a));
    for k in a do parse(cat("unassign(", k, "):"), 'statement') end do end proc;
> Unwith(Slode);
> map(eval, [DEdetermine, FPseries, FTseries]); => [DEdetermine, FPseries, FTseries]
```

Процедура **Unwith** использует нашу процедуру **With6**, которая устраняет недостатки стандартной процедуры **with** при вызове ее в теле процедуры либо модуля [45].

Вызов **MapAt(F, L, n)** простой процедуры, реализованной *функциональным* оператором, возвращает результат применения к элементам списка/множества *L*, *позиции* которых определены положительным целым или их множеством *n*, процедуры либо функции *G*. Следующий фрагмент представляет исходный код процедуры и ее применение. В определенном отношении процедура – аналог одноименной функции *Mathematica*.

```

> MapAt := (F::{symbol, function, procedure}, L::{set, list}, n::{posint, set(posint)}) ->
    convert([seq(`if` (member(k, Flatten({n}, 1)), `if` (type(eval(F), symbol),
    F(L[k]), `if` (type(F, {function, procedure}) and type(F, arity(nops(L[k])),
    F(op(L[k])), L[k])), L[k]), k = 1 .. nops(L)), whattype(L));
> MapAt(G, [69, 64, 44, 22, 15], {2, 4}); => [69, G(64), 44, G(22), 15]
> MapAt(G, {69, 64, 44, 22, 15}, {2, 4}); => {15, 44, 69, G(64), G(22)}
> W := (x, y) -> x + y: MapAt(W, {69, 64, 44, 22, 15}, 3); => {15, 22, 44, 64, 69}
> MapAt(W, [69, 64, [75, 400], 22, {15, 22}], {3, 5}); => [69, 64, 475, 22, 37]
> S := (x, y, z) -> x + y + z: MapAt(S, [6, [5, 7], 22, {15, 22, 70}], {2, 4}); => [6, [5, 7], 22, 107]

```

При этом, если в качестве первого аргумента F выступает процедура или функция, то она ассоциируется с соответствующим элементом L только в том случае, если арность F равняется числу элементов данного элемента L , иначе такой элемент игнорируется.

В целом ряде задач, обусловленных обработкой строчного представления *определений* процедур, определенный интерес представляет задача разбиения данного строчного представления на две основные компоненты, а именно: *заголовок* и *описательная часть*, а также *тело* процедуры с закрывающей процедурной скобкой «*end proc*». Например, такая процедура может быть эффективно использована при выполнении различных подстановок над *телом* процедуры, не затрагивающих ее заголовка и описательной. В этом отношении определенный интерес представляет процедура *partproc*, исходный код которой и примеры применения представлены следующим фрагментом:

```

> partproc := proc(P::procedure)
    local a, b, c, d, f, k, t;
        t := interface(verboseproc);
    if member(true, map(Pref1, {op(3, eval(P))}, "Copyright")) then interface(verboseproc = 2)
    end if;
    assign(a = table([1 = "", 2 = "local ", 3 = "option ", 5 = "description ", 6 = "global "]),
        b = [1, 2, 3, 5, 6], c = "", f = convert(eval(P), string));
    for k in b do d := op(k, eval(P));
        if evalb(d <> NULL) then a[k] := cat(a[k], seqstr(d), "; ") else a[k] := "" end if
    end do;
        c := `if` (a[1] = "", "proc () ", cat("proc (", a[1][1 .. -3], ") "));
        seq(assign('c' = cat(c, `if` (a[k] <> "", a[k], ""))), k = b[2 .. -1]);
        interface(verboseproc = t), [c, f[length(c) .. -1]]
    end proc;
> P:=proc(x,y,z) local a,b,c; global f,g,h; options system; description "ab"; x+y+z end proc:
> partproc(P);
    ["proc (x, y, z) local a, b, c; option system; description "abcd"; global f, g, h; ",
    " x+y+z end proc"]
> G:=(x,y) -> x+y: partproc(G); => ["proc (x, y) option operator, arrow; ", "; x+y end proc"]

```

Вызов процедуры *partproc(P)* возвращает 2-х элементный список, чей *первый* элемент в строчном формате представляет *заголовок* и *описательную* часть процедуры *P*, *второй* элемент – в строчном формате *тело* с закрывающей процедурной скобкой «*end proc*». Приведенные примеры применения процедуры достаточно наглядно иллюстрируют суть реализуемого ею алгоритма и особых пояснений не требуют. В качестве примера использования процедуры *partproc* приведем реализацию процедуры *WithM*, которая оказывается полезной в целом ряде приложений, использующих подстановки.

В отличие от *With* функции *Mathematica* функция *subs* и процедура *algsubs* *Maple* не обеспечивают безусловных подстановок в *тело* процедур. Тогда как данная проблема успешно решается следующей процедурой *WithM*, чей вызов *WithM(E, X)* возвращает результат замены *всех* вхождений в выражение *X* символов, задаваемых *левыми* частями *уравнений* списка *E*, на соответствующие им *правые* части. При этом, в качестве *E* может выступать и отдельное уравнение. Следующий фрагмент представляет исходный код процедуры наряду с примерами применения *WithM*, а также *subs* и *algsubs*. Следует отметить, что функция *With* пакета *Mathematica*, в свою очередь, подобно функциям *Replace* и *ReplaceAll* достаточно существенно уступает нашей процедуре *Subs* [90].

```
> Pref1 := (S::{symbol, string}, p::{string, symbol}) -> `if` (length("'" | | S) >= length("'" | | p) and
    "'" | | S[1..length("'" | | p)] = "'" | | p, true, false);

> Pref1(RANS_IAN_RAE_RAC_2011, "RANS_IAN"); => true

> WithM := proc(E::`=`, list(`=`), set(`=`)), X::anything)
    local a, c, t;
    if not type(X, `procedure`) then Subs(E, X) else
        if member(true, map(Pref1, {op(3, eval(X))}, "Copyright")) then
            interface(verboseproc = 2); t := interface(verboseproc)
        else t := interface(verboseproc)
        end if;
        assign(a = partproc(X), c = "proc () ");
        c := Subs(E, parse(cat(c, a[2]))); interface(verboseproc = t);
        parse(cat(a[1], convert(eval(c), string)[8 .. -1]))
    end if
end proc;

> subs([x = a+b, y = (c+d), z = 75, h = 420], proc(x, y, z) x+y+z end proc);
Error, invalid substitution in proc

> subs([x = a+b, y = (c+d), z = 75, h = 420], proc() x+y+z end proc);
    proc() a + b + c + d + 75 end proc

> subs([x = a+b, y = c+d, z = 75], proc(p, q, r) x+y+z end proc);
    proc(p, q, r) a + b + c + d + 75 end proc

> algsubs(x = a+b, proc() x+y+z end proc);
    proc() x + y + z end proc

> WithM(x = a+b, proc(x, y, z) x+y+z end proc);
    proc(x, y, z) a + b + y + z end proc
```

```

> WithM([x = a+b, y = (c+d), z = 75, h = 420], x^2+y^2+z^2);
      (a + b)^2 + (c + d)^2 + 5625
> WithM([x^2 = a+b, y = (c+d), z = 75, h = 420], x^2+y^2+z^2);  ⇒ a + b + (c + d)^2 + 5625
> WithM([x = a+b, y = (c+d), z = 75, h = 420], 2011);          ⇒ 2011
> WithM([x = a+b, y = (c+d), z = 75, h = 420], 2011 + xyz);    ⇒ 2011 + xyz
> WithM([x = a+b, y = (c+d), z = 75, h = 420], proc(x, y, z) x + y + z end proc);
      proc(x, y, z) a + b + c + d + 75 end proc
> G := proc(x, y, z) local m, n, p; x + y + z end proc;
> V := proc() option `Copyright W`; x + y + z end proc;
> WithM([x = a+b, y = (c+d), z = 75, h = 420], G);
      proc(x, y, z) local m, n, p; a + b + c + d + 75 end proc
> WithM([x = a+b, y = (c+d), z = 75, h = 420], V);
      proc() option `Copyright W`; a + b + c + d + 75 end proc
> WithM([x^2 = a, y^3 = b], proc(y, a) 1/x^2+1/y^3 end proc);
      proc(y, a) 1/a + 1/b end proc
> WithM([x = a, y = b], proc(x, y) 1/x^2 + 1/y^3 end proc);
      proc(x, y) 1/a^2 + 1/b^3 end proc
> WithM([x=a, y=b], proc() 1/x^2 + 1/y^3 end proc);  ⇒ proc() 1/a^2 + 1/b^3 end proc
> WithM([x^2 = a+b, y = b+d], proc(x, y) 1/x^2 + 1/y^3 end proc);
      proc(x, y) 1/(b + a) + 1/(b + d)^3 end proc
> WithM([x = a+b, y = (c+d), z = 75, h = 420], 2011 + xyz);
      Subs([x = b + a, y = c + d, z = 75, h = 420], 2011 + xyz)

```

Фрагмент начинается с процедуры, реализованной функциональным оператором, чей вызов *Pref1(S, p)* возвращает *true*, если строка/символ *S* начинается со строки/символа *p*, и *false* в противном случае. Процедура довольно полезна при работе со строками. В следующей за ней процедуре *WithM* существенно используются как процедура *Pref1*, так и процедура *partproc*, рассмотренная выше. Процедура *WithM* расширяет пакет в его применениях для решения задач, использующих подстановки в выражения. При этом, в случае невозможности выполнить подстановку возвращается невычисленным вызов процедуры *Subs*, используемой *WithM* для реализации подстановок. Между тем, процедуру *WithM* возможно несложно модифицировать, используя для реализации подстановок вместо нашей процедуры *Subs* стандартную процедуру *subs*. Естественно, в этом случае, с одной стороны, сузится возможность выполнения целого ряда весьма важных типов подстановок, но, с другой стороны, расширится круг обрабатываемых подстановками выражений. Оставляем это читателю в качестве полезного упражнения.

```

> Split := proc(L::list)
  local a, b, k;
    a := {op(L)}; b := table([seq(k = NULL, k = a)]); for k in L do b[k] := b[k], k end do;
    sort([entries(b)], length)
  end proc;
> Split([a, g, d, a, a, h, g, d, f, f, h, d, a, a]);  ⇒ [[f, f], [g, g], [h, h], [d, d, d], [a, a, a, a, a]]

```

```

> Split1 := proc(L::list)
  local a, b, c;
  if L = [] then [] else a := NULL; b := [L[1]];
    for c from 2 to nops(L) do
      if b[-1] = L[c] then b := [op(b), L[c]] else a := a, b; b := [L[c]] end if
    end do;
    [a, b]
  end if
end proc:

> Split1([a, a, a, b, b, a, a, c, b, b, b, c, c, c, d, d, d, d]), Split1([75, 75]), Split1([]);
[[a, a, a], [b, b], [a, a], [c], [b, b, b], [c, c, c], [d, d, d, d]], [[75, 75]], []

> SplitBy := proc(L::list, F::{function, procedure})
  local a, b, k;
  if L = [] then [] else a := Split1(map(F, L)); b := [1 .. nops(a[1])];
  for k in a[2 .. -1] do b := [op(b), rhs(b[-1]) + 1 .. rhs(b[-1]) + nops(k)] end do;
    [seq(L[k], k = b)]
  end if
end proc:

> SplitBy([1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5], floor); ⇒ [[1, 3/2], [2, 5/2], [3, 7/2], [4, 9/2], [5]]
> SplitBy([75], floor), SplitBy([], floor); ⇒ [[75]], []

```

Вызов первого формата функции **Split[L]** пакета *Mathematica* возвращает вложенный список, чьи подсписки содержат следующие подряд идентичные элементы списка *L*, в случае пустого списка *L* возвращается пустой список, т.е. []. В то время как процедура **Split1(L)**, представленная предыдущим фрагментом, решает ту же задачу; между тем, процедура **Split(L)** решает до некоторой степени *подобную* задачу, однако с возвратом вложенного списка, чьи подсписки, отсортированные в порядке возрастания их длин, содержат идентичные элементы списка *L* безотносительно их положения в списке. В свою очередь, процедура **SplitBy(L, F)**, базирующаяся на процедуре **Split1**, возвращает *вложенный* список, чьи подсписки содержат следующие подряд идентичные элементы списка *L* при условии, что идентичность определяется относительно списка **map(F, L)**; в случае пустого списка *L* процедура **SplitBy** возвращает пустой список, т.е. [].

Math-функция **Composition[f1, ..., fn]** определяет *композицию* функций {f1, ..., fn}, тогда как следующая одноименная *Maple*-процедура выполняет ту же самую функцию:

```

> Composition := proc(F::list(symbol), V::anything)
  local a, k;
  a := ""; seq(assign('a' = cat(a, seqstr(k), "(")), k = F);
  eval(parse(cat(a, `if` (type(V, list), seqstr(op(V)), seqstr(V)), seq(")", k = 1 .. nops(F)))))
end proc:

> Composition([F, G, H, S, W], [x, y, z, t, h]), Composition([F, G, H, S, W], (a+b)/(c+d));
F(G(H(S(W(x, y, z, t, h))))), F(G(H(S(W((a+b)/(c+d)))))

```

```

> ComposeList := (F::list(symbol), x::anything) ->
    [x, seq(Composition(Rlss(F)[1 .. k], x), k = 1 .. nops(F))]
> ComposeList([F, G, S, V], x);  => [x, V(x), V(S(x)), V(S(G(x))), V(S(G(F(x))))]
> FixedPointList := proc(F::{function, `procedure`}, x::anything, n::posint)
    local a, b, c;
    assign(a = evalf(F(x), n), b = [evalf(F(x), n)], c = convert([22], 'bytes'));
    while c <> a do a := evalf(F(a), n); c := evalf(F(a), n); b := [op(b), a] end do;
    [op(b), c]
end proc;
> FixedPointList(sin, 75, 2);  => [-0.39, -0.38, -0.37, -0.36, -0.35, -0.34, -0.33, -0.32, -0.31, -0.31]
> FixedPointList(sqrt, 75, 4);
    [8.660, 2.943, 1.716, 1.310, 1.145, 1.070, 1.034, 1.017, 1.008, 1.004, 1.002, 1.001, 1.000, 1.000]
> map(F, [x, y, z], n);  => [F(x, n), F(y, n), F(z, n)] # Maple 6 и выше
> map(evalf, [1.1234, 2.1234, 3.1234, 4.1234, 5.1234, 6.1234], 3); # Maple 8
    [1.1234, 3., 2.1234, 3., 3.1234, 3., 4.1234, 3., 5.1234, 3., 6.1234, 3.]
> map(evalf[3], [1.1234, 2.1234, 3.1234, 4.1234, 5.1234, 6.1234]); # Maple 8
    [1.12, 2.12, 3.12, 4.12, 5.12, 6.12]
> map(evalf, [1.1234, 2.1234, 3.1234, 4.1234, 5.1234, 6.1234], 3); # Maple 11
    [1.12, 2.12, 3.12, 4.12, 5.12, 6.12]
> FixedPoint := proc(F::{function, procedure}, x::anything, n::posint)
    local a, b, c;
    assign(a = evalf(F(x), n), b = [evalf(F(x), n)], c = convert([22], 'bytes'));
    while c <> a do a := evalf(F(a), n); c := evalf(F(a), n) end do
end proc;
> [FixedPoint(sin, 75, 2), FixedPoint(sqrt, 75, 4)];  => [-0.31, 1.000]

```

В то время как процедура *ComposeList*, реализованная функциональным оператором на основе *Composition*, эквивалентна одноименной процедуре пакета *Mathematica*. В завершение фрагмента представлены исходный код процедуры *ComposeList* наряду с примером ее применения, из которого смысл результата ее вызова вполне очевиден, и процедура *FixedPointList(W, x, n)*, функционально идентичная одноименной функции *Mathematica*; процедура генерирует список результатов повторного применения *W* к исходному значению *x* с точностью до *n* десятичных знаков до тех пор, пока результат не будет повторяться. Здесь же вполне уместно сделать одно достаточно существенное замечание. Формально применение встроенной функции *map* во всех релизах *Maple* идентично, однако, например, в сочетании с функцией *evalf* результат в 8-м релизе *Maple* некорректен, как иллюстрируют последние примеры фрагмента. И получение корректного результата требует индексации функции *evalf* третьим аргументом, как иллюстрирует пример фрагмента. В последующих релизах данная особенность была устранена возвращением к принятому для *map* формализму. Более детально на этом вопросе останавливаться не считаем целесообразным. Это еще один довольно веский

довод в подтверждение остроты проблемы несовместимости релизов пакета *Maple*. В отличие от *FixedPointList*, вызов процедуры *FixedPoint(W, x, n)* возвращает не список, а единственное значение – фиксированную точку – постоянный результат повторного применения к x функции/процедуры W .

Следующая процедура служит некоторым аналогом *Distribute* функции *Mathematica*, а также расширением встроеной функции *map* пакета *Maple*. Следующий фрагмент представляет исходный код процедуры и примеры ее применения. Вызов процедуры *mapt(F, expr, t)* возвращает список результатов применения функции/процедуры F к операндам выражения *expr*, тип которых соответствует типу, заданному аргументом t . В случае отсутствия требуемых операндов вызов *mapt* возвращается невычисленным.

```
> mapt := proc(F::[function, `procedure`, symbol], expr::anything, t::type)
  local a, b, d, k;
  assign(a = convert(expr, list), b = NULL);
  for k to nops(a) do
    d := op(k, a); if type(eval(d), t) then b := b, d end if
  end do;
  `if`(b = NULL, 'procname(args)', `if`(nops([b]) = 1, F(b), map(F, [b])))
end proc;

> mapt(G, [a, sin, c, cos, 75, tan], procedure);  => [G(sin), G(cos), G(tan)]
> mapt(G, a*cos(x), anything);                  => [G(a), G(cos(x))]
> mapt(G, 75/420, `*`);                          => mapt(G, 5/28, `*`)
> mapt(G, (a+b)/(c+d), `^`);                     => G(1/(c+d))
> mapt(G, (a+b)*sin, procedure);                 => G(sin)
> mapt(G, (a+b)*sin, `+`);                       => G(a + b)
> mapt(G, a/(c+d), anything);                   => [G(a), G(1/(c+d))]
```

В обработке списочных структур интерес представляет задача группировки элементов вложенных списков *listlist*-типа на основе n -х элементов их подписков. Данную задачу решает процедура, вызов которой *ListListGroup(x, n)* возвращает вложенный список – результат группировки *listlist*-списка x по n -му элементу его подписков. Процедура аналогична одноименной процедуре *Mathematica*, представленной несколько ниже.

```
> ListListGroup := proc(L::listlist, n::posint)
  local a, b, c;
  if nops(L[1]) < n then 'procname(args)' else a := table([]);
    for b in L do a[b[n]] := a[b[n]], b end do; b := [entries(a)]; [seq(c[2 .. -1], c = b)]
  end if
end proc;

> ListListGroup([[[75, 2], [420, 6], [15, 2], [22, 2], [69, 6]], 2);
  [[75, 2], [15, 2], [22, 2]], [[420, 6], [69, 6]]
> ListListGroup([[[75, 2], [420, 6], [15, 2], [22, 2], [69, 6]], 6);
  ListListGroup([[[75, 2], [420, 6], [15, 2], [22, 2], [69, 6]], 6)
```


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 *ExprOfStr*, исходный код которой с примерами ее применения представляет нижеследующий фрагмент.

```

> ExprOfStr := proc(S::string, m::posint, n::{-1, 1}, L::{set(string), list(string)})
local a, b, c, d, k;
if not belong(m, 1 .. length(S)) then return 'procname(args)'
else assign(a = "", b = true, c = length(S),
    d = interface(warnlevel)), interface(warnlevel = 0)
end if;
if n = 1 then for k from m to c do a := cat(a, S[k]);
    try parse(a, 'statement') catch: b := false end try;
    if b = false then b := true; next
    elif k = c or member(S[k], L) then
        if k = c then a := a[1 .. -1] end if; break else b := true; next
    end if
end do
else for k from m by -1 to 1 do a := cat(S[k], a);
    try parse(a, 'statement') catch: b := false end try;
    if b = false then b := true; next
    elif k = 1 or member(S[k], L) then
        if 1 < k then a := a[2 .. -1] end if; break
    else b := true; next
    end if
end do
end if;
interface(warnlevel = d);
if b = false then false else try parse(a, 'statement') catch: return false end try;
    if n = 1 and member(a[-1], L) then a[1 .. -2]
    elif member(a[1], L) then a[2 .. -1] else a
    end if
end if
end proc;

> ExprOfStr("123456 avz := proc(x) local a; a:=x^2+75; a end proc; 789", 8, 1, {";"});
    "avz := proc(x) local a; a:=x^2+75; a end proc;"
> ExprOfStr("123456a:=x^2+75;789", 3, 1, {";"});  => false
> ExprOfStr("123456a:=x^2+75;789;", 7, 1, {";"});  => "a:=x^2+75;"
> ExprOfStr("12 agn3456789abc;", 14, -1, {" "});  => "agn345678"

```

Вызов процедуры *ExprOfStr(w, m, n, L)* возвращает результат извлечения из строки *w*, ограниченной ее *m*-й позицией и концом, *первого* корректного выражения с условием, что поиск производится *влево* (*n=-1*)/*вправо* (*n=1*) от заданной позиции и следующий

или предыдущий за найденным выражением символ должен принадлежать списку *L*. Результат вызова процедуры *ExprOfStr* возвращается в *строчном* формате; при этом, в отсутствие корректного выражения возвращается *false*, в то время как вызов *ExprOfStr* на *недопустимых* фактических аргументах возвращается *невычисленным*. Приведенные выше примеры иллюстрируют сказанное. Ниже будет представлен *аналог* процедуры *ExprOfStr* в программной среде пакета *Mathematica*.

На основе вышепредставленной процедуры *ExprOfStr* достаточно несложно получить решение интересной задачи по определению списка подпроцедур для произвольной процедуры *P*. Данную задачу успешно решает процедура *SubProcs(P)*, вызов которой возвращает 2-элементную последовательность, чей *первый* элемент определяет список *заголовков* подпроцедур, составляющих *главную* процедуру *P*, включая имя и самой *P*, тогда как второй элемент – список сгенерированных имен всех таких подпроцедур в форме *Name__*, включая *главную* процедуру *P*; более того, все процедуры с указанным форматом именами активизируются в текущем сеансе пакета. Между 2-я элементами этой *последовательности* имеет место взаимно-однозначное соответствие. Следующий фрагмент представляет исходный код процедуры с примерами ее применения.

```
> SubProcs := proc(P::`procedure`)
  local a, b, c, d, h, k, t;
    assign67(t = interface(verboseproc)), interface(verboseproc = 3);
    assign(a = cat(" ", P, " := ", convert(eval(P), string), ";"),
    assign(b = Search3(a, {" := proc ("), h = [], d = []);
    c := [seq(k[1], k = op(b)[2 .. -1]); interface(verboseproc = t);
  for k in c do
    h := [op(h), ExprOfStr(a, k - 1, -1, {" "})];
    d := [op(d), ExprOfStr(a, k + 4, 1, {";", ":", " "})]
  end do;
    map(parse, [seq(cat(h[k], "__", " := ", d[k]), k = 1 .. nops(h)), 'statement'];
    map(parse, h, 'statement'), map(parse, [seq(cat(k, "__"), k = h)], 'statement')
  end proc;
> P := proc(x, y) local a, b, P1, P2; P1 := proc(z, h) local m, n; z^2 + h^2 end proc;
    P2 := proc(z) local P3; P3 := proc(z) z^4 + P1(z, 75) end proc; P3(z) end proc;
    x*P2(x) + P1(x, y) + P2(y) end proc;
> SubProcs(P); => [P, P1, P2, P3], [P__, P1__, P2__, P3__]
> eval(P2__); => proc(z) local P3; P3 := proc(z) z^4 + P1(z, 75) end proc; P3(z) end proc
> G := proc(x, y, z) x + y + z end proc: SubProcs(G); => [G], [G__]
> eval(G__); => proc(x, y, z) x + y + z end proc
```

На основе процедуры *SubProcs* достаточно несложно запрограммировать процедуру, чей вызов *SubsProcQ(P, V)* возвращает *true*, если *V* определяет *подпроцедуру* процедуры *P*, и *false* в противном случае. Для *Maple* эта задача существенно проще аналогичной задачи для *Mathematica*, т.к. в отличие от второй она идентифицирует процедуры по их именам, тогда как во второй – по их заголовкам, о чем говорится детальнее ниже.

В качестве примера приведем определения трех идентичных по смыслу процедур, из

```
In[868]:= P[x_] := Module[{}, x]; P[y_] := Module[{}, y]; P[y_ /; PrimeQ[y]] := Module[{}, y]
In[869]:= InputForm[DefFunc[P]]
Out[869]//InputForm=
      P[y_ /; PrimeQ[y]] := Module[{}, y]
      P[y_] := Module[{}, y]
```

которых пакет полагает две различными из-за наличия у них разных заголовков; при этом, заголовки процедур должны различаться по существу, что и видно из примера.

```
> SubsProcQ := proc(x::`procedure`, y::`procedure`)
  local a, b, k;
  assign(a = [SubProcs(P)], b = []);
  for k in a[2] do b := [op(b), cat(convert(k, string)[1 .. -3], " := ", convert(eval(k), string))]
  end do;
  member(cat(convert(y, string), " := ", convert(eval(y), string)), b)
end proc;
> P := proc () local a, P1, P3; P1 := proc () local P2; `+`(args); P2 := proc () local b; [args] end
proc; b = 75 end proc; P3 := proc() nops(args)*6 end proc; 420 end proc: P2:=proc() local b;
[args] end proc: P3 := proc(x) local b; [args]*x end proc: map2(SubsProcQ, P, [P2, P3, AW]);
      [true, false, false]
```

Представленные в данном разделе средства вполне обозримы и довольно прозрачны, составляя лишь очень небольшую часть нашей Библиотеки [45]. Они предназначены для иллюстрации как организации процедур в пакете *Maple*, так и для используемых ими ряда полезных приемов программирования. Немалое количество нестандартных приемов программирования приведено в процедурах данной Библиотеки, исходные тексты которых не только доступны из Библиотеки непосредственно, но и приведены в отдельных *mws*-файлах, которые дополнительно содержат как нетипичные, так и типичные примеры их применения наряду с различными вариантами одной и той же процедуры. Между тем, при использовании нашей Библиотеки следует иметь в виду следующее важное обстоятельство. Ввиду достаточно частого изменения внутренней организации целого ряда стандартных средств пакета (часто, не вполне обоснованного) от релиза к релизу, а также изменений самой среды *Windows* некоторые средства из нашей Библиотеки могут вести себя не совсем в соответствии с их описанием. Между тем, достаточно опытный пользователь всегда имеет возможность изменять нужным образом требуемое программное средство и обновлять его в Библиотеке либо в своей собственной библиотеке, или создать на его основе новое. Более того, представленные примеры процедур иллюстрируют достаточно простой механизм как модификации, так и пополнения стандартных библиотечных средств пакета. В заключение раздела вкратце остановимся на вопросах отладки *Maple*-процедур, которая используется, в основном, при разработке достаточно сложных и объемных процедур. Между тем, из нашего опыта многолетнего использования *Maple* следует, без данных средств можно обходиться, используя возможности интерактивного режима программирования.

6.1.9. Элементы отладки процедур и функций в среде Maple

Проблема отладки может возникать либо при появлении рассмотренных выше, либо других особых и аварийных ситуаций, а также при получении некорректных с точки зрения решаемой задачи результатов. В настоящее время проблема отладки довольно хорошо разработана и на этом вопросе нет *особого* смысла останавливаться, поскольку имеющий определенный компьютерный навык *пользователь* вполне знаком с данным вопросом. Из простых средств отладки можно отметить методы контрольных точек, трассировки (*прокрутки*) и др. Пакет *Maple* в качестве подобного средства предлагает метод *трассировки* вычислений, поддерживаемый процедурами `{trace, debug}`, которые представляют на самом деле одну и ту же процедуру, но с *альтернативными* именами. Поэтому, говоря о *debug*-процедуре, будем иметь в виду также и альтернативную ей *trace*-процедуру, и наоборот.

В тестирующем режиме, определенном процедурой `{trace | debug}(P1, ..., Pn)`, делается *трассировка* каждого вызова *Pk*-процедур/функций, указанных в списке фактических аргументов функции до тех пор, пока не будет вызова `{untrace | undebug}(P1, P2, ..., Pn)` соответственно, отменяющего режим тестирования всех либо части тестируемых *Pk*-процедур/функций. Для трассируемой процедуры/функции на печать выводятся: (1) точки ее вызова, (2) результаты всех выполняемых *промежуточных* вычислений вместе с предложениями, а также (3) точки выхода. В точках входа указываются фактические значения аргументов, а в выходных – возвращаемые ими результаты. Более детально вопросы использования данных средств для отладки процедур рассмотрены в [28-38].

Для отладки механизма вызовов процедур из других процедур достаточно полезным средством может оказаться и процедура *where*(| *Число*), по вызову которой выводится содержимое *стэка вызовов процедур* на глубину, определяемую значением (*целое число*) фактического аргумента функции. Вызов процедуры *where*() определяет трассировку вызовов процедур, начиная с верхнего уровня; выводятся выполняемые предложения процедур и значения передаваемых процедурам фактических аргументов. Тогда как при определении *глубины* стэка выводятся лишь элементы заданного *числа* его *нижних* уровней. С учетом сказанного, вызов *where*-процедуры должен указываться внутри процедуры, чья *трассировка* вызовов должна отслеживаться, как иллюстрирует весьма простой фрагмент трассировки вызовов *Kr*-процедуры на глубину 5 стэка вызовов:

```
> G := proc() S(args) end proc: S:=proc() V(args, 95) end proc: V:= proc() Kr(args, 99) end
proc: Kr := proc() local k; where(5): [nargs, sum(args[k], k=1..nargs)] end proc:
G(68, 63, 43, 14, 21, 48);  => [8, 451]
TopLevel: G(68, 63, 43, 14, 21, 48)
          [68, 63, 43, 14, 21, 48]
G: S(args)
          [68, 63, 43, 14, 21, 48]
S: V(args, 95)
          [68, 63, 43, 14, 21, 48, 95]
V: Kr(args, 99)
```

```
[68, 63, 43, 14, 21, 48, 95, 99]
Currently in Kr.
> Kr(Kr(68, 63, 43, 14, 21, 48)); ⇒ [1, [6, 257]]
TopLevel: Kr(Kr(68, 63, 43, 14, 21, 48))
[68, 63, 43, 14, 21, 48]
Currently in Kr.
TopLevel: Kr(Kr(68, 63, 43, 14, 21, 48))
[[6, 257]]
Currently in Kr.
```

Следует подчеркнуть, что пакетный отладчик *Debugger* располагает *where*-командой, аналогичной *where*-процедуре, за исключением того, что *первую* можно использовать интерактивно в отладочном режиме, инициируемом отладчиком. При этом, лучший эффект от использования *where*-процедуры можно получить при *трассировке* именно рекурсивных вызовов процедур и/или функций. Наряду с процедурами, механизм трассировки на основе *where*-процедуры можно использовать и для пользовательских функций, как это иллюстрирует пример определения *Fnc*-функции, реализованной посредством функционального оператора «->», а именно:

```
> Fnc := () -> [where(5), evalf(sqrt(sum(args['k']^2, 'k' = 1 .. nargs)), 6)]:
> V(Kr(Fnc((42, 47, 67, 89, 96, 62)))) ⇒ [2, 99 + [1, [171.590]]]
TopLevel: V(Kr(Fnc(42, 47, 67, 89, 96, 62)))
[42, 47, 67, 89, 96, 62]
Currently in Fnc.
TopLevel: V(Kr(Fnc(42, 47, 67, 89, 96, 62)))
[[171.590]]
Currently in Kr.
TopLevel: V(Kr(Fnc(42, 47, 67, 89, 96, 62)))
[[1, [171.590]]]
V: Kr(args, 99)
[[1, [171.590]], 99]
Currently in Kr.
```

Фрагмент использует *две* процедуры *V* и *Kr*, определенные в предыдущем фрагменте, и определяет новую *Fnc*-функцию, содержащую *вызов where*-процедуры, обеспечивая использование для нее описанного механизма тестирования вызовов через стек.

Встроенный *отладчик DEBUG* ориентирован на отладку довольно сложных процедур в интерактивном режиме и базируется на механизме контрольных точек (*check points*), допуская 2 варианта исполнения. По первому варианту режим отладки активируется по функции *DEBUG*(*{Комментарий}*), чья идентификация *Input*-параграфа делается «DBG>»-метками; отмена режима отладки делается по команде {quit | stop | done}; при этом, необходимо иметь в виду, что при вводе команд режима отладки кодирование после них разделителя не допускается. В режиме *DEBUG* можно использовать более 16 команд: *cont*, *next*, *step*, *into*, *stopwhen*, *return*, *stopat*, *list*, *where*, *stopwhenif*, *stoperror*,

showstop, *showstat*, *stop* и другие, обеспечивающих целый ряд важных отладочных функций, с которыми детально можно ознакомиться в [8-11,14] либо в определенной степени по справке пакета, а также по поставляемой с пакетом документации. Там же достаточно детально можно ознакомиться с ограничениями по применению данного отладчика. Между тем, новые релизы пакета часто расширяют функции отладчика.

При этом следует иметь в виду, в режиме **DEBUG** допускается выполнять вычисления с участием *переменных* процедуры, выводить промежуточные результаты вычислений и присваивать значения переменным процедуры, обеспечивая *мониторинг* вариантов вычислений. Перед *выходом* из отладочного режима нужно отменить установки точек всех типов, т.е. действие *stop{at | when | error}*-команд, и только после этого выполнить *{stop | done | quit}*-команду, отменяющую режим отладки с выводом соответствующего сообщения и *возвратом* в вычислительную среду пакета. В противном случае делается выход из отладочного режима, но ядро пакета остается в так называемом *оперативном DEBUG*-режиме, активируемом лишь в момент вызова соответствующих процедур и функций, определенных указанными командами, которые в таком случае выступают уже на уровне функций языка, требуя соответствующего синтаксиса. В оперативном **DEBUG**-режиме допускается использование всех упомянутых 16 команд/функций, а внутри режима отладки определенных ими процедур/функций использование всех команд отладочного режима. Установка контрольных точек *любого* из рассмотренных 3 типов может производиться и внутри самих процедур, иницилируя режим отладки в моменты их вызовов, однако на наш взгляд, это не самая лучшая технология отладки. В книгах [8-11] можно найти целый ряд весьма поучительных примеров применения **DEBUG**-средств языка пакета для отладки достаточно простых процедур. Из данных примеров не только четко прослеживаются *основные* принципы тестирования на базе контрольных точек, но из них также следует, что средство функции **DEBUG** уже для относительно несложных процедур является малообозримым и рекомендуется лишь в достаточно сложных для отладки обычными средствами пакета случаях.

В качестве довольно простого средства отладки *Maple*-процедур можно использовать и модифицированный метод контрольных точек. Такой метод состоит в следующем. Для генерации процедур, *обеспечивающих* возвращение значений нужных выражений в установленных контрольных точках, служит процедура *ChkPnt* [8-11], кодируемая в самом начале тела тестируемой процедуры *P* в виде вызова *ChkPnt(args)*. После чего в требуемых местах процедуры *P* кодируются вызовы следующего формата, а именно:

chkpntN(_x1, _x2, _x3, ..., _xn, x1, x2, x3, ..., xn)(); (1)

chkpntN(_x1, _x2, _x3, ..., _xn, x1, x2, x3, ..., xn)(x1, x2, x3, ..., xn); (2)

где *N* – номер контрольной точки и *x_j* (*j*=1 .. *n*) – имена выражений, значения которых нужно получить в данной точке процедуры *P*. Вызов тестируемой процедуры *P(args)* на кортеже основных фактических аргументов не включает механизма контрольных точек, тогда как вызов процедуры *P(args, chkpnt = N)* для *N* из диапазона [1 .. 72] будет обеспечивать вывод значений требуемых выражений в контрольной точке с номером *N* (*формат 1*) или вывод значений выражений с их *возвратом* (*формат 2*), как наглядно иллюстрирует следующий достаточно прозрачный фрагмент.

```

> ChkPnt := proc() local a, k;
    unassign(seq(cat('chkpnt', k), k = 1 .. 72));
    assign(a = {seq('if (type(args[k], 'equation'), args[k], NULL), k = 1 .. nargs)});
    seq('if (lhs(a[k]) = 'chkpnt' and type(rhs(a[k]), 'posint'), RETURN(assign(cat('chkpnt',
    rhs(a[k])) = () -> [WARNING("in %1 variables %2 have the following values %3 ",
    procname, [seq(args[k], k = 1 .. 1/2*nargs)], [seq(eval(args[k]), k = 1/2*nargs + 1 .. nargs)]),
    RETURN(RETURN)])), eval(cat('chkpnt', eval(rhs(a[k])))), NULL), k = 1 .. nops(a))
end proc;
> Proc := proc(x::numeric, y::numeric, z::numeric) local a, b, c; ChkPnt(args);
a := evalf(sqrt(x^2 + y^2 + z^2), 4); chkpnt1(_x, _y, _z, _a, x, y, z, a)();
b := evalf(sqrt(x^3 + y^3 + z^3), 4); chkpnt2(_a, _b, a, b)(a, b); c := evalf(sin(a) + cos(b), 4);
chkpnt3(_a, _b, _c, a, b, c)(); WARNING("Results: a = %1, b = %2, c = %3", a, b, c) end proc;
> Proc(68.42, 63.47, 43.67);
Warning, Results: a = 103.1, b = 812.0, c = 0.6433
> Proc(68.42, 63.47, 43.67, chkpnt = 2); => 103.1, 812.0
Warning, in chkpnt2 variables [_a, _b] have the following values [103.1, 812.0]
> Proc(68.42, 63.47, 43.67, chkpnt = 1);
Warning, in chkpnt1 variables [_x, _y, _z, _a] have the following values
[68.42, 63.47, 43.67, 103.1]
> Proc(68.42, 63.47, 43.67, chkpnt = 3);
Warning, in chkpnt3 variables [_a, _b, _c] have the following values [103.1, 812.0, 0.6433]

```

Данный механизм контрольных точек позволяет достаточно несложно устанавливать контрольные точки и получать в них значения требуемых выражений, и возвращать эти значения, что обеспечивает простой механизм запрограммированного выхода из любой точки процедуры с возвращением значений требуемых выражений. Данный механизм отладки достаточно эффективен с учетом структурированности основного количества *Maple*-процедур, которые не используют *goto*-механизм. Наконец, вызов процедуры *maplemint(P)* обеспечивает вывод протокола результатов семантического анализа *P*-процедуры, включая ее никогда не выполняемые предложения [8-11], что в целом ряде случаев позволяет проводить анализ процедур на их корректность.

В качестве весьма полезного упражнения читателю рекомендуется проверить разные варианты отладки *Maple*-процедур на основе упомянутых средств. Между тем, наш опыт работы с программными средствами различных уровня и назначения позволяет констатировать, при достаточно полном соответствии описания функционирования конструкций средств их реализации, хороших знаниях сущности погружаемой в их среду задачи и возможностей этого средства существует немало более эффективных средств отладки, чем *методы* трассировки и контрольных точек в их чистом виде. Так, наша практика создания различного рода *ПС* не использовала данной методологии в ее классической форме, как не отвечающей основной задаче эффективной отладки. В целом же, даже довольно сложные процедуры возможно отлаживать и без указанных выше средств, имея ввиду *интерактивный* характер *Maple*-языка, позволяющий *вполне*

успешно проводить пошаговую интуитивную отладку **Maple**-процедур параллельно с их написанием, используя методику и приемы т.н. *эвристического программирования* (точнее, его элементов). Именно подобным способом создавалось большинство наших программных средств, включая и многие средства из нашей Библиотеки [45].

В заключение раздела кратко остановимся еще на вопросе оптимизации процедур. В качестве определенного подспорья здесь может оказаться процедура **maplemint**, чей вызов **maplemint(Pro)** генерирует отчет с семантической информацией по заданной процедуре **Pro** (активной в текущем сеансе или находящейся в **Maple**-библиотеке, которая логически связана с главной библиотекой **Maple**) и выводит коды, которые невыполнимы при вызове процедуры. Вызов процедуры **maplemint(Proc)** генерирует информацию по таким важным аспектам **Proc** как константы, присваиваемые в качестве значений; декларированные глобальные переменные, начинающиеся с символа '_' (подчеркивания); неиспользуемые декларированные глобальные переменные; используемые, однако не декларированные глобальные переменные; неиспользуемые декларированные локальные переменные; переменные цикла, повторно используемые во вложенных циклах; код, который недостижим; ключевые слова **break** и/или **next**, обнаруженные вне циклов; отсутствие знака умножения '*' и др. Несколько детальнее с данным средством можно ознакомиться по конструкции **?maplemint**, тогда как с исходным текстом процедуры (листингом) можно ознакомиться по следующему предложению, а именно:

> interface(verboseproc = 3); eval(`maplemint/recurse`);

Между тем, текущая реализация процедуры **maplemint** не поддерживает получения семантической информации по процедурам, содержащим модульные объекты и/или средства обработки особых и ошибочных ситуаций. Однако и перечисленное может оказаться достаточно полезным при решении ряда вопросов оптимизации процедур. Следующий фрагмент иллюстрирует примеры применения процедуры **maplemint**.

```
> maplemint(sin);
Error, (in maplemint/expression) the module system is not yet supported
> maplemint(MkDir);
Error, (in maplemint/statement) exception handling is not yet supported
> maplemint(save1);
This code is unreachable:
while not Fend(p) do h:=readline(p); writeline(a,`if` (h[-1] <> ";", h, cat(h[1..-2], ":")) end do
null(close(f, a), writebytes(f, readbytes(a, infinity)), close(f), fremove(a))
These local variables were never used: h, k
These local variables were used before they were assigned a value: a, p
These local variables were used before they were assigned a value: k
These parameters were never used explicitly: F
These names were used as global names, but were not declared: c
These local variables were never used: x
These local variables were used before they were assigned a value: s, b, a, zeta, nu
These parameters were never used explicitly: E
> maplemint(diff);
```


These names were used as global names, but were not declared: SubstituteAll
 These local variables were never used: b
 These names were used as global names, but were not declared: k

Из приведенного фрагмента можно сделать ряд весьма полезных выводов, а именно: (1) достаточно сложные процедуры, как правило, используют различные механизмы обработки ошибочных и особых ситуаций, посему они не являются доступными для анализа процедурой *maplemint*, (2) часто не всегда результаты семантического анализа соответствуют конкретному алгоритму тестируемой процедуры, например, в случае формальных аргументов подпроцедур, глобальных переменных, индексов цикла или суммирования и т.д., (3) в ряде случаев выводимая информация не вполне корректна с точки зрения *Maple*-языка, как это иллюстрирует следующий простой фрагмент:

```
> P:=proc() assign('libname' = op([libname, "D:/RANS/LAN"])); `+`(args)/nargs end proc;  
> maplemint(P);
```

These names were used as global names, but were not declared:

"C:\\Program Files\\Maple 11/lib",

"C:/Program Files/Maple 11/LIB/UserLib"

В данном простом фрагменте в качестве имен *глобальных* переменных указывается не предопределенная *libname*-переменная, а ее значение, между тем, это совсем не одно и то же. Имеются и другие некорректности. Однако, для случая достаточно простых процедур вышеуказанное средство пакета *Maple* может оказаться довольно *полезным*, прежде всего, для не столь искушенного пользователя. Наконец, *синтаксический Mint*-контролер анализирует программу и генерирует сообщение о возможных ошибках в *mpl*-файле данных с исходным *Maple*-текстом. Если же данный файл не задан, то для чтения исходного *Maple*-текста используется стандартный ввод. Анализ завершается по достижении конца текста. Детальнее с таким средством синтаксической проверки *Maple*-программ можно ознакомиться в текущем сеансе по конструкции *?mint*.

Между тем, наш опыт и опыт наших коллег показывают, что при вполне достаточной квалификации применение для отладки и оптимизации средств, созданных в *Maple*, вышеперечисленных стандартных средств совершенно не обязательно, тем более, что они имеют ряд довольно существенных *ограничений*. Достаточный программистский опыт, возможность программирования интерактивно-эвристического типа, хорошее знание самой сути программируемых задач зачастую *вполне* достаточны для создания достаточно сложных и эффективных программных средств в среде пакета *Maple*.

Действительно, средства и организация *языка* пакета предоставляют возможность для пользователя в интерактивном режиме запрограммировать алгоритм своей задачи по этапам, также поэтапно производить отладку полученных документов или процедур. Он же позволяет и относительно несложно производить их постепенную доводку до нужных требований; при этом, сам процесс такого программирования обеспечивает уточнение отдельных составляющих программируемого алгоритма. При том, *отладка* программного комплекса в целом сводится к отладке составляющих, последовательно связанных между собой, которая делается одновременно с их программированием. А так как составляющие с учетом достаточно крупных (*в плане поддерживаемых операций*)

функциональных средств *Maple*-языка, как правило, хорошо обозримы, то и процесс их отладки существенно упрощается, а с ним и *весь* процесс отладки в целом. Именно такая методика и использовалась нами при создании средств нашей Библиотеки [45]. Опыт ее применения со всей очевидностью доказал ее жизнеспособность и во многих случаях большую эффективность относительно имеющихся для этих целей средств стандартной отладки. Правда, описанный подход *априори* предполагает как хорошее знание сути программируемого приложения, так и саму программную среду пакета.

Рассмотрев процедурные объекты *Maple*-языка, обеспечивающие довольно высокий уровень модульности программирования в среде пакета, и базируясь на собственном опыте работы с обоими пакетами, процедурное программирование представляется в среде *Maple* нам более развитым и отвечающим *современным* технологиям разработки программного обеспечения. Представим теперь новый тип (начиная с релиза 6 пакета) объектов *Maple*-языка, не только повышающих уровень модульности программных средств, разрабатываемых в среде *Maple*, но и в определенной мере обеспечивающих объектно-ориентированную технологию программирования в его среде. Такого типа объектами являются *программные модули*, в отличие от пакетных модулей, о которых речь будет идти в настоящей книге несколько ниже.

6.1.10. Организация программных модулей в среде Maple-языка

В настоящем разделе рассматриваются *модульные* объекты *Maple*-языка, предполагая, что читатель в определенной степени имеет представление о работе в среде пакета в пределах, например, книг [42-44, 46-56] или подобных им изданий. Все используемые понятия и определения полностью соответствуют книгам [42-44]. При этом, акцент в данном разделе делается на модульных объектах языка, определяющих возможность использования элементов объектно-ориентированного программирования в *Maple*.

Начиная с *Maple 6*, пакет включает средства по обеспечению ряда базовых элементов объектно-ориентированного программирования, которые в целом поддерживаются механизмом *программных модулей* (или в дальнейшем просто *программных модулей – ПМ*). Эти модули не следует ассоциировать с более широким понятием пакетных модулей, которые употреблялись на протяжении книг [8-11], рассматривающих *пятый* релиз пакета. Вместе с тем, можно считать, что программные модули при оформлении их в библиотечные структуры составляют *подмножество* всех пакетных модулей. Итак, если *пакетные* модули вполне могут рассматриваться как *внешние* по отношению к его ядру хранилища определений функциональных средств, тогда как механизм *программных модулей* прежде всего ориентирован на обеспечение определенного *уровня* объектно-ориентированного программирования в среде *Maple*-языка пакета в текущем сеансе.

Так, если процедуры позволяют определять последовательность предложений языка, описывающих некоторый законченный алгоритм (*той либо иной степени сложности*), в виде единого объекта, к которому впоследствии можно обращаться с передачей ему фактических аргументов, не интересуясь собственно самой реализацией алгоритма, то механизм *программных модулей* обеспечивает существенно более высокий уровень *абстрагирования*, позволяя «*скрывать*» от пользователя уже целые наборы тематически

связанных процедур и данных. Итак, относительно *Maple*-языка *ПМ* являются новым типом выражений подобно числам, спискам, уравнениям, отношениям, процедурам. Тестирование модульных объектов производится функциями *typematch*, *type*, а также процедурой *whattype*, как иллюстрирует следующий достаточно простой фрагмент:

```
> Gr := module() end module;
> type(Gr, `module`), typematch(Gr, `module`), whattype(eval(Gr)); => true, true, module
> type(module() end module, 'module');
Error, unexpected single forward quote
> type(module() end module, `module`); => true
> type(module() end module, 'moduledefinition'); => false
> type('module() end module', 'moduledefinition'); => true
```

При этом, четыре последних примера иллюстрируют возможность тестирования не только готового модуля, а также его определения. С этой целью определение модуля должно указываться *невычисленным*, в противном случае возвращается *false*-значение. Такая организация *ПМ* позволяет успешно использовать их при программировании *параметрических* алгоритмов, создании пакетных модулей наряду с предоставлением возможности использования в *Maple*-программах *Pascal*-подобных записей или *C++*-подобных структур [1-3]. В среде *Maple*-языка *ПМ* могут использоваться несколькими способами, из которых следует отметить 4 наиболее широко используемых, а именно: (1) инкапсуляция, (2) создание модулей пакета, (3) моделирование объектов, а также (4) параметрическое программирование. Детально эти способы с рекомендациями по их практическому использованию рассмотрены в наших книгах [10-14].

Инкапсуляция в значительной степени гарантирует то, что уровень абстрагирования процедур и данных *ПМ* строго соответствует определенному для них *интерфейсу*, что позволяет пользователю программировать *весьма* сложные мобильные и многократно используемые (*reusable*) программные системы с четко определенными интерфейсами пользователя. При этом, обеспечиваются лучшие сопровождение и четкое понимание *исходных* программных текстов, что является очень важной характеристикой сложных программных систем. Механизм инкапсуляции обеспечивает возможность определения процедур и данных, *видимых* извне модуля (*т.е. экспортируемых модулем*), а также тех, которые составляют внутреннюю сущность самого модуля и недоступны вне модуля, *т.е. являются невидимыми* вне самого модуля.

Пакетные модули (*в отличие от ПМ*) обеспечивают механизм совместного хранения тематически связанных *Maple*-процедур и других объектов. Такие наборы процедур обеспечивают достаточно развитые функциональные средства, ориентированные на конкретные области приложений, например: *linalg* и *LinearAlgebra* (*линейная алгебра*), *stats* (*статистический анализ данных*), *plots* и *plottools* (*графические средства и средства анимации*) и др. Значительное количество функциональных средств пакета находится именно в его *модулях* как *внутренних*, так и *внешних*. Основной организацией модулей пакета ранних релизов являлась табличная организация, в которой входами являлись *имена* процедур, а выходами их *определения*. Тогда как *ПМ* обеспечивают совершенно иной механизм организации модулей пакета, который детально рассмотрен в [28-38].

Так, пакетный модуль *LinearAlgebra*, обеспечивающий функции линейной алгебры, имплантирован в среду пакета релизов 6 и выше именно как программный модуль.

Посредством *ПМ* достаточно легко программируются объекты. В программной среде под *объектом* понимается некоторая программная единица, определяемая как своим состоянием, так и поведением. Вычисления над подобными объектами производятся передачей им некоторых *управляющих* сообщений, на которые они должны отвечать соответствующими *действиями* (*вычисление, управление, изменение состояния и др.*). При этом, параметрические программы пишутся без знания того, как организованы сами объекты, обработку которых они производят. Более того, *параметрическая* программа будет работать с любым объектом, имеющим с ней общий интерфейсный протокол, безотносительно того, как объект удовлетворяет данному протоколу. Перечисленные выше *четыре* аспекта, поддерживаемые механизмом *программного модуля*, определяют непосредственное практическое применение *Maple*-технологии, которая достаточно широко иллюстрируется в книгах [10,28-38] на ряде практически полезных примеров. Тогда как основы механизма *ПМ* и их использования в *Maple*-среде иллюстрируются соответствующими довольно наглядными примерами, которые можно найти там же.

Организация *программного модуля (ПМ)* весьма напоминает организацию процедуры *Maple*, которая достаточно детально рассмотрена выше. В общем случае организация (*структура*) программного модуля имеет следующий весьма прозрачный вид:

```

module()
    export {экспортируемые переменные}
    local {локальные переменные}
    global {глобальные переменные}
    options {опции модуля}
    description {описание модуля}
    Т Е Л О  М О Д У Л Я
end module { : | ; }
    
```

Определение каждого *ПМ* начинается с ключевого слова **module**, за которым следуют круглые скобки "()", подобно тому, как это имеет место для определения процедуры без заданных *формальных* аргументов. Завершается определение модуля скобкой **end module**. Все остальные компоненты структуры модуля, находящиеся между **module()** и **end module**, являются необязательными. При этом, взаимное расположение пяти указанных *деклараций* в определении модуля несущественно, однако все они должны предварять (*при их наличии*) собственно тело модуля. В результате *компиляции* модуля устанавливается принятый языком *выходной* порядок его существующих деклараций, определяемый релизом пакета и вычислительной платформой.

Программный модуль является типом *Maple*-выражения, которое создается в процессе вычисления *определения модуля*. Определение создается программой синтаксического анализа языка на основе представленной выше структуры модуля. Подобно *процедуре* определение *ПМ* может включать ряд необязательных деклараций типа *global*, *local*, *option* и *description*, имеющих тот же смысл, что и в случае процедуры (*между тем, их специальные опции могут отличаться*). При этом, *ПМ* можно представить себе в форме

набора тематически связанных переменных. При этом, некоторые из этих переменных доступны для текущего сеанса вне определения модуля после его вычисления. Такие переменные задаются в *определении* модуля как *глобальные* (*global*) или *экспортируемые* (*export*). Другие переменные определяются явно либо неявно *локальными* (*local*), они доступны только в рамках определения модуля в период его реализации. Локальные переменные используются только алгоритмом, реализуемым предложениями *Maple*-языка, составляющими тело самого модуля. Переменные такого типа рекомендуется определять явно посредством *local*-декларации, иначе это сделает сам *Maple*-язык на основе принятых *неявных* правил синтаксического и семантического анализа. Другие переменные, находящиеся в определении модуля, относятся к *глобальным*, параметрам либо к *локальным* в зависимости от возможных приложений. Глобальные переменные могут быть определены *глобальными* явно через *global*-декларации модуля либо через *неявные* правила синтаксически-семантического анализа в момент упрощения модуля. При этом следует иметь в виду, что множества *глобальных*, *локальных* и *экспортируемых* переменных модуля не должны попарно пересекаться.

Каждый *ПМ* имеет тело, которое может быть как *пустым*, так и содержать выражения и/или предложения. Конструкции тела обеспечивают как необходимые вычисления в процессе *реализации* данного модуля, так и определяют начальные значения для его *экспортируемых* переменных. Множество переменных тела модуля и их соотношений составляет *полный* лексический набор модуля. Это полностью *аналогично* определению тела процедуры; в обоих случаях используются одни и те же лексические и неявные правила синтаксического и семантического анализа. При этом, в определении модуля и процедуры допускается широкий произвол в их взаимной вложенности (*вложенности типов процедура-процедура, процедура-модуль, модуль-процедура, модуль-модуль*). Модуль, содержащийся в другом модуле, называется *подмодулем* относительно *второго* модуля. Переменным модуля (*локальным, глобальным или экспортируемым*) могут присваиваться любые допустимые *выражения*, включая процедуры и модули. При этом, простейший модульный объект имеет следующий вид: **module() ... end module**. Данный модуль не имеет особого смысла, т.к. не экспортирует переменных, не имеет ни глобальных, ни локальных переменных, ни даже своего тела, производящего какие-либо вычисления. В общем случае *ПМ* можно рассматривать как результат вызова процедуры, которая возвращает некоторые из своих локальных переменных, определенных в модуле как его *экспортируемые* переменные (*или экспорты*).

Модули подобно *процедурам* могут быть как *поименованными*, так и *непоименованными*. *Непоименованные* модули используются, как правило, непосредственно в выражениях, тогда как *поименованный* модуль поддерживает существенно более *развитый* механизм работы с ним. Имя модулю можно присваивать двумя способами, а именно:

- (1) присвоением определения модуля некоторой переменной;
- (2) указывая имя между ключевым словом **module** и **()**-скобками.

Однако, между обоими способами *именования* существуют достаточно существенные различия. Поименованный *первым* способом модуль может впоследствии многократно *переименовываться*, создавая свои копии. Тогда как поименованный *вторым* способом модуль не может быть переименован, а его имя имеет *protected*-атрибут. Более того,

если при выполнении *поименованного* первым способом модуля возникает ошибочная ситуация, то имя модуля идентифицируется как неизвестное (*unknown*), тогда как для второго случая *ошибка* привязывается к имени модуля. Это вполне разумное решение, т.к. модули могут быть *непоименованными* либо иметь разноименные копии, тогда как второй способ именованного модуля определяет модуль с *фиксированным* именем. Следующий достаточно простой фрагмент наглядно иллюстрирует вышесказанное, а именно:

```
> module () error Mod_1 end module;
Error, (in unknown) Mod_1
> Chap := module () error Mod_2 end module;
Error, (in unknown) Mod_2
> module Art () error Mod_3 end module;
Error, (in Art) Mod_3
> Kr := module B () end module;
> type(Art, `module`), type(B, `module`); => true, true
> A := module B () end module;
Error, (in B) attempting to assign to `B` which is protected
```

С учетом сказанного данный *фрагмент* достаточно прозрачен и особых пояснений не требует. Между тем, предпоследний пример фрагмента иллюстрирует возможность одновременного *именования* модуля как *первым*, так и *вторым* способом. Тогда как уже повторная аналогичная попытка вызывает ошибочную ситуацию. Обусловлено это, что имя *поименованного* вторым способом модуля имеет *protected*-атрибут. Именно здесь уместно рассмотреть компоненты определения модуля несколько детальнее.

Ряд неудобств работы с программными модулями *второго* типа, которые в некоторых случаях требуют более сложных алгоритмов обработки при создании программного обеспечения с их применением, ставит на повестку дня вопрос создания процедуры, конвертирующей их в модули *первого* типа. В дальнейшем, под *программным* модулем *первого* типа мы будем понимать модуль, именованный конструкцией вида «**Name := module () ...**», тогда как *программный* модуль *второго* типа характеризуется именуемой конструкцией следующего вида «**module Name () ...**». Но здесь имеется также третий способ именованного программного модуля, который за одно определение позволяет задавать функционально эквивалентные разноименные модули второго типа, как это весьма наглядно иллюстрирует следующий простой фрагмент, а именно:

```
> M := module M1 () export x; global y; x := () -> `(args)/nargs; y := 2010 end module;
5*M:- x(65, 63, 43, 21, 14), y; => 206, 2010
> map(type, [M, M1], `module`), map(type, [M, M1], 'mod1'); => [true, true], [false, false]
> map(mod21, [M, M1]), map(type, [M, M1], 'mod1'), M:- x(63, 43), 3*M1:- x(68, 21, 14);
[], [true, true], 53, 103
```

Некоторые полезные соображения по использованию программных модулей второго типа могут быть найдены в наших книгах [28-38,42]. Достаточно полезная процедура *mod21* решает данную проблему конвертирования программных модулей, а именно: вызов процедуры *mod21(M)* возвращает *NULL*-значение, т.е. ничего, с обеспечением в текущем сеансе конвертацию программного модуля *второго* типа, заданного именем

M, в эквивалентный ему программный модуль *первого* типа с тем же самым именем *M*. Если же вызов процедуры *mod21(M,f)* использует *второй* необязательный аргумент *f*, то процедура рассматривает его как каталог, в котором должен быть сохранен файл «*M.mod1*» с определением отконвертированного программного модуля *M* (если такой каталог *f* отсутствует, то он будет создан с произвольным уровнем вложенности). В таком случае вызов процедуры *mod21(M,f)* возвращает полный путь к файлу с сохраненным отконвертированным модулем *M* и с выводом соответствующего сообщения, которое информирует о полном пути к созданному файлу «*M.mod1*». Файл с сохраненным *M*-модулем имеет входной *Maple*-формат. Примеры следующего фрагмента достаточно прозрачно иллюстрируют вышесказанное, а именно:

```
> mod21 := proc(M::`module`) local a, b, c, d, t, p, k, v, sf, h, nu;
  assign(a = convert(eval(M), 'string'), t = {}, p = []);
  sf = ((x, y) -> `if`(length(y) <= length(x), true, false));
  assign(b = Search2(a, {"module "}), assign('b' = [seq(k + 5, k = b)]);
  assign(c = {seq(nexts(a, b[k], "() ", k = 1 .. nops(b)))};
  seq(assign('p' = [op(p), a[c[k][1] + 1 .. c[k][2] - 1]]), k = 1 .. nops(c)); assign('p' = sort(p, sf));
  p := [seq(op([assign('r' = Search2(p[k], {":-"})), `if`(r <> [] and p[k] <> ":-", p[k][2 .. r[-1] + 1],
    `if`(p[k] = " ", NULL, cat(p[k][1 .. -2], ":-")))), k = 1 .. nops(p))];
  p := [seq(k = "", k = p)]; seq(`if`(2 < c[k][2] - c[k][1], assign('t' = {op(t),
    v $ (v = c[k][1] + 1 .. c[k][2] - 1)), NULL), k = 1 .. nops(c));
  eval(parse(cat("unprotect(", M, ")", assign("", M, "=", SUB_S(p, dsps(a, t)), ")")));
  if nargs = 2 then
    if type(args[2], 'dir') then h := cat(args[2], "\\ ", M, ".mod1"); save M, h; h
    else assign(nu = interface(warnlevel)), null(interface(warnlevel = 0)),
      assign('h' = cat(MkDir(args[2]), "\\ ", M, ".mod1"));
    (proc() null(interface(warnlevel = nu)); save M, h end proc)(), h;
    WARNING("module <%1> has been converted into the first type,
      and saved in datafile <%2>", M, h)
  end if
end if
end proc:

> module A () local C; export B; B := C[y](1, 2, 3) + C[z]: module C () local H; export y, z;
y := () -> sum(args[k], k=1..nargs); z := H[h](6, 7, 8); module H () export h;
h := () -> sqrt(`+`(args)) end module end module; end module: A1 := module () local C;
export R; R:=C[y](1,2,3) + C[z]: C:=module () local H; export y, z; y:=() -> sum(args[k],
k=1 .. nargs); z:=H[h](6, 7, 8); H:=module () export h; h:= () -> `*`(args) end module end
module end module: A2:=module () local C; export R; R:=C[y](1,2,3)+C[z]: module C ()
local H; export y, z; y:=() -> sum(args[k], k=1..nargs); z:=H[h](6, 7, 8); H:= module ()
export h; h:= () -> `*`(args) end module end module; end module: A3:= module A3 ()
export h; h := () -> sum(args[k], k=1 .. nargs)/nargs end module:
Error, attempting to assign to `A3` which is protected
> M := module M1 () export G; G := () -> sum(args[k], k=1 .. nargs)/nargs end module:
```

```

N := module N1 () export Z; local M; Z := () -> M:- h(args); M:= module () export h;
h := () -> sum(args[k], k=1..nargs)/nargs end module end module: mod21(A, "C:/aaa"),
mod21(A1,"C:/temp/aaa/ccs"), mod21(A2,"C:/temp/aaa/ccs"), mod21(A3,"C:/temp/aaa"),
mod21(M,"C:/temp/aaa/ccs"), mod21(M1,"C:/temp/aaa/ccs"),
mod21(N1,"C:/temp/aaa/ccs"), mod21(N, "C:/temp/aaa/ccs");
Warning, module <A> has been converted into the first type,
and saved in datafile <c:\\aaa\\A.mod1>
Warning, module <A1> has been converted into the first type,
and saved in datafile <c:\\temp\\aaa\\ccs\\A1.mod1>
Warning, module <M> has been converted into the first type,
and saved in datafile <c:\\temp\\aaa\\ccs\\M.mod1>
"C:/temp/aaa/ccs\\A2.mod1", "C:/temp/aaa\\A3.mod1",
"C:/temp/aaa/ccs\\M1.mod1", "C:/temp/aaa/ccs\\N1.mod1",
"C:/temp/aaa/ccs\\N.mod1"
> map(type, [A,A1,A2,A3,M,M1,N,N1], 'mod1'); => [true, true, true, true, true, true, true, true]

```

Следует отметить, именно на данной процедуре мы обнаружили недостатки работы пакетного *стека*, что привело к необходимости соответствующим образом изменять процедуру, чтобы обеспечить ее совместимость в рамках пакета *Maple* релизов 6 – 11. Вопросы работы с модулями обоих типов, а также соответствующие для этого случая средства рассмотрены в наших книгах [10-14,28-38,42-44].

Декларация *description*. Определение *программного* модуля допускает использование декларации *description*, включающей текстовую строку, представляющую своего рода краткое *документирование* данного модуля. Например, такая строка может содержать краткое описание модуля. Эта компонента определения модуля вполне аналогична *одноименной* компоненте описания процедуры, которая в результате автоматического упрощения модуля становится самой *последней* среди всех его деклараций. При этом, декларация определяет одну либо несколько строк, составляющих единое описание модуля. Следующий весьма простой фрагмент иллюстрирует сказанное, а именно:

```

> Grsu := module()
    export Sr;
    local k;
    description "Sr - средняя аргументов";
    Sr := () -> `+`(args)/nargs
end module;

```

Grsu := module () local k; export Sr; description "Sr - средняя аргументов "; end module

Из приведенного фрагмента следует, что тело *Grsu*-модуля скрыто от пользователя, в свою очередь описание характеризует его экспортируемую *Sr*-переменную.

Декларация *options*. Определение *программного* модуля может содержать также *опции* аналогично случаю процедуры, определяющие режим работы с модулем. Между тем, в отличие от процедур опции *remember*, *system*, *arrow*, *operator* и *inline* не имеют для *модуля* смысла. *Опции* кодируются в форме переменной или уравнения с переменной

в качестве его левой части. Если в *options*-декларации определены нераспознаваемые языком опции, то они игнорируются, что дает возможность пользователю определять опции для собственных нужд, которые распознаются языком как атрибуты.

Из *опций* модуль допускает лишь *trace*, *package* и *`Copyright ...`*, аналогичные случаю *Maple*-процедур, и специальные опции *record*, *load = name* и *unload = name*, где *name* – имя локальной или экспортируемой переменной модуля, определяющей процедуру. Такая процедура вызывается при первоначальном создании модуля или при чтении его из системы хранения функциональных средств пакета. Как правило, такая опция используется для какой-либо инициализационной цели модуля. Простой фрагмент иллюстрирует применение *load*-опции для инициализации *Sr*-процедуры модуля:

```
> module Q () local Sr; export Ds; option load=Sr; Ds:=() -> sqrt(sum((args[k]-Sr(args))^2,
  k = 1 .. nargs)/nargs); Sr := () -> sum(args[k], k = 1 .. nargs)/nargs end module:
> 36*Q:- Ds(43, 48, 14, 21, 68, 63)^2;  => 14249
```

Итак, опция *load* позволяет создавать наборы тематически связанных *экспортируемых* процедур, для чьей инициализации используются требуемые локальные процедуры и/или функции. Такая опция выполняет своего рода инициализационные функции модуля. Тогда как опция *unload = name* определяет имя *локальной* или *экспортируемой* процедуры модуля, которую требуется вызвать, когда модуль более недоступен либо производится выход из пакета. Модули с опцией *`package`* понимаются системой как *пакетные* модули и их *экспорты* автоматически получают *protected*-атрибут. Наконец, опция *record* используется для идентификации записей. Детально с опциями модуля можно ознакомиться в текущем сеансе в справке пакета по *?module,option*.

Локальные и глобальные переменные ПМ. Подобно процедурам, программные модули поддерживают механизм *локальных (local)* и *глобальных (global)* переменных, которые используются в определении модуля. Глобальные переменные *областью* определения имеют текущий сеанс, а *область* определения *локальных* переменных ограничена *лишь* самим модулем. Однако, в отличие от процедур, модуль поддерживает более гибкий механизм локальных переменных; при этом, недопустимо определение одной и той же переменной как в *local*-декларации, так и в *export*-декларации. Попытка данного рода приводит к ошибочной ситуации. Результатом вычисления определения *модуля* является *модуль*, к *экспортируемым* членам (*переменным*) которого можно программно обращаться *вне области* модуля. Следующий весьма простой фрагмент иллюстрирует результат определения локальных и глобальных переменных программного модуля:

```
> module() local a,b; global c; assign(a=68, b=63); c:=proc() `+(args) end proc end module:
> a, b, c(68, 63, 43, 48, 14, 21);  => a, b, 257
> N := module () export d; d := proc() `+(args) end proc end module:
> d(68, 63, 43, 48, 14, 21), N: -d(68, 63, 43, 48, 14, 21);  => d(68, 63, 43, 48, 14, 21), 257
> module () local a,b; export a,b,c; assign(a=68,b=63); c:=proc() nargs end proc end module:
Error, export and local `a` have the same name
```

Из этого фрагмента следует, между глобальными и экспортируемыми переменными модуля имеется весьма существенное различие. Если *глобальная* переменная доступна

вне области модуля, то для доступа к экспортируемой переменной механизм является более сложным и в общем случае требует ссылки на модуль, экспортирующий такую переменную, следующего формата, а именно:

<Имя модуля>:- <Экспортируемая переменная>{(Фактические аргументы)}
<Имя модуля>[<Экспортируемая переменная>]{{(Фактические аргументы)}}

Это так называемый *связывающий* формат обращения к экспортируемым переменным модуля. По вызову встроенной функции *exports(M)* возвращается *последовательность* всех экспортов модуля *M*, в то время как по вызову процедуры *with(M)* возвращается *список* всех экспортов модуля *M*; при этом, во *втором* случае экспорты модуля *M* будут доступны в текущем сеансе и к ним можно обращаться без ссылок на содержащий их модуль, как это иллюстрирует следующий довольно простой фрагмент, а именно:

```
> M := module() local k; export Dis, Sr; Dis := () -> sqrt(sum((args[k] - Sr(args))^2,
  k=1..nargs)/nargs); Sr := () -> sum(args[k], k=1..nargs)/nargs; end module: exports(M),
  [eval(Dis), eval(Sr)]; => Dis, Sr, [Dis, Sr]
> 6*M[Sr](68, 63, 43, 48, 14, 21), 36*M[Dis](68, 63, 43, 48, 14, 21)^2; => 257, 14249
> 36*M:- Dis(68, 63, 43, 48, 14, 21)^2, with(M); => 14249, [Dis, Sr] # Maple 8
> 36*M:- Dis(68, 63, 43, 48, 14, 21)^2, with(M); => 14249, [Dis, Sr] # Maple 10
Warning, the `with` command does not work inside procedures or modules
> 36*M:- Dis(68, 63, 43, 48, 14, 21)^2, with(M); => 14249, [Dis, Sr] # Maple 11
Warning, M is not a correctly formed package - option `package` is missing
> 6*Sr(68, 63, 43, 48, 14, 21), 36*Dis(68, 63, 43, 48, 14, 21)^2; => 257, 14249
> G := module() global X; export Y; Y := () -> `(args)/nargs; X := 68 end module:
> X, Y(1, 2, 3), G:- Y(1, 2, 3); => 68, Y(1, 2, 3), 2
> V := module() global Y; Y := () -> `(args)/nargs end module;
      V := module() global Y; end module
> V1 := proc() `(args)/nargs end proc: Y(1, 2, 3, 4, 5, 6, 7), V1(1, 2, 3, 4, 5, 6, 7); => 4, 4
```

Фрагмент представляет модуль *M* с двумя экспортами *Dis* и *Sr*. Вызов *exports(M)* дает последовательность всех *экспортов* модуля *M*, однако это лишь имена. Затем по связке «:-» получаем доступ к экспорту *Dis* с передачей ему фактических аргументов. Тогда как вызов *with(M)* возвращает список всех экспортов модуля *M*, делая их доступными в текущем сеансе. Между тем, здесь необходимо сделать одно довольно существенное замечание. Корректно возвращая результат, вызов *with(M)*, между тем, в зависимости от *релиза* выводит или нет диагностические сообщения, как иллюстрирует фрагмент. Это говорит о том, что внутренняя организация *модулей* изначально серьезно не была проработана, вызывая постоянные модификации. В частности, для *Maple 11* наличие в модуле *M* уже орции *package* устранил вывод указанного сообщения.

Таким образом, *глобальность* той или другой переменной модуля можно определять либо через *global*-декларацию, либо через *export*-декларацию модуля. Однако, если в первом случае мы к данной переменной можем обращаться непосредственно после *вычисления* определения модуля, то во втором случае мы должны использовать связку «:-» или вызов формата *M[<Экспорт>]{{(Аргументы)}}*. Наконец, модуль *V* и процедура

V1 иллюстрируют функциональную эквивалентность обоих объектов – модульного и процедурного с тем лишь отличием, что модуль *V* позволяет «*скрывать*» свое тело в отличие от процедуры *V1*. Таким образом, возможно создавать модули без экспортов, возвраты которых обеспечиваются через их *глобальные* переменные, что обеспечивает *скрытие внутреннего* механизма вычислений. Однако, при сохранении такого модуля в *Maple*-библиотеке последующий доступ к его глобальным переменным средствами пакета становится невозможным, т. е. указанный прием работает только при условии вычисления определения модуля в текущем сеансе. Следовательно, описанный выше подход довольно искусственен и ограничен, и носит в своей основе иллюстративный характер. Как уже отмечалось, минимальным объектом, распознаваемым пакетом как *модуль*, является определение одного из следующих видов, а именно:

M := module() end module или **module M1 () end module**

о чем свидетельствуют и результаты их тестирования, а именно:

```
> restart; M := module() end module: module M1 () end module:
> map(type, [M, M1], `module`), map(whattype, map(eval, [M, M1])),
    [true, true], [module, module]
```

Между тем, попытка получить их *экспорты* посредством процедуры *with* инициирует ошибочную ситуацию со следующей релизо-зависимой диагностикой, а именно:

```
> M:=module () end module: module M1 () end module: with(M); with(M1); # Maple 6 – 8
Error, (in pacman:- with) module `M` has no exports
Error, (in pacman:- with) module `M1` has no exports
> with(M); with(M1); # Maple 9 – 10
Error, (in with) module `M` has no exports
Error, (in with) module `M1` has no exports
> with(M); with(M1); # Maple 11
Error, (in with) module `M` has no exports
Warning, M1 is not a correctly formed package – option `package` is missing
Error, (in with) module `M1` has no exports
> lasterror; => "module `%1` has no exports" # Maple 6 – 11
> map(exports, [M, M1]); => []
```

Тогда как во *всех* релизах переменная *lasterror* пакета получает идентичное значение. В этой связи для обеспечения более простой работы с *программными* модулями можно рекомендовать для получения их *экспортов* применять встроенную функцию *exports*, как иллюстрирует последний пример фрагмента, или использовать нашу процедуру *With*, являющуюся расширением стандартной процедуры *with*, и которая не только в такой ситуации возвращает корректный результат, но и позволяет использовать себя внутри процедур в отличие от пакетной процедуры *with*, как хорошо иллюстрирует следующий достаточно наглядный фрагмент. Это обстоятельство следует учитывать при работе с *модулями* в программном режиме.

```
> With := proc(P::{package, `module`}, F::symbol) local a, b, c, h;
    if nargs = 0 then error "'With' uses at least the 1st argument, which is missing"
```

```

else assign(a = interface(warnlevel), b = cat([libname][1], "/_$_Art22_Kr15$_"),
        interface(warnlevel = 0)
end if;
if nargs = 1 then
    try c := with(args)
    catch "module `%1` has no exports": return []
    end try;
    h := cat(cat(seq(cat("unprotect(`", k, "`):`", k, "`:=eval(`", args[1], "`[`, k, "`]"):
        protect(`", k, "`):", k = c))[1 .. -2], ":")
else h := cat(cat(seq(cat("unprotect(`", args[k], "`):`", args[k], "`:=eval(`", args[1], "`[`,
    args[k], "`]:protect(`", args[k], "`):", k = 2 .. nargs))[1 .. -2], ":")
end if;
writeline(b, h), close(b);
(proc() read b end proc)(), null(interface(warnlevel = a)), remove(b),
    if (1 < nargs, [args[2 .. -1]], c)
end proc:
> With(M), With(M1); => [], []
# Maple 6 - 11

```

Так как *модули*, как правило, содержат процедуры, а все они поддерживают механизм *локальных* переменных, то при наличии неявно определенных *локальных* переменных, *Maple*-язык при выводе предупреждений об этом привязывает их к *объектам*, которые их содержат, как это иллюстрирует следующий достаточно простой фрагмент:

```

> GRSU := module() local a; export b; global t; a := proc() c := 68; h := 14 end proc;
    b := proc() d := 21 end proc end module:
Warning, `c` is implicitly declared local to procedure `a`
Warning, `h` is implicitly declared local to procedure `a`
Warning, `d` is implicitly declared local to procedure `b`
Warning, `d` is implicitly declared local to procedure GRSU:-b
# Maple 11
> e := proc() g := 63 end proc: t := proc() v := 68 end proc:
Warning, `g` is implicitly declared local to procedure `e`
Warning, `v` is implicitly declared local to procedure `t`

```

Как видно из настоящего фрагмента и в этом случае имеет место релизо-зависимость выводимых *Maple* сообщений, что, учитывая их суть, не делает чести разработчикам.

Декларация *export* определяет последовательность локальных имен объектов модуля, к которым возможен доступ *извне* модуля, т.е. она определяет локальные переменные (*присущие сугубо модулю и скрывающиеся от внешней среды*) *глобальными*. Однако, в отличие от *локальных* *экспортируемые* переменные не могут быть определены таковыми *неявно*. Их следует определять явно через *export*-декларацию. При этом важно помнить, что при экспортировании модулем неопределенной локальной переменной она не будет тождественна *одноименной глобальной* переменной текущего сеанса, что иллюстрирует следующий весьма простой пример, а именно:

```

> module R() export VG; VG := () -> `+(args) end module: evalb(VG = R:- VG); => false

```

```
> VG := 68: VG, R:- VG(68, 63, 43, 14, 21); ⇒ 68, 209
```

Из примера видно, что *экспортируемая* неопределенная переменная не тождественна одноименной глобальной *VG*-переменной. Данное свойство модульного механизма языка пакета позволяет, в частности, определять в модуле одноименные с пакетными функциональные средства, но определяющие *различные* вычислительные алгоритмы, как это наглядно иллюстрирует следующий простой фрагмент, а именно:

```
> G:=module() local k; export ln; ln:=->evalf(sum(log(args[k]),k=1..nargs)) end module:
> ln(42, 47, 67, 89, 96, 68, 63, 43, 14, 21);
Error, (in ln) expecting 1 argument, got 10
> G[ln](42, 47, 67, 89, 96, 68, 63, 43, 14, 21); ⇒ 38.65291672
```

Такое свойство, в частности, достаточно полезно при создании собственных функций пользователя, подобных стандартным функциям, но отличающихся определенными особенностями. Более того, имеется возможность сохранять за ними пакетные имена.

Вне программных *модулей* обращение к *экспортируемым* переменным выполняется по одной из двух конструкций, представленных несколько выше. При этом, выполнение предложения *with(Имя модуля)* дает возможность обращаться ко всем *экспортируемым* переменным модуля только по их *именам*, делая их доступными в текущем сеансе для любого активного документа либо находящегося в очереди готовых документов, как это наглядно иллюстрирует следующий весьма простой фрагмент, а именно:

```
> Gr := module () export a, b, c; assign(a = 42, b = 47, c = 67) end module:
> Gr:- a, Gr:- b, Gr:- c, {a, b, c}, with(Gr); ⇒ 42, 47, 67, {a, b, c}, [a, b, c]
> a, b, c; ⇒ 42, 47, 67
```

Как следует из последнего примера фрагмента, *программные* модули могут выступать и на уровне модулей пакета. Именно такой механизм в значительной мере позволяет облегчить имплантирование в среду пакета функциональных средств из ряда других программных систем, в частности, именно так было и с модулем **NAG LinearAlgebra**.

Каждое определение *Maple*-процедуры ассоциируется с *неявными* переменными *args*, *nargs*, *procname*, рассмотренными выше. Тогда как с определением *ПМ* ассоциируется лишь одна неявная *thismodule*-переменная. В рамках тела модуля данной переменной присваивается содержащий ее модуль. Это позволяет ссылаться на *модуль* в рамках *его* собственного определения. Довольно простой пример иллюстрирует использование модульной переменной *thismodule*, а именно:

```
> module AV () export a, b; a := () -> sum(args[k], k=1..nargs); b := thismodule:- a(42,
47, 67, 62, 89, 96) end module: AV:- a(68, 63, 43, 48, 21, 14), AV:- b; ⇒ 257, 403
```

Посредством *thismodule*-переменной имеется возможность организации *рекурсивных* выполнений модуля внутри *самого* модуля, что существенно расширяет возможности модульного программирования в программной среде *Maple*-языка пакета.

По функции *op* можно получать доступ к трем компонентам модуля *M*, а именно:

- (1) *op(1, eval(M))* – последовательность экспортов программного модуля *M*
 - (2) *op(2, eval(M))* – оболочка определения программного модуля *M*
 - (3) *op(3, eval(M))* – последовательность локальных переменных модуля *M*
-

Приведем пример на применение функции *op* относительно простого модуля:

```
> M := module() local a, b; export x; a, b := 68, 63; x := (y) -> a*y + b*y end module:
> [op(1, eval(M)), op(2, eval(M)), [op(3, eval(M))],
    [x], module() local a, b; export x; end module, [a, b]
```

Следует отметить, что по вызову *op(2, eval(M))* возвращается не *исходное* определение модуля, а скорее его упрощенная копия (*оболочка определения модуля*) без самого тела. Копия используется, как правило, только для качественной печати модуля. С другой стороны, собственно определение модуля имеет 8 операндов, доступных по функции *op(k, 'M')*, где 'M' – определение программного модуля *M*, а именно:

- (1) *op(1, 'M')* – последовательность неявных параметров модуля *M*
- (2) *op(2, 'M')* – последовательность локальных переменных модуля *M*
- (3) *op(3, 'M')* – последовательность опций модуля *M*
- (4) *op(4, 'M')* – последовательность экспортируемых переменных модуля *M*
- (5) *op(5, 'M')* – последовательность описательных строк из *description*-секции
- (6) *op(6, 'M')* – последовательность глобальных переменных модуля *M*
- (7) *op(7, 'M')* – последовательность лексических параметров модуля *M*
- (8) *op(8, 'M')* – имя поименованного программного модуля

Простой фрагмент иллюстрирует получение всех операндов определения модуля:

```
> seq(lprint(op(k, 'module Avz () local a, b; global z, h, w; export x; option package;
description "RANS_IAN_2010"; a, b, z, h, w := 68, 63, 43, 21, 14; x := (y) -> a*y + b*y
end module')), k=1 .. 8);
thismodule
a, b
package
x
"RANS_IAN_2010"
z, h, w
NULL
Avz
```

Дополнительно к ранее рассмотренным для программных модулей, начиная с релиза 10, введен ряд дополнительных переменных, а именно. Если модуль *M* экспортирует переменную *ModuleApply*, то вызов формата *M(args)* обеспечивает вызов процедуры *M:- ModuleApply(args)*, как иллюстрирует следующий весьма простой пример:

```
> M:=module() export ModuleApply; ModuleApply:= () -> `+(args)/nargs end module:
> 6*M(68, 63, 43, 48, 21, 14); ⇒ 257
```

Если модуль содержит локальную или экспортируемую переменную *ModuleLoad*, то определяемая ею процедура вызывается, когда модуль читается из *Maple*-библиотеки, содержащей его. Если модуль имеет локальную или экспортируемую *ModuleUnload*-переменную, определенная ею процедура вызывается, когда модуль недоступен либо в случае завершения сеанса с *Maple*. При этом, две последние переменные являются

зеркальными средствами опций *load* и *unload* модуля. Если модуль имеет локальную или экспортируемую *ModulePrint*-переменную, вместо самого модуля возвращается результат вызова *ModulePrint()*. Детальнее с этими *переменными* можно ознакомиться в справочной системе *Maple* по запросу *?module*. Как следует из их описания, данные переменные ничего особо существенного не несут, однако в некоторых случаях могут оказаться достаточно полезными средствами при программировании модулей.

Механизм вызовов *Maple* не позволяет использовать *динамически* генерируемые имена для *экспортов* модуля. Данную задачу вполне успешно решает процедура *dcemod* [45]. Исходный текст процедуры и пример ее использования приведены ниже. Детальнее с данной процедурой и ее возможностями можно ознакомиться в книгах [28-38].

```
> dcemod := proc(M::symbol, ex::symbol) local a; global _v63g58;
  unassign('_v63g58'), if (nargs < 2, ERROR("quantity of arguments should be more
  than 1 but has been received <%1>", nargs), if (type(M, `module`),
  if (member(ex, {exports(M)}), assign67(a = "_$EE_", '_v63g58' = NULL),
  ERROR("<%1> does not export <%2>", M, ex)),
  ERROR("<%1> is not a module", M))), null(writeline(a, cat("`_v63g58`:=", M, "[", ex, "]((",
  if (2 < nargs, seqstr(args[3 .. nargs]), ``), "):"))),
  (proc() close(a); read a; remove(a) end proc()), _v63g58, unassign('_v63g58')
  end proc;
> 6*dcemod(SimpleStat, SR, [68, 63, 43, 48, 14, 21], 1, 10); ⇒ 257
```

Параметризация модулей. В отличие от процедур *программные модули* не используют механизма *формальных аргументов*. По этой причине для использования *ПМ* в задачах параметрического программирования используется следующая общая конструкция:

```
Proc := proc(Параметры {::Типы})
  Module () export {Переменные};
    <ТЕЛО модуля, содержащее Параметры>
  end module
end proc;

> SveGal := proc(a::integer, b::integer)
  module ()
  export Gal, Sv;
    Gal := () -> '+'(args)/(a*nargs + b); Sv := () -> '*'(args)(a + b^nargs)
  end module
end proc;

> Res := SveGal(420, 72): 222*Res:- Gal(63, 68, 43), Res:- Sv(43, 48, 14, 21); ⇒ 29, 6068160
```

Механизм параметризации позволяет создавать модули, весьма легко настраиваемые на *конкретные* условия применения, что обеспечивает их гибкость и мобильность при программировании задач из различных приложений. Вышеприведенный фрагмент иллюстрирует использование данной конструкции для *параметризации* конкретного *программного* модуля, определенного в процедуре *SveGal*. Данная процедура *SveGal* в

качестве формальных аргументов a , b использует параметры *вложенного* в нее модуля, экспортирующего 2 функции *Gal* и *Sv*. Присвоение вызова процедуры с конкретными фактическими аргументами некоей переменной генерирует программный модуль, поименованный первым способом, параметры a и b которого получают конкретные значения. Итак, мы получаем программный модуль, настроенный на определенные значения его параметров. В дальнейшем такой модуль используется вышеописанным способом. Описанный механизм позволяет несложно производить параметризацию программных модулей, обеспечивая решение разнообразных задач *параметрического* программирования в среде пакета *Maple*.

Процедуры нашей Библиотеки [45] представляют целый ряд весьма полезных средств для работы с программными и пакетными модулями, существенно дополняющими имеющиеся стандартные средства пакета. Вместе с этим, данные средства позволяют детализировать сами модульные *Maple*-объекты и использовать их особенности для программирования задач с использованием подобных объектов. Указанные средства поддерживают такие виды обработки как: конвертирование модулей в процедуры, проверка наличия в файлах некорректных модулей, проверка аргументов процедур и модулей, проверка активности (*годности к непосредственному использованию*) модуля или процедуры, проверка типа модульной таблицы, преобразование файлов данных входного формата *Maple*, содержащего модули, преобразование модуля второго типа в первый тип, конвертирование файла входного формата *Maple* в файл внутреннего формата *Maple*, и наоборот, и т.д. Представленные в [45] инструментальные средства позволяют обеспечить набор разнообразных полезных операций с процедурными и модульными объектами *Maple*. Такие средства используются достаточно широко при расширенном программировании различных приложений в пакете *Maple* и которые в целом ряде случаев весьма существенно упрощают сам процесс программирования.

6.1.11. Сохранение процедур и программных модулей в файлах

Пакет *Maple* располагает средствами сохранения в файлах процедур и программных модулей с возможностью их последующего чтения как в текущем сеансе, так и после перезагрузки пакета. С целью сохранения процедур и программных модулей в файле служит предложение **save** языка пакета, имеющее следующие форматы кодирования:

save $N1, N2, \dots, Nk, <Файл>$ или **save($N1, N2, \dots, Nk, <Файл>$)**

где $N1, N2, \dots, Nk$ – последовательность *имен* сохраняемых объектов и *Файл* – имя файла или полный путь к нему типа {string | symbol}. Объекты сохраняются в файлах в одном из 2 форматов (*входном Maple-формате* и *внутреннем m-формате*). Вызов **save($N1, N2, \dots, Nk, <Файл>$)** либо выполнение предложения **save $N1, N2, \dots, Nk, <Файл>$** пишет в файл определения имен $N1, N2, \dots, Nk$ в виде последовательности предложений присвоения. При попытке сохранения неопределенного имени A в файл запишется предложение вида $A := A$. Успешный вызов **save** возвращает *NULL*-значение, т.е. ничего.

При этом, если для файла определено *m*-расширение имени, то файл сохраняется во *внутреннем m-формате* пакета, иначе используется *входной* формат языка; *m-формат*

используется для сохранения процедур, модулей и других объектов в компактном и более простом для чтения *Maple* формате. Объекты, сохраненные во внутреннем *m*-формате, смогут читаться быстрее (*прежде всего, файлы большого объема*), чем объекты, сохраненные во входном *Maple*-формате языка. Следующий весьма простой пример иллюстрирует сохранение целочисленного *L*-списка из 1000.000 элементов в файлах обоих форматов, из которого следует, что файл «*file*» входного формата занимает 7.71 MB, тогда как файл «*file.m*» внутреннего *Maple*-формата только 5.29 MB.

```
> L := [k$к=1 .. 1000000]: save(L, "C:\\temp\\file"); save(L, "C:\\temp\\file.m");
```

При этом, для больших сохраняемых объектов такая разница может быть достаточно существенной. При задании в **save**-предложении неопределенных идентификаторов, они сохраняются в виде **Id := 'Id'**; с выводом соответствующих сообщений, например:

```
> save(H, G, S, "D:\\MathMaple\\datafile");
Warning, unassigned variable `H` in save statement
Warning, unassigned variable `G` in save statement
Warning, unassigned variable `S` in save statement
```

Именно *внутренний m-формат (m-файлы)* представляет *Maple*-объекты, сохраняемые в библиотеках пакета (*т.н. Maple-библиотеках*) и идентичных с ними по организации. Однако здесь имеется одно довольно существенное «*но*». Файлы с *Maple*-объектами, сохраненными во входном *Maple*-формате посредством предложения **save**, мобильны относительно всех версий пакета (*хотя они и могут быть синтаксически зависимыми от релиза*), тогда как файлы внутреннего *m*-формата всегда релизо-зависимы. Попытка чтения *m*-файла (*не соответствующего текущему релизу*) предложением **read** вызывает ошибочную ситуацию, тогда как чтение файла во входном *Maple*-формате корректно в любом релизе, если определения процедур и модулей, находящихся в нем, не будут содержать релизо-зависимых синтаксических элементов. В нынешней реальности *m*-файл, созданный в среде *Maple 6*, несовместим с *m*-файлом, созданным в среде релизов 7-11, тогда как в рамках релизов 7 – 11 имеет место полная совместимость. Обусловлено это изменением соответствия между идентификационными номерами внутренних структур данных и их именами. Согласно нашей терминологии [8-14,28-38] файлы, корректно читаемые предложением **read**, называются файлами пакета либо *Maple*-файлами.

Загрузка в текущий сеанс сохраненного по **save**-предложению файла выполняется по предложению **read**, которое имеет следующие простые форматы кодирования:

read <Файл> или **read(<Файл>)**

где *Файл* – имя файла либо полный путь к нему типа {string | symbol}; более того, если файл указывается лишь именем, а не полным путем к нему, то предполагается, что он находится в текущем каталоге. Более того, результат чтения файла посредством **read**-предложения зависит от формата файла. Общим является только тот факт, что после загрузки файла содержащиеся в нем определения становятся доступными в текущем сеансе, если впоследствии не определяется противного. Между тем, если загружается файл входного *Maple*-формата, в случае завершения **read**-предложения разделителем «;» на монитор выводится содержимое файла, а вызов **read**-предложения возвращает

значение последнего его предложения. Тогда как по «:»-разделителю информации не выводится, но также возвращается значение последнего предложения загруженного файла. В случае загрузки *m*-файла информации не выводится и возвращается *NULL*-значение. На первых порах работы с *Maple* средства доступа к *внутренним m*-файлам наиболее востребованы при создании различных библиотек функций или процедур пользователя, а также сохранения часто используемых конструкций. Независимо от формата (*m*-формат либо *ASCII*-формат) сохраненного по предложению **save** файла последующая его загрузка по **read**-предложению вызывает вычисление *всех* входящих в него определений, делая их доступными для текущего сеанса работы с пакетом.

В *Maple 6* предложение **save** допускает формат *save(F)*, по которому в файл *F* пишет все вычисленные в текущем сеансе имена в форме последовательности предложений присвоения. Кроме того, данная возможность предоставляется только для *F*-файлов, чьи имена имеют расширение «.m», т.е. файлы внутреннего *Maple*-формата. Однако, начиная с *Maple 7*, данная возможность отсутствует. Между тем, в целом ряде случаев подобное средство сможет упростить программирование и облегчить работу с *Maple*. В этом плане может оказаться достаточно полезной наша процедура *saveall* [41-45]. В следующем фрагменте представлены исходный текст и пример применения *saveall*.

```
> saveall := proc(F::{string, symbol}) local _avz_agn_asv_, a, k, j, t, psi, p, f;
    _avz_agn_asv_ := ((({anames()}) minus {anames('environment')}) minus
{anames('builtin')}) minus {anames('package')}) minus {'_libobj', '_warning', 'interface',
'_warnings', 'type/interfaceargs', 'lasterror', 'lastexception', 'StandardContext', 'Empty',
'Context/InitProcs', 'ContextKey', 'context/InitProcs', 'ContextData', 'stack', 'ContextMenu'};
    _avz_agn_asv_ := {seq('if'(cat("", k)[1 .. 3] = "CM:" or type(k, 'libobj'), NULL, k),
    k = _avz_agn_asv_)};
    assign(a = interface(warnlevel), p = packages(), null(interface(warnlevel = 0)),
    assign(psi = (a -> convert(eval(a), 'string'))); _avz_agn_asv_ := {seq('if'(Search2(psi(k),
{"the University of Waterloo", "Waterloo Maple Inc"})<>[], NULL, k), k = _avz_agn_asv_)};
    _avz_agn_asv_ := {seq('if'(search(cat("", k), "/", 't'), 'if'(type(cat(`, cat("", k)[1 .. t - 1]),
'libobj'), NULL, k), k), k = _avz_agn_asv_)};
    null(interface(warnlevel = a));
    if p <> [] then for j in p do for k in _avz_agn_asv_ do
        if type(cat(j, "/", k), 'libobj') then _avz_agn_asv_ := subs(k = NULL, _avz_agn_asv_)
        end if
    end do
    end do
    end if;
    if _avz_agn_asv_ = {} then WARNING("current session does not contain user definite
objects suitable for saving")
    else _avz_agn_asv_ := op(_avz_agn_asv_); f := pathtf(F);
        (proc(_avz_agn_asv_) save args, f end proc)(_avz_agn_asv_), f
    end if
end proc;
```

```
> saveall("c:\\temp\\tallinn\\ riga\\vilnius\\grodno\\save2114.m");
Warning, current session does not contain user definite objects suitable for saving
> saveall("c:\\temp\\tallinn\\ riga\\vilnius\\grodno\\save2114.m");
"c:\\temp\\grodno\\bbb\\ccc\\save2114.m"
```

Вызов процедуры *saveall(F)* возвращает реальный путь к принимающему файлу (*имя которого или путь к нему определены фактическим аргументом F*), который содержит все вычисленные в текущем сеансе имена в формате последовательностей предложений присвоения. При этом, процедура не сохраняет встроенные средства, библиотечные средства в пределах всех библиотек, определенных *libname*-переменной, переменные среды пакета, пакетные модули. При отсутствии *сохраненных* имен вызов процедуры возвращает *NULL*-значение, т. е. ничего, с выводом соответствующего сообщения, как иллюстрирует предыдущий фрагмент. В целом ряде случаев процедура *saveall* имеет достаточно полезные приложения, с некоторыми из них можно ознакомиться в [28].

Еще на одном существенном аспекте следует заострить внимание. Предложение *save* некорректно сохраняет программные модули в *m*-формате, например:

```
> restart; M := module () export x; x := () -> `(args)/nargs end module; M1 := module ()
  export y; y:=() -> `+(args)/nargs end module; M2:=module () export z; z:=() -> `+(args)
  end module; save(M, M1, M2, "C:/temp/M.m"); restart; read("C:/temp/M.m");
> map(type, [M, M1, M2], `module`), with(M), with(M1), with(M2);
      [true, true, true], [x], [y], [z]
> M:- x(68, 63, 43, 21, 14, 48), x(68, 63, 43, 21, 14, 48);
      x(68, 63, 43, 21, 14, 48), x(68, 63, 43, 21, 14, 48)
> M1:- y(68, 63, 43, 21, 14, 48), y(68, 63, 43, 21, 14, 48);
      y(68, 63, 43, 21, 14, 48), y(68, 63, 43, 21, 14, 48)
> M2:- z(68, 63, 43, 21, 14, 48), z(68, 63, 43, 21, 14, 48);
      z(68, 63, 43, 21, 14, 48), z(68, 63, 43, 21, 14, 48)
> M:- x(68, 63, 43, 21, 14, 48), 6*M1:- y(68, 63, 43, 21, 14, 48), M2:- z(68, 63, 43, 21, 14, 48);
      433266624, 257, 257
```

Из приведенного фрагмента следует, что модули *M*, *M1* и *M2*, сохраненные по *save*-предложению в *m*-файле, затем читаются *read*-предложением в текущий сеанс и как модули действительно распознаются *type*-функцией. Более того, вызов процедуры *with* вполне корректно возвращает списки экспортируемых модулями переменных. Тогда как стандартные вызовы этих переменных возвращаются невычисленными, т.е. экспортируемые данными модулями переменные оказываются неопределенными. И причина лежит в том, что *save*-предложение не сохраняет тела *программных* модулей в *m*-файлах внутреннего *Maple*-формата языка пакета.

Весьма детальное обсуждение данного вопроса может быть найдено в наших книгах [28-38,42-44]. Для устранения подобного недостатка нами был предложен целый ряд полезных средств, расширяющих функциональные возможности предложений *save* и *read*, с которыми можно ознакомиться в отмеченных книгах и в нашей Библиотеке [45]. В частности, предложение *save* не позволяет также корректно сохранять в файле

динамически вычисляемые имена, что в целом ряде случаев представляется довольно существенным недостатком. Наша процедура *save1* устраняет этот недостаток. Вызов процедуры *save1(N, E, F)* пишет указанные переменные, определенные фактическим аргументом *N*, в файл, указанный фактическим аргументом *F*, как *последовательность* предложений присвоения. Если некоторому элементу *n* из *N* не делалось присвоения, то в файл записывается предложение присвоения *n := n*; с выводом соответствующего сообщения. Если некоторый элемент *n* из *N* защищен (*имеет protected-атрибут*), то элемент игнорируется с выводом соответствующего сообщения. Более того, если все элементы из *N* защищены, то инициируется ошибочная ситуация. Успешный вызов процедуры *save1* возвращает полный путь к файлу *F*, обеспечивающему присвоения выражений *E* соответствующим символам из *N* в текущем сеансе *Maple* с выводом *всех* необходимых информационно-диагностических сообщений.

Эта процедура существенно расширяет возможности предложения *save*, обеспечивая сохранение в файлах и входного формата *Maple*, и внутреннего *Maple*-формата всех присвоений переменным с динамически генерируемыми именами. В ряде задач это – весьма важная возможность. Более того, процедура поддерживает сохранение *Maple*-объектов в файлах с произвольными путями к ним. Примеры представленного ниже фрагмента иллюстрируют некоторые из наиболее типичных применений *save1*.

```
> save1 := proc(N::{symbol, list(symbol)}, E::anything, F::{string, symbol})
  local a, b, c, k, nu, omega, zeta, psi, r, s, x;
  assign(b = (x -> null(interface(warnlevel = x))), zeta = "_$Euro$_",
  nu = "save(", s = "symbols %1 are protected");
  if not type(eval(F), {'symbol', 'string'}) then
    ERROR("argument <%1> can't specify a datafile", F)
  elif not type(F, 'file') then c := interface(warnlevel); b(0); r := CF1(MkDir(F, 1)); b(c)
  else r := CF1(F)
  end if;
  psi := proc(f) local a, k, p, h;
    if (f[-2 .. -1] = ".m", RETURN(), assign(p = fopen(f, 'READ', 'TEXT'), a = "$$$_"));
    while not Fend(p) do h := readline(p); writeline(a, if (h[-1] <> ";", h, cat(h[1 .. -2], ":")))
    end do;
    null(close(f, a), writebytes(f, readbytes(a, infinity)), close(f), remove(a))
  end proc;
  omega := proc(N::{symbol, list(symbol)}, E::anything, F::{string, symbol}) local k;
    if (type(N, 'symbol'), assign('nu' = cat(nu, N, ",")), seq(assign('nu' = cat(nu, N[k], ",")),
    k = 1 .. nops(N))), writeline(zeta, cat(nu, "", r, "", ":")), close(zeta);
    if type(N, 'symbol') then assign(N = eval(E)) elif type(E, 'list') then
      for k to min(nops(N), nops(E)) do assign(N[k] = eval(E[k])) end do
    else assign(N[1] = eval(E))
    end if;
    (proc(zeta) read zeta; remove(zeta) end proc)(zeta)
  end proc;
```

```

if type(N, 'symbol') and type(N, 'protected') then ERROR("symbol <%1> is protected", N)
elif type(N, 'list') then assign(a = []);
  for k to nops(N) do `if` (type(N[k], 'protected'), NULL, assign('a' = [op(a), N[k]])) end do;
  `if` (a = [], ERROR("all symbols %1 are protected", N), op([omega(a, args[2], r), psi(r),
  `if` (nops(a) = nops(N), NULL, WARNING(s, {op(N)} minus {op(a)})))))
else omega(args[1 .. 2], r), psi(r)
end if;
r, WARNING("the saving result is in datafile <%1>", r)
end proc;
> save1([G, cat(x, y, z), cat(a, b, c), cat(h, t), convert("RANS", symbol), cat(S, v)], [63, 68, 43,
48, 14, 21], "C:\\Academy\\RANS\\IAN.m"); G, xyz, abc, ht, RANS, Sv;
Warning, the saving result is in datafile <c:/academy/rans/ian.m>
"c:/academy/rans/ian.m", 63, 68, 43, 48, 14, 21
> restart; read("C:\\Academy\\RANS\\IAN.m"); G, xyz, abc, ht, RANS, Sv;
63, 68, 43, 48, 14, 21
> save1([cat(x,y,z), cat(y,z), cat(z,6)], [proc() `+` (args)/nargs end proc, 63, 43], "C:/TEMP");
Warning, the saving result is in datafile <c:/_temp>
"c:/_temp"
> restart; read("C:\\_Temp"); 2*xyz(1, 2, 3, 4, 5, 6), yz, z6; => 7, 63, 43

```

Приведем еще пару процедур, расширяющих предложения **save** и **read** для работы с программными модулями пакета. Обе процедуры *savem1* и *readm1* имеют форматы кодирования следующего достаточно простого вида, а именно:

savem1(F, M) и *readm1(R)*

соответственно, где *F* – имя или полный путь к файлу *внутреннего Maple-формата (m-файлу)* и *M* – программный модуль первого типа либо их последовательность. Более того, модули не должны иметь *protected*-атрибута. Процедура *savem1* предназначена для сохранения программных модулей первого типа в файлах внутреннего формата (*m-файлах*); если в качестве *F*-аргумента указан не *m-файл*, к его имени добавляется расширение «.m». Успешный вызов процедуры *savem1* возвращает путь к созданному *m-файлу* с сохраненными в нем модулями *M*. Тогда как процедура *readm1(R)* читает в текущий сеанс файл *R* с корректной активацией сохраненных в нем по процедуре *savem1* программных модулей. Итак, успешный вызов процедуры *readm1* возвращает *NULL*-значение. Ниже представлен фрагмент с исходными текстами обеих процедур и примерами их применения для сохранения/чтения модулей первого типа.

```

> savem1 := proc(F::[string, symbol], M::mod1) local a;
  if not (map(type, {args[2 .. -1]}, 'mod1') = {true}) then
    error "modules should be of the first type"
  end if;
  if "" || F[-2 .. -1] <> ".m" then a := F || ".m" else a := F end if;
  seq(assign(args[k] = convert(eval(args[k]), 'string')), k = 2 .. nargs);
  (proc() save args, a end proc)(args[2 .. -1]), a

```

```

end proc;
> readm1 := proc(F::file) local b, c, k;
  assign(b = {}, c = 14/21);
  if "" || F[-2 .. -1] <> ".m" then error "file should has internal Maple format, but had
                                     received '%1'-type", Ftype(F)
  end if;
  do
    if c <> 0 then c := readline(F); if c[1] = "I" then b := {op(b), Iddn1(c)} end if
    else break
    end if
  end do;
  close(F), assign('b' = map(convert, b, 'string')); read F;
  for k in b do
    try parse(cat(k, ":", eval(` || k, ":"), 'statement'); NULL
    catch: error "file <%1> should be saved by procedure `savem1`, F
    end try
  end do
end proc;
> M := module () export x; x := () -> `(args)/nargs end module: M1 := module () export y;
  y := () -> `+(args)/nargs end module: M2 := module () export z; z := () -> `+(args) end
  module: M3 := module () export h; h := () -> evalf(sqrt(`+(args)), 5) end module:
> savem1("C:\\temp\\Academy", M, M1, M2, M3); => "C:/temp/Academy.m"
> restart; readm1("C:\\temp\\Academy.m");
> map(type, [M, M1, M2, M3], `module`), with(M), with(M1), with(M2), with(M3);
      [true, true, true, true], [x], [y], [z], [h]
> M:- x(68, 63, 43, 21, 14, 48), x(68, 63, 43, 21, 14, 48); => 433266624, 433266624
> 6*M1:- y(68, 63, 43, 21, 14, 48), 6*y(68, 63, 43, 21, 14, 48); => 257, 257
> M2:- z(68, 63, 43, 21, 14, 48), z(68, 63, 43, 21, 14, 48); => 257, 257
> M3:- h(68, 63, 43, 21, 14, 48), h(68, 63, 43, 21, 14, 48); => 16.031, 16.031

```

Обе процедуры обрабатывают основные особые и ошибочные ситуации. В качестве довольно полезного упражнения читателю рекомендуется рассмотреть организацию обеих приведенных процедур, использующих ряд практически полезных приемов.

В целом ряде случаев возникает необходимость анализировать *m*-файл внутреннего формата на предмет находящихся в нем объектов. Данную задачу решает процедура *mnames*, исходный текст и примеры применения которой приведены ниже. Ее вызов *mnames(F)* для анализа в среде *Maple 6* файла *F*, подготовленного в среде *Maple 7-11*, возвращает лишь множество имен объектов с выводом соответствующего сообщения.

```

> mnames := proc(F::file) local a, b, c, h, t, tab;
  if Ftype(F) <> ".m" then error "datafile <%1> has a type different from m-type", F
  else close(F), assign(a = {}, b = Release(), c = parse(readline(F)[2]));

```

```

do h := readline(F);
  if h = 0 then close(F); break
  else if h[1] = "I" then a := {op(a), Iddn1(h)} minus {false} end if
  end if
end do;
if b = 6 and 7 <= c then a, WARNING("procedure return presents the names of Maple
                                objects only, because m-file had been created in release >= 7
                                whereas the current release is 6", b)

else read F;
  for h in a do t := whattype(eval(h));
    if type(tab[t], 'assignable') then tab[t] := {}; tab[t] := {op(tab[t]), h}
    else tab[t] := {op(tab[t]), h}
    end if
  end do;
  `if`(nargs = 1, NULL, Unassign(op(a))), eval(tab)
end if
end if
end proc;

> mnames("C:/Temp/Academy\\Test72.m", 14); AGN, L1, S, Akr, Grod, AVZ, eval(m);
table([float = {AGN}, integer = {AVZ}, list = {L1, L}, fraction = {Akr}, set = {S, S1}, module = {M,
M1}, array = {a, v, m, v1, m1, a1}, table = {T1, T}, complex = {Grodno}, procedure = {P}])
AGN, L1, S, Akr, Grodno, AVZ, m

```

Процедура *mnames* неоднократно успешно использовалась для анализа незнакомых файлов *внутреннего Maple-формата*. Целый ряд других полезных средств для работы с программными модулями *внутреннего Maple-формата* представлен в наших книгах [28-38,41] и в Библиотеке [45] для пакета *Maple* релизов 6 - 11. Наряду с познавательным интересом данные средства представляют определенный интерес для практического программирования в *Maple*, предлагая целый ряд полезных и эффективных приемов программирования наряду с освоением программирования в *Maple-языке* в качестве достаточно хороших примеров программирования различных системных задач.

Между тем, справедливости ради вполне уместно привести уточнение относительно наших процедур *save1*, *savem1* и *readm1* наряду с другими, упомянутыми в контексте создания средств, расширяющих функциональные возможности стандартных *save* и *read*, и устраняющих некоторые их недостатки. Действительно, до 11-й версии *Maple* упомянутые средства не только закрывали все недоработки стандартных средств для сохранения программных модулей в файлах *внутреннего Maple-формата*, но и в ряде случаев весьма существенно упрощали программирование целого ряда задач в *Maple* версий 6-10. И лишь в *Maple 11* стандартное предложение *save* было доведено до ума.

Наконец, сохранение процедур и модулей возможно производить непосредственно в библиотеках пользователя, структурно подобных главной библиотеке пакета. С этой целью была создана процедура *UpLib* [45], чей успешный вызов *UpLib(F,N)* обновляет

библиотеку, указанную *первым* фактическим аргументом *F* процедурами, таблицами и/или пакетными модулями, чьи имена определены *вторым* фактическим аргументом *N* (список символов). Успешный запрос процедуры возвращает *NULL*-значение, то есть ничего, с выводом соответствующих сообщений. Процедура обрабатывает основные ошибочные и особые ситуации, связанные с отсутствием библиотеки или отсутствием сохраняемых средств. При возникновении таких ситуаций возникают ошибки. Ниже представлен исходный текст процедуры *UpLib* и примеры ее использования.

```
> UpLib := proc(L::{string, symbol}, N::list(symbol)) local a, b, c, d, h, k, p, t, n;
assign(n = nops(N), a = [], p = [libname], t = cat(CDM(), "/lib/", L));
if member(cat("", L)[2..3], {"/", ":\\"}) and type(L, {'mla', 'mlib'}) then h := cat("", L)
elif type(t, {'mla', 'mlib'}) then h := t else
  for k to nops(p) do
    if Search1(Case(cat("\\", L)), CF(p[k]), 'd') and d = ['right'] then h := p[k]; break end if
  end do
end if;
if (type(h, 'symbol'), ERROR("<%1> is not a Maple library", L),
seq(if (type(N[k], {'module', 'table', 'procedure'}), assign('a' = [op(a), N[k]]),
WARNING("<%1> is not a procedure and not a module, and not a table", N[k])), k=1..n));
if (nops(a) = 0, ERROR("procedures, modules or tables do not exist for saving"),
assign(b = NLP(L)[1]));
for k to nops(a) do
  if member(a[k], b) or Search1(cat(a[k], `:-`), a[k], 'd') and d = ['left']
  then WARNING("<%1> does exist and will be updated", a[k])
  else WARNING("<%1> does not exist and will be added", a[k])
  end if
end do;
assign(c = savelibname), assign('savelibname' = h), savelib(op(a));
unassign('savelibname'), assign(savelibname = c),
WARNING("Library update has been done!")
end proc;
> UpLib("F:\\userlib6789\\userlib11\\userlib", [saveall]);
Warning, <saveall> does exist and will be updated
Warning, Library update has been done!
```

Наконец, процедура *MapleLib* предназначена для обновления и главной библиотеки *Maple* и библиотек пользователя, имеющих структурную организацию, аналогичную главной библиотеке *процедурами, таблицами* и программными модулями, определения которых были вычислены в текущем сеансе *Maple*. Предполагается, что вычисленные определения *программных* модулей, подлежащих сохранению в библиотеке, содержат *package*-опцию. Со средствами работы с библиотеками пакета возможно достаточно детально познакомиться в наших книгах [28-39, 41-44] и в Библиотеке [45].

Стандартная процедура *with* применима к пакетным модулям любого типа, позволяя на интерактивном уровне не только возвращать список имен экспортов модуля, но и обеспечивать их доступность в текущем сеансе; *with* обеспечивает те же возможности,

что и предложение **use**, однако она работает только на интерактивном уровне. Тогда как предложение **use** поддерживает работу лишь с модулями как программными, так и пакетными, как вне процедур/модулей, так и в их телах. Таким образом, процедура **with** эффективна только на *высшем* уровне, применяясь на интерактивном уровне (*т. е. в режиме работы интерактивно с текущим документом*), прежде всего. Поскольку **with** использует лексическое правило просмотра, то она не работает в телах процедур или модулей, в целом ряде случаев создавая определенные неудобства пользователю. При этом, в ранних версиях отсутствовало предложение **use** в качестве некоей альтернативы **with**. По этой причине нами был предложен ряд достаточно простых модификаций [9] стандартной процедуры **with**, одну из которых представляет следующий фрагмент:

```
> With6 := proc(P::{package}) local a, b, k;
  if not type(P, `module`) then a := interface(warnlevel); interface(warnlevel = 0);
    b := with(P); interface(warnlevel = a); b else a := [exports(P)]; unprotect(op(a));
    for k in a do assign(k = eval(P[k])) end do; protect(op(a)), a
  end if
end proc;
> restart; G := proc(P::package, M::Matrix) with(P); Determinant(M) end proc: M :=
LinearAlgebra[RandomMatrix](4, 4): G(LinearAlgebra, M);
Warning, the assigned name Map now has a global binding
Determinant  $\begin{pmatrix} -41 & -34 & -56 & 62 \\ 20 & -62 & -8 & -79 \\ -7 & -90 & -50 & -71 \\ 16 & -21 & 30 & 28 \end{pmatrix}$ 
> G := proc(P::package, M::Matrix) With6(P); Determinant(M) end proc: M :=
LinearAlgebra[RandomMatrix](4, 4): G(LinearAlgebra, M);  $\Rightarrow$  1611078
> restart; S := proc(P::package, L::list) with(P); Reverse(L) end proc: S(ListTools,
[68, 63, 43, 48, 21, 14]);
Warning, the assigned name Group now has a global binding
Reverse([68, 63, 43, 48, 21, 14])
> restart; S := proc(P::package, L::list) With6(P); Reverse(L) end proc: S(ListTools,
[68, 63, 43, 48, 21, 14]);  $\Rightarrow$  [14, 21, 48, 43, 63, 68]
```

Фрагмент приведен для *Maple* релизов 6 – 11; между тем, в этой связи вполне уместно отметить, что уже на примере различных версий данной процедуры мы в который раз убедились, что разработчики пакета *Maple* (также как и пакета *Mathematica*) довольно часто игнорируют такой важный аспект, как совместимость *снизу-вверх*, что, вероятно, обусловлено частой, порой не вполне мотивированной модификацией стандартных средств – или это результат *непродуманной* стратегии, или *коммерческая* составляющая, ориентированная на расширение рынка. В последнем случае такая политика может в полной мере рассматриваться как весьма неприглядная. Справедливости ради нужно отметить, что на сегодня версий пакета *Maple*, практически, в два раза больше, версий пакета *Mathematica*. И такая гонка версий зачастую не совсем мотивирована.

6.1.12. Создание пользовательских пакетных модулей в Maple

С каждым новым релизом пакета в нем появляются новые *пакетные* модули, средства которых ориентированы на некоторый определенный круг приложений. Получить список *пакетных* модулей, включенных в текущий релиз *Maple*, можно оперативно по запросу *?index,package*. В частности, таким способом несложно убедиться, что релизы 8, 10 и 11 *Maple* располагают соответственно 71, 101 и 102 пакетными модулями. При этом, в большинстве изданий и технической документации по пакету *Maple* данные структурированные наборы средств называются «пакетами», хотя на наш взгляд, это и не вполне правомочно. Более того, по большому счету именно сам *Maple* является пакетом. Наша *мотивировка* подобной терминологии представлена, например, в [12–14, 28–37, 42–44]. В данном смысле и *программные*, и *пакетные* модули в общем случае являются различными объектами. Модульная организация обеспечивает целый ряд преимуществ, детально не обсуждаемых здесь. В частности, она позволяет проводить независимую разработку средства с ориентацией на конкретный круг приложений, допускает использование одноименных со стандартными средств. Имеются и другие немаловажные преимущества модульной организации программных средств.

```
> M_Type := proc(M::symbol) local a, T, omega, nu, t, k;
    assign(a = {anames(procedure)}, omega = interface(warnlevel)), `if` (M = liesymm,
        RETURN("Tab"), assign(nu = (h -> null(interface(warnlevel = h)))));
    try t := [exports(M)];
        if t = [] then return WARNING("<%1> is not a module", M) else unassign('Fin');
            goto(Fin)
        end if
    catch "wrong number (or type) of parameters in function %1": NULL
    catch "invalid input: ":
        try nu(0); t := with(M)
            catch "%1 is not a package": nu(omega); return WARNING("%1 is not a module", M)
            catch "invalid input: %1": nu(omega); return WARNING("<%1> is not a module or
                can't be initialised", M)
            catch "system level initialisation for": nu(omega); return WARNING("module %1
                can't be initialised", M)
            end try
        end try;
    for k in t do if member(k, a) then next else unprotect(k); unassign(k) end if end do;
    Fin;
    nu(omega), assign(T = convert(eval(M), string)), `if` (searchtext("table", T, 1..5) = 1, 'Tab',
        `if` (searchtext("module", T, 1..6) = 1, 'Mod',
            `if` (searchtext("proc", T, 1..4) = 1, 'Proc', NULL))), nu(omega)
    end proc;
> map(M_Type, [PolynomialTools, ExternalCalling, LinearFunctionalSystems,
    MatrixPolynomialAlgebra, ScientificErrorAnalysis, CodeGeneration, LinearOperators,
```

```
CurveFitting, Sockets, Maplets, Student, Matlab, Slode, LibraryTools, Spread, codegen,
context, finance, genfunc, LinearAlgebra, geom3d, group, linalg, padic, plots, process,
simplex, student, tensor, MathML, Units, DEtools, diffalg, Domains, stats, Groebner,
GaussInt, LREtools, PDEtools, Ore_algebra, algcurves, orthopoly, combinat, diffoms,
combstruct, geometry, intrans, networks, numapprox, numtheory, plottools, sumtools,
XMLTools, powseries, ListTools, RandomTools, RealDomain, SolveTools, SumTools,
StringTools, TypeTools, Worksheet, OrthogonalSeries, FileTools, ScientificConstants,
RationalNormalForms, VariationalCalculus]); # Maple 10
```

```
[Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Tab, Mod, Mod, Mod, Tab,
Mod, Mod, Mod, Mod, Mod, Mod, Mod, Tab, Mod, Tab, Tab, Tab, Mod, Mod, Tab, Tab, Mod,
Proc, Mod, Mod, Tab, Tab, Mod, Tab, Tab, Mod, Tab, Tab, Mod, Tab, Tab, Mod, Mod, Mod, Mod,
Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod, Mod]
```

```
> Mulel(%); ⇒ [[Proc, 35, 1], [Tab, 13, 16], [Mod, 1, 50]]
```

Организационно пакетные модули в настоящее время бывают 3-х типов – *процедуры*, *программные модули* (как правило, *второго типа*) и *таблицы*. В частности, посредством процедуры *M_Type* [45] возможно тестировать *тип* пакетного модуля; ее применение в среде пакета *Maple 10* дает результат, представленный предыдущим фрагментом.

Таким образом, пакет *Maple 10* из 67 проанализированных располагает 16 модулями табличного типа (*Tab*), 50 модулями модульного типа (*Mod*) и только одним модулем процедурного типа (*Proc*). Распределение *пакетных* модулей по типам их *организации* зависит от релиза пакета и для *Maple 6*, например, типы распределились следующим образом: *Proc* - 1, *Tab* - 34 и *Mod* - 2, т.е. явно превалирует *табличная организация*. Выше были рассмотрены основные вопросы создания процедур и программных модулей, в этой связи останавливаться на этом аспекте не имеет смысла. Напомним только, что при создании пакетного модуля *процедурного* или *модульного* типа в его секции **option** необходимо кодировать опцию **package**, что позволит после сохранения его в *Maple-библиотеке* в последующем обращаться к содержащимся в нем средствам принятыми в пакете *Maple* стандартными способами.

Здесь мы рассмотрим лишь создание пакетного модуля *табличного* типа, как первого наиболее массового типа модулей в среде ранних релизов пакета *Maple*. Как можно будет впоследствии заметить, именно основные методы доступа к *пакетным* модулям определяются форматом, используемым при обращении к *табличным* объектам. При этом, общая схема создания пакетного модуля табличного типа простая и сводится к следующему. На первом этапе производятся тщательные тестирование, апробация и отладка процедур, предназначенных к включению в создаваемый пакетный модуль. Затем готовые процедуры погружаются в табличную структуру, у которой в качестве *выходов* выступают имена процедур, а *выходов* – их определения, т.е. создается таблица следующего простого формата, а именно:

```
Имя := table([P1 = proc(...) ... end proc, P2 = proc(...) ... end proc, ..., Pn = proc(..) ... end proc])
```

где *Имя* – имя таблицы (*впоследствии пакетного модуля*) и *Pj* – имена процедур (*j=1..n*). При этом, в качестве *выходов* таблицы наряду с определениями процедур могут быть также вызовы процедур, функций или произвольные *Maple-выражения*. На третьем

заключительном этапе *таблица* сохраняется описанным выше способом в библиотеке *Maple*. Если данная библиотека логически связана с главной библиотекой пакета, то сохраненная в ней таблица и будет являться вашим пакетным модулем. Рассмотрим типичный пример создания простого пакетного модуля табличного типа.

```
> TabMod := table([Sr = proc() evalf(`+`(args)/nargs, 6) end proc, Ds = proc() local k;
    evalf(sqrt(sum((Sr(args) - args[k])^2, k = 1..nargs)/nargs), 6) end proc]):
    TabMod[Art] := () -> `(args)/nargs^2: TabMod[SV]:=
    define(SV, SV()) = nargs*sum(args['k'], 'k' = 1..nargs)) , eval(SV): eval(TabMod):
> TabMod[SV](68, 63, 43, 14, 21, 48), TabMod[Sr](68, 63, 43, 14, 21, 48); => 1542, 42.8333
> TabMod[Ds](68, 63, 43, 14, 21, 48);
    0.166667 ( 6. ( Sr( 64, 59, 39, 10, 17, 44 ) - 64. )^2 + 6. ( Sr( 64, 59, 39, 10, 17, 44 ) - 59. )^2
    + 6. ( Sr( 64, 59, 39, 10, 17, 44 ) - 39. )^2 + 6. ( Sr( 64, 59, 39, 10, 17, 44 ) - 10. )^2
    + 6. ( Sr( 64, 59, 39, 10, 17, 44 ) - 17. )^2 + 6. ( Sr( 64, 59, 39, 10, 17, 44 ) - 44. )^2 )^(1/2)
> with(TabMod), TabMod[Ds](68, 63, 43, 14, 21, 48); => [Art, Ds, SV, Sr], 19.8948
> savelibname := "C:/Program Files/Maple 10/LIB/UserLib": savelib(TabMod); restart;
> libname := libname, "C:\\BOOK\\UserLib"; with(TabMod);
    libname := "C:\\Program Files\\Maple 8/lib", "C:/BOOK/UserLib"
    [Art, Ds, SV, Sr]
> seq(F(68, 63, 43, 14, 21, 48), F = [Art, SV, Ds, Sr]); restart;
    72211104, 1542, 19.8948, 42.8333
> seq(F(68, 63, 43, 14, 21, 48), F = [Art, SV, Ds, Sr]);
    Art(68,63,43,14,21,48), SV(68,63,43,14,21,48), Ds(68,63,43,14,21,48), Sr(68,63,43,14,21,48)
```

Так, *первые* три примера фрагмента иллюстрируют поэтапное определение таблицы *TabMod*, в которой *входами* являются имена процедур и *выходами* – их определения. В следующем примере производится вызов погруженных в таблицу *процедур SV* и *Sr* на конкретном наборе фактических аргументов с *возвратом* соответствующих значений, тогда как аналогичный вызов процедуры *Ds* возвращает результат *невычисленным*, не считая очевидных упрощений, обусловленных, прежде всего, использованием *evalf*-функции. Объясняется данная ситуация тем, что в этого типа *табличных* объектах не производится *полного* *вычисления* их входов в том понимании, что при использовании одним из выходов какого-либо входа таблицы либо иного вызова, он рассматривает данный вход *неопределенным*. Для устранения этой ситуации следует перед вызовами входов таблицы использовать вызов стандартной процедуры *with* либо нашей *With6*.

В частности, для инициации в текущем сеансе процедур таблицы *TabMod* (пример 6) используется вызов *with(TabMod)*; в результате последующий вызов *Ds*-процедуры получаем вполне определенным. При этом, вызов *with(TabMod)* возвращает список имен входов таблицы *TabMod*. Если пакетный модуль *Id* находится в библиотечном файле, путь к которому определен в переменной *libname* пакета, по вызову *with(Id)* производится его загрузка в *рабочую область пакета*, тем самым обеспечивая доступ ко всем поддерживаемым им функциональным средствам. Тогда как по конструкциям:

Id_Модуля[Id_Функции] и *Id_Модуля[Id_Функции](Фактические аргументы)*

можно загружать в рабочую область пакета лишь конкретную модульную функцию, вычисляя вызов конкретной *модульной* функции в конкретной точке соответственно, определяемой заданными фактическими аргументами *при условии*, что содержащий требуемую функцию/процедуру пакетный модуль находится в **Maple**-библиотеке, логически связанной с главной библиотекой пакета. Но, и в данном случае требуется выполнение приведенного выше условия – *предварительный* вызов *with*-процедуры.

Так, из двух последних примеров фрагмента, иллюстрирующих сказанное, наглядно видно, что после сохранения таблицы **TabMod** в **Maple**-библиотеке, которая логически сцеплена с *главной* библиотекой пакета, выполнение вызова *with(TabMod)* делает все процедуры, определенные в пакетном модуле **TabMod**, полностью доступными. При этом, без вызова *with(TabMod)* процедуры этого модуля остаются неопределенными. Более того, по вызову процедуры **packages()** возвращается *упорядоченный* список имен всех пакетных модулей, чей хоть один экспорт был активирован в текущем сеансе.

Процедура **ListPack** [45] позволяет получать *список пакетных* модулей, содержащихся в главной **Maple**-библиотеке пакета или в библиотеке, подобной *главной* библиотеке. Вызов процедуры **ListPack()** без фактических аргументов предполагает, что *пакетные* модули разыскиваются в главной **Maple**-библиотеке, в то время как вызов **ListPack(F)** обеспечивает *поиск* в **Maple**-библиотеке, чье имя или полный путь к ней определены фактическим аргументом *F*. Имеются и другие полезные приложения процедуры. В частности, она позволяет получать более точную классификацию *пакетных* модулей, чем по вызову *?index, package* для **Maple 8**. Так, классифицируемый по такому вызову пакетный модуль **Rif** на самом деле таким не является, что констатирует процедура. Следующий фрагмент представляет исходный текст процедуры и ее применение.

```
> ListPack := proc() local a, b, c, d, g, h, f, k, n, m, s, v, w, t, tst, gs, qt, p; global _modpack;
  if 0 < nargs and type(args[1], {'mlib', 'mla'}) then f := cat("", args[1], "/maple.hdb");
    if not belong(CF(args[1]), map(CF, [libname])) then
      assign67(p = 8, f = cat("", args[1], "/maple.hdb"), 'libname' = libname, args[1])
    end if;
    if type(f, 'file') then assign(d = {}, n = 120000, s = {}, v = {})
    else error "Maple library <%1> does not contain help database", args[1]
    end if
  else type(sin, 'libobj'), assign(d = {}, f = cat(_libobj, "/maple.hdb"), n = 120000, s = {}, v = {})
  end if;
  tst := proc(x) try parse(x); eval(%); parse(x) catch : NULL end try end proc;
  gs := proc(x::{set, list}) local a, f;
    assign(f = cat([libname][1][1 .. 2], "\_ $Art16_Kr9$ \_")),
    assign(a = cat("_000:=proc(x::{list, set}) local a, b, c, k; global _modpack;
    assign(c = interface(warnlevel), '_modpack' = {}), null(interface(warnlevel = 0));
    for k in x do try b:= With(k); if type(b, 'list') then _modpack:= {op(_modpack), k}
    end if catch : NULL end try end do; _modpack, null(interface(warnlevel = c)) end
    proc: _000(" , convert(x, 'string'), "):");
    writeline(f, a), close(f); read f; remove(f), _modpack,
```

```

unassign('_000', '_modpack'), `if`(p = 8, assign('libname' = op([libname][1 .. -2])), NULL)
end proc;
qt := x -> `if`(x = {}, NULL, x);
assign(m = filepos(f, infinity)), close(f), assign(b = open(f, 'READ'), h = {"With(", w = {});
while filepos(b) < m do assign('a' = convert(subs({0 = NULL, 13 = NULL, 10 = NULL,
    readbytes(b, n)), 'bytes')), assign('c' = Search2(a, h));
    if c <> [] then for k in c do g := Search2(a[k .. k + 35], {"", " "); " "});
        if g <> [] then d := {op(d), FNS(a[k + 5 .. k + g[1] - 2], " ", 3)} end if
    end do
end if
end do;
seq(`if`(search(k, " ", 't') or search(k, ":-", 't'), assign('v' = {cat(`, k[1 .. t - 1]), op(v)}),
    `if`(search(k, "[", 't'), assign('v' = {cat(`, k[1 .. t - 1]), op(v)},
        'w' = {k, op(w)}), assign('v' = {op(v), cat(`, k)})), k = d);
assign('v' = {seq(`if`(type(k, 'package'), k, NULL), k = v)}, 'w' = map(tst, w)), qt(v),
    `if`(w = {}, NULL, qt(gs(w)))
end proc:

```

> restart; ListPack();

Maple 8

{powseries, LREtools, plottools, numapprox, Ore_algebra, DEtools, Slode, polytools, algcurves,
 LinearOperators, diffalg, geometry, ListTools, linalg, plots, codegen, finance, Groebner, intrtrans,
 stats, padic, tensor, StringTools, RealDomain, numtheory, simplex, LibraryTools, TypeTools,
 queue, CurveFitting, OrthogonalSeries, process, SoftwareMetrics, VariationalCalculus, group,
 RandomTools, CodeGeneration, genfunc, Worksheet, RationalNormalForms, student, combinat,
 VectorCalculus, Maplelets, geom3d, SumTools, SolveTools, diffforms, LinearFunctionalSystems,
 Units, sumtools, networks, orthopoly, ScientificConstants, PolynomialTools, combstruct, SNAP,
 Spread, RootFinding, Matlab, Domains, liesymm, priqueue, LargeExpressions, PDEtools, MathML,
 ExternalCalling, LinearAlgebra, XMLTools, GaussInt, MatrixPolynomialAlgebra, context, Sockets},
 {Converter, Tools, Hypergeometric, Examples, Natural, Calculus1, stats[statplots], stats[describe],
 stats[fit], stats[anova], Elements, Standard}

> ListPack("C:\\Program Files\\Maple 11\\LIB\\UserLib");

Maple 10

{boolop, DIRAX, SoLists, SimpleStat, ACC}

Механизм *with*-процедуры, в частности, состоит в том, что ее вызов непосредственно влечет за собой вычисление (всех или заданных) выходов соответствующей модульной таблицы, в качестве которых могут выступать любые допустимые *Maple*-выражения, включая вызовы функций/процедур, как это иллюстрирует следующий пример:

```

> Kr[Sv] := 3: Kr[S] := proc() 'procname(args)' end proc: Kr[W] := readlib(ifactors):
    with(Kr), Sv, S(68, 63, 43, 14, 21, 48), ifactors(2010);
    [S, Sv, W], 3, S(68, 63, 43, 14, 21, 48), [1, [[2, 1], [3, 1], [5, 1], [67, 1]]]

```

Это обстоятельство можно довольно эффективно использовать в программировании в среде *Maple*-языка. Например, определив в качестве *выходов T*-таблицы *определения* разных процедур, можно весьма просто организовывать их *условное* выполнение, как это иллюстрирует следующий весьма простой фрагмент, а именно:

```
> T[x]:= proc() `+(args) end proc: T[y]:= proc() nargs end proc: T[z]:= proc() `*(args) end
proc: if whattype(AGN)='symbol' then op(with(T, x)(68,63,43))/op(with(T, y)(68,63,43))
else op(with(T, z)(68, 63, 43)) end if; ⇒ 58
```

Наряду с отмеченными имеется еще ряд весьма интересных применений табличных структур языка, полезных для *практического* программирования в среде *Maple* [28-37]. Более того, при организации *пакетных модулей Maple* допускает использование также нестандартных подходов. В качестве практического примера нами представлен один довольно простой и в ряде случаев полезный тип пакетных модулей, базирующийся на структурах *array*-типа. Для обеспечения доступа к данному типу нестандартным пакетным модулям создана довольно простая процедура *ArtMod* [45]. В ряде случаев подобные нестандартные подходы к организации *пакетных* модулей могут оказаться существенно эффективнее стандартных, поддерживаемых средствами пакета *Maple*, позволяя выполнять над структурами таких модулей нужные *символьные* вычисления и преобразования. Наш опыт апробации и эксплуатации пакета *Maple* релизов с 4-го по 11-й со всей определенностью говорят в пользу данного утверждения.

В заключение данного раздела кратко акцентируем внимание на способах обращения к *пакетному* модулю и содержащимся в нем объектам в целом, предполагая, что наш модуль находится в библиотеке, логически связанной с главной библиотекой пакета:

M[Function](args), M:- Function(args) – вызов функции *пакетного* модуля на заданных фактических аргументах *args* при условии, что пакетный модуль *M* имеет *модульную* структуру, иначе второй формат *вызова* инициирует ошибочную ситуацию формата «Error, `M` does not evaluate to a module». Тогда как *первый* формат корректен для модуля любого допустимого типа;

with(M) – возврат списка имен объектов, находящихся в модуле *M*, с их активацией в текущем сеансе. После такого вызова доступ к функциям модуля *M* обеспечивается в обычном формате *function(args)*; при этом, появление *предупреждений* вида «Warning, the name N has been redefined» говорит, что одноименная с библиотечной процедура *N* пакетного модуля заменила библиотечное средство. Во избежание данной ситуации рекомендуется использовать разовые вызовы пакетных средств по нижеследующего формата конструкциям **M[Function](args), M:- Function(args)**;

with(M, f1, f2, f3, ...) – возврат списка имен [f1, f2, ...] объектов модуля *M* с активацией в текущем сеансе приписанных им определений. После данного вызова доступ к этим функциям *f1, f2, ...* модуля *M* обеспечивается в обычном формате *fj(args)*;

packages() – возврат *упорядоченного* (согласно порядка обращения к модулям) списка имен всех пакетных модулей, чей хоть один экспорт был активирован в текущем сеансе;

ListPack() – возврат списка имен пакетных модулей, расположенных в *главной Maple*-библиотеке, **ListPack(F)** – возврат списка имен всех пакетных модулей, находящихся в *Maple*-библиотеке, чье имя или полный путь к которой определены аргументом *F*.

Некоторые примеры нижеследующего фрагмента иллюстрируют вышесказанное:

```
> with(DIRAX); ⇒ [conv, delete, empty, extract, insert, new, printd, replace, reverse, size, sortd]
> restart; with(linalg, det, rank); ⇒ [det, rank]
> restart; m:=matrix(3, 3, [x, y, z, a, b, c, d, g, h]): with(linalg, det, rank), det(m), rank(m);
```

```
[det, rank], x b h - x c g - a y h + a z g + d y c - d z b, 3
> restart; map(with, [plots, linalg, LinearAlgebra, plottools]): packages();
[plots, linalg, LinearAlgebra, plottools]
> restart; p := x^3+14*x^2+21: m := matrix(3, 3, [x, y, z, a, b, c, d, g, h]): norm(p, 1),
linalg[norm](m); ⇒ 36, max(|x|+|y|+|z|, |a|+|b|+|c|, |d|+|g|+|h|)
```

В заключение сделаем еще одно довольно существенное замечание. Процедура *with* пакета используется для обеспечения удобного доступа к его модульным средствам, в удобной форме и на интерактивном уровне; *with* – команда, которая обеспечивает функциональные возможности, аналогичные предложению *use* пакета, но работает на интерактивном уровне и применима ко всем пакетным и программным модулям. При этом, предложение *use* работает только в случае с программными модулями.

Процедура *with* эффективна только на верхнем уровне и служит, прежде всего, для интерактивного использования. Поскольку *with* работает, используя специальный лексический предпросмотр, она не работает в телах процедур либо модулей. Между тем, в релизах 6 – 9 формат *with* работает корректно в телах процедур и модулей, но в *Maple 10, 11* она не функционирует корректно в телах *процедур* или *модулей*, выводя соответствующие предупреждения. Между тем, наша процедура *With* [45] устраняет этот недостаток, позволяя корректно использовать вышеупомянутый формат в телах процедур и модулей. В целом ряде случаев это обеспечивает намного более удобное представление алгоритмов в среде *Maple*-языка пакета. Более того, вызов процедуры *With(P, Fo {, F1 {...}})* присваивает именам *{Fo, F1,...}* *protected*-атрибут, отсутствующий для стандартных средств пакета. Следующий фрагмент иллюстрирует сказанное.

```
> With := proc(P::{package, `module`}, F::symbol)
local a, b, c, h;
if nargs = 0 then error "'With' uses at least the 1st argument, which is missing"
else assign(a = interface(warnlevel), b = cat([libname][1], "/_$_Art21_Kr14$_")),
interface(warnlevel = 0)
end if;
if nargs = 1 then
try c := with(args) catch "module `%1` has no exports" : return [] end try;
h := cat(cat(seq(cat("unprotect(`", k, "`):`", k, "`:=eval(`", args[1], "`[`, k, "`"):protect(`",
k, "`):"), k = c))[1 .. -2], ":")
else h := cat(cat(seq(cat("unprotect(`", args[k], "`):`", args[k], "`:=eval(`", args[1], "`[`,
args[k], "`"):protect(`", args[k], "`):"), k = 2 .. nargs))[1 .. -2], ":")
end if;
writeline(b, h), close(b); (proc() read b end proc)(), null(interface(warnlevel = a)),
fremove(b), `if`(1 < nargs, [args[2 .. -1]], c)
end proc;
> With(plots, display, animate); ⇒ [display, animate]
> display := 63; animate := 68;
Error, attempting to assign to `display` which is protected
```



```

Error, attempting to assign to `animate` which is protected
> Kr:=module() export sr, N; N:=() -> [[nargs], args]; sr:=() -> `(args)/nargs end module:
> With(Kr, sr, N);  => [sr, N]
> N(68, 63, 43), sr(68, 63, 43);  => [[3], 68, 63, 43], 58
> M := module () end module: with(M);
Warning, M is not a correctly formed package - option 'package' is missing
Error, (in with) module `M` has no exports
> With(M);  => []

```

При этом, при отсутствии у модуля *M* экспортов вызов *with(M)* вызывает ошибочную ситуацию с диагностикой «*module `%1` has no exports*» для всех версий *Maple*. Поэтому для более удобной работы с экспортами модуля рекомендуется использовать *exports-*функцию, возвращающую *NULL*-значение на модулях без экспортов или процедуру *With*, возвращающую в аналогичной ситуации пустой список, т.е. [].

Использование вызова процедуры *with(P)* для проверки экспортов пакетного модуля *P* не является целесообразным, ибо в таком случае производится загрузка в рабочую область пакета всех его экспортов. Поэтому для этих целей рекомендуется применять процедуру *tpacmod*, имеющую формат вызова следующего вида *tpacmod(P {, Name})*, где *P* – имя пакетного модуля и *Name* – имя экспорта [45]. Вызов процедуры с одним аргументом *P*, в качестве которого допустимо лишь имя пакетного модуля из *Maple*-библиотеки, логически сцепленной с *главной* библиотекой пакета, возвращает список экспортов модуля *P*. Тогда как вызов процедуры *tpacmod(P, Name)* с 2-я аргументами возвращает *true*, если *Name* является экспортом модуля *P*, и *false* в противном случае. При этом, в любом случае экспорты модуля *P* не загружаются в рабочую область и не становятся активными в текущем сеансе. Данная возможность весьма актуальна при работе с пакетными модулями. Следующий фрагмент представляет исходный текст процедуры *tpacmod* и примеры ее использования для проверки экспортов модулей.

```

> tpacmod := proc(P::package)
    parse(cat("`if (belong(Release(), 6 .. 11), `pacman`, `PackageManagement`):- ",
    convert(`if` (nargs = 1, pexports, pmember)(`if` (nargs = 1, args, op([args[2], args[1]]))),
    'string')), 'statement')
end proc:
> tpacmod(stats);  => [anova, describe, fit, importdata, random, statevalf, statplots, transform]
> packages();  => []

```

Из примеров фрагмента следует, что вызовы процедуры *tpacmod(M)* для получения списка экспортов пакетного модуля *M*, или для тестирования имени быть экспортом пакетного модуля не загружают ни экспортов пакетного модуля в рабочую область, ни модуля в целом. Данное средство во многих случаях работы с пакетными модулями оказывается в значительной мере достаточно полезным. Рассмотрев вопросы ведения библиотек пользователя в среде *Maple*, довольно важным представляется их анализ на предмет *эффективности*, ответ на который в значительной степени может прояснить статистический анализ динамики функционирования библиотек.

6.1.13. Статистический анализ библиотек в среде пакета Maple

Создав *собственную Maple-библиотеку* процедур с использованием вышеупомянутых подходов либо иным способом, вполне естественно возникает задача ее *оптимизации*, в частности, с целью раскрытия частоты использования средств, содержащихся в ней, и основных ресурсов компьютера, используемых ими. В данном контексте, проблема оптимизации библиотек пользователя довольно актуальна. Для этих целей довольно полезной представляется процедура **StatLib(Lb)** [45], обеспечивающая сбор основной статистики по заданной **Lb** библиотеке с возвратом статистики для ее последующего анализа. Прежде всего, процедура **StatLib** предполагает, анализируемая библиотека **Lb** находится в каталоге **LIB** главной библиотеки **Maple**. При этом, в процессе своего выполнения процедура **StatLib** требует некоторых дополнительных ресурсов памяти компьютера и времени для выполнения своих функций.

Процедура **StatLib(Lb)** в качестве *первого* обязательного аргумента **Lb** использует имя анализируемой **Maple-библиотеки**, которая располагается в подкаталоге **Lib** главного каталога пакета и которая логически связана с *главной Maple-библиотекой*. Другие ее формальные аргументы (*в количестве до 3*) являются дополнительными и назначение кортежей их значений определяется следующим образом, а именно:

- StatLib(Lb)** – инициация сбора статистики по средствам библиотеки **Lb** в целом;
- StatLib(Lb, 0)** – удаление всех файлов со статистической информацией по анализируемой библиотеке **Lb**;
- StatLib(Lb, 1)** – отмена сбора статистики по библиотеке **Lb** с сохранением в файлах полученных результатов;
- StatLib(Lu, P)** – инициация сбора статистики по процедуре **P** из библиотеки **Lb**;
- StatLib(Lb, T, 2)** – возврат собранной статистики по библиотеке **Lb** или по процедуре **P** в заданном разрезе **T**;
- StatLib(Lb, T, N)** – возврат собранной статистики по заданной процедуре **N** в требуемом разрезе {calls, bytes, depth, maxdepth, time};
- StatLib(Lb,abend)** – завершение процедуры при появлении критических ошибок;
- StatLib(Lb, T, 2, m)** – возврат собранной статистики по библиотеке **Lb** либо по ее процедуре **P** в заданном **T** разрезе, характеризуемом числом **m**.

Вызов процедуры **StatLib(Lb)** с 1 фактическим аргументом **Lb** инициирует процесс по сбору статистики по вызовам составляющих библиотеку процедур. Данный процесс сбора сможет быть прекращен вызовом процедуры **StatLib(L, 1)**, который обязателен при необходимости продолжения сбора статистики в последующих сеансах работы с пакетом, ибо он производит сохранение собранной статистики в 5 файлах, которые размещаются в каталоге с библиотекой **Lb**. Имена таких файлов имеют одинаковый «\$@_»-префикс. Вызов процедуры **StatLib** с нулевым значением второго фактического аргумента возвращает **NULL**-значение с удалением файлов сбора статистики по этой библиотеке. Этот вызов рекомендуется выполнять перед инициацией *нового* процесса профилирования процедур для *сохранения* его результатов в каталоге с библиотекой **Lb**. При этом, существующие статистические файлы с *ранее* собранной информацией

могут быть предварительно сохранены в другом каталоге либо остаться в старом под другими именами, т.е. возможно сохранение истории профилирований библиотеки.

Вызов процедуры *StatLib(Lb, P)* инициирует процесс сбора статистики по вызовам в текущем сеансе процедур, составляющих библиотеку *Lb* и определяемых списком *P* их имен. Между тем, запрашиваемые к профилированию процедуры не обязательно должны принадлежать библиотеке *Lb* – они могут находиться и в любой библиотеке, логически связанной с главной библиотекой пакета, или быть активными в текущем сеансе *Maple*. Иначе инициируется *ошибочная* ситуация с возвратом соответствующей диагностики. Это позволяет производить анализ как всей библиотеки *Lb* в целом, так и в *разрезе* составляющих библиотеку процедур (*не совмещая оба эти процесса*), а также любых активных либо доступных процедур текущего сеанса пакета.

Процедура *StatLib* обеспечивает возврат статистической информации в пяти разрезах, определяемых ключевыми словами $T = \{calls, bytes, depth, maxdepth, time\}$, определяющими соответственно показатели: (1) количество вызовов, (2) размер используемой памяти, (3) глубина и (4) максимальная глубина вложенности совместно с (5) использованным временем в с. Вызов процедуры *StatLib(Lb, T, 2)* возвращает массив характеристик для всех *профилируемых* процедур библиотеки *Lb* {или *процедур, определенных аргументом P при вызове StatLib(L, P)*} на текущий момент в заданном *T*-разрезе (*в частности, calls – количество вызовов процедуры*), тогда как вызов процедуры *StatLib(Lb, T, 2, m)* возвращает массив для *T*-разреза тех *профилируемых* процедур, значения характеристик которых, соответствующие *T*-разрезу, не меньше *m*. В случае отсутствия таких строк у массива процедура возвращает *lack*-значение, информируя о том, что на данный момент ни одна из профилируемых процедур не обладает характеристикой, соответствующей *T* разрезу со значением, не меньшим, чем *m*. Данный массив построчно отсортирован в порядке убывания значений характеристик; более того, для равных значений строки массива сортируются лексикографически.

Наконец, вызов *StatLib(Lb, T, N)* процедуры (*если в качестве 3-го аргумента N процедуры указано значение {symbol, string}-mina*) возвращает значение характеристики процедуры с именем *N* (*если такая существует и ранее профилировалась*) в заданном *T*-разрезе. Так, например, вызов *StatLib(UserLib, calls, Rmdir)* возвращает число вызовов процедуры *Rmdir* библиотеки *UserLib*, если данная библиотека либо процедура предварительно профилировались. На остальных кортежах значений фактических аргументов *NULL*-значение возвращается. В целях обеспечения больших и надежности, и сохранности собираемой статистической информации рекомендуется периодически ее сохранять посредством вызовов *StatLib(Lb, 1)* с последующим возобновлением профилирования требуемых средств вызовом *StatLib(Lb)* процедуры.

Процедура *StatLib* допускает два режима мониторинга результатов профилирования процедур – динамический и разовый. Динамический режим обеспечивает довольно удобную возможность мониторинга посредством вышеописанных вызовов процедуры в процессе профилирования, точнее, между двумя вызовами процедуры *StatLib(Lb)* и *StatLib(Lb, 1)*. При этом, не нарушается процесс профилирования. Тогда как разовый режим дает возможность проводить выборочную проверку результатов предыдущего профилирования без возобновления самого процесса профилирования. Этот подход

позволяет более гибко производить мониторинг процесса профилирования процедур как внутри него, так и вне самого процесса.

В некоторых версиях релизов 6 – 11 пакета использование процедуры *StatLib* может в ряде случаев вызывать критические ошибки, связанные с переполнением пакетного стека, прежде всего. В этом случае рекомендуется выполнить вызов *StatLib(Lb,abend)* с последующим выполнением предложения **restart** пакета. Данный прием позволяет сохранять в указанных статистических файлах информацию по количеству вызовов профилированных процедур на момент аварийной ситуации по крайней мере. Тогда как следующий вызов *StatLib(Lb)* процедуры обеспечивает возобновление процесса профилирования с прерванного момента. Упомянутые ошибочные ситуации вполне могут быть в значительной степени объяснены следующим образом.

Наш достаточно длительный опыт использования *Maple* релизов 4 – 11 в различных приложениях, включая развитие средств, расширяющих основные средства пакета, со всей определенностью выявил одно довольно существенное обстоятельство. Многие из часто используемых стандартных средств были обеспечены *недостаточно* развитой системой обработки особых и ошибочных ситуаций, которые в большинстве случаев завершаются ошибками с очевидно некорректной *диагностикой*, например, «*Execution stopped: Stack limit reached*» с последующим аварийным завершением текущего сеанса. Таким образом, или *Maple* имеет стек недостаточной глубины, или вышеупомянутая ситуация была вызвана (*при отсутствии каких-либо циклических вычислений*) ошибкой, имеющей причину другого характера. К большому сожалению, увеличение номеров релизов пакета *Maple* пока сопровождается лишь уменьшением их робастности, да и не только этого. Надежная работа пакета остается на уровне благих пожеланий.

Реализация алгоритма процедуры *StatLib* существенно базируется на представленных в [45] наших процедурах *DoF, Fremove, Plib, belong, _SL, tabar*, а также на специальных процедурах *profile* и *unprofile* пакета для профилирования вызовов процедур. Отметим, использование процедуры *StatLib* вызывает замедление вычислений при увеличении размера используемой памяти, величина которых зависит, прежде всего, как от числа профилируемых процедур, так и частоты их использования. Но, ввиду относительно небольших библиотек пользователя (*до 200 – 300 процедур и частоте их использования не более 100 за сеанс, исключая циклические конструкции*) это обстоятельство не приводит к критическим ситуациям, связанным с использованием *основных* ресурсов компьютера и времени обработки. Так, использование данного механизма для нашей библиотеки [45] (*более 850 процедур*) не дает каких-либо оснований рассматривать такой механизм профилирования в качестве причины достаточно серьезных дополнительных издержек основных ресурсов ПК – эксперименты проводились на ПК *MicroLink 500 (Pentium 4)* с частотой 3042 Mhz, RAM 1.5 GB и HDD 150 GB в процессе 5 – 6 часов работы.

Механизм использования *StatLib* процедуры сводится к следующему. В самом начале сеанса работы с пакетом выполняется вызов процедуры *StatLib(Lb {, P})*, где *Lb* – имя анализируемой библиотеки пользователя, удовлетворяющей указанным условиям (*P-аргумент может определять список имен процедур из библиотеки Lb либо вне ее*). Тогда как перед завершением текущего сеанса работы выполняется вызов процедуры *StatLib(Lb, 1)*, который обеспечивает сохранение статистической информации в 5 специальных

файлах с именами «\$@#_h» (где $h \in \{depth, calls, bytes, maxdepth, time\}$), помещаемых в **Lb** каталог с библиотекой **Lb**. В произвольный момент (динамически или разово) на основе вызовов процедуры **StatLib(Lb, T, {2 | name} {, m})** можно получать справку по нужным характеристикам вызовов процедур библиотеки **Lb** в указанных выше разрезах.

Наиболее эффективным режимом представляется следующий. Каждый сеанс работы с пакетом начинается вызовом **StatLib(Lb)**, после чего выполняется требуемая работа в среде пакета. Периодически рекомендуется производить пары вызовов **{StatLib(Lb, 1), StatLib(Lb)}** для обеспечения надежности сохранности результатов профилирования процедур библиотеки. Затем перед завершением текущего сеанса выполняется вызов процедуры **StatLib(Lb, 1)**, обеспечивающий прекращение процесса профилирования и сохранение его результатов в упомянутых файлах. Анализ собранной информации дает возможность улучшать и организацию библиотеки в целом, и эффективность ее отдельных составляющих процедур, имеющих высокую частоту использования либо достаточно существенно использующих основные ресурсы компьютера.

Опыт использования процедуры **StatLib** со всей определенностью свидетельствует о ее довольно высокой эффективности при решении вопросов *оптимизации* библиотек пользователя. Методика использования процедуры **StatLib** довольно подробно была рассмотрена в наших предыдущих книгах [10–14, 28–38, 42, 44]. Представленные ниже примеры достаточно наглядно иллюстрируют принципы и результаты применения процедуры **StatLib** для задачи профилирования пользовательских библиотек.

```
> StatLib := proc(L::{string, symbol})
local a, k, h, S, P, K, G, H, t, pf, unp, u, V, T, R, W, Z, calls1, bytes1, depth1, maxdepth1,
time1;
global profile_maxdepth, profile_calls, profile_bytes, profile_depth, profile_time,
`$Art16_Kr9`, calls2, bytes2, depth2, maxdepth2, time2;
unp := () -> unassign(`$Art16_Kr9`, 'profile_proc', op(T), seq(cat(profile_, k), k = R));
`if` (nargs = 2 and args[2] = 'abend', RETURN(unprofile(),
unp(), "Abend! Execute `restart` command!"), NULL);
W := table([1 = true, 2 = `if` (nargs = 2 and type(args[2], {'binary', 'list'({'symbol'}))), true,
false), 3 = `if` (nargs = 3 and member(args[2], {'calls', 'maxdepth', 'depth', 'time', 'bytes'})
and (args[3] = 2 or type(args[3], 'symbol')), true, false), 4 = `if` (nargs = 4
and member(args[2], {'calls', 'maxdepth', 'depth', 'time', 'bytes'})
and args[3] = 2 and type(args[4], 'numeric'), true, false)];
`if` (W[nargs] = true, unassign("W"), ERROR("invalid arguments %1 have been passed to
the StatLib", [args]));
assign(K = Plib(L)), assign(R = ['calls', 'maxdepth', 'depth', 'time', 'bytes']),
assign('calls1' = cat(K, "/$@_", R[2]), 'bytes1' = cat(K, "/$@_", R[1]),
'depth1' = cat(K, "/$@_", R[3]), 'maxdepth1' = cat(K, "/$@_", R[4]),
'time1' = cat(K, "/$@_", R[5])), assign(T = [seq(cat(R[h], `2`), h = 1 .. 5)],
V = ['calls', 'maxdepth', 'depth', 'time', 'bytes'],
Z = [seq(cat(profile_, R[h]), h = 1 .. 5)], G = []);
```

```

`if` (nargs = 2 and args[2] = 0, RETURN(WARNING("datafiles with statistics have been
removed out of directory with library <%1>", L), unprofile(), op(map(Fremove,
[bytes1, calls1, depth1, maxdepth1, time1])), unp()), NULL);
if nargs = 2 and args[2] = 1 then
    `if` (type(eval(profile_calls), 'table'), assign('calls2' = eval(cat(profile_, R[2])),
        'bytes2' = eval(cat(profile_, R[1])), 'depth2' = eval(cat(profile_, R[3])),
        'maxdepth2' = eval(cat(profile_, R[4])), 'time2' = eval(cat(profile_, R[5])),
        ERROR("profiling does not exist"));
    (proc()) save maxdepth2, maxdepth1; save calls2, calls1; save bytes2, bytes1;
        save depth2, depth1; save time2, time1 end proc()), RETURN(unprofile(), unp()),
        WARNING("profiling of library <%1> has been completed", L))
else NULL
end if;
if nargs = 3 and belong(cat(``, args[2]), R) and
    member(whattype(args[3]), {'symbol', 'string'}) then
    assign(`$Art16_Kr9` = eval(cat(profile_, args[2])), pf = cat(K, "/$@", args[2])),
    `if` (type(eval(`$Art16_Kr9`), 'table'), NULL, `if` (DoF(pf) = 'file',
    [(proc()) read pf end proc()], assign(`$Art16_Kr9` = eval(cat(``, args[2], `2`))),
        RETURN("a profiling information does not exist"));
    assign(a = [cat(``, args[2]), `$Art16_Kr9`[cat("", args[3])]],
        RETURN(`if` (type(a[2], 'numeric'), a, "procedure has been not profiled"),
        `if` (type(eval(profile_proc), 'symbol'), unp(), NULL))
else NULL
end if;
if 3 <= nargs and belong(cat(``, args[2]), R) and args[3] = 2 then
    `if` (nargs = 4 and type(args[4], 'numeric'), assign(a = args[4]), assign(a = 0));
    assign(pf = cat(K, "/$@", args[2]), `$Art16_Kr9` = eval(cat(profile_, args[2])),
    `if` (type(eval(`$Art16_Kr9`), 'table'), NULL, `if` (DoF(pf) = 'file',
    [(proc()) read pf end proc()], assign(`$Art16_Kr9` = eval(cat(``, args[2], `2`))),
        RETURN("a profiling information does not exist"));
    RETURN(tabar(`$Art16_Kr9`, 'Procedures', cat(`Procedure's `, args[2]), a),
        `if` (type(eval(profile_proc), 'symbol'), unp(), NULL))
else NULL
end if;
`if` (K <> false, [assign(P = march('list', K)), assign('h' = nops(P))],
    ERROR("library <%1> does not exist or is not linked with the main Maple library", L));
`if` (nargs = 2 and type(args[2], 'list'({'symbol'})), assign(S = args[2]),
    assign(S = [seq(`if` (P[k][1][1 .. 2] <> ":-", cat(``, P[k][1][1 .. -3]), NULL), k = 1 .. h)]));
for k in S do
    try `if` (type(eval(k), 'procedure'), assign('G' = [op(G), k]), NULL)
    catch "" : NULL("Exception handling with program modules")
    end try

```

```

end do;
`if (G = [], ERROR("procedures ordered for profiling do not exist both in library <%1>
and in libraries logically linked with the main Maple library", L,
unprofile(), unp()), NULL);
try if DoF(calls1) = 'file' then null((proc() profile(op(G));
seq((proc(x) read x end proc)(eval(V[h])), h = 1 .. 5);
seq(_SL(Z, eval(T[h]), h), h = 1 .. 5) end proc))
else profile(op(G))
end if
catch "%1 is already being profiled": WARNING("profiling is already being executed!")
end try
end proc:
> restart; StatLib(UserLib);
> Mkdir("C:/Temp/Art/Kr"), type("C:\\Temp/Art/Kr", dir), type("C:/Temp/Art/Kr", file);
"c:\\temp\\art\\kr", true, false
> StatLib(UserLib, calls, 2, 7);


| Procedures | Procedure's calls |
|------------|-------------------|
| Red_n      | 33                |
| Case       | 31                |
| Subs_all1  | 16                |
| Subs_All   | 16                |
| Search     | 10                |
| holdof     | 9                 |
| CF         | 7                 |


> StatLib(UserLib, time, Case); ⇒ [time, 0.0]
> StatLib(UserLib, bytes, StatLib); ⇒ [bytes, 17980]
> seq(StatLib(UserLib, calls, h), h = [belong, tabar, Case, CF]);
[calls, 4], [calls, 1], [calls, 61], [calls, 7]
> StatLib(UserLib, time, 2, 0.05);


| Procedures | Procedure's time |
|------------|------------------|
| Adrive     | 0.157            |
| tabar      | 0.062            |


> StatLib(UserLib, calls, 2, 20);


| Procedures | Procedure's calls |
|------------|-------------------|
| Case       | 76                |
| Red_n      | 33                |
| sub_1      | 20                |


```

> StatLib(UserLib, bytes, 2, 20000);

<i>Procedures</i>	<i>Procedure's bytes</i>
<i>SLj</i>	3591160
<i>tabar</i>	2097700
<i>Red_n</i>	355168
<i>Case</i>	176172
<i>type/nestlist</i>	115256
<i>Search</i>	74416
<i>Adrive</i>	64752
<i>StatLib</i>	60944
<i>Plib</i>	42456
<i>Subs_all1</i>	38856
<i>type/dir</i>	35676
<i>CF</i>	35372
<i>belong</i>	33132
<i>sub_1</i>	28392
<i>type/file</i>	20244

> StatLib(UserLib, 1);

Warning, profiling of library <UserLib> has been completed

[Новый Maple-сеанс:

> StatLib(UserLib); CureLib("C:\\rans\\academy\\libraries\\ArtKr", x, y):

Warning, library file <C:/rans/academy/libraries/ArtKr/Maple.ind> does not exist

Warning, Analysis of contents of library lib-file is being done. Please, wait!

Warning, library contains multiple entries of the following means

[Atr, CCM, CureLib, Currentdir, DAclose, DAopen, DAread, DULib, FSSF, F_atr1, F_atr2, FmF, Imaple, Is_Color, LibElem, LnFile, ModFile, NLP, ParProc1, RTfile, Reduce_T, SDF, SSF, Suffix, Uninstall, Vol, Vol_Free_Space, WD, WS, cdt, conSA, dslib, ewsc, gelist, sfd, mapTab, readdata1, redlt, sorttf, type/dir, type/...];

the sorted nested list of their names with multiplicities appropriate to them is in predefined variable `_mulvertools`

Warning, Analysis of contents of library lib-file has been completed!

> _mulvertools;

[[F_atr2, 6], [Atr, 4], [CureLib, 3], [FSSF, 3], [SDF, 3], [SSF, 3], [Suffix, 3], [conSA, 3], [ewsc, 3], [gelist, 3], [sorttf, 3], [type/file, 3], [CCM, 2], [Currentdir, 2], [DAclose, 2], [DAopen, 2], [DAread, 2], [DULib, 2], [F_atr1, 2], [FmF, 2], [Imaple, 2], [Is_Color, 2], [LibElem, 2], [LnFile, 2], [ModFile, 2], [NLP, 2], [ParProc1, 2], [RTfile, 2], [dslib, 2],


```

[Reduce_T, 2], [Uninstall, 2], [Vol, 2], [Vol_Free_Space, 2], [WD, 2], [WS, 2], [cdt, 2],
[mapTab, 2], [readdata1, 2], [redlt, 2], [sfd, 2], [type/dir, 2]]
> SLj([Vic, 68], [Gal, 63], [Sv, 43], [Arn, 48], [Art, 21], [Kr, 14], 2);
      [[Kr, 14], [Art, 21], [Sv, 43], [Arn, 48], [Gal, 63], [Vic, 68]]
> StatLib(UserLib, calls, 2, 15);
      [Procedures  Procedure's calls]
      [Case         96]
      [Red_n        39]
      [sub_1        26]
      [Search       23]
      [Subs_all1    19]
      [Subs_All     19]
      [belong       15]
> StatLib(UserLib, bytes, 2, 63000);
      [Procedures  Procedure's bytes]
      [SLj         7143544]
      [tabar       4929356]
      [Red_n       407032]
      [type/nestlist 340600]
      [Case        252076]
      [Search      99476]
      [Adrive      84024]
      [StatLib     82572]
      [Plib        67216]
> StatLib(UserLib, 0);
Warning, datafiles with statistics have been removed of directory with library <UserLib>

```

Ввиду ранее сделанных пояснений примеры данного фрагмента весьма прозрачны и не требуют каких-либо дополнительных разъяснений. В частности, представленная процедура *StatLib* была достаточно эффективно использована нами для улучшения функциональных характеристик пользовательской Библиотеки *UserLib6789* [45].

Процедуры, представленные в Библиотеке [45], обеспечивают пользователя набором средств для обработки библиотек пользователя, имеющих структурную организацию, аналогичную главной *Maple*-библиотеке. Данные средства обеспечивают целый ряд функций, упрощающих проблему восстановления поврежденных библиотек. Наряду с этим, они также поддерживают и другие структурные организации, полезные в ряде важных приложений. На сегодня библиотека *UserLib6789* имеет *Freeware*-статус.

6.2.1. Определения процедур в пакете Mathematica и их типы

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 рассмотрим его основные элементы процедурного программирования в среде пакета *Mathematica*.

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[54]:= R[x_] := Module[h := Sin[x] + Cos[x]; h]
In[55]:= R[68]
Module::argr: Module called with 1 argument; 2 arguments are expected. >>
Out[55]= Module[h := Sin[68] + Cos[68]; h]
In[56]:= R[x_] := Module[{}, h := Sin[x] + Cos[x]; h]
In[57]:= R[68]
Out[57]= Cos[68] + Sin[68]
```

Этим определение процедур в *Mathematica* существенно отличается от определения процедур в *Maple*, где произвол представления составляющих их компонент является значительно 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[343]:= GS[x_Integer, y_ : 68, z_Integer] := Module[{a}, a = x + y^2 + z; a]
In[344]:= {GS[42, 2, 47], GS[42, 47]}
Out[344]= {93, 4713}
In[345]:= GS[x_, y_Integer: 68, z_] := Module[{a}, a = x + y^2 + z; a]
In[346]:= {GS[42, 2, 47], GS[42, 47], GS[42, 72, 420, 47]}
Out[346]= {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[372]:= G[x_ : 68, y_ : 63] := x^2 + y^3
In[373]:= {G[], G[14]}
Out[373]= {254671, 250243}
```

т.е., в результате кодирования единственного фактического аргумента при вызове *G*, опускается именно второй формальный аргумент функции.

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[416]:= Options[G] = {Art -> 22, Kr -> 14}
Out[416]= {Art -> 22, Kr -> 14}
In[417]:= G[Z_, OptionsPattern[]] := H[Z, Z + OptionValue[Art] + OptionValue[Kr]]
In[418]:= {G[68], G[68, Art -> 72, Kr -> 420]}
Out[418]= {H[68, 104], H[68, 560]}
```

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 $\langle Type \rangle$
- 3) $X_;/ Test[X]$ – формальный аргумент X допускает *фактическое* значение, на котором $Test[X]$ возвращает значение True.

В случае получения при вызове процедуры *фактического* значения типа, отличного от типа, определенного для соответствующего для него *формального* аргумента, то вызов процедуры возвращается *невывчисленным*, 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
```

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 *невывчисленным*, как это иллюстрирует предыдущий фрагмент. Между тем, в данном вопросе между пакетами *Maple* и *Mathematica* имеет место *определенный* паритет. Отсутствующие в обоих пакетах типы можно средствами пакетов определять как *типы* пользователя с последующим *подключением* их к пакету, правда, *Mathematica* позволяет делать это разово, кодируя определения типов прямо в *заголовке* процедуры, тогда как *Maple* требует определение делать в виде отдельной булевой процедуры либо функции. Однако, по большому счету это несущественно. В то же время, если недопустимый *фактический* аргумент для *Mathematica*-процедуры инициирует возврат ее вызова *невывчисленным*, то для *Maple*-процедуры инициируется ошибочная ситуация с возвратом соответствующей диагностики, давая возможность ее программно обрабатывать 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. Следующий весьма простой пример достаточно наглядно иллюстрирует вышесказанное:

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

```
In[634]:= {Sv[75, 420], Sv[75], Sv[]}
```

```
Out[634]= {495, 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[609]:= Agn[x_ /; Quiet[If[SymbolQ[x], x = 75; True, x; True]],
           y_ /; Quiet[If[SymbolQ[y], y = 2011; True, y; True]],
           z_ /; Quiet[If[SymbolQ[z], z = 420; True, z; True]]] :=
           Module[{}, {x + y + z, Clear["x", "y", "z"]}]][[1]]
In[610]:= {Agn[22, 420, 15], Agn[15, 100, z], Agn[x, 100, z], Agn[x, 100, 200], Agn[x, y, z]}
Out[610]= {457, 535, 595, 375, 2506}
```

Таким образом, пакет *Mathematica* подобно *Maple* (и даже в несколько больших пределах) располагает достаточно широкими возможностями по использованию *нестандартных* приемов программирования. Правда, вопрос лишь в том, насколько они оправданны 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[637]:= x = 15; If[x == 15, agn]
```

```
Out[637]= agn
```

```
In[638]:= x = 22; If[x == 15, avz]
```

```
In[639]:= x = 2; 100*If[x == 15, x^2]
```

```
Out[639]= 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, 17, Null}
```

Отдельно следует рассмотреть вопрос определения процедур с произвольным числом формальных аргументов; если в *Maple* данный вопрос довольно просто решается при использовании процедурных переменных *args*, *nargs* и ряда других, отмеченных выше:

```
> G := () -> [[args], nargs, sum(args[k], k=1..nargs)]: G(), G(42, 47, 67), G(x, y, z);
[[], 0, 0], [[42, 47, 67], 3, 156], [[x, y, z], 3, x + y + z]
```

то в программной среде *Mathematica* ситуация обстоит несколько иначе. Подобными процедурными переменными пакет не располагает. Между тем, некоторые аналоги в этой связи все же могут быть определены, как иллюстрирует следующий фрагмент:

In[607]:= Fargs[x_] := Module[{args = {x}, nargs = Length[{x}], k}, {args, nargs, Sum[args[[k]], {k, 1, nargs}]}]	(A)
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[{##}]}, Sum[args[[k]], {k, 1, nargs}]] &	(C)
In[782]:= {GG[], GG[42, 47, 67, 15, 22], args, nargs}	
Out[782]= {0, 193, args, nargs}	

Между тем, определенные 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 (≥ 1) фактических аргументов; при этом, для указанных переменных использовать простые конструкции *args = {x}, nargs = Length[{x}]* (A). Наиболее простым является определение неопределенного количества формальных аргументов для *чистых* функций (B), тогда как для *блочных* объектов (C) блочные переменные *args* и *nargs* определяются способом, аналогичным способу для процедур, представленному в начале предыдущего фрагмента (A).

Между тем, организация процедурных конструкций в пакетах *Maple* и *Mathematica* 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 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
```

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[168]:= V := 68; G := 63; S := 43; (Module[{}, V/2] + Module[{}, G/3])/(Module[{}, S] + 12)
Out[168]= 1
In[169]:= Sin[(14*Module[{}, V] + 21*Module[{}, G])/(Module[{}, S]^2 + 72)] // N
Out[169]= 0.926228
In[170]:= Module[{}, V]*Y^2 + (Log[Module[{}, G]] // N)*Y + Module[{}, S]^2 + 72
Out[170]= 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.

6.2.2. 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[172]:= 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[173]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[173]= {9088, {68 a$9088 + 63 b$9088 + 43 c$9088, a$9088, b$9088, a$9088$9088}, 9089}
In[174]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[174]= {9090, {68 a$9090 + 63 b$9090 + 43 c$9090, a$9090, b$9090, a$9090$9090}, 9091}
In[175]:= {$ModuleNumber, G[68, 63, 43], $ModuleNumber}
Out[175]= {9092, {68 a$9092 + 63 b$9092 + 43 c$9092, a$9092, b$9092, a$9092$9092}, 9093}
In[176]:= n = 1; While[n < 4, Print[$ModuleNumber]; n++]
9277
9277
9277
In[177]:= {$ModuleNumber, $ModuleNumber}
Out[177]= {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 наглядный фрагмент, а именно:

```
In[430]:= 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[431]:= S[68, 63]
Out[431]= {126588, 959, 959, 959, 959, 959, 959}
In[432]:= g := {a$959, b$959, c$959}
In[433]:= S[68, 63]
Out[433]= {126984, 962, 962, 962, 962, 962, 962}
In[434]:= d := {g, {a$962, b$962, c$962}}
In[435]:= S[68, 63]
Out[435]= {127380, 965, 965, 965, 965, 965, 965}
In[436]:= {d, {a$965, b$965, c$965}}
Out[436]= {{{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*. Следующий простой фрагмент иллюстрирует сказанное достаточно наглядно:

```
In[624]:= Kr[x_, y_] := Module[{a, b}, h := a*x + b*y; {{a, b, h}, h}]
In[625]:= Kr[14, 21]
Out[625]= {{a$846, b$846, 14 a$846 + 21 b$846}, 14 a$846 + 21 b$846}
In[626]:= First[First[%]]^2 + Take[First[%], {2, 2}]^2
Out[626]= {a$846^2 + b$846^2}
In[627]:= Kr[x_, y_] := Module[{a, b}, a = 96; b = 89; h := a*x + b*y;
                                Print[{"a$"} <> ToString[$ModuleNumber - 1],
                                "b$"} <> ToString[$ModuleNumber - 1]];
                                {Symbol["a$"] <> ToString[$ModuleNumber - 1],
                                Symbol["b$"] <> ToString[$ModuleNumber - 1]]}]

In[628]:= Kr[14, 21]
Out[628]= {96, 89}
In[629]:= First[%]^2 + Last[%]^2
Out[629]= 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 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[100]:= G:= 63; {Unique[], Unique[S], Unique[{x, y, z}], Unique["G"]}
```

```
Out[100]= {$18, S$1422, {x$1423, y$1423, z$1423}, G19}
```

```
In[101]:= n = 1; S = {}; While[n < 6, S = Append[S, Unique[G]]; n++]; S
```

```
Out[101]= {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 текст с примером приводится ниже:

```
In[271]:= Un[x_] := Module[{}, If[x === Null, Symbol["$" <> ToString[$ModuleNumber]],  
    If[StringQ[x], Symbol[x <> ToString[$ModuleNumber]], If[Head[x] === Symbol,  
        Symbol[ToString[x] <> "$" <> ToString[$ModuleNumber]], x]]]
```

```
In[272]:= {Un[Null], Un[S], Un["G"], Un[{x, y, z}]}
```

```
Out[272]= {$11399, S$11400, G11401, {x, y, z}}
```

В качестве достаточно полезного упражнения читателю рекомендуется обобщить *Un*-процедуру на случай допустимости в качестве фактического аргумента списка имен и/или строк. В предложенном примере вызов **Un[{x, y, z, ...}]** возвращает {*x, y, z, ...*}. 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[314]:= n = 1; While[n < 5, Unique[{x, y, z}]; n++]
```

```
In[315]:= ?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[426]:= Clear[a]; M[x_] := Module[{a = 14}, h = x + a^2]
```

```
In[427]:= {M[21], a, h}
```

```
Out[427]= {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[201]:= SerNum := ToExpression[ToString[$SessionID] <> ToString[$ModuleNumber]]
```

```
In[202]:= $ModuleNumber = SerNum
```

```
Out[202]= 22951317499871323246755
```

```
In[203]:= $ModuleNumber
```

```
Out[203]= 22951317499871323246756
```

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 имен объектов.

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 весьма простой фрагмент, а именно:

```
In[324]:= 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[325]:= HSR[68, 63]
```

```
Out[325]= {68 a$1033 + 63 b$1033, 68 a$1033 + 63 b$1033}
```

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 большие ограничения, чем пакет *Maple*, 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^a 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]
> A := proc(x, y) local a, b, c, B; B := proc(x, y) local a; a := x^2 + y^2; sqrt(a) end proc;
          a := x*y; b := x + y; B(a, b) end proc;
> A(63, 68); => 18369817^(1/2)

```

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 *последний* пример 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*.

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[324]:= a = 1; b = 2; c = 3; {67*Block[{a = 42, b = 47, c = 67}, (a*x + x^b)/c], {a, b, c}}
Out[324]= {42 x + x^47, {1, 2, 3}}
In[325]:= 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[325]= {14 x + x^21, {1, 2, 3}}
In[326]:= Clear[a, b, c]; a = 1; b = 2; c = 3; {3*Block[{a, b, c}, (a*x + x^b)/c], {a, b, c}}
Out[326]= {x + x^2, {1, 2, 3}}
In[327]:= Clear[a, b, c]; {c*Block[{a, b, c}, (a*x + x^b)/c], {a, b, c}}
Out[327]= {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[71]:= Clear[a]; a = 68; Module[{b, c}, b = a; a = Sin[x] + Cos[x]; c = a; If[a === b, a, a = b; c]]
Out[71]= Cos[x] + Sin[x]
In[72]:= a
Out[72]= 68
In[73]:= Clear[a]; a = 68; Module[{b}, b = a; a = Sin[x] + Cos[x]; First[{a, a = b}]]
Out[73]= Cos[x] + Sin[x]
In[74]:= a
Out[74]= 68
```

При этом, возможны некоторые другие варианты реализации указанного механизма *локализации* глобальных переменных, используемого блочными структурами. Тут же уместно отметить, механизм локализации переменных, используемый процедурами *Maple*, существенно более универсален, не требуя использования структур, подобных блочным структурам пакета *Mathematica*.

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 процесса в целом. Эта возможность может быть использована довольно эффективно.

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

```
In[410]:= 14*x + 21*y
Out[410]= 14 x + 21 y
In[411]:= {Block[{x = a + 68, y = b + 63}, Expand[%]], %}
Out[411]= {2275 + 14 a + 21 b, 14 x + 21 y}
In[412]:= Avg = 63
Out[412]= 63
In[413]:= {Module[{Avg}, Avg + 14], Avg}
Out[413]= {14 + Avg$832, 63}
In[414]:= {Block[{Avg}, Avg + 14], Avg}
Out[414]= {77, 63}
In[415]:= Clear[a, b, h]; h := a^2 + b^3; a = 21
Out[415]= 21
In[416]:= {Block[{a = 72}, h + 14], h, a}
Out[416]= {5198 + b^3, 441 + b^3, 21}
In[417]:= Clear[Avz]; {Block[{Avz = 42}, Avz + 68], Avz}
Out[417]= {110, Avz}
```

Приведенные примеры достаточно просты и прозрачны, и с учетом вышесказанного не требуют каких-либо особых пояснений.

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[274]:= n := 68; {Sum[n^2, {n, 14}], Product[n, {n, 21}], n}
Out[274]= {1015, 51090942171709440000, 68}
```

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, например:

```
In[842]:= G[x_, y_] := Sin[x] + h*Log[y]
In[843]:= 1/G[Pi/4, 1]^2
```

Out[843]= 2

In[844]:= S[x_] := Block[{y = a, h = b}, G[Pi/2, y]]

In[845]:= S[10]

Out[845]= 1 + b Log[a]

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[846]:= a := 14; {Block[{a = 21}, a], a}

Out[846]= {21, 14}

In[847]:= a := 14; {Block[{a}, a := 21; a], a}

Out[847]= {21, 14}

In[848]:= a := 14; {Block[{a}, a], a}

Out[848]= {14, 14}

In[849]:= a := 14; {Block[{a}, a^2], a}

Out[849]= {196, 14}

In[850]:= Clear[a]; {Block[{a}, a^2], a}

Out[850]= {a^2, a}

In[851]:= Clear[a]; {Block[{a = 72}, a^2], a}

Out[851]= {5184, a}

In[852]:= Clear[a]; {Block[{a = 420}, a], a}

Out[852]= {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* организации. Нижеследующий фрагмент иллюстрирует как простые примеры процедур на основе обеих организаций, так и процедуру **ProcQ[x]**, обеспечивающую тестирование x -объектов пакета на предмет их процедурности.

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, 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[SameQ[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[SameQ[ToExpression[c], x], True, False]]; d || h
In[347]:= {ProcQ[Kr], ProcQ[Art], ProcQ[D], ProcQ[Sin], ProcQ[Block], ProcQ[Module]}
Out[347]= {True, True, False, False, False, False}

```

Перейдем к рассмотрению механизмов *глобальных* переменных и *возврата* результатов процедурами (*модулями*) пакета *Mathematica* относительно аналогичного в *Maple*.

Между тем, отождествление процедурных объектов на основе *Module* и *Block* следует проводить весьма осмотрительно, т.к. в общем случае они не являются эквивалентны, о чем свидетельствует следующий достаточно прозрачный фрагмент, а именно:

```

In[441]:= 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[442]:= 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[443]:= Subs[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[443]= (b^2 + c)/Sqrt[z]
In[444]:= Subs1[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[444]= ("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[73]:= 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[74]:= Map4[F, {a, b, c, d, h}, x]
Out[74]= {"F[F,x]", "F[x,x]", "F[{F[F,x], F[x,x]},x]", "F[d,x]", "F[h,x]"}
In[75]:= 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[75]= {$Failed, F[x, x], $Failed, F[d, x], F[h, x]}
In[76]:= 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[77]:= Map4[F, {a, b, c, d, h}, x]
Out[77]= {F[a, x], F[b, x], F[c, x], F[d, x], F[h, x]}
In[78]:= 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[79]:= Map4[F, {a, b, c, d, h}, x]
Out[79]= {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*₁, *x*], *F*[*a*₂, *x*], *F*[*a*₃, *x*], ...}, где *L*={*a*₁, *a*₂, *a*₃, ...}. 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**.

Функции пакета *Mathematica* располагают рядом интересных средств для поддержки работы с динамическими объектами. Так, динамический модуль **DynamicModule**[{*x*, *y*, ...}, *W*] представляет собой объект, который поддерживает тот же самый локальный статус для переменных *x*, *y*, ... в процессе вычисления всех динамических объектов из тела *W*. Указанные в **DynamicModule** переменные по умолчанию имеют значения на протяжении всего текущего сеанса с пакетом. При этом, динамический объект может выступать не только непосредственно в качестве выражения, но также, в частности, в качестве координаты в графическом примитиве, объекта типа «ползунок», в качестве установки для опции. Между тем, в отличие от стандартного модуля **DynamicModule** непосредственно не дает возможности получать его определение ни по стандартной функции **Definition**, ни по нашей процедуре **DefFunc**, представленной ниже. 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, для иллюстрации реализуя 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["2011"]
Definition::notfound: Symbol 2011 not found. >>
Out[746]=
In[747]:= {DefFunc["2011"], DefFunc1["2011"], DefFunc2["2011"]}
Definition::notfound: Symbol 2011 not found. >>
Out[747]= { , DefFunc1["2011"], DefFunc2["2011"]}
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["2011", w]}
Out[793]= {True, True, False}
In[794]:= {t, g, w}
Out[794]= {"DynamicModule", "Module", w}

```

В случае некорректного аргумента x вызовы процедур завершаются *невычисленными*, 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.

```
In[818]:= ProcQ1[x_ /; StringQ[x], t_ /; ! HowAct[t]] := Module[{b = " := ", c = DefFunc2[x],
    d, a = Quiet[ToString[Args[x]]], p, x1 = "Block[{" , 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[819]:= {ProcQ1["G", y1], ProcQ1["ProcQ", y2], ProcQ1["Dm", y3], ProcQ1["BB", y4]}
Out[819]= {True, True, True, True}
In[820]:= {y1, y2, y3, y4}
Out[820]= {"Others", "Module", "DynamicModule", "Block"}
In[821]:= H = 2011; V := 69; {ProcQ1["75", x1], ProcQ1["H", x2], ProcQ1["V", x3]}
Out[821]= {False, False, False}
In[822]:= {x1, x2, x3}
Out[822]= {"Others", "Others", "Others"}
```

Вызов **ProcQ1**[*x*, *t*] возвращает True, если *x* есть **Block**, **Module** либо **DynamicModule**, и "Others" либо False в противном случае; через аргумент *t* возвращается тип объекта *x*.

6.2.3. 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 «:=», то в *Mathematica*-процедуре глобальными полагаются все переменные, явно не определенные локальными. Таким образом, *Maple* имеет более надежный механизм защиты глобальных переменных от их модификации; точнее, от пользователей требуется больше внимательности при программировании своих алгоритмов в формате процедур и автоматически переменные, которым в теле процедуры присваиваются значения по оператору присваивания «:=», присваивается статус локальных переменных с выводом соответствующего сообщения, что позволяет пользователю эффективно определиться со статусом переменных, используемых его процедурой. Следующий фрагмент весьма наглядно иллюстрирует сказанное.

```
> Sv := proc(x::anything) y := 68; assign(z = 63); [y, z] end proc;
Warning, `y` is implicitly declared local to procedure `Sv`
> y, z := 42, 47; [Sv(2010), y, z];
Error, (in assign) invalid arguments
```

```
> Sv := proc(x::anything) y := 68; assign('z' = 63); [y, z] end proc:
```

Warning, `y` is implicitly declared local to procedure `Sv`

```
> y, z := 42, 47: [Sv(2010), y, z]; => [[68, 63], 42, 63]
```

```
In[321]:= Sv[x_] := Module[{}, y := 68; z := 63; {y, z}]
```

```
In[322]:= {y, z} = {42, 47}; {Sv[2010], y, z}
```

```
Out[322]= {{68, 63}, 68, 63}
```

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 (модули) *Mathematica*, представляется нам несколько более низким.

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 простой фрагмент, а именно:

```
In[651]:= Art[x_;/ If[! IntegerQ[x], h = 75; True, h = 420; True], y_] :=
```

```
Module[{a = 100}, x + y + h + a]
```

```
In[652]:= {Art[75, 420], Art[15.22, 420]}
```

```
Out[652]= {1015, 610.22}
```

```
> Vsv := proc(x::if `(type(x, integer), [assign('h' = 75), integer][2], [assign('h' = 420), integer][2]), y) x+y+h end proc:
```

```
> Vsv(100, 200);
```

Error, invalid input: Vsv expects its 1st argument, x, to be of type if(type(x, integer), ...

```
In[711]:= Kr[x_, y_] := Module[{a = If[IntegerQ[x], 75, 420]}, x + y + a]
```

```
In[712]:= {Kr[7.5, 420], Kr[100, 200]}
```

```
Out[712]= {847.5, 375}
```

```
In[713]:= Sv[x_, y_] := Module[{a = If[IntegerQ[x] && PrimeQ[y], 75, 420]}, x + y + a]
```

```
In[714]:= {Sv[75, 420], Kr[15, 127]}
```

```
Out[714]= {915, 217}
```

```
> VG := proc(x, y) local a::if `(x + y > 100, integer, float); a := 4.5; x + y + a end proc:
```

```
> [VG(100, 200), VG(15, 22)]; => [304.5, 41.5]
```

```
> kernelopts(assertlevel = 2): VG(100, 200);
```

Error, (in VG) assertion failed in assignment, expected if(100 < x+y, integer, float), got 4.5

```
> A:=proc(x, y::integer:=75, z:=100) x+y+z end proc: A(75,420,100), A(100,200); => 595, 400
```

```
> A := proc(x, y::integer:=75, z) x + y + z end proc:
```

Error, required parameter `z` cannot appear after optional or expected parameters in ...

```
In[888]:= H[x_: 75, y_, z_] := Module[{}, x + y + z]
```

```
In[889]:= {H[100, 200, 300], H[100, 200], H[100]}
```

```
Out[889]= {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. 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. С другой стороны, и *Maple* позволяет задавать *типы* для локальных переменных процедур в зависимости от значений фактических аргументов, получаемых процедурами в момент их вызова. Правда, данный механизм активизируется только после соответствующей установки *kernelopts(assertlevel=2)*. Во фрагменте этот момент проиллюстрирован процедурой *VG*. Наконец, если *Maple* не допускает присвоения значений по умолчанию промежуточным аргументам, то пакет *Mathematica* допускает, расширяя возможность определения процедур. Детально этот механизм здесь не обсуждается, но последний пример дает некоторую иллюстрацию.

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 нашей Библиотеки [45] для *Maple* был использован именно подобный подход для возврата результатов через глобальные переменные. В целом ряде случаев данный подход оказывается достаточно эффективным.

С возвратом результатов процедурами *Maple* и *Mathematica*, как нам представляется, предпочтение также следует отдать *первому* средству. В *Maple* процедура может *одним* или *несколькими* способами одновременно возвращать результаты своего выполнения

- (1) *через результат выполнения последнего предложения процедуры;*
- (2) *через глобальные переменные процедуры;*
- (3) *через фактические evaln-аргументы процедуры;*
- (4) *через фактические optional-аргументы процедуры;*
- (5) *через return-предложение процедуры.*

Следующий простой пример иллюстрирует все 5 способов возврата одновременно:

```
> restart; Art := proc(x, y::evaln) global a; a := 6; assign('z' = 68); y := 63; if nargs > 2 then  
  assign(args[3] = 420) else return x end if; x^2 end proc: {Art(72, h, w), a, h, z, w}, Art(72,v);  
                    {5184, a, z, h, w}, 72
```

В отличие от процедур *Maple*, механизм возврата результатов выполнения процедур пакета *Mathematica* более ограничен и включает только *три* возможности, а именно:

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

Следующий простой фрагмент иллюстрирует первые 3 перечисленные возможности

в одной *Mathematica*-процедуре, а именно:

```
In[1]:= GS[x_, y_] := Module[{a}, a := 68; b := 63; If[x + y < a, Return[x^2 + y^2], c = x*y]; a]
In[2]:= {{GS[14, 21], b, c}, {GS[42, 47], b, c}}
Out[2]= {{637, 63, c}, {68, 63, 1974}}
In[3]:= 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[4]:= {{AK[47], b, c}, {AK[72], b, c}}
Out[4]= {{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. Первый пример простой процедурой иллюстрирует стандартный способ присвоения (по оператору присвоения `:=`) результата необязательному второму аргументу; однако, такой способ оказывается *недопустимым* по причине, хорошо усматриваемой из ниже приведенной диагностики. Для устранения данной причины возможно использовать стандартную процедуру *assign*, как это иллюстрирует *второй* пример фрагмента. В то же время такой подход корректен только при условии, что 2-й фактический аргумент удовлетворяет условию *type(eval(args[2]), symbol) => true*.

Между тем, имеется возможность избежать требования данного условия; 3-й пример фрагмента достаточно наглядно иллюстрирует такую возможность.

```
> A := proc(x::anything) if nargs = 2 and type(eval(args[2]), symbol) then args[2] := x^3
                                end if; x^2 end proc;

Warning, `args` is implicitly declared local to procedure `A`
Error, cannot use reserved name `args` as a local variable in procedure A
> A := proc(x::anything) if nargs = 2 and type(eval(args[2]), symbol) then
                                assign(args[2] = x^4) end if; x^2 end proc:
> A(75), [A(75, t), t];  => 5625, [5625, 31640625]
> A := proc(x::anything) local a; if nargs = 2 then a := nvalue(args[2]);
                                if not a = {} then unassign(a[1]); assign(a[1] = x^4) else assign(args[2] = x^4)
                                end if end if; x^2 end proc:
> t := 75: A(75), [A(75, t), t];  => 5625, [5625, 31640625]
> h := 75: A(75), [A(75, h), h];  => 5625, [5625, 31640625]
> b := 88: A(75), [A(75, b), b];  => 5625, [5625, 31640625]
In[755]:= Agn[x_] := Module[{a = {x}}, If[Length[a] == 2, If[! HowAct[a[[2]]], a[[2]] =
                                a[[1]]^4, Null], Null]; a[[1]]^2]
In[756]:= {Agn[75], {Agn[75, t], t}}
Out[756]= {5625, {5625, t}}
In[757]:= 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[758]:= {Agn[75], {Agn[75, t], t}}
Out[758]= {5625, {5625, 31640625}}
```

В среде *Mathematica* для возможности возвращения результата через необязательный *k*-й фактический аргумент, которому не присваивалось значений, вполне достаточно применять конструкцию следующего вида (*{x}* – список фактических аргументов):

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

Как достаточно наглядно иллюстрируют два последних примера фрагмента.

Здесь еще раз уместно напомнить следующий весьма существенный момент. В книге приводятся примеры процедур и функций, созданных как в среде пакета *Maple*, так и в среде *Mathematica* в процессе подготовки настоящей книги и собранных в пакете «AVZ_Package.nb» [90] и Библиотеке [45], свободно распространяемых. При этом, ряд примеров, приведенных в иллюстрационных целях, содержат *оригинальные* средства, разработанные нами в иных целях, требуя для их корректного выполнения загрузки и инсталляции указанных средств в зависимости от используемого пакета, а также его версии. Напомним, что гарантированно Библиотека [45] работает корректно с *Maple* версий 6–11 и пакет "AVZ_Package.nb" [90] с *Mathematica* версий 7–8. При этом, целый ряд средств из этих источников вполне корректны и в последующих версиях пакетов, являясь релизо-независимыми с достаточно пролонгированным сроком действия.

6.2.4. Некоторые примеры оформления Mathematica-процедур

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. Между тем, целый ряд примеров *Mathematica*-процедур различного назначения был представлен также в других разделах книги. Здесь рассматриваются лишь процедуры т.н. «системного» характера, которые, однако, представляют самый непосредственный прикладной интерес в программировании различных приложений. Более того, здесь представлены такие процедуры, которые имеют обозримый объем для возможности относительно несложного анализа. Их анализ может послужить достаточно полезным упражнением читателю, начинающему программирование в среде *Mathematica*.

В качестве первого примера приведем процедуру **Uprocs**, как достаточно полезную в практическом отношении, так и иллюстрирующую подход к некоторому *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[1]:= Gs[x_] := Block[{a, b, c}, Evaluate[(a*x + x^b)/c]]
In[2]:= S[x_] := Block[{y = a, h = b}, G[Pi/2, y*x]]
In[3]:= S[x_] := Module[{y = a, h = b}, G[Pi/2, y*x]]
In[4]:= S[x_, y_] := Block[{z = a, h = b}, G[Pi/2, (y*x)/z]]
In[5]:= Bl[y_] := Block[{h = z}, G[Pi/2, y]]
In[6]:= MM[x_, y_] := Module[{}, x + y]
In[7]:= BlockQ[x_// Definition] := Module[{a},
  If[Evaluate[x] == ToExpression[StringTake[ToString[Definition[x]],
    {1, First[First[StringPosition[ToString[Definition[x]], " := Block["] - 1]}]], True, False]]
In[8]:= Sv[x_] := Module[{}, y := 68; z := 63; {y, z}]
In[9]:= Kr[x_] := Block[{y = a, h = b}, (y^2 + h^2)*x]; Art[x_] := Module[{a, a = 68; x + a]
In[10]:= 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[SameQ[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[SameQ[ToExpression[c], x], True, False]]; d | | h]
```

```

In[11]:= Map[ProcQ, {Gs, S, Bl, MM, BlockQ, Sv, Kr, 2010, "420", V72, Sin}]
Out[11]= {True, True, True, True, True, True, True, False, False, False, False}
In[12]:= GS[x_, y_] := Module[{a}, a := 68; b := 63; If[x + y < a, Return[x^2 + y^2], c = x*y]; a]
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]:= Clear[a, b, c]; b = 420; c = 72; AK[x_] := Block[{a}, a := 68; b := 63; c = 42*x; b + c]
In[15]:= a := 68; b := 56; P[] := Module[{a, b, c}, ToString[Definition[P]]]
In[16]:= 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[17]:= {StringEnd["123456", "3456"], StringEnd["123456", "3476"], StringEnd["123456", ""]}
Out[17]= {True, False, True}
In[18]:= Uprocs[] := Module[{a, b, c, d, h, g, k}, a := "_$Art21_Kr14$.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 = 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[19]:= Uprocs[]
Out[19]= {AK, Art, Bl, BlockQ, Gs, GS, Kr, MM, P, ProcQ, S, S, StringEnd, Sv, Uprocs, ProcName}
In[20]:= ProcName[] := Module[{}, Names["*" <> ToString[$ModuleNumber]]]
In[21]:= Uprocs[] := Module[{a, b, c, d, h, g, k, t1, t2}, a := "_$Art21_Kr14$.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 = Symbol[StringTake[d, First[First[StringPosition[d, "["]] - 1]]];
      t1 = If[StringCount[d, " := Module[" != 0, Module, Block];
      t2 = StringTake[d, Last[First[StringPosition[d, "]" ]]]];
      If[ProcQ[h], g = Append[g, {h, t2, t1}], Null]; k = k + 1]; DeleteFile[a]; Return[g]
In[22]:= Uprocs[]
Out[22]= {{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.

Вызов достаточно простой процедуры **StringEnd[x, y]**, представленной в предыдущем фрагменте, возвращает значение **True**, если только подстрока **y** завершает строку **x**, и значение **False** в противном случае. Более того, в случае вызова процедуры **StringEnd** на *недопустимых* фактических аргументах вызов возвращается невычисленным. Ниже представлена процедура **UprocQ[x]**, чей вызов возвращает значение **False**, если объект **x** не является *Mathematica*-процедурой, и *двухэлементный* список в противном случае; в данном случае его *первым* элементом возвращается значение **True**, тогда как *вторым* возвращается тип процедуры {Module | Block}. Фрагмент представляет исходный текст процедуры и результат ее применения к процедурам предыдущего фрагмента.

```
In[142]:= UprocQ[x_] := Module[{a, b, c}, If[ProcQ[x], a := ToString[Definition[x]];
  {b, c} = {StringPosition[a, " := Module[{"], StringPosition[a, " := Block[{"]};
  {b, c} = {Quiet[Check[First[Flatten[First[b]]], 0]], Quiet[Check[First[Flatten[First[c]]], 0]]};
  {True, If[c == 0, Module, If[b == 0, Block, If[b < c, Module, Block]]]}, False]]
In[143]:= Map[UprocQ, {ProcQ, StringEnd, Gs, Bl, Sv, Kr, UprocQ, Uprocs, Avz72}]
Out[143]= {{True, Module}, {True, Module}, {True, Block}, {True, Block}, {True, Module},
  {True, Block}, {True, Module}, {True, Module}, False}
```

Следующая процедура **Locals[x]** возвращает значение **False**, если объект **x** не является *Mathematica*-процедурой, иначе возвращается список локальных переменных в виде строки. Фрагмент представляет исходный текст процедуры и результат применения к некоторым процедурам двух предыдущих фрагментов.

```
In[125]:= 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[126]:= Map[Locals, {UprocQ, StringEnd, Uprocs, Kr, Gs, Art}]
Out[126]= {"{a, b, c, d, h}", "{a, b}", "{a, b, c, d, h, g, k, t1, t2}", "{y = a, h = b}", "{a, b, c}", "{a}"}
```

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 текст процедуры и пример ее применения.

```
In[647]:= Spos[x_String, y_String, p_Integer, dir_Integer] := Module[{a, b, c},
  If[StringLength[y] > 1 || dir != 0 && dir != 1, Return[False], b = StringLength[x]];
  If[p < 1 || p > b, False, If[p === 1 && dir === 0, c = 0, If[p === b && dir === 1, c = 0,
  If[dir === 0, For[a = p, a >= 1, a -- 1, If[StringTake[x, {a}] === y, Return[a], c]],
  For[a = p, a <= b, a += 1, If[StringTake[x, {a}] === y, Return[a], c]]]]];
  If[a === 0 || a === b + 1, 0, a]]
In[648]:= 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[648]= {7, 10, 12, 3, 0, 0}
```

Следующий фрагмент иллюстрирует достаточно простую процедуру **SortNL**[*L*, *p*, *b*], производящую сортировку *числового* списка *ListList*-типа по элементам в *p*-позиции его подписков. Сортировка *L* производится по убыванию (*Greater*) либо возрастанию (*Less*) значений элементов в *p*-позиции подписков. Основным интерес представляет в этом фрагменте программная обработка всех основных *особых* и *ошибочных* ситуаций.

```
In[115]:= SortNL[L_List, p_Integer, 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[116]:= L := {{42, 68, 420, 72}, {47, 63, 14, 21}, {67, 43, 21, 14}, {62, 48, 14, 21}}
In[117]:= SortNL[L, 2, Less]
Out[117]= {{67, 43, 21, 14}, {62, 48, 14, 21}, {47, 63, 14, 21}, {42, 68, 420, 72}}
In[118]:= SortNL[L, 2, Greater]
Out[118]= {{42, 68, 420, 72}, {47, 63, 14, 21}, {62, 48, 14, 21}, {67, 43, 21, 14}}
In[119]:= SortNL[L, 12, Less]
SortNL::incorrect second argument, should lay in interval {1, 4}
In[120]:= SortNL[{L, h}, 2, Less]
SortNL::incorrect first argument, should be listlist, but has received
  {{{42, 68, 420, 72}, {47, 63, 14, 21}, {67, 43, 21, 14}, {62, 48, 14, 21}}, Less}}
In[121]:= SortNL[L, 2, Art_Kr]
SortNL::incorrect third argument, should be {Less, Greater}, but has received {Art_Kr}
In[122]:= SortNL[{{a, b, c}, {68, 72, 420}, {63, 43, 14}}, 2, Greater]
SortNL::incorrect first argument, should be numerical listlist, but has received
  {{{a, b, c}, {68, 72, 420}, {63, 43, 14}}}
```

```

In[123]:= SortLpos[L_List, n_Integer, SF_] := If[1 <= n <= Min[Map[Length, L]], $Failed,
      If[! MemberQ[{Greater, Less}, SF], $Failed, Sort[L, SF[#2[[n]], #1[[n]]] &]]]
In[124]:= L = {{63, 68, 43}, {14, 22, 48}, {42, 47, 67}, {96, 72, 420}}; L1 = {{42, 47}, {14, 22, 72}}
In[125]:= SortLpos[L, 2, Greater]
Out[125]= {{14, 22, 48}, {42, 47, 67}, {63, 68, 43}, {96, 72, 420}}
In[126]:= SortLpos[L, 2, Less]
Out[126]= {{96, 72, 420}, {63, 68, 43}, {42, 47, 67}, {14, 22, 48}}
In[127]:= {SortLpos[L, 2, GreaterEqual], SortLpos[L, 7, Less], SortLpos[L1, 2, Greater]}
Out[127]= {$Failed, $Failed, {{14, 22, 72}, {42, 47}}}

```

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.

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* ≡ {*y*₁, *y*₂, ..., *y*_{*n*}} и *z* ≡ {*z*₁, *z*₂, ..., *z*_{*n*}}, определяя множество подстановок {*y*₁ → *z*₁, *y*₂ → *z*₂, ..., *y*_{*n*} → *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[68]:= 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[69]:= (c + x^2)/x^2 /. x^2 -> a
Out[69]= (a + c)/x^2

```

```

In[70]:= Subs[(c + x^2)/x^2, x^2, a]
Out[70]= (a + c)/a
In[71]:= (c + b^2)/x^2 /. x^2 -> Sqrt[z]
Out[71]= (b^2 + c)/x^2
In[72]:= Subs[(c + b^2)/x^2, x^2, Sqrt[z]]
Out[72]= (b^2 + c)/Sqrt[z]
In[74]:= (a + x^2)/(b + a/x^2) /. x^2 -> Sqrt[a + b]
Out[74]= (a + Sqrt[a + b])/(b + a/x^2)
In[75]:= Subs[(a + x^2)/(b + a/x^2), x^2, Sqrt[a + b]]
Out[75]= (a + Sqrt[a + b])/(b + a/Sqrt[a + b])
In[76]:= (a + x^2)/(b + 1/x^2) /. x^2 -> Sqrt[a + b]
Out[76]= (a + Sqrt[a + b])/(b + 1/x^2)
In[77]:= Subs[(a + x^2)/(b + 1/x^2), x^2, Sqrt[a + b]]
Out[77]= (a + Sqrt[a + b])/(b + 1/Sqrt[a + b])
In[78]:= Replace[1/x^2 + 1/y^3, {{x^2 -> a + b}, {y^3 -> c + d}}]
Out[78]= {1/x^2 + 1/y^3, 1/x^2 + 1/y^3}
In[79]:= Subs[1/x^2 + 1/y^3, {x^2, y^3}, {a + b, c + d}]
Out[79]= 1/(a + b) + 1/(c + d)
In[80]:= Replace[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {{x^2 -> a + b}, {y^3 -> c + d}}]
Out[80]= {Sqrt[Cos[1/y^3] + Sin[1/x^2]], Sqrt[Cos[1/y^3] + Sin[1/x^2]]}
In[81]:= Subs[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {x^2, y^3}, {a + b, c + d}]
Out[81]= Sqrt[Cos[1/(c + d)] + Sin[1/(a + b)]]
In[82]:= With[{x = a + c, y = b}, Module[{}, x^2 + y]]
Out[82]= b + (a + c)^2
In[83]:= 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[83]= With[{x^2 = a + c, y = b}, Module[{}, x^2 + y]]
In[84]:= Subs[Module[{}, x^2 + y], {x, y}, {a + c, b}]
Out[84]= b + (a + c)^2
In[85]:= Subs[Module[{}, x^2 + y], {x^2, y}, {a + c, b}]
Out[85]= a + b + c
In[86]:= Replace[(a + x^2/y^3)/(b + y^3/x^2), {{y^3 -> m}, {x^2 -> n}}]
Out[86]= {(a + x^2/y^3)/(b + y^3/x^2), (a + x^2/y^3)/(b + y^3/x^2)}
In[87]:= Subs[(a + x^2/y^3)/(b + y^3/x^2), {y^3, x^2}, {m, n}]
Out[87]= n(a m + n)/(m(m + b n))
In[88]:= Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
      Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]
In[89]:= Df[(a + x^2)/(b + a/x^2), x^2]
Out[89]= (a^2 + 2 a x^2 + b x^4)/(a + b x^2)^2
In[90]:= Df[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]
Out[90]= (-4 x - 2 Sqrt[y] + y)/((2 + Sqrt[y])^3 y^(3/2))

```

```

In[81]:= D[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]
General::ivar: Sqrt[y] is not a valid variable. >>
Out[81]=  $\frac{\partial}{\partial y} \left( \frac{x + \sqrt{y}}{(y + 2\sqrt{y})^2} \right)$ 
In[82]:= Df[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[82]=  $\frac{(d + 2\sqrt{x})/\sqrt{a + \sqrt{x}} - 8(\sqrt{a + \sqrt{x}} + x)/(2(d + 2\sqrt{x})^3)}{1}$ 
In[83]:= Df[(x + Sqrt[x + b])/(d + 2*Sqrt[x + b])^2, Sqrt[x + b]]
Out[83]=  $\frac{(d - 2(2x + \sqrt{b + x}))}{(d + 2\sqrt{b + x})^3}$ 
In[84]:= 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[85]:= {ReplaceAll[c/x^2 + x^2, x^2 -> t], ReplaceAll[(1 + c/x^2)/(b + x^2), x^2 -> t]}
Out[85]= {t + c/x^2, (1 + c/x^2)/(b + t)}
In[86]:= {ReplaceAll1[c/x^2 + x^2, x^2, a+b], ReplaceAll1[(1 + c/x^2)/(b + x^2), x^2, c+d]}
Out[86]= {a + b + c/(a + b), (1 + c/(c + d))/(b + c + d)}
In[87]:= Df1[x_, y_] := Module[{a, b, c = "$Sart22$Kr14$$"}, 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[88]:= Df2[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
Simplify[ReplaceAll1[D[ReplaceAll1[x, y, a], a], a, y]]]]
In[89]:= Df1[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[89]=  $\frac{(d + 2\sqrt{x})/\sqrt{a + \sqrt{x}} - 8(\sqrt{a + \sqrt{x}} + x)/(2(d + 2\sqrt{x})^3)}{1}$ 
In[90]:= Df2[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]
Out[90]=  $\frac{(d + 2\sqrt{x})/\sqrt{a + \sqrt{x}} - 8(\sqrt{a + \sqrt{x}} + x)/(2(d + 2\sqrt{x})^3)}{1}$ 
In[91]:= Df2[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[91]=  $-\frac{((a - c)x^2)}{(1 + c)^2}$ 
In[92]:= Df1[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[92]=  $-\frac{((a - c)x^2)}{(1 + c)^2}$ 
In[93]:= Df[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]
Out[93]=  $-\frac{(2(a - c)x^6)}{(1 + cx^4)^2}$ 
In[94]:= Df2[(a + b)/(Sin[x] + Cos[x]), Sin[x] + Cos[x]]
Out[94]=  $-\frac{(a + b)}{(\cos[x] + \sin[x])^2}$ 
In[95]:= Df2[Cos[x]/(Sin[x] + Cos[x]), Cos[x]]
Out[95]=  $\frac{\sin[x]}{(\cos[x] + \sin[x])^2}$ 

```

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. Наряду с этим, они в ряде ситуаций оказываются более полезны, чем процедура **Df**, о чем свидетельствует приведенный пример. Вместе с тем, все три процедуры достаточно существенно расширяют стандартную функцию **D**. Данный фрагмент представляет исходный текст отмеченных процедур и некоторые примеры их применения там, где они превосходят функции **D**, **ReplaceAll**, **Rule** и **With**.

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 тексты с примерами их применения представляет следующий фрагмент, а именно:

```
In[1]:= Int[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y],
                                Simplify[Subs[Integrate[Subs[x, y, a], a], a, y]]]
In[2]:= Int1[x_, y_] := Module[{a}, If[! HowAct[y], Integrate[x, y],
                                Simplify[ReplaceAll1[Integrate[ReplaceAll1[x, y, a], a], a, y]]]
In[3]:= {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[3]= {-Cos[a + 1/x^2] + c Log[1/x^2], c/x^4 - Cos[a + 1/x^2]}
In[4]:= {Int[Sin[n/x^2] + m/x^2, x^2], Int1[Sin[n/x^2] + m/x^2, x^2]}
Out[4]= {-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[5]:= {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[5]= {(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])}
```

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**.

Наконец, следующий фрагмент представляет процедуру, вызов **NamesProc[]** которой возвращает отсортированный список *имен* процедур пользователя, активированных в текущем сеансе. В ряде случаев процедура может оказаться полезным средством.

```
In[408]:= 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[409]:= NamesProc1[] := Select[Map[ToExpression, Sort[Names["`*"]]], ProcQ[#] &]
In[410]:= NamesProc1[]
Out[410]= {Df, Dff, NamesProc, ProcQ, SortNL, Spos, Subs, Uprocs}

```

Вызов простой функции **GV[]** возвращает список *специальных* символов и глобальных переменных пакета текущего релиза; в используемом релизе их *общее* количество **175**. Следующий фрагмент представляет исходный текст функции и ее использование.

```

In[383]:= GV[] := Select[Names["*"], StringTake[#, 1] == "$" &]
In[384]:= GV[]
Out[384]= {"$Aborted", "$ActivationGroupID", "$ActivationKey", ..., "$VersionNumber"}
In[385]:= Length[Select[GV[], Attributes[#] != {} &]]
Out[385]= 90
In[386]:= Length[Select[GV[], Attributes[#] == {Protected} &]]
Out[386]= 34
In[387]:= DeleteDuplicates[Select[Map[Attributes, Select[GV[], Attributes[#] != {} &]],
# != {Protected} &]]
Out[387]= {{Locked, Protected}, {Protected, ReadProtected}, {HoldAll, Protected}}

```

Так, последние 3 примера фрагмента констатируют, из **175** глобальных переменных и специальных символов пакета **90** наделены специальными атрибутами, например, **34** наделены только *Protected*-атрибутом. Остальные приходятся на три типа атрибутов, представленных последним примером данного фрагмента.

Как уже отмечалось, средства процедурного программирования пакета *Mathematica* имеют относительно пакета *Maple* более ограниченные выразительные возможности, что в целом ряде случаев создает определенные затруднения при создании процедур *специального* назначения. В частности, пакет *Maple* при программировании процедур допускает использование в их телах специальных параметров (*args, nargs, procname* и др.), о которых говорилось выше и которые позволяют весьма существенно облегчать программирование целого ряда процедур. Немало процедур нашей Библиотеки [45] существенно использовали данные средства. Естественно, каждая программная среда требует своего развития под конкретные нужды пользователя, включая оба пакета.

Это мы делали и в *Maple*, и в значительно *меньшей* степени в *Mathematica* по причине менее длительного периода работы со вторым пакетом. И с учетом вышесказанного в зависимости от типа расширяющих программную среду процедур одни из них более эффективно реализуемы в *Maple*, тогда как другие в среде пакета *Mathematica*. Ниже приводится фрагмент, содержащий пример процедуры некоего аналога упомянутых параметров, используемых в теле процедур программной среды *Maple*.

```

In[826]:= SubDelStr[x_String, L_List] := Module[{k = 1, a = {}},
If[! L == Select[L, ListQ[#] && Length[#] == 2 &], x, For[k, k <= Length[L], k++,
a = Append[a, StringTake[x, L[[k]]] -> ""]; StringReplace[x, a]]]
In[827]:= ListListQ[L_] := If[ListQ[L] && Length[Select[L, ListQ[#] && Length[#] ==

```

```

Length[L[[1]] &]] == Length[L], True, False]
In[828]:= Nproc[] := Module[{a, b, c, g, f = "$$Art22$Kr15$.txt", k = 1, h = ""},
  a = Upprocs[]; 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[829]:= Avz[x_Integer, y_Integer, z_Real] := Module[{a, b, c}, b = c*(x + y);
  b = Sqrt[GS[x] + y + z*b]; Nproc[]
In[830]:= Avz[14, 22, 67.62]
Out[830]= {"Avz", 3}

```

Процедура **SubDelStr**[*x*, *L*] обеспечивает удаление из строки *x* подстрок, ограниченных номерами позиций, заданных *списком списков* *L* из двухэлементных списков. Функция **ListListQ**[*L*] тестирует список *L* быть *списком списков* (*ListList*). Тогда как вызов **Nproc**[] процедуры, закодированной в теле процедуры, при вызове последней обеспечивает получение списка $\{N, m\}$, где *N* - имя главной процедуры и *m* - число ее аргументов.

В качестве еще одного довольно полезного примера можно привести процедуру, чей вызов **Args**[*M*] возвращает список формальных аргументов для процедуры/функции *M*. Исходный текст и пример применения процедуры представлены ниже. При этом, в случае невозможности вычислить аргументы возвращается пустой список, т.е. {}.

```

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}];
    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[{}],
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, 72, Length, x, (a + b)/(c+d), 420}]
```

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

Attributes[Length] = {Protected}

```
Out[1340]= {{S_String, h_ /; CharacterQ[h], p_ /; MemberQ[{1, 2, 3}, p]}, {x_}, {}, {}, {}, {}, {} }
```

Более того, в качестве фактического аргумента процедуры **Args**[W] может выступать произвольное *Mathematica*-выражение. При этом, на стандартных функциях пакета возвращается пустой список с выводом по ним справочной информации.

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; при этом, обработка файла производится на месте.

```

In[419]:= DeCod[F_String] := Module[{a,b=F, k=1,Kr}, Kr[N_Integer] := 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[420]:= DeCod["D:\\Math_myLib\\Addition.doc"]
```

```
Out[420]= "D:\\Math_myLib\\Addition.doc"
```

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 N.

```

In[44]:= ExtrCall[N_, P_] := Module[{a}, a[x_] := ! StringFreeQ[ToString[Definition[P]], " "
<> ToString[x] <> "["; If[! ListQ[N], a[N], Select[N, a[#] &]]]

```

```
In[45]:= Map3[ExtrCall, Run, {ActiveProcess, Attrb, SearchDir, SearchFile, Df, Uprocs}]
```

```
Out[45]= {True, True, True, True, False, False}
```

```
In[46]:= ExtrCall[{Run, Write, Read, If, Return}, Attrb]
```

```
Out[46]= {Run, Read, If, Return}
```

Вызов следующей весьма простой процедуры **ExtrExpr**[S, N, M] возвращает в строчном

формате корректное выражение, содержащееся в подстроке строки S , ограниченной позициями с номерами N и M . В отсутствие выражения возвращается пустой список.

```
In[121]:= ExtrExpr[S_String, N_Integer, M_Integer] := 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[122]:= ExtrExpr["z=(Sin[x+y] + Log[x])+G[x,y];", 4, 13]
Out[122]= "Sin[x + y]"
In[123]:= ExtrExpr["z=(Sin[x+y] + Log[x])+F[x,y];", 1, 21]
Out[123]= "Log[x] + Sin[x + y]"
In[124]:= ExtrExpr["z = (Sin[x + y] + Log[x]) + F[x, y];", 1, 29]
Out[124]= "F[x, y] + Log[x] + Sin[x + y]"
```

Данная процедура оказывается достаточно полезной в целом ряде приложений.

```
In[2]:= BinaryListQ[L_List] := MemberQ[{{0},{1},{0,1}}, Sort[DeleteDuplicates[Flatten[L]]]]
In[3]:= L = {0, 1, 0, 1, {0, 0, 1}, 1, 0, 1, 1}; BinaryListQ[L]
Out[3]= True

In[4]:= Bits[x_, P_Integer] := Module[{a, k}, If[StringQ[x] && StringLength[x] == 1,
If[1 <= P <= 8, PadLeft[IntegerDigits[ToCharacterCode[x]][[1]], 2], 8][[P]],
If[P == 0, PadLeft[IntegerDigits[ToCharacterCode[x]][[1]], 2], 8],
Defer[Bits[x, P]]], If[BinaryListQ[x] && 1 <= Length[Flatten[x]] <= 8, a = Length[x];
FromCharacterCode[Sum[x[[k]]*2^(a - k), {k, 1, a}], Defer[Bits[x, P]]]]

In[5]:= {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[5]= {{0, 1, 0, 0, 0, 0, 0, 1}, 1, "A", Bits["A", 9], "="}
```

Предыдущий фрагмент представляет функцию **BinaryListQ[L]**, тестирующую список L на предмет *бинарности* как единичных его элементов, так и подсписков. Определение факта бинарности возвращает значение **True**, иначе возвращается **False**. Эта функция достаточно полезна в целом ряде приложений. Так, процедура **Bits[x,P]** использует ее довольно существенно, обеспечивая ряд полезных функций при работе с символами. На кортеже фактических аргументов $\langle x, P \rangle$, где x – односимвольная строка (*character*) и P – целое в диапазоне $0..8$, процедура возвращает *бинарное* представление x в форме списка, если $P=0$, и P -й бит такого представления x в противном случае. Тогда как на кортеже фактических аргументов $\langle x, P \rangle$, 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** возвращается невычисленным.

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 обеих концов; при других значениях N вызов возвращается *невычисленным*. 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 обоих концов строку W при $p=3$; при других значениях p вызов **StrDelEnds**[W, h, p] возвращается невычисленным. Следующий фрагмент представляет исходные тексты обоих процедур с некоторыми примерами их конкретного использования.

```
In[355]:= DelEl[L_List, 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[356]:= DelEl[{a, a, a, a, a, 63, 68, 43, 14, 42, 47, 22, a, a, a, a}, a, 1]
Out[356]= {63, 68, 43, 14, 22, a, a, a, a}
In[357]:= DelEl[{a, a, a, a, a, 63, 68, 43, 14, 22, 42, 47, a, a, a, a}, a, 2]
Out[357]= {a, a, a, a, a, 63, 68, 43, 14, 22}
In[358]:= DelEl[{a, a, a, a, a, 63, 68, 43, 14, 22, 42, 47, a, a, a, a}, a, 3]
Out[358]= {63, 68, 43, 14, 22, 42, 47}
In[359]:= StrDelEnds[S_String, 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[360]:= StrDelEnds["xxxxxxx123456789xxxxxxxxxxx", "x", 1]
Out[360]= "123456789xxxxxxxxxxx"
In[361]:= StrDelEnds["xxxxxxx123456789xxxxxxxxxxx", "x", 2]
Out[361]= "xxxxxxx123456789"
In[362]:= StrDelEnds["xxxxxxx123456789xxxxxxxxxxx", "x", 3]
Out[362]= "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 процедура **StrStr**[x] обеспечивает возврат выражения x , отличного от строки, в строчном формате и двойной строки в противном случае. Процедура **StrStr** полезна при работе со строчными конструкциями, в частности, с функцией **StringReplace**.

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. Вызов **TestProcCalls**[*P*, *L*] процедуры возвращает результат вызова процедуры **P** на *фактических* аргументах, определенных списком *L*, если они допустимы, иначе возвращается список из значений {True, False}, порядком соответствующих *порядку* фактических аргументов при вызове процедуры *P*. Реализация процедуры базируется на ранее рассмотренной процедуре **Args**.

```
In[676]:= StrStr[x_] := If[StringQ[x], "\"" <> x <> "\"", ToString[x]]
In[677]:= Map[StrStr, {"RANS", a + b, IAN, {68, 63, 43}, F[x,y]}]
Out[677]= {"\"RANS\"", "a + b", "IAN", "{68, 63, 43}", "F[x, y]"}
In[678]:= SuffPref[S_String, s_String, 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[679]:= SuffPref["IAN_RANS_RAC_REA_72_420", "72_420", 2]
Out[679]= True
In[680]:= SuffPref["IAN_RANS_RAC_REA_72_420", "IAN_RANS", 1]
Out[680]= True
In[681]:= TestProcCalls[P_ /; ProcQ[P], L_List] := Module[{a, b = {}, c, d, k = 1}, a = Args[P];
    If[a == {}, Return[P[]], a = Map[ToString, a]; For[k, k <= Length[L], k++, d = a[[k]];
        If[SuffPref[d, "_", 2], b = Append[b, True], c = StringPosition[d, "_/"];
        If[c == {}, b = Append[b, ToString[Head[L[[k]]]] ==
            StringTake[d, {Flatten[StringPosition[d, "_"]][[-1]] + 1, -1}]],
            b = Append[b, ToExpression[StringTake[d, {1, c[[1]][[1]] - 1}] <> " = " <>
                ToString[L[[k]]] <> "," <> StringTake[d, {c[[1]][[2]] + 1, -1}]]]]];
        If[DeleteDuplicates[b] != {True}, b, ToExpression[ToString[P] <> "[" <>
            StringTake[ToString[Map[StrStr, L]], {2, -2}] <> "]" ]]]]
In[682]:= TestProcCalls[SuffPref, {"IAN_RANS_RAC_REA_72_420", "72_420", 2}]
Out[682]= True
```

Во многих случаях актуально знание о доступности в текущем сеансе пакета того или иного объекта (*переменной, процедуры и др.*). В этом плане процедура **HowAct**[*Q*] может оказаться намного полезней и эффективнее стандартной функции **ValueQ**[*x*] пакета.

```
In[325]:= HowAct[Q_] := If[Quiet[Check[ToString[Definition[Q]], True]] ===
    "Null", False, True]
```

```
In[326]:= x = 6; {Map[ValueQ, {ProcQ, Sin, 75, x}], Map[HowAct, {ProcQ, Sin, 75, x}]}
Out[326]= {{False, False, False, False}, {True, True, True, True}}
```

Вызов **HowAct**[*Q*] возвращает True на активизированном в текущем сеансе объекте *Q*, включая конкретные значения любого допустимого типа, и False в противном случае.

Для пакета *Maple* была создана процедура **Email**, обеспечивающая детальный анализ файлов типов {doc, rtf, htm} и в общем случае *rlb*-типа на предмет наличия 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[436]:= 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[437]:= Email["D:\\Math_myLib\\Rans_Ian.mht", Res]
Out[437]= {"aladjev@yandex.ru", "aladjev@gmail.com", "valadjev@yahoo.com",
  "noosphere_academy@yahoo.com", "iansd.staff@inbox.ru", "vkristo@live.com"}
In[438]:= Res
Out[438]= {"@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.

```
In[379]:= SubStr[S_String, p_Integer, 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[380]:= SubStr["12345abcdefg6789sewrt", 14, "3", "r", Error]
Out[380]= "12345abcdefg6789sew"
```

```
In[381]:= SubStr["12345abcdefg6789sewrt", 14, {"1", "2", "0"}, {"s", "w", "x"}, Error]
Out[381]= "345abcdefg6789"
In[382]:= SubStr["12345abcdefg6789sewrt", 22, "0", "x", Error]
Out[382]= $Failed
In[383]:= Error
Out[383]= "Argument p should be in range 1..21 but received 22"
```

Предыдущий фрагмент представляет довольно 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.

Следующий фрагмент представляет реализацию процедуры **Email** с использованием процедуры **SubStr**, позволяя упростить алгоритм и расширить *типы* анализируемых файлов данных на предмет наличия в них корректных *email*-адресов.

```
In[1349]:= Email[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, $Art22$Kr14$]];
  a = DeleteDuplicates[a]; t = Select[a, # == "@" | | StringFreeQ[StrDelEnds[#, "@"],
    3], {"@", "."] | | StringFreeQ[#, "."] | | StringTake[#, {1, 1}] == "." | |
  StringTake[#, {-1, -1}] == "." &]; Select[a, ! MemberQ[t, #] &]]
In[1350]:= Email["D:\\Math_myLib\\Noosphere_Academy.nb", Res]
Out[1350]= {"valadjev@yahoo.com", "vaganov@yandex.ru", "vkristo@live.com"}
In[1351]:= Res
Out[1351]= {"@", "@font-face", "@page", "Art@Kr", "@SimSun", "U@", "rans@", "ian@"}
```

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 этой задачи, поэтому можно использовать следующие процедуры.

```
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[1409]:= Range2[x_, y_Integer /; 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]:= Agn[Range3[z, 10]] := Module[{}, (z1 + z3 + z5 + z7 + z9)/(z2 + z4 + z6 + z8 + z10)]
In[1413]:= Range3[x_, y_Integer /; 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[1414]:= Range3[h, 14]
Out[1414]= {h1_, h2_, h3_, h4_, h5_, h6_, h7_, h8_, h9_, h10_, h11_, h12_, h13_, h14_}
In[1415]:= Agn[Range[10]]
Out[1415]= 5/6

```

Вызов **Range1**[*J1, Jp*] возвращает список переменных в форме {*J1, J2, J3, ..., Jp*}, тогда как вызов **Range2**[*J, p*] возвращает список переменных в аналогичной форме; тем временем вызов процедуры **Range3**[*J, p*] возвращает список в форме {*J1_, J2_, J3_, ..., Jp_*}, где *X* – имя и *p* – целое число. При этом, процедура **Range3** в ряде случаев создает достаточно 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 данной книге, можно найти в нашем небольшом пакете «*AVZ_Package*» [90], содержащем более 200 средств.

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** дает возможность реализовать достаточно полезный механизм, аналогичный *procname*-механизму пакета *Maple*. Один из таких подходов наглядно иллюстрирует процедура *Avz* последнего примера фрагмента.

Из примера использования **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 сгенерированного списка {*Z*₁, *Z*₂, ..., *Z*_{*n*}} (*n* ≥ 1).

```
In[367]:= AssignToList[y_List, z_ /; ! HowAct[z], n_Integer /; 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[368]:= {AssignToList[Range[9], z, 9], z1, z3, z5, z6, z9}
Out[368]= {{1, 2, 3, 4, 5, 6, 7, 8, 9}, 1, 3, 5, 6, 9}
```

Механизм, обеспечивающий работу с объектами типа «последовательность» (*sequence*), в *Mathematica* отсутствует, что создает определенные затруднения. Между тем, пакет предоставляет функцию **Sequence**[*a*, *b*, *c*, ...], которая определяет последовательность аргументов, автоматически передаваемых произвольной функции/процедуре. В этом контексте процедура **SequenceQ**[*s*] обеспечивает *тестирование* объектов, созданных на основе функции **Sequence**, возвращая True, если *s*-объект определен этой функцией, и False в противном случае; при этом, имя *s*-объекта кодируется 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, a72, b420, as69, z, a, b, c, m, n, p, x, y, z]
```

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.

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 представлены следующим образом, а именно:

- *первый* элемент подписки определяет контекст, соответствующий пакету, который был загружен в *текущем* сеансе пакета на момент вызова процедуры **NamesCS**;
- *все* последующие элементы подписки определяют объекты данного пакета, которые были в *текущем* сеансе пакета активизированы.

```

In[613]:= 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},
{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++, If[Length[c[[k]]] == 1, c[[k]] = Null,
c[[k]] = Select[c[[k]], StringQ[#] || ToString[Definition[#]] !=
"Null" &]]; Pobj := Select[c, Length[#] > 1 && ! # === Null &]]

In[614]:= NamesCS[P, Pr, Pobj]
In[615]:= {P, Pr}
Out[615]= {"AladjevProceduresAndFunctions`", "ResourceLocator`", "GetFEKernelInit`",
"DocumentationSearch`", "JLink`", "PacletManager`", "WebServices`", "System`", "Global`",
{Avz, Svetla, NamesCS, ProcName, Art22Kr14, Noosphere}}
In[616]:= Pobj
Out[616]= {"AladjevProceduresAndFunctions`", ActiveProcess, Adrive, Aobj, Args, Args0,
AssignToList, Attrib, AutoLoadPack, BinaryListQ, Bits, BlockQ, CALL, CallsInProc, CDir,
CharacterQ, Closes, CopyDir, DeCod, DelEl, Df, Df1, Df2, DirName, DirQ, Email, Email1,
ExtrCall, ExtrExpr, ExtrPackName, FileFormat1, FileOpenQ, FunCompose, FunctionQ, Int,
Help, HowAct, Int1, ListListQ, LoadFile, LoadNameFromM, Locals, Map1, Map2, Map3,
Map4, Map5, Map6, Mapp, MixCaseQ, NamesCS, NamesNbPackage, NamesNbPackage1,
NamesProc, NbName, Need, Nobj, Nproc, PackNames, ProcQ, PureFuncQ, Range1,
Range2, Range3, ReadFullFile, ReplaceAll1, SearchDir, SearchFile, SequenceQ, SeqUnion,
SetDir, SortLpos, SortNL, Spos, StrDelEnds, StringEnd, StrStr, SubDelStr, Subs, SubStr,
SuffPref, TestArgsTypes, TestProcCalls, UpdateContextPaths, UpdatePackages,
UpdatePath, UprocQ, Uprocs, Ver}, {"DocumentationSearch`", Information}, {"JLink`",
Information}, {"PacletManager`", Information}, {"WebServices`", Information}, {"System`",
ComplexExpand, Environment, InterpolatingFunction}}

```

Более того, список, возвращаемый через **Pobj**-аргумент, содержит только подписки, чьи соответствующие пакеты имеют в текущем сеансе активизированные объекты.

```

In[488]:= $ProcName := Block[{a, b, c, d, h, t = " := ", k = 1}, a = ToString[Stack[_][[1]]];
b = Names["`*"]; For[k, k <= Length[b], k++, h = ToString[ToExpression["Definition["
<> b[[k]] <> "]"]]; If[StringFreeQ[h, t], Continue[], d = Flatten[StringPosition[h, t]];
If[a == StringTake[h, {d[[2]] + 1, -1}], c = StringTake[h, {1, Flatten[StringPosition[h,
"["]][[1]] - 1}]; Break[], Continue[]]; c]

In[489]:= Agn[x_, y_, z_] := Module[{a, b}, {$ProcName, Args[$ProcName],
ProcQ[ToExpression[$ProcName]], Locals[ToExpression[$ProcName]]}]

In[490]:= Agn[42, 47, 67]
Out[490]= {"Agn", {x_, y_, z_}, True, "{a, b}"}

```

Предыдущий фрагмент иллюстрирует пример применения глобальной переменной **\$ProcName**, определенной специальной блочной конструкцией, которая возвращает *имя* содержащей ее процедуры. Вызов следующей процедуры **WhatObj[x]** возвращает местоположение объекта *x*, активизированного (*доступного*) в текущем сеансе пакета, а именно: **None** – неопределенный объект, **"Standard"** – стандартная функция пакета, **"Current Session"** – объект, чье определение вычислено в текущем сеансе, **"Контекст"** – контекст, определяющий пакет, активизированный в текущем сеансе и содержащий определение *x*-объекта, в противном случае вызов процедуры **WhatObj** возвращается невычисленным. Данная информация довольно востребована в целом ряде случаев.

```
In[839]:= 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 = $Art22Kr14$; Clear[$Art22Kr14$];
    t = SubStr[a, b[[1]] - 1, {"[", "{"}, "", $Art22Kr14$] <> ""; $Art22Kr14$ = h; t]]]
In[840]:= Map[WhatObj, {Map3, Break, Agn, Sin, 68, GS, Continue, Goto, StrStr}]
Out[840]= {"AladjevProceduresAndFunctions`", "Standard", "Current Session", "Standard",
    WhatObj[68], None, "Standard", "Standard", "AladjevProceduresAndFunctions`"}
```

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 выше 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.

```
In[420]:= ToString1[x_] := Module[{a = "$Art22Kr$.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[421]:= Kr[x_] := Module[{a = "Art_Kr", b = ""}, a <> b <> ToString[x]]
In[422]:= ToString[Definition[Kr]]
Out[422]= "Kr[x_] := Module[{a = Art_Kr, b = ""}, a <> b <> ToString[x]]"
In[423]:= ToExpression[%]
ToExpression::sntx: Invalid syntax in or before "Kr[x_] := Module[{a = Art_Kr, b = ""} ... ".
Out[423]= $Failed
```

```

In[424]:= ToString1[Definition[Kr]]
Out[424]= "Kr[x_] := Module[{a = \"Art_Kr\", b = \"\" = \"\"}, StringJoin[a, b, ToString[x]]]"
In[425]:= ToExpression[%]; Kr[2]
Out[425]= "Art_Kr = 2"
In[426]:= 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[427]:= DefFunc[Kr]
Out[427]= Kr[x_] := Module[{a = "Art_Kr", b = "" = ""}, a <> b <> ToString[x]]

```

Непосредственное применение процедуры *ToString1* позволяет в целом ряде случаев, порой довольно существенно, упрощать алгоритмы решаемых задач. Так, применение процедуры оказывается весьма эффективным при реализации алгоритма процедуры **DefFunc[x]**, обеспечивающей возврат в компактном формате определения *x*-объекта, содержащегося в пакете либо *nb*-документе, загруженном в текущий сеанс.

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**, чей вызов возвращает вложенный список, чьи подписки в строчном формате содержат имена активных объектов и в качестве последнего элемента – их тип, распознаваемый пакетом либо определенный нами, в частности, {"procedure", "function"} [90].

```

In[444]:= TypeActObj[] := Module[{a = Names["*"], b = {}, c, d, h, p, k = 1},
    For[k, k <= Length[a], k++, h = a[[k]]; c = ToExpression[h]; p = Head[c];
    If[! StringFreeQ[h, "$"] || p === Symbol && Definition[c] === Null,
    Continue[], b = Append[b, {h, If[ProcQ[c], "procedure",
    If[FunctionQ[c], "function", p]}]];
    a = Quiet[Gather1[Select[b, ! #[[2]] === Symbol &], 2]];
    ToExpression[StringReplace[ToString1[Map[DeleteDuplicates,
    Map[Sort, Map[Flatten, a]]],
    "AladjevProceduresAndFunctions`TypeActObj`" -> ""]]]
In[445]:= TypeActObj[]
Out[445]= {{{"A", "b", "G", Integer}, {"Art", "Cv", "Gather1", "TypeActObj", "x", "procedure"},
    {"c", "d", "h", Real}, {"F", "W", "function"}, {"Kt", Plus}, {"L", List}, {"Z", Times}, {"T", Cos}}}
In[460]:= Gather1[L_ /; ListListQ[L], n_Integer] := 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[461]:= L = {{42, V, 1}, {47, G, 2}, {67, S, 1}, {68, V, 2}, {63, G, 3}, {43, S, 2}}; Gather1[L, 2]
Out[461]= {{{42, V, 1}, {68, V, 2}}, {{47, G, 2}, {63, G, 3}}, {{67, S, 1}, {43, S, 2}}}

```

С целью упрощения алгоритма процедуры **TypeActObj** была определена процедура, чей вызов **Gather1[L, n]** возвращает вложенный список *ListList*-типа, образованный на основе *ListList*-списка *L* путем группировки его подсписков по их *n*-му элементу. Эта процедура несколько расширяет стандартную функцию **Gather** пакета и оказывается достаточно полезной в целом ряде приложений. Тогда как весьма простая процедура **GC** может оказаться полезной в ряде приложений, связанных, в частности, с задачами сортировки. Вызов **GC[x]** возвращает уникальный десятичный код *x*-выражения. При этом, из примеров применения процедуры следует, что результаты, возвращаемые на выражениях идентичного содержания в *символьном* и *строчном* форматах различны.

```
In[4]:= GC[x_] := ToExpression[StringReplace[ToString[ToCharacterCode[ToString[
    InputForm[x]]], {" " -> "", " " -> "", " " -> ""}]]]

In[5]:= Map[SC, {"RANS_IAN_RAE_RAC", 72, 42.47, 2011, Avz}]
Out[5]= {1438069418780295530484157500000, 2750, 342056000, 5762400, 935740}

In[6]:= Map[GC, {{GS, "GS"}, {a/b, "a/b"}, {a + b, "a+b"}}]
Out[6]= {869057467240512000, 4995361808033786112000, 4281619809256576647168000}

In[7]:= SortNL1[L_List, p_Integer, 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[8]:= L = {{42, V, 1}, {47, G, 2}, {67, 67, 15}, {68, 72, 2}, {63, 42, 3}, {43, S, 2}}; SortNL1[L, 2, Less]
Out[8]= {{47, G, 2}, {43, S, 2}, {42, V, 1}, {63, 42, 3}, {68, 72, 2}, {67, 67, 15}}
```

Вызов процедуры **SortNL1[L, p, b]** возвращает результат сортировки *вложенного* списка *L ListList*-типа по *p*-элементам его подсписков на основе их уникальных десятичных кодов, определяемых процедурой **GC**, представленной выше. При этом, 3-й аргумент допускает 2 значения {*Greater*, *Less*}. Процедура **SortNL1** является обобщением **SortNL** на случай сортировки в соответствии с уникальными десятичными кодами. Тогда как довольно простая процедура **Arity[x]** возвращает арность процедуры/ функции *x*.

```
In[44]:= Arity[x_] := Module[{a = ToString[DefFunc[x]]}, If[! ProcQ[x] && ! FunctionQ[x]
    && a != "Null", Defer[Arity[x]], If[ProcQ[x] || FunctionQ[x],
    a = Args[x]; If[StringFreeQ[ToString[a], {"_", "___"}], Length[a],
    "Undefined"], Defer[Arity[x]]]]]

In[45]:= Avz[x_] := Module[{}, Plus[x]]; Avz[68, 63, 43, 48, 22, 14]
Out[45]= 258

In[46]:= Map[Arity, {Cos, Agn, ProcQ, TypeActObj, Subs, SubStr, Avz, Arity}]
Out[46]= {Arity[Cos], Arity[Agn], 1, 0, 3, 5, "Undefined", 1}
```

При этом, для процедур/функций с неопределенным числом аргументов вызов **Arity** возвращает "Undefined", в остальных случаях вызов возвращается невычисленным.

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 **63** и **68** 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 = 63; Avz := 68; Art := 22; Kr = 14; 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]= {63, 68, 14, 22}

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[$Art$], 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 = 63; Avz := 68; Art := 22; Kr = 14; SetAttributes["Agn", Protected];
SetAttributes["Art", Protected];
In[1559]:= Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]
Out[1559]= {{Protected}, {}, {Protected}, {}]
In[1560]:= {$Art$, $Kr$} = {72, 420}; VarExch1[{"Agn", "Avz", {"x", "y"}, {"Art", "Kr"}]]
In[1561]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {"Agn", "Avz", "Art", "Kr"}]]
Out[1561]= {{68, 63, 14, 22}, {}, {Protected}, {}, {Protected}}]
In[1562]:= {x, y, $Art$, $Kr$}
Out[1562]= {x, y, 72, 420}

```

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 подсписков 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 в текущем сеансе в *Input*-параграфе, и не поддерживается в случае загрузки в текущий сеанс пакета, содержащего процедуры с глобальными переменными в их определениях.

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 = 63; SetAttributes[Agn, Protected]; Attributes[Agn];
In[495]:= Art22[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
Return[Defer[Art22[p]]]], b = 420, c}, c = N[Sin[72]]; {a, b, c}]
In[496]:= Art22[{"Agn", 68}]
Out[496]= {Art22[{"Agn", 68}], 420, 0.253823}
In[497]:= Art22[p_List] := Module[{a = Quiet[Check[Map[Attributes, p],
Print[Defer[Art22[p]]]; Abort[]], b = 420, c}, c = N[Sin[72]]; {a, b, c}]
In[498]:= Art22[{"Agn", 68}]
Art22[{"Agn", 68}]
Out[498]= $Aborted
In[499]:= Art22[p_List] := Module[{a = Quiet[Check[Map[Attributes, p], $Aborted]],
b = 420, c}, If[a == $Aborted, Return[Defer[Art22[p]], Null]; c = N[Sin[72]]; {a, b, c}]
In[500]:= Art22[{"Agn", 68}]
Out[500]= Art22[{"Agn", 68}]
```

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 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, {68, 63, 14, 22, 72, 420, Null, 2011}]
Out[456]= {"Agn", "Avz", "Art", "Kr", "Varn", "$Art$", "Vsv", "$Kr$", "Grsu", {}}
In[457]:= 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[458]:= Contexts1[]
Out[458]= {"AladjevProceduresAndFunctions", "Algebra", ..., "WrappersDump", "XML"}
In[459]:= Npackage[x_String] := If[MemberQ[Contexts1[], x], Names[x <> "*"], $Failed]
In[460]:= Npackage["AladjevProceduresAndFunctions"]
Out[460]= {"AcNb", "ActiveProcess", "Adrive", "Aobj", "Args", ..., "$Help", "$ProcName"}
In[461]:= SortString[x_String, y_ /; MemberQ[{Greater, Less}, y]] := StringJoin[
    Sort[Characters[x], y[ToCharacterCode[#1][1], ToCharacterCode[#2][1]]] &]]
In[462]:= SortString["AVZ321AGN456Art789Kr", Less]
Out[462]= "123456789AAAGKNVZrrt"
```

Процедура **Contexts1**, являясь модификацией функции **Contexts1** пакета, возвращает список контекстов, соответствующих пакетам, компоненты которых активизированы в текущем сеансе. Тогда как весьма простая функция **Npackage[x]** возвращает список имен в строчном формате всех объектов, определения которых содержатся в пакете *x*, активизированном в текущем сеансе. В случае неактивности в текущем сеансе пакета *x* или его отсутствия вызов функции **Npackage[x]** возвращает значение **\$Failed**. Тогда как функция **SortString[x, y]** возвращает результат *посимвольной* сортировки *x*-строки в соответствии со вторым аргументом, принимающим значение {*Greater* | *Less*}.

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.

```

In[1478]:= StringReplace1[S_String, L_List /; ListListQ[L] && Length[L[[1]]] == 2 &&
          MatrixQ[L, IntegerQ] && Sort[Map[Min, L]] [[1]] >= 1, P_List] :=
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[1479]:= StringReplace1["avz123456789agn", {{4, 7}, {8, 10}, {11, 12}}, {" RANS ", Tampere,
Sqrt[(a + b)*(c + d)]]]
Out[1479]= "avz RANS TampereSqrt[(a + b) (c + d)]agn"

```

Средства *Mathematica* позволяют весьма 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 как для организации пакетных модулей *Maple*, так 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]*, которая рассматривает 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 другого типа. В первом случае при наличии в *L* подписка с первым элементом *a* он заменяется на $\{a, b\}$, иначе он дополняет *L*; второй случай возвращает список $\{indices | entries\}$ соответственно; 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}}

```

В ряде случаев при работе с файлами данных может оказаться достаточно полезной и 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. {}.

```
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.22",
"C:/Temp/Agn.63", "C:/Academy/RANS/IAN/Tallinn.txt"}}
```

В ряде случаев при работе с файлами данных может оказаться достаточно полезной и процедура **IsFileOpen[F, h]**, чей вызов возвращает True, если файл *F*, определенный его полным путем, открыт, и False в противном случае. Если же аргумент *F* не определяет файл и/или второй аргумент *h* является определенным, вызов процедуры **IsFileOpen** возвращается *невычисленным*. При этом, через второй аргумент в случае возврата True возвращается список режимов открытия файла данных *F*, а именно {"read", "write"}.

```
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"}}
```

Следующая процедура **NestQL[L]** оказывается весьма полезной в работе со списками, возвращая True, если список *L* является вложенным, и False в противном случае.

```
In[489]:= NestQL[L_/: ListQ[L]] := If[MemberQ[Map[ListQ[#] &, L], True], True, False]
In[490]:= {NestQL[{a, b, c}], NestQL[{a, {b, x}, c}], NestQL[{a, b, x, c}]}
Out[490]= {False, True, True}
In[531]:= MaxNestLevel[L_/: ListQ[L]] := Module[{a = Flatten[L], b = L, c = 0},
    While[! a == b, b = Flatten[b, 1]; c = c + 1]; c];
In[532]:= L = {{a, {b, {m, {x, y}, n}, x}, c, {{{{{{63, 68}}}}}}}}; Map[MaxNestLevel, {L, {a, b, c}}]
Out[532]= {8, 0}
```

```

In[560]:= L = {m, {m, {b, {m, {x, y}, n}, x}, c, {{{{{{63, m, 68}}}}}}}}; MemberQ[L, x]
Out[560]= False

In[561]:= 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[562]:= {{MemberQ1[L, x, y], y}, {MemberQ1[L, 68, z], z}, {MemberQ1[L, m, h], h}}
Out[562]= {{True, {2, 4}}, {True, {8}}, {True, {0, 1, 3, 8}}}

In[563]:= 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[564]:= L = {42, 47, 63, 68, 6, 68, 68, 72, {6, 68}, 420, 68}; {MemberQ2[L, 68, g], g}
Out[564]= {True, 5}

```

С другой стороны, 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. элементы которого не являются списками. В завершение предыдущего фрагмента представлены процедура **MemberQ1**, которая в определенной степени расширяет стандартную функцию **MemberQ** на вложенные списки, и процедура **MemberQ2[L,x,y]**, которая расширяет эту 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 невлеченный список имеет уровень вложенности 0*); в противном случае возвращается значение **False**. 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* есть элемент *L*-списка; в противном случае возвращается **False**. Более того, при возврате значения **True** через третий аргумент *y* возвращается *число* вхождений *x*-значения в список *L*. В принципе, процедуры предыдущего фрагмента допускают ряд довольно интересных модификаций, существенно расширяющих *сферу* применения этих средств. Оставим это заинтересованному читателю в качестве достаточно полезного упражнения.

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 различных 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 *Со* предшественников, т.е. блочных конфигураций, которые под действием *глобальной функции* перехода структуры в следующий момент времени переводятся в конфигурацию *Со*. 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.

```
In[558]:= PredecessorsR[Ltf_List, Co_String, 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 = 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[559]:= PredecessorsL[Ltf_List, Co_String, 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[560]:= Ltf = {"0000", "0011", "0101", "0111", "1000", "1011", "1101", "1110"}
Out[560]= {"0000", "0011", "0101", "0111", "1000", "1011", "1101", "1110"}
In[561]:= Ltf1 = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[561]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[562]:= Clear[Cf]; {PredecessorsR[Ltf, "010100", 3, Cf], Cf}
Out[562]= {0, {}}
In[563]:= Clear[Cf]; {PredecessorsL[Ltf, "010100", 3, Cf], Cf}
Out[563]= {0, {}}
In[564]:= Clear[Cf]; {PredecessorsL[Ltf1, "010100", 3, Cf], Cf}
Out[564]= {4, {"01110000", "10101011", "00011101", "11000110"}}
In[565]:= Clear[Cf]; {PredecessorsR[Ltf1, "010100", 3, Cf], Cf}
Out[565]= {4, {"00011101", "01110000", "10101011", "11000110"}}
In[566]:= Clear[Cf]; {PredecessorsL[Ltf, "0101", 3, Cf], Cf}
Out[566]= {1, {"111001"}}
In[567]:= Clear[Cf]; {PredecessorsR[Ltf, "0101", 3, Cf], Cf}
Out[567]= {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 " $x_1x_2...x_nx_1$ " (эти строки отвечают набору параллельных подстановок $x_1x_2...x_n \rightarrow x_1 \setminus \{x_n, x_1 \notin A\}$, 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 γ -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.

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 γ -конфигурацией, и печатает об этом соответствующее сообщение. В случае отсутствия γ -конфигураций размера m вызов процедуры TestOnNCF возвращает Null и печатает об этом соответствующее сообщение. Следующий фрагмент представляет исходный текст процедуры и наиболее типичные примеры ее применения.

```
In[1558]:= TestOnNCF[Ltf_List, n_Integer, m_Integer] := 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 3051 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 места средства по этому показателю.

```
In[557]:= Ltf = {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
Out[557]= {"0000", "0011", "0101", "0110", "1001", "1010", "1100", "1111"}
In[558]:= t = TimeUsed[]; TestOnNCF[Ltf, 3, 15]; TimeUsed[] - t
All block configurations of size 15 possess 4 different predecessors
Out[558]= 626.36

> TestOnNCF := proc(ltf::table, n::posint, m::posint)
local a, b, c, k;
a := map(op, {entries(ltf)});
c := (n - 1)^nops(a);
for k in tuples(a, m) do
b := nops([Predecessors(ltf, k, n)][1]);
if b <> c then return k, b end if
end do;
WARNING("All block configurations of length %1 have %2 predecessors", m, c)
end proc;

> ltf := table(["000"="0", "001"="1", "010"="1", "011"="0", "100"="1", "101"="0", "110"="0", "111"="1"]);
> t := time(): TestOnNCF(ltf, 3, 15); time() - t;
Warning, All block configurations of length 15 have 4 predecessors
195.268

In[554]:= t = TimeUsed[]; Tuples[{"0", "1"}, 15]; TimeUsed[] - t
Out[554]= 0.031

> t:=time(): tuples({"0", "1"}, 15): time() - t; ⇒ 94.671
```

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 выше, 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[642]:= 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[643]:= Tuples1[{1, 2, 3}, 3]
Out[643]= {"111", "112", "113", "121", "122", "123", "131", "132", "133", "211", "212", "213", "221",
  "222", "223", "231", "232", "233", "311", "312", "313", "321", "322", "323", "331", "332", "333"}
In[644]:= t = TimeUsed[]; Tuples1[{0, 1}, 15]; TimeUsed[] - t
Out[644]= 75.89

> t := time(): tuples({1, 2}, 15): time() - t; => 65.266 #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[599]:= 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[600]:= t = TimeUsed[]; NestCycles[9]; TimeUsed[] - t
Out[600]= 923.047

> NestCycles := proc(n::posint)
  local b, k;
  parse(cat(cat(seq(cat("for k", convert(b, string), " to ", convert(n, string), " do "),
  b = 1 .. n), " 75 "), cat(seq("end do ", b = 1 .. n, ":")), 'statement')
end proc;

> t := time(): NestCycles(9): time() - t; => 95.016
```

Следует отметить, использованный в обоих процедурах прием довольно эффективен при необходимости динамической генерации программных кодов.

Для тестирования символов на предмет отсутствия у них значений, нами определены две простые процедуры **HowAct** и **SymbolQ**. Первая из них корректно тестирует факт активности переменной текущего сеанса, однако на локальных переменных **HowAct** возвращает значение False независимо от наличия значений у таких переменных, как это весьма наглядно иллюстрирует следующий простой фрагмент, а именно:

```
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}
```

В таком случае следует использовать процедуру **SymbolQ**, корректно выполняемую в любом месте текущего документа. На примере процедуры **TestOnNCF**, существенно использующей процедуру **PredecessorsR**, несложно убедиться, что в заголовке второй тип *Cf* аргумента, через который процедура возвращает дополнительный результат, должен определяться именно как **SymbolQ**, а не **HowAct**, обеспечивая корректный ее вызов внутри других процедур, у которых *Cf* определено локальной переменной.

```
In[581]:= ExpLocals[P_/: ProcQ[P], L_/: ListQ[L] &&
      If[DeleteDuplicates[Map[SymbolQ, L]] == {True}, True, False]] :=
      Module[{a = ToString1[DefFunc[P]], b, h = Locals[P], d = {}, m, t = 1, v,
c = Map[ToString, Args[P]]}, b = StringPosition[a, " := Module[\""];
      For[t, t <= Length[L], t++, v = ToString[L[[t]]];
      If[StringPosition[h, {"\" <> v, \" <> v <> \", \" <> v <> \"}] === {} &&
      Select[c, SuffPref[#, v <> "_", 1] &] === {}, d = Append[d, v], Null];
      d = ToString[Map[ToExpression, d]]; If[d == "{}", Return[],
      m = If[ListListQ[b], b[[1]][[1]], b[[1]]]; a = StringInsert[a, StringTake[d,
      {2, -2}] <> If[Locals[P] == "{}", "", ", ", "], m + 12]; ToExpression[a];
      StringReplace[StringSplit[d, ","], {"\" -> \"\", \"\" -> \"\", \"\" -> \"\"}]]

In[582]:= Avz[x_] := Module[{a = 6}, b = 75; c = 420; d = b*c; a*d/x]
In[583]:= ExpLocals[Avz, {b, c, d}]
Out[583]= {"b", "c", "d"}
In[584]:= Avz[100]
Out[584]= 1890
In[585]:= {b, c, d}
Out[585]= {b, c, d}
In[586]:= DefFunc[Avz]
Out[586]= Avz[x_] := Module[{b, c, d, a = 6}, b = 75; c = 420; d = b*c; (a d)/x]
```

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 предыдущим 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 **ExpLocals** к очень простой процедуре *Avz* с целью расширения ее списка локальных переменных на 3 переменные. Следует отметить, что процедуру **ExpLocals** можно довольно несложно обобщить на возможность расширения *списков* локальных переменных процедур на *дополнительные* локальные переменные даже с начальными условиями. Оставляем это читателю в качестве достаточно интересного 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[623]:= MemberT[L_ /; ListQ[L], x_] := Length[Select[Flatten[L], # == x &]]
In[624]:= 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[625]:= L = {a, b, {c, d, {d, x, d}, x}, {d, x, h, d}, p, {{d, {d, {a, d, c, d}}}}, d, w}
Out[625]= {a, b, {c, d, {d, x, d}, x}, {d, x, h, d}, p, {{d, {d, {a, d, c, d}}}}, d, w}
In[626]:= {MemberLN[L, d], MemberLN[L, t], MemberT[{a, h, {c, h, g}, s}, h]}
Out[626]= {{{1, 1}, {2, 3}, {3, 3}, {4, 1}, {5, 2}}, {{1, 0}, {2, 0}, {3, 0}, {4, 0}, {5, 0}}, 2}
In[627]:= {MemberLN[{{{{x, {{{y}}}}}}, 2], MemberT[{a, h, {c, {{{y}}}}, h, v}, s], 4]}
Out[627]= {{{1, 0}, {2, 0}, {3, 0}, {4, 0}, {5, 0}, {6, 0}, {7, 0}, {8, 0}}, 0}
```

Отметим, что последние средства **MemberT** и **MemberLN** полагают, что невложенный список имеет уровень *вложенности* 1. Следующий фрагмент представляет процедуру **ListToString[x,y]**, которая возвращает результат *конвертирования* в единую строку всех элементов списка *x* без учета его вложенности, разделенных строкой *y*; в то время как строка *x* конвертируется в список подстрок строки *x*, разделенных строкой *y*.

```
In[5]:= 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[6]:= ListToString[{a + b, {"Agn", 63}, Kr, 14, Art, 22, "RANS", {{{Avz | | 69}}}, "&"]
Out[6]= "a + &\\"Agn\\"&63&Kr&14&Art&22&\\"RANS\\"&Avz | | 69"
In[7]:= ListToString["a + b&\\"Agn\\"&63&Kr&14&Art&22&\\"RANS\\"&Avz | | 69", "&"]
Out[7]= {a + b, "Agn", 63, Kr, 14, Art, 22, "RANS", Avz | | 69}
```

Для работы со строчными структурами достаточно полезной оказывается процедура **InsertN**[*S*, *L*, *n*] (аналог одноименной процедуры для *Maple*, рассмотренной выше), которая обеспечивает вставку в строку *S* подстроку 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 текст 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[583]:= 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[584]:= InsertN["123456789Rans_Ian", {Agn, Avz, Vsv, Art, Kr}, {6, 9, 3, 0, 3, 15}]
Out[584]= "Agn123AvzVsv456Art789KrRans_Ian"
In[585]:= InsertN["123456789", {a, b, c, d, e, f, g, h, n, m}, {4, 2, 3, 0, 1, 5, 6, 7, 8, 9, 15}]
Out[585]= "a1b2c3d4e5f6g7h8n9m"
```

В противоположность предыдущей процедуре следующая процедура **DelSubStr**[*S*, *L*] обеспечивает удаление из строки *S* подстроку, 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[738]:= 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[739]:= DelSubStr["123456789abcdfh", {{3, 4}, {7}, {9}, {13}, {15}}]
Out[739]= "12568abcf"
In[740]:= DelSubStr["123Rans4IAN575420", {{4, 8}, {9, 11}, {12}}]
Out[740]= "12375420"
In[741]:= DelSubStr["123456789abcdgh", Range[1, 15]]
Out[741]= ""
```

```
In[742]:= DelSubStr["123Rans4IAN575420", {4, 8}]
Out[742]= "123ansIAN575420"
In[743]:= DelSubStr["123Rans4IAN575420", {{4, 8}}]
Out[743]= "123IAN575420"
In[744]:= DelSubStr["123456789abcdgh", {6}]
Out[744]= "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 текст процедуры и ее применения.

Выше приведенной *Maple*-процедуре *AllMatrices(m, n, A, G)*, обеспечивающей возврат списка всевозможных (mxn) -матриц, чьи элементы принадлежат конечному алфавиту *A*, в качестве аналога можно привести *Mathematica*-процедуру *AllMatrices[m, n, A]*, с исходным текстом которой и примером применения можно ознакомиться ниже.

```
In[72]:= 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[73]:= AllMatrices[3, 2, {1, 2}]
```

Out[73]=

[illegible]

```
In[121]:= t = TimeUsed[]; Tuples[{0, 1}, 25]; TimeUsed[] - t
```

General::nomem: The current computation was aborted because there was insufficient memory available to complete the computation.

```
Out[121]= SystemException["MemoryAllocationFailure", {HoldForm[t = TimeUsed[];  
Tuples[{0, 1}, 25]; TimeUsed[] - t], HoldForm[Tuples[{0, 1}, 25]]}]
```

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 памяти, как иллюстрирует последний пример фрагмента.

Аналогичная ситуация возникает и с одноименной *tuples*-процедурой, однако в этом случае компьютер, как правило, «зависает», что говорит о более развитой диагностике подобных ситуаций в пакете *Mathematica* по отношению к *Maple*.

Начиная с *Maple 10*, для очень важной *map*-функции была определена опция *inplace*, допустимая лишь при применении этой функции к прямоугольным *rtable*-объектам и обновляющая такие объекты на месте. Между тем, для объектов другого типа такой механизм не поддерживается, как иллюстрируют первые 3 примера нижеследующего фрагмента. С целью устранения данного недостатка предлагается простая процедура *MapInSitu*, чей исходный однострочный экстракод представляет 4-й пример:

```
> y := [75.3, 42.69, 3.3]: [map[inplace](sin, y), y]; # Maple 11
      [[-0.09806582040, -0.9614691168, -0.1577456941], [75.3, 42.69, 3.3]]
> M := Matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9]): map[inplace](^, M, 2): M;
      
$$\begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{bmatrix}$$

> M1 := matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9]): map[inplace](^, M1, 2): eval(M1);
      
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

> MapInSitu:=(x,y::string) -> parse(cat(y,":= map(",convert(x,string),"",y,"):"),'statement'):
> y := [75.3, 42.69, 3.3]: h := {a, b, c}: [MapInSitu(sin, "y"), MapInSitu(G, "h")];
      [[-0.09806582040, -0.9614691168, -0.1577456941], {G(c), G(a), G(b)}]
> [y, h]; => [[-0.09806582040, -0.9614691168, -0.1577456941], {G(c), G(a), G(b)}]
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, -0.999998}, {0.0896956, -0.34307}}
In[1440]:= y
Out[1440]= {{0.0217746, -0.999998}, {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}}}
```

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 **MapInSitu1** 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 функцией **map** | **Map**. Тогда как, во многих случаях в качестве второго фактического аргумента выступает непосредственно *само* обрабатываемое выражение. Как правило, в такой ситуации результат вызова либо присваивается некоторому идентификатору, или непосредственно входит в состав некоторого выражения. Следующая процедура **MapInSitu1**(*x*,*y*) обеспечивает присвоение *всем* идентификаторам, которым в текущем сеансе пользователем были присвоены значения, совпадающие со значением второго фактического аргумента *y*, результат вызова **map**(*x*,*y*), обновляя их значения *на месте*. Следующий фрагмент представляет исходный код процедуры и ее применение.

```
> MapInSitu1 := proc(x, y)
  local a, b, c, d, h, k, _Art_Kr_;
    assign(a = [anames('user')], _Art_Kr_ = []);
    b := parse(cat("[", [Sub_st(["MapInSitu1(" = ""],
      convert('procname(args)', string)[1 .. -2], c])[1], "]"))[2];
  for k in a do if eval(k) = b then _Art_Kr_ := [op(_Art_Kr_), k] end if end do;
  if _Art_Kr_ <> [] then map(x, y);
    seq(assign(_Art_Kr_[k] = map(F, eval(_Art_Kr_[k])), k = 1 .. nops(_Art_Kr_))
  end if; map(x, y)
end proc;

> h := {a, "b", c}; s := {a, "b", c}; t := {a, "b", c}; MapInSitu1(F, {a, "b", c}); h, s, t;
      {F(a), F(c), F("b")}
      {F(a), F(c), F("b")}, {F(a), F(c), F("b")}, {F(a), F(c), F("b")}
> h := [a, "b", c]; s := [a, "b", c]; t := [a, "b", c]; MapInSitu1(F, [a, "b", c]); h, s, t;
      [F(a), F("b"), F(c)]
      [F(a), F("b"), F(c)], [F(a), F("b"), F(c)], [F(a), F("b"), F(c)]
> MapInSitu1(sin, [42.69, 47.64, 420.75]); => [-0.961469117, -0.493500741, -0.221561600]
In[722]:= {H, G} = {{7.5, 47.64, 15.22}, {7.5, 47.64, 15.22}}
Out[722]= {{7.5, 47.64, 15.22}, {7.5, 47.64, 15.22}}
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, 15.22}]
Out[720]= {0.938, -0.493501, 0.468828}
```

```
In[721]:= {H, G}
Out[721]= {{0.938, -0.493501, 0.468828}, {0.938, -0.493501, 0.468828}}
```

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. Вызовы этих процедур возвращают в качестве результата **Map(x, y)** и **map(x, y)** соответственно.

Функция **Args1** – довольно полезная модификация рассмотренной выше 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 тест на строчный формат возвращенного **Args1[P]** списка формальных аргументов процедуры **P**. В отличие от процедур **Args** и **Args0** **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"
```

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[{}, m + n + p; h = Avg[]; h]
In[8]:= Z[69, 64, 44]
Out[8]= {}
```



```

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 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 текст процедуры с различными примерами ее применения. В ряде приложений данная процедура оказывается достаточно полезным средством.

```

In[788]:= ActRemObj[x_ /; StringQ[x], y_ /; MemberQ[{"Act", "Rem"}, y]] :=
Module[{a = Directory[] <> "\\\" <> 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 = 420}, x + a]; A[x_, y_] := Module[{a = 75}, x + y + a]
In[790]:= {A[100], A[100, 200]}
Out[790]= {520, 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]= {520, 375}
In[815]:= Fam := {69, 64, 44, 15, 22}; Fam
Out[815]= {69, 64, 44, 15, 22}
In[816]:= {ActRemObj["Fam", "Rem"], Fam}
Out[816]= {"Remove", Removed["Fam"]}
In[817]:= ActRemObj["Fam", "Act"]; Fam
Out[817]= {69, 64, 44, 15, 22}
In[822]:= GrSu = 2011; {ActRemObj["GrSu", "Rem"], GrSu}
Out[822]= {"Remove", Removed["GrSu"]}
In[823]:= ActRemObj["GrSu", "Act"]; GrSu
Out[823]= 2011
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, не был найден в текущем по умолчанию каталоге, то вызов процедуры **ActRemObj** возвращается невычисленным.

Здесь вполне уместно отметить еще одно немаловажное обстоятельство, отличающее *Maple* от *Mathematica* в контексте процедурного (модульного) программирования. Так, если в *Maple* определено несколько одноименных процедур, активной является лишь *последняя* в цепочке вычисления их определений, тогда как в случае *Mathematica* имя процедуры не является определяющим, а процедуры различаются их *заголовками*, как это иллюстрирует следующий фрагмент. Более того, при вызове процедур в качестве выбранной принимается та *первая* из списка одноименных процедур, определяемого функцией **Definition**, на которой допустимым кортеж фактических аргументов. Это обстоятельство следует учитывать при программировании и оно было учтено нами в вышеперассмотренной процедуре **ActRemObj** использованием функции **ReadList**.

```

> A:= proc(x) local a; a:=420; x + a end proc: A:= proc(x,y) local a; a:=75; x + y + a end proc:
> eval(A); => proc(x, y) local a; a := 75; x + y + a end proc
In[601]:= A[x_] := Module[{a = 420}, x + a]; A[x_, y_] := Module[{a = 75}, x + y + a];
          A[x_/, IntegerQ[x]] := Module[{a = 420}, x + a];
In[602]:= Definition[A]
Out[602]= A[x_/, IntegerQ[x]] := Module[{a = 420}, x + a]
          A[x_] := Module[{a = 420}, x + a]
          A[x_, y_] := Module[{a = 75}, x + y + a]
In[603]:= {A[100], A[100, 200], A[420.75]}
Out[603]= {520, 375, 840.75}

```

В целях 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*, активизированных в текущем сеансе. Следующий фрагмент представляет исходный текст процедуры и ее применение.

```
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_]"}

```

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* подобных процедур в строчном формате. Следующий фрагмент представляет исходный текст процедуры и пример ее применения. В ряде приложений данные средства представляются нам достаточно полезными.

```
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[{}, Length[x]]
In[912]:= {ScanLikeProcs[], ScanLikeProcs[S], S}
Out[912]= {{G, ScanLikeProcs, V}, {G, ScanLikeProcs, V}, {"G[x_;/ PrimeQ[x]]", "G[x_]",
    "G[x_, y_]", "G[x_, y_;/ ListQ[y], z_]"}, {"V[x_;/ ListQ[x]]", "V[x_]"}]}

```

Следующий фрагмент представляет аналог рассмотренной процедуры **ActRemObj** в *Maple* с использованием аналогичного алгоритма, но с учетом особенностей пакета. В результате вызова **ActRemObj('x', {"Act" | "Rem"})** делается {восстановление | удаление} относительно текущего сеанса объекта с именем *x*; более того, фактический аргумент при вызове кодируется в *невычисленном* формате, т.е. 'x'. Успешное удаление объекта из текущего сеанса возвращает значение "Remove", тогда как восстановление объекта в текущем сеансе пакета возвращает значение "Activate". Если же файл, содержащий определение удаленного объекта, не был найден в каталоге пакета, вызов процедуры **ActRemObj** возвращается невычисленным. Нижеследующий фрагмент представляет исходный текст процедуры **ActRemObj** наряду с примерами ее применения.

```

> ActRemObj := proc(x::uneval, y::{ "Rem", "Act"})
  local a, b, c;
  a := cat(convert(eval(x), string)); b := cat(kernelopts(mapledir), "\\", a, ".$ArtKr$");
  if y="Act" then if isFile(b) then read b; remove(b); "Activate" else 'procname(args)' end if
  else
    c := cat(a, " := ", convert(eval(parse(convert(x, string)[2 .. -2])), string), ":");
    writeline(b, c), close(b); eval(parse(cat("unassign(", convert(x, string), ")"))), "Remove"
  end if
end proc;

> x := 420.75: ActRemObj('x', "Rem"), x;  => "Remove", x
> ActRemObj('x', "Act"), x;  => "Activate", 420.75

```

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, ToExpression["Remove[" <> ToString1[a] <> ""]];
  Map[ToExpression, c]; "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 текст 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[620]:= 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[621]:= Select[Names["*"], ProcQ[ToExpression[#]] &]
Out[621]= {"G", "V"}
In[622]:= Definition[G]
Out[622]= 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[623]:= Definition[V]
Out[623]= V[x_;/ ListQ[x]] := Module[{}, Length[x]]
          V[x_] := Module[{}, x^2]
In[624]:= 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[625]:= MdP[]
Out[625]= {"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.

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[84]:= 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]];
    If [a == {}, True, 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[85]:= {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[85]= {True, True, False}
In[86]= {HeadingQ["D[x_, y_ /; ListQ[y], z_:75, h]"], HeadingQ["[x_, y_ /; ListQ[y], z:75]"]}
Out[86]= {False, False}
In[87]:= {HeadingQ["D[]"], HeadingQ["Z[x_]"], HeadingQ["Q[x_]"]}
Out[87]= {True, True, True}
In[88]:= {HeadingQ["D[_x_]"], HeadingQ["Z[x_]"], HeadingQ["Q[x_]"]}
Out[88]= {False, True, True}
In[89]:= {HeadingQ["D[x_, y_ /; ListQ[y], z_:75, h]"], HeadingQ["V[x_, y_ /; ListQ[y], z_]"]}
Out[89]= {False, True}

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, 13], Agn[42, 47, 75]}
Out[2544]= {196, 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.

```

> Agn := proc(x, y, z) [args], args[k]$k=1..nargs end proc: Agn(3, 4, 5); => [3, 4, 5], 3, 4, 5
In[657]:= ProcCall[] := Block[{}, ToExpression[StringReplace[ToString[DefFunc[
    $ProcName]], "ProcCall[]" -> ToString[Map[ToExpression, Args1[$ProcName]]]]]]
In[658]:= ProcCall[]
SetDelayed::shape: Lists {} and Block[{}, ToExpression[StringReplace[ToString[DefFunc[
    $ProcName]], {} -> ToString[ToExpression/@Args1[$ProcName]]]]] are not the same shape.
Out[658]= $Failed
In[659]:= Mn[x_, y_ /; PrimeQ[y], z_, h_, g_ /; PrimeQ[g]] := Module[{args = ProcCall[],
    {args, Length[args], Quiet[Take[args, {1, Length[args]}]]}]
In[660]:= Mn[42, 2, 47, 67, 13]
Out[660]= {Null, 0, Take[Null, {1, 0}]}
In[661]:= Mn[42, 2, 47, 67, 13]
Out[661]= {{42, 2, 47, 67, 13}, 5, {42, 2, 47, 67, 13}}
In[662]:= DefFunc[Mn]
Out[662]= 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[663]:= Mn[75, 7, 420, 15, 41]
Out[663]= {{75, 7, 420, 15, 41}, 5, {75, 7, 420, 15, 41}}
In[664]:= Agn[] := Module[{args = ProcCall[], {args, Length[args]}]
In[665]:= Agn[]
Out[665]= {Null, 0}
In[666]:= Agn[]
Out[666]= {}, 0
In[667]:= Avz[x_] := Module[{args = ProcCall[], {args, Length[args]}]
In[668]:= Avz[75]
Out[668]= {Null, 0}
In[669]:= Avz[69, 64, 44, 15, 22, 6]
Out[669]= {{69, 64, 44, 15, 22, 6}, 6}

```

Как иллюстрирует *первый* пример предыдущего фрагмента, *Maple* располагает рядом процедурных переменных, рассмотренных выше, которые обеспечивают, в частности,

возможность получать список переданных ей 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, содержащей в определении вызов **ProcCall[]**, играет в определенном контексте смысл *инициализационного*, в то время как все последующие в текущем сеансе обеспечивают корректное выполнение *основной* процедуры. Фрагмент представляет *исходный* код процедуры **ProcCall** и примеры ее применения. При этом, для получения в процедуре списка фактических аргументов в точке ее вызова можно обойтись и без **ProcCall**, для чего можно использовать в теле процедуры конструкцию формата *{последовательность формальных аргументов без приписанных им шаблонов}*, как было показано выше, или использовать стандартную функцию **With**, как следующий довольно простой фрагмент достаточно 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.

Следующая простая процедура 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 процедуры и ее применение.

```
In[815]:= ExprQ[W_;/ StringQ[W]] := If[Quiet[ToExpression[W]] === $Failed, False, True]
In[816]:= Map[ExprQ, {"(a+b)", "(a^^c+b)", "a\\b+c", "a**b+cos[x]/c"}]
Out[816]= {True, False, False, True}
```


Процедура **ExprQ** оказывается довольно полезной при программировании процедур (*и, прежде всего, системного характера*) для обработки различного рода выражений.

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[1]:= Affiliate[x_ /; StringQ[x]] := Module[{a = Quiet[Context[x]]},
      If[ToString[a] === "Context[" <> x <> "]", "Undefined", a]]
In[2]:= g = 64; Map[Affiliate, {"ProcQ1", "Sin", "g", "Z", "Affiliate"}]
Out[2]= {"AladjevProceduresAndFunctions", "System", "Global", "Undefined", "Global"}
In[3]:= {V, G = 75, 75}; Map[Affiliate, {"V", "G", "75", "Sin[15]", "Q", "Map"}]
Out[3]= {"Global", "Global", "Undefined", "Undefined", "Undefined", "System"}
```

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 иллюстрирует простой пример:

```
> map(whattype, [a+b, 5, a/b, proc() end proc, G(6)]); => [+ , integer, *, procedure, function]
```

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[914]:= 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[915]:= t = 420; x = 75; y := 42.47; z = a + b; Map[WhatType, {"Kr", "x", "y", "z", "ProcQ",
      "Sin", "F[r]", "WhatType"}]
Out[915]= {"Undefined", "Integer", "Real", "Plus", "Module", "System", "F", "Module"}
In[916]:= Map[WhatType, {"a^b", "a**b", "3+5*I", "{42, 47}", "a&&b", "a->b"}]
Out[916]= {"Power", "NonCommutativeMultiply", "Complex", "List", "And", "Rule"}
In[917]:= Map[WhatType, {"a_/b", "a<>b", "a<->b", "a | b", "a`b"}]
Out[917]= {"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.

Последовательности выражений (или *просто последовательности*) в среде пакета *Maple* образуются на основе оператора *занятой* (,) 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. Следующий фрагмент представляет ряд процедур, обеспечивающих работу с объектом **Seq[x]** как последовательностью элементов *x*. В частности, процедура **SeqToList[x]** обеспечивает конвертирование *Seq*-объекта *x* в список, вызов процедуры **ListToSeq[x]** обеспечивает конвертирование списка *x* в *Seq*-объект, вызовы процедуры **SeqIns[x, y, z]** возвращают результат вставки в *Seq*-объект *x* элемента *y* (*список, Seq-объект, и др.*) в соответствии с заданной позицией *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 и др.*). Средства для работы с *Seq*-объектами могут быть расширены, обеспечивая пользователя полезным программным инструментарием.

```
In[683]:= A := Seq[a, b, c, d, h, r, x, y, z]
In[684]:= SeqQ[x_] := Block[{a = ToString[x]}, If[StringLength[a] >= 4 &&
StringTake[a, {1, 4}] == "Seq[" && StringTake[a, {-1, -1}] == "]", True, False]]
In[685]:= Map[SeqQ, {A, {a, b, c}, Agn}]
Out[685]= {True, False, False}
In[686]:= SeqToList[x_ /; SeqQ[x]] := ToExpression["{" <> StringTake[ToString[x],
{5, -2}] <> "}"]
In[687]:= SeqToList[A]
Out[687]= {a, b, c, d, h, r, x, y, z}
In[688]:= ListToSeq[x_ /; ListQ[x]] := ToExpression["Seq[" <> StringTake[ToString[x],
{2, -2}] <> "]" ]
In[689]:= ListToSeq[{42, 47, 67, 15, 22}]
Out[689]= Seq[42, 47, 67, 15, 22]
In[690]:= SeqIns[x_ /; SeqQ[x], y_ /; 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[691]:= SeqIns[A, {75, 420}, 22]
Out[691]= Seq[a, b, c, d, h, r, x, y, z, 75, 420]
In[692]:= SeqIns[A, Seq[42, 47, 67, 15, 22], 5]
Out[692]= Seq[a, b, c, d, h, 42, 47, 67, 15, 22, r, x, y, z]
In[693]:= SeqDel[x_ /; SeqQ[x], y_] := Module[{a = SeqToList[x], b = If[SeqQ[y],
SeqToList[y], y]}, ListToSeq[Select[a, ! MemberQ[Flatten[{b}], #] &]]]
In[694]:= SeqDel[A, Seq[a, b, c, d]]
Out[694]= Seq[h, r, x, y, z]
```

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 текст процедуры и примеры 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[z]; {TwoHandQ[a | | b + c, p], p}
Out[841]= {True, Or}
```

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 подстрок.

```
In[715]:= 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[716]:= StringPosition1["abcdefghijklmnopstw", {a, c, h, w}]
Out[716]= {"a", {{1, 1}}, "c", {{3, 3}}, "h", {{8, 8}}, "w", {{12, 12}}}
```

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. {}.

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 текст 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 ;; 420, "Rhs", s], s}
Out[738]= {420, 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 выражение x на левую и правую части. Процедура **RhsLhs** довольно легко модифицируется в свете расширения анализируемых операторов **Head[x]**. Оставляем это читателю в качестве достаточно полезного практического упражнения.

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 тексты приведены 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[747]:= 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[748]:= StringTake1["ransianavzagnvsvartkr", {ian, agn, art}]
Out[748]= {"rans", "avz", "vsv", "kr"}
In[749]:= StringTake1["ransianavzagnvsvartkr", {ian, 420, art}]
Out[749]= {"rans", "avzagnvsv", "kr"}
In[750]:= StringTake1["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]
Out[750]= {"s", "avz", "vsv"}

In[751]:= 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[752]:= StringTake2["ransianavzagnvsvartkr", {ian, agn, art}]
Out[752]= {"rans", "avz", "vsv", "kr"}
In[753]:= StringTake2["ransianavzagnvsvartkr", {ian, 420, art}]
Out[753]= {"rans", "avzagnvsv", "kr"}
In[754]:= StringTake2["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]
Out[754]= {"s", "avz", "vsv"}
```

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

Программирование в пакетах Maple и Mathematica: Сравнительный аспект

```

In[901]:= {OverLap["123456789abcd", "3456789abcd", z], z}
Out[901]= {11, "3456789abcd"}
In[902]:= {OverLap["123456789abcd", "3456789abcd", z], z}
Out[902]= {OverLap[{"123456789abcd", "3456789abcd", "3456789abcd"}], "3456789abcd"}
In[903]:= Clear[z]; {OverLap["123456789abcd", "3456789abcd", z], z}
Out[903]= {11, "3456789abcd"}
In[904]:= OverLap["555555555", "123456789"]
Out[904]= $Failed

In[799]:= 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[802]:= LongestCommonSubString["abdfgavzagntrehgfgtjkt", "ghtavzagnderty"]
Out[802]= {"avzagn"}
In[803]:= LongestCommonSubString["abdfgtrehgfgtjkt", "ghtf2g6derty"]
Out[803]= {"d", "e", "f", "g", "h", "r", "t"}
In[804]:= LongestCommonSubString["abdfgtrehgfgtjkt", "Z123456789"]
Out[804]= {}

In[841]:= 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[842]:= LongestCommonSubSequence["abcdexyzthg", "uvabxycde"]
Out[842]= {"abcde"}
In[843]:= LongestCommonSubSequence["abcdeefg", "123456789"]
Out[843]= {}

In[854]:= PrefixQ[x_;/StringQ[x], y_;/StringQ[y]] := If[x == "", True,
    If[! StringFreeQ[y, x] && StringTake[y, StringLength[x]] == x, True, False]]
In[855]:= {PrefixQ["avz", "avzransian"], PrefixQ["", "avzransian"]}
Out[855]= {True, True}

In[856]:= SuffixQ[x_;/StringQ[x], y_;/StringQ[y]] := If[x == "", True,
    If[! StringFreeQ[y, x] && StringTake[y, -StringLength[x]] == x, True, False]]
In[857]:= {SuffixQ["avz", "avzransianavz"], SuffixQ["", "avzransian"]}
Out[857]= {True, True}

In[873]:= PalindromeQ[x_;/StringQ[x]] := If[x == StringReverse[x], True, False]
In[874]:= Map[PalindromeQ, {"123456789", "abcdeffedcba"}]
Out[874]= {False, True}

```

```

In[875]:= 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[876]:= Map[MaximalPalindromicSubstring, {"12345436", "abcdefgh"}]
Out[876]= {"34543"}, {"a", "b", "c", "d", "e", "f", "g", "h"}
In[888]:= IsPermutation[x_ /; StringQ[x]] := Block[{a = Characters[x], If[Length[a] ==
    Length[DeleteDuplicates[a]], True, False]]
In[889]:= Map[IsPermutation, {"abc", "abcba", "123456789", "1233456"}]
Out[889]= {True, False, True, False}
In[900]:= 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[901]:= Clear[h]; {IsMonotonic["abcdefgh", h], h}
Out[901]= {True, "Increase"}
In[902]:= Clear[h]; {IsMonotonic["987654321", h], h}
Out[902]= {True, "Decrease"}
In[903]:= IsMonotonic["qwertyuiopasdfghjklzxcvbnm"]
Out[903]= False
In[904]:= IsMonotonic[qwertyuiopasdfghjklzxcvbnm, k]
Out[904]= 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*. Вызов функции **SortQ[s]** возвращает значение True, если *s* является *посимвольно* отсортированной строкой, и значение False в противном случае. 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 и префиксом для *y*. Более того, вызов с тремя аргументами **OverLap[x, y, z]**, где *z* неопределенный символ, возвращает через *z* само наложение. При отсутствии *наложения* вызов процедуры возвращает \$Failed, в других ошибочных и особых ситуациях вызов процедуры возвращается *невычисленным*. При этом, в целом *Math*-процедура **OverLap** оказывается *реактивнее* своего *Maple*-аналога.

Между тем, вызов процедуры **LongestCommonSubString**[x, y], одноименной с *Maple*-процедурой, возвращает список подстрок максимальной длины, общих для строк x, y . Процедура **LongestCommonSubSequence**[x, y] подобна предыдущей, однако ее вызов возвращает список максимальных подпоследовательностей, общих для строк x и y . В то время как вызов функции **{PrefixQ | SuffixQ}**[x, y] возвращает True, если строка x – {префикс | суффикс} строки y , в противном случае возвращается значение False. В свою очередь простая функция **PalindromeQ**[x] возвращает True, если строка x – палиндром, и False в противном случае. С другой стороны, вызов **MaximalPalindromicSubstring**[x] возвращает список подстрок максимальной длины, являющиеся палиндромами. Вызов процедуры **IsPermutation**[x] возвращает True, если строка x – перестановка символов, ее составляющих, и False в противном случае; при этом, строка полагается перестановкой, если она содержит строго по одному вхождению составляющих ее символов. Наконец, вызов одноименной с *Maple*-процедурой 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 **IsMonotonic**[x, t] с 2-я аргументами в случае основного результата True через 2-й неопределенный символ t , возвращается порядок следования символов в строке x {"Increase", "Decrease"}. В случае пустой строки x вызов **IsMonotonic**[x, t] возвращает True; при этом, через t возвращается "Decrease". В остальных случаях вызов процедуры **IsMonotonic** возвращается невычисленным.

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.

Списки являются *центральными* структурами 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*, пакет *Mathematica* располагает целым рядом весьма эффективных средств работы со списочными структурами различного вида, в частности, большинство *встроенных* функций пакета имеют атрибут *Listable*, который определяет, что они эффективно применимы по отдельности как к каждому элементу списка, так и к соответствующим элементам каждого списка в случае их множества. В то же время все аргументы *List*-типа в функции такого типа должны быть *идентичной* длины, в противном случае идентифицируется ошибочная ситуация, как достаточно наглядно иллюстрируют первые четыре примера следующего фрагмента, а именно:

```
In[810]:= Sin[{a, b, c, d, h}]
Out[810]= {Sin[a], Sin[b], Sin[c], Sin[d], Sin[h]}
In[811]:= {x, y, z, h}/{a, b, c, d}
Out[811]= {x/a, y/b, z/c, h/d}
In[812]:= {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[812]= {x, y, z, h} + {a, b, c, d, g}
In[813]:= 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[814]:= ListOp[{x, y, z, h, w}, {a, b, c, d, e}, #1 + #2 &]
Out[814]= {a + x, b + y, c + z, d + h, e + w}
In[815]:= ListOp[{x, y, z, h, w}, {a, b, c}, #1^#2 &]
Out[815]= {x^a, y^b, z^c, h, w}
In[816]:= ListOp[{x, y, z, h, w}, {a, b, c, d, e}, #1 * #2 &]
Out[816]= {a x, b y, c z, d h, e w}
In[817]:= ListOp[{x, y, z, h, w}, {}, #1 * #2 &]
Out[817]= {x, y, z, h, w}
In[769]:= {True, False, True} && {True, True, False}
Out[769]= {True, False, True} && {True, True, False}
In[770]:= ListOp[{True, False, True}, {True, True, False}, #1 && #2 &]
Out[770]= {True, False, False}
In[771]:= ListOp[{True, False, True}, {True, True, False}, #1 || #2 &]
Out[771]= {True, True, True}
```

С целью устранения данного недостатка предлагается процедура **ListOp**[*x*, *y*, *z*], чей вызов возвращает список, элементами которого являются результаты применения **z**-процедуры/функции к соответствующим элементам списков *x* и *y*; при этом, в случае различных длин данных списков описанная процедура применяется к обоим спискам в рамках минимальной длины списков, оставляя остальные элементы большего списка без изменения. Фрагмент представляет исходный текст процедуры **ListOp** и примеры ее применения, довольно наглядно иллюстрирующие сказанное. Процедура **ListOp** в значительной степени предполагает в качестве третьего аргумента чистые функции, что в значительной степени позволяет расширить класс функций в качестве третьего аргумента, например, на логические функции, как иллюстрируют последние 3 примера.

В среде *Maple* определен тип индексированных выражений, что в целом ряде случаев дает возможность более эффективно обрабатывать индексированные выражения. Выше 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[806]:= 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[807]:= {IndexedQ[Quiet[15[[2 ;; 4]]], IndexedQ[Quiet[ArtKr[[15 ;; 22]]]]}
Out[807]= {False, True}
In[808]:= Index[x_/, IndexedQ[x]] := Module[{a = Quiet[ToString[x]], b, c},
      b = Flatten[StringPosition[a, "["];
      ToExpression[StringTake[a, {b[[2]] + 1, -3}]]]
In[809]:= {Index[Quiet[VSV[[15 ;; 22]]], Index[Quiet[SVEGAL[[75]]]]}
Out[809]= {15 ;; 22, 75}
In[810]:= Index[Quiet[420[[2 ;; 4]]]]
Out[810]= Index[420[[2 ;; 4]]]
In[811]:= H[1] = 75; H[2] = 47; H[6] = 15; H[9] = 22; H[15] = 42; H[22] = 69;
In[812]:= (H[6] + H[2])/(H[9] + H[22])
Out[812]= 435/91
In[813]:= ArrayInd[x_/, StringQ[x]] := Module[{a = StringSplit[ToString[InputForm[
      DefFunc[x]]], "\n \n"]}, If[a == {"Null"}, {}, a]]
In[814]:= Map[ArrayInd, {"H", "G"}]
Out[814]= {"H[1] = 75", "H[2] = 47", "H[6] = 15", "H[9] = 22", "H[15] = 42", "H[22] = 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 $\{ "H[n1]=c1", \dots, "H[nk]=ck" \}$, 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]
In[75]:= SymbolToList[x_;/; SymbolQ[x]] := Map[ToExpression, Characters[ToString[x]]]
In[76]:= SymbolToList[Art22Kr15]
Out[76]= {A, r, t, 2, 2, K, r, 1, 5}
> convert("Art22Kr15", list), convert(Art22Kr15, list);
      {"A", "r", "t", "2", "2", "K", "r", "1", "5"}, [Art22Kr15]
> SymbolToList := (x::symbol) -> map(parse, convert(convert(x, string), list));
> SymbolToList(Art22Kr15); => [A, r, t, 2, 2, K, r, 1, 5]
```

Оба пакета обеспечивают стандартные средства *конвертации* строк в списки, тогда как для обеспечения конвертации *символов* в списки в обоих пакетах можно предложить 2 одноименные функции **SymbolToList**, возвращающие результат *конвертации* символа в список. Конец предыдущего фрагмента представляет исходные коды функций.

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[873]:= 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[874]:= PartialSums[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}]
Out[874]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
In[875]:= GS = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; PartialSums["GS"]
Out[875]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
In[876]:= GS
Out[876]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
In[877]:= SV = {a, b, c, d, e, f}; PartialSums["SV"]
Out[877]= {a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
In[878]:= SV
Out[878]= {a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
```

Следующая процедура **Seq** служит аналогом одноименной процедуры пакета *Maple*.

```
In[898]:= 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[900]:= Seq[F[k], k, 15 ;; 22]
Out[900]= {F[15], F[16], F[17], F[18], F[19], F[20], F[21], F[22]}
In[901]:= Seq[F[t], t, 1 ;; 75 ;; 6]
Out[901]= {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[902]:= Seq[F[t], t, 100 ;; 90]
Out[902]= {F[100], F[99], F[98], F[97], F[96], F[95], F[94], F[93], F[92], F[91], F[90]}
```

```
In[903]:= Seq[F[t], t, 42.69 ;; 100 ;; 6.47]
Out[903]= {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[904]:= {Seq[F[k], k, 42 ;; 69 ;; 0], Seq[F[k], k, 42 ;; 69 ;; 6]}
Out[904]= {Seq[F[k], k, 42 ;; 69 ;; 0], {F[42], F[48], F[54], F[60], F[66]}}
In[905]:= Seq[F[k], k, 42 ;; 69 ;; -6]
Out[905]= 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.

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[420, {{3, 1}, {6, 0}, {9, 1}}], BitSet1[75, {4, 0}], BitSet1[74, {7, 1}]}
Out[927]= {485, 67, 75}
In[928]:= BitSet1[420, {{3, 1}, {6, 0}, {9, 2}}]
Out[928]= BitSet1[420, {{3, 1}, {6, 0}, {9, 2}}]
In[8]:= BitGet1[x___, n_/: IntegerQ[n] && n >= 0, p_/: IntegerQ[p] && p > 0 | | ListQ[p]]:=
    Module[{b = 1, c = {}, d, a = ToExpression[Characters[IntegerString[n, 2]]],
    h = If[ListQ[p], p, {p}], For[b, b <= Length[a], b++,
    c = Append[c, If[MemberQ[h, b], a[[b]], Null]]];
    If[! HowAct[x], x = Length[a], Null]; Select[c, ToString[#] != "Null" &]]
In[871]:= {BitGet1[h, 75, {1, 5, 7}], h, BitGet1[75, {1, 5, 7}], BitGet1[p, 420, {1, 3, 5, 7, 9}], p}
Out[871]= {{1, 0, 1}, 7, {1, 0, 1}, {1, 0, 0, 1, 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*. Во фрагменте представлен исходный код процедуры **BitGet1** и примеры ее применения, хорошо иллюстрирующие сказанное.

Списочная структура является одной из базовых в *Maple*, но еще в большей степени в *Mathematica*. Как *Maple*, так еще в большей степени *Mathematica* располагают весьма развитым набором средств обработки списочных структур. Одним из таких довольно важных средств является конвертация выражений в списки; для *Maple* такое средство имеет вид **convert**(*Ex*, *list*), тогда как *Mathematica* аналогичного средства не имеет, и в качестве его может выступить процедура, чей вызов **ToList**[*Ex*] возвращает результат конвертации выражения *Ex* в список. Следующий фрагмент представляет исходный код процедуры **ToList** с некоторыми примерами ее применения в сравнении с *Maple*.

```
In[833]:= ToList[expr_] := Module[{a, b, c = {}, d, k = 1, n}, 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[834]:= ToList[(a*Sin[x] + g[b])/(c + d) + (d + c)/(Cos[y] + h)]
Out[834]= {(c + d)/(h + Cos[y]), (g[b] + a Sin[x])/(c + d)}
In[835]:= ToList[(a*Sin[x] + g[b])*(c + d) + (d + c)/(Cos[y] + h)]
Out[835]= {(c + d)/(h + Cos[y]), (c + d) (g[b] + a Sin[x])}
> convert((a*sin(x) + g(b))/(c + d) + (d + c)/(cos(y) + h), list);
    [(a*sin(x) + g(b))/(c + d), (c + d)/(cos(y) + h)]
> convert((a*sin(x) + g(b))*(c + d) + (d + c)/(cos(y) + h), list);
    [(a*sin(x) + g(b))*(c + d), (c + d)/(cos(y) + h)]
```

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 применения.

```
In[989]:= 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[990]:= Map[Ind, {Sin[x], (a + b)/(c + d), Sin[x] + Cos[x], AG[a + b], SV[x, y, z]}]
Out[990]= {{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 типичными примерами ее применения.

```
In[820]:= 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[420]; 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[420]]]
In[821]:= OP[(a + b)/(c + g[d])]
Out[821]= {-1, a, b, c, d, g}
In[822]:= OP[Cos[x] + Sin[y]*Tan[z]]
Out[822]= {Cos, Sin, Tan, x, y, z}
In[823]:= OP[Sqrt[(a + b)/(c + d)] + Sin[x]*Cos[y]]
Out[823]= {-1, 1/2, a, b, c, Cos, d, Sin, x, y}
In[824]:= OP[(Log[(a + b)/(c + d)] + Sin[x]*Cos[y])/(G[h, g, t] - w^2)]
Out[824]= {-1, 2, a, b, c, Cos, d, g, G, h, Log, Sin, t, w, x, y}
In[825]:= Map[OP, {{Sin[x]}, G[h, g, t], A[m, p]/G[t, q]}]
Out[825]= {{Sin, x}, {g, G, h, t}, {-1, A, G, m, p, q, t}}
```

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", "x:", "x:\\"}. Вызовы формата **FreeSpaceVol[{" " | "..."}]** возвращают указанную выше информацию относительно текущего устройства прямого доступа.

```
In[924]:= FreeSpaceVol[x_/, StringQ[x]] := Module[{a = "C:\\$$Art$$Kr$$txt", b, c},
    b = Run["Dir", If[SymbolQ[x], x <> ":\\", 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[925]:= FreeSpaceVol["F:"]
Out[925]= "Drive F: is not ready"
In[926]:= Map[FreeSpaceVol, {"d:\\", "C", "G:"}]
Out[926]= {{34036736000, bytes}, {100766498816, bytes}, {5426442240, bytes}}
```

Фрагмент представляет исходный текст процедуры с примерами ее применения.

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.

```
In[859]:= 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[860]:= ListListGroup[{{75, 2}, {420, 6}, {15, 2}, {22, 2}, {69, 6}}, 2]
Out[860]= {{{75, 2}, {15, 2}, {22, 2}}, {{420, 6}, {69, 6}}}
In[861]:= ListListGroup[{{75, 2}, {420, 6}, {15, 2}, {22, 2}, {69, 6}}, 6]
Out[861]= ListListGroup[{{75, 2}, {420, 6}, {15, 2}, {22, 2}, {69, 6}}, 6]
```

Фрагмент представляет исходный код процедуры с примерами ее применения.

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[955]:= 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[956]:= P[x_, y_] := Module[{a, P1}, P1[z_, h_] := Module[{n}, z^2 + h^2]; x*y + P1[x, y]]
In[957]:= x = DefFunc1[ToString[P]]; {ExprOfStr[x, 44, 1, {" ", ";", ", ", ""}],
    ExprOfStr[x, 39, -1, {" ", ";", ", ", ""}]}
Out[957]= {"Module[{n}, z^2 + h^2]", "P1[z_, h_]"}
In[958]:= y = "123456789abcdefghAVZ_2011"; ExprOfStr[y, 75, -1, {" ", ";", ", ", ""}]
Out[958]= ExprOfStr["123456789abcdefghAVZ_2011", 75, -1, {" ", ";", ", ", ""}]
In[959]:= ExprOfStr[x, 10, 1, {" ", ";", ", ", ""}]
Out[959]= $Failed
```

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 на недопустимых фактических аргументах возвращается невычисленным. Примеры иллюстрируют сказанное.

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[929]:= 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[930]:= {F[Sequence[{x, y, z}]], F[Sequences[{x, y, z}]], F[Sequences[x, y, z]]}
Out[930]= {F[{x, y, z}], F[x, y, z], F[x, y, z]}
```

На основе ранее представленной процедуры **ExprOfStr** несложно получить решение достаточно интересной задачи по определению списка *подпроцедур* для произвольной процедуры *Module*-типа. Данную задачу успешно решает процедура **SubProcs** $[P]$, чей вызов возвращает вложенный 2-элементный список *ListList*-типа, чей *первый* элемент представляет *список заголовков* подпроцедур, составляющих *главную* процедуру P типа *Module*, тогда как второй элемент – список сгенерированных имен всех подпроцедур, включая главную процедуру P , и активизированных в текущем сеансе.

```
In[6]:= 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[7]:= 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[8]:= P[75, 420]
Out[8]= 33490188900
In[9]:= SubProcs[P]
Out[9]= {"P[x_, y_]", "P1[z_, h_]", "P2[z_]", "P3[h_]", {P$4438, P1$4443, P2$4449, P3$4454}}
In[10]:= DefFunc[P2$4449]
Out[10]= P2$4449[z_] := Module[{P3}, P3[h_] := Module[{}, h^4]; P3[z]]
```

Между элементами обоих списков имеет место взаимно-однозначное соответствие. В представленном выше фрагменте приведен исходный текст процедуры и достаточно типичные примеры ее применения. Процедура допускает интересные расширения.

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[864]:= 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[865]:= {P[75, 420], P[7, 420]}
Out[865]= {False, 179389}
In[866]:= Map[ProcQ, {P, F}]
Out[866]= {True, False}
In[867]:= 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[868]:= {P[75, 420], P[7, 420]}
Out[868]= {True, 179389}
In[869]:= Map[ProcQ, {P, F}]
Out[869]= {True, True}
> P:= proc(x, y) local a, b, c, F; F := proc(x, y) local a, b; x^2 + y^2 end proc; x*y + F(x, y);
                                         ParProc(F) end proc;
> P(75, 420);  => [ Arguments = (x, y)
                  locals = (a, b) ]
> ParProc(F);
Error, (in ParProc) <F> is not a procedure and not a module
```

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*.

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[884]:= 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[885]:= Locals1[ProcQ]
Out[885]= {"a", "b", "c", "d", "h"}
```

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.

Естественным дополнением к **Locals1** выступает процедура, вызов которой **Globals**[*P*] возвращает список *глобальных* переменных в строчном формате процедуры *P*. Более того, из вышерассмотренных *наших* средств процедура использует четыре процедуры **Locals1**, **DefFunc**, **ExprOfStr** и **ListListQ**. 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 со следующей процедурой **MinusList**[*x, y*], чей вызов возвращает

результат вычитания списка y из списка x , состоящего в удалении из списка x первых вхождений элементов из списка y . Следующий фрагмент представляет исходный код процедуры **MinusList** наряду с типичным примером ее применения.

```
In[894]:= 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[895]:= 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[895]= {"a", "P3", "b", Avz, "m", Kr, Agn, "n", "P3", Art, g}
```

Данная процедура представляет и самостоятельный интерес при работе со списками. Тогда как один из достаточно простых вариантов обобщения процедуры **Globals** для случая *вложенных* процедур представлен довольно простой процедурой **Globals1**, чей вызов **Globals1[P]** возвращает список глобальных переменных в строчном формате P процедуры; при этом, в качестве P может выступать как процедура, не содержащая в своем теле подпроцедур различного уровня вложенности, так и процедура, имеющая такие подпроцедуры. Следующий фрагмент представляет исходный код процедуры **Globals1** наряду с наиболее типичными примерами ее применения.

```
In[887]:= 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[888]:= 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[888]= {"Q", "T", "V"}

In[889]:= 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[889]= {"gs", "h", "t", "z"}
```

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[929]:= 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]]] <> " = " <>
ToString1[ Res]], Null]; If[Res == {}, False, True]]
```

```
In[930]:= {SubsProcQ[P, P3, x], x}
```

```
Out[930]= {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 *напутства* 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 текстом, представленным ниже:

```
In[624]:= ProcsAct[] := Module[{a = Names["*"], b = Names["System`*"], c, d = {}, k, h, x = t},
c = Select[Select[a, ! MemberQ[b, #] &], ProcQ[ToExpression[#]] &];
For[k = 1, k <= Length[c], k++, h = c[[k]]; Clear[t];
ProcQ1[ToString[h], t]; d = Append[d, {ToExpression[h], t}]; t = x; d]
In[625]:= ProcsAct[]
Out[625]= {{ActiveProcess, "Module"}, {ActRemObj, "Module"}, {Adrive, "Module"}, ...
{VarExch, "Module"}, {VarExch1, "Module"}, {Ver, "Block"}, {WhatObj, "Module"}}
```

Вызов процедуры **ProcsAct[]** возвращает вложенный список, 2-элементные подписки которого своим первым элементом определяют имена объектов, активизированных в текущей сессии, тогда как вторые элементы определяют их типы в разрезах **"Module"**, **"Block"**, **"DynamicModule"** и **"Others"**. Процедура представляет вполне определенный интерес в целом ряде приложений, прежде всего, системного характера.

6.2.5. 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 λ -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 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 *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 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 λ -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 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;

– *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 С указатели на функцию вполне могут быть использованы для обеспечения эффекта функций высшего порядка. Более того, функции высшего порядка и *отложенная* списочная структура реализованы также и в библиотеках языка C++. В языке C# возможно использовать λ -функции для создания программ в функциональном стиле. Некоторые языки, имеющие довольно развитые средства метапрограммирования, позволяют создать объекты данных и программные конструкции, специфичные для функционального стиля, предоставляя возможность на их основе писать достаточно эффективные функциональные программы.

И если *Mathematica*-язык, исповедующий смешанную концепцию *функционального и процедурного* программирования, поддерживает функциональное программирование, то *Maple*-язык, исповедующий концепцию сугубо процедурного программирования, в то же время допускает целый ряд элементов функционального программирования. 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* 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[116]:= {Plus[#2 &][a, b, c], Plus[## &][a, b, c, d, f]], Plus[##4 &][a, b, c, d, g, h, p]]}
Out[116]= {b, a + b + c + d + f, d + g + h + p}
In[117]:= {Map[##^2 &, {a, b, c, d}], Map[##^2 &, {a + b + c + d}]}
Out[117]= {{a^2, b^2, c^2, d^2}, {(a + b + c + d)^2}}
In[118]:= {Nest[##^2 &, {a, b, c, d}, 3], Nest[##^2 &, {a + b + c + d}, 3]}
Out[118]= {{a^8, b^8, c^8, d^8}, {(a + b + c + d)^8}}
In[119]:= G[##2, ##4] &[x, y, z, h, d, t]
Out[118]= G[y, z, h, d, t, h, d, t]
In[119]:= #2 &[a, b, c] + ##2 &[x, y, z, h, d, w, v]
Out[119]= 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[40]:= Art := Function[{x, y}, x*Sin[y]]
In[41]:= Kr := (#1^2 + #2^4) &
In[42]:= {Art[68, 42.68], Kr[14, 21]}
Out[42]= {-65.5636, 194677}
```

```
In[43]:= {Function[{x, y}, x*Sin[y]][68, 42.68], (#1^2 + #2^4) &[14, 21]}
Out[43]= {-65.5636, 194677}
In[44]:= Map[H[#]*G[#] + w, {x, y, z}]
Out[44]= {(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 терминах языка *Maple*. Между тем, пакет *Mathematica* не располагает стандартными средствами для тестирования поименованных функций и данную задачу решает достаточно простая процедура **FunctionQ[x]**, чей исходный текст и примеры использования представляет нижеследующий достаточно прозрачный фрагмент.

```
In[142]:= F := Function[{x, y, z, h}, Sqrt[x^2 + y^2 + z^2 + h^2]]
In[143]:= F[42, 68, 72, 420]^2
Out[143]= 187972
In[144]:= Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &
In[145]:= Art[42.68, 420.72];
Out[145]= -10.7004
In[146]:= RANS := 2011; IAN[x_] := Module[{}, x^2]; Sv := 43
In[147]:= FunctionQ[x_] := Module[{a, b}, a = ToString[InputForm[x]];
    If[StringFreeQ[a, "#1"] == False && StringTake[a, {-3, -1}] == " & ", True,
    a = Quiet[Check[ToString[InputForm[Definition[x]]], 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, FunctionQ[43], FunctionQ[IAN], 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 обеих процедур на функции *S[x, y]* и ее *&*-аналоге.

```
In[854]:= PureFuncQ[F_] := StringTake[ToString[F], {-3, -1}] == " & " && !
    StringFreeQ[ToString[F], "#"]
In[855]:= 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[856]:= 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]]], ToString[d]]]; ToExpression[d]]
In[857]:= S[x_, y_] := a[x]*b[y] - c[x, y]
In[858]:= Map5[S, {{x1, y1}, {x2, y2}, {x3, y3}}]
Out[858]= {a[x1] b[y1] - c[x1, y1], a[x2] b[y2] - c[x2, y2], a[x3] b[y3] - c[x3, y3]}
In[859]:= Map6[a[#1]*b[#2] - c[#1, #2] &, {{x1, y1}, {x2, y2}, {x3, y3}}]
Out[859]= {a[x1] b[y1] - c[x1, y1], a[x2] b[y2] - c[x2, y2], a[x3] b[y3] - c[x3, y3]}
```

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 $t_j \in \{_Integer, _Real, Complex, True, False\}$; 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[877]:= V := Compile[{{x, _Real}, {y, _Real}}, (x + y)^2]; Art := Function[{x, y}, x*Sin[y]];
    Kr := (#1^2 + #2^4) &; GS[x_Integer, y_Integer] := Sin[75] + Cos[42];
    Sv[x_Integer, y_Integer] := x^2 + y^2; S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
In[878]:= 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[879]:= Map[FunctionQ1, {"V", "S", "Art", "Kr", "Pi", "75", "GS", "Sv"}]
Out[879]= {True, True, True, True, False, FunctionQ1["75"], False, False}
```

```

In[880]:= QFunction[x_ /; SymbolQ[x]] := Module[{a = StringLength[x],
    b = ToString[InputForm[DefFunc[x]]], c}, If[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[881]:= Map[QFunction, {"V", "S", "Art", "Kr", "Pi", "42.75", "GS", "Sv"}]
Out[881]= {True, True, True, True, "Pi", QFunction["42.75"], 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 применения.

В принципе, чистые функции просто определяются и в среде пакета *Maple*, для этого вполне достаточно, чтобы их тела не содержали глобальных переменных; при этом, и здесь допускается возможность обращения к ним не по их именам, а применением их к заданным фактическим аргументам. В *Maple* для таких целей служат конструкции следующего общего формата, а именно:

$$\text{proc}(\text{formal args}) \text{ <Тело процедуры> end proc } (\text{actual args})$$

$$((\text{formal args}) \rightarrow \text{<Тело функции>}) (\text{actual args})$$

Более того, на основе таких конструкций можно определять различные конструкции, в основе которых лежит указанный принцип обращения к функциям (*процедурам*) не по их именам, а применением их к заданным фактическим аргументам. Следующий простой пример достаточно прозрачно иллюстрирует сказанное на основе довольно простого расширения стандартной встроенной функции *map* пакета *Maple*:

```

> proc(x, y, z) x^2 + y^2 + z^2 end proc (42, 47, 67);  => 8462
> ((x, y, z) -> x^2 + y^2 + z^2) (42, 47, 67);  => 8462
> Map := proc(F::{function, procedure}, L::listlist) local a, b, c;
    if nops(L[1]) <> nops([op(1, eval(P))]) then error "mismatching of number
    of formal and actual arguments" end if;
    assign67(a = convert(eval(F), string), b = NULL);
    for c in L do b := b, parse(cat(a, "(", convert(c, string)[2 .. -2], ");"), 'statement') end do;
    b
end proc:
> P := proc(x, y, z) local a; x*y + y*z end proc: T := (x, y, z) -> x*y + y*z:
> Map(P, [[a, b, c], [h, t, w], [68, 63, 43]]);  => a b + b c, h t + t w, 6993
> Map(T, [[a, b, c], [h, t, w], [68, 63, 43]]);  => a b + b c, h t + t w, 6993
In[19]:= Mapp[F_, L_] := ToExpression[ StringReplace[ToString[Map[F, L, {1}]],
    {"[" -> "[", "]" -> "]"}]]

```

```
In[20]:= L := {{a, b, c}, {h, t, w}, {68, 63, 43}}; Mapp[F, L]
Out[20]= {F[a, b, c], F[h, t, w], F[68, 63, 43]}
In[21]:= G[x_, y_, z_] := x^2 + y^2 + z^2; Mapp[G, L]
Out[21]= {a^2 + b^2 + c^2, h^2 + t^2 + w^2, 10442}
```

В качестве формальных аргументов процедуры **Map** выступают имя процедуры либо функции, а в качестве второго *вложенный* список, чьими элементами являются списки значений передаваемых функции фактических аргументов. В завершение фрагмента приведен *исходный* текст простой чистой функции **Mapp**, обобщающей стандартную функцию **Map** пакета *Mathematica* на число аргументов большее, чем 1.

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_, ...]:=<Тело>**, обращаясь впоследствии к ней по имени **G**. Тогда как при однократном применении функцию можно задавать в ее *короткой* форме, т.е. в формате **<Тело> &**. Следующий весьма простой фрагмент иллюстрирует вышесказанное, а именно:

```
In[180]:= G[x_] := x^2; a := {}; For[k = 0, k < 21, k++, a = Append[a, G[k]]]; a
Out[180]= {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
In[181]:= #1^2 &[14]
Out[181]= 196
In[182]:= a := {}; For[k = 0, k < 21, k++, a = Append[a, #1^2 & [k]]]; a
Out[182]= {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

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 <i>G</i> -функции
G[x_String]	- <i>//</i> - строка <i>//</i> - <i>//</i> - <i>//</i> - <i>//</i> -

G[x_Integer]	-	//-	целое	//-	//-	//-
G[x_Complex]	-	//-	комплексное	//-	//-	//-
G[x_Number]	-	//-	число	любого типа	//-	//-
G[x_Real]	-	//-	действительное	с плавающей точкой		//-
G[x_;/ ЛУ]	-	для x-аргумента допускается значение, для которого некоторое логическое условие (ЛУ) принимает значение True (Истина)				
G[x_ША /; ЛУ]	-	для x-аргумента с допустимым типом, соответствующим ША, выбирается такое значение, для которого логическое условие (ЛУ) принимает значение True.				

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 производятся два ее вызова с различными фактическими значениями для ее формальных аргументов, а именно:

```
In[225]:= GS[x_Integer, y_]; MatrixQ[y, z_String] := y^(x + StringLength[z])
```

```
In[226]:= GS[6.8, {1, 2, 3}, "RANS_IAN"]
```

```
Out[226]= GS[6.8, {1, 2, 3}, "RANS_IAN"]
```

```
In[227]:= GS[2, {{68, 63}, {14, 21}}, "IAN_72"]
```

$$\text{Out}[227] = \{\{457163239653376, 248155780267521\}, \{1475789056, 37822859361\}\}$$

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 – (2×2) -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. Второй 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[511]:= RANS[x_, n_Integer, y_List] := {{x, n, y}, a*x^(n + Length[y])}
In[512]:= RANS[2, 3, {68, 63, 43}]
Out[512]= {{2, 3, {68, 63, 43}}, 64 a}
In[513]:= RANS[x_, n_Integer, y_List] := {x, n, y, a*x^(n + Length[y])}
In[514]:= RANS[2, 3, {68, 63, 43}]
Out[514]= {2, 3, {68, 63, 43}, 64 a}
In[515]:= RANS[x_, n_Integer, y_List] := {x, n, y; a*x^(n + Length[y])}
In[516]:= RANS[2, 3, {68, 63, 43}]
Out[516]= {2, 3, 64 a}
In[517]:= RANS[x_, n_Integer, y_List] := {x, n; y; a*x^(n + Length[y])}
In[518]:= RANS[2, 3, {68, 63, 43}]
Out[518]= {2, 64 a}
In[519]:= RANS[x_, n_Integer, y_List] := {x; n; y; a*x^(n + Length[y])}
In[520]:= RANS[2, 3, {68, 63, 43}]
Out[520]= {64 a}
In[521]:= RANS[x_, n_Integer, y_List] := {a*x^(n + Length[y])}
In[522]:= RANS[2, 3, {68, 63, 43}]
Out[522]= {64 a}
In[523]:= RANS[x_, n_Integer, y_List] := {F = a*x^(n + Length[y]); d = Sqrt[f]; R = d*F}
In[524]:= RANS[2, 3, {68, 63, 43}]
Out[524]= {64 a Sqrt[f]}
In[525]:= {F, d, R}
Out[525]= {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 *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. Сказанное иллюстрирует следующий достаточно простой фрагмент:

```

In[83]:= GS[x_List, y_/; IntegerQ[y]] := First[{z=Length[x]; R=x+IntegerDigits[y]; R^z}]
In[84]:= GS[{68, 63, 43}, 420]
Out[84]= {373248, 274625, 79507}
In[85]:= z = 3; R = {2, 4, 6}; {Options[GS], Attributes[GS]}
Out[85]= {{}, {}}
In[86]:= SetAttributes[GS, {Protected, Listable}]; Attributes[GS]
Out[86]= {Listable, Protected}
In[87]:= GS = 2011; N[Sqrt[GS]]
Set::wrsym: Symbol GS is Protected. >>
Out[87]= 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=2011*, 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.

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[{68, 63, 43, 14, 21}], AR["RANS - IAN - 2011"], AR[68]}
Out[97]= {377.136, {68, 63, 43, 14, 21}, 17, 314432}
In[98]:= AR[x_] := 2*x + 68; AR[x_String] := N[Sqrt[StringLength[x]]]
In[99]:= {AR[19.42], AR["RANS - IAN - 2011"]}
Out[99]= {106.84, 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[100]:= Sum1[0] := 0; Sum1[n_Integer] := Sum1[n - 1] + n
In[101]:= Sum1[68]; Sum1[2011]
$RecursionLimit::reclim: Recursion depth of 256 exceeded. >>
Out[101]= 2011 + (2010 + (2009 + (2008 + (2007 + (2006 + (2005 + (2004 + (2003 .... ))))))))
In[102]:= $RecursionLimit = 3000; Sum1[2011]
Out[102]= 2023066
```

В данном фрагменте определяется рекурсивная *Sum1*-функция (с начальным условием рекурсии *Sum1[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*] \Rightarrow **F**[*x*]. Полезным данный оператор может быть и при организации функций пользователя, позволяя передавать в качестве фактических значений для ее формальных аргументов заголовки функций и/или их формальные аргументы. Между тем, данное средство в целом не представляет особого интереса, что и побудило вывести его за рамки пакета. 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 иллюстрирует вышесказанное, а именно:

```
In[85]:= FunCompose[L_List, 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[86]:= FunCompose[{F, G, H, T, W, Q, V, U}, Sin[z]]
Out[86]= F[G[H[T[W[Q[V[U[Sin[z]]]]]]]]]
In[87]:= {FunCompose[{Sin, Cos, Log}, 19.42], FunCompose[{Sin, Cos, Log, Sqrt}, 2011.72]}
Out[87]= {-0.833093, -0.709578}

> FunCompose := proc(L::list(symbol), x::anything) local a, b, c;
      a, b := cat("(", convert(x, 'string'), ")"), map(convert, Rssl(L), 'string');
      for c in b do a := cat("(", c, a, ")") end do;
      parse(a, 'statement')
end proc;

> FunCompose([F, G, H, T, W, Q, V, U], sin(z));  $\Rightarrow$  F(G(H(T(W(Q(V(U(sin(z))))))))))
> FunCompose([sin, cos, ln, sqrt], 2011.72);  $\Rightarrow$  -0.7095783435
```

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[4]:= SV[z_] := F@z + z^3; VSV[Id_] := Module([], Id@z); {VSV[F, h], SV[68]}
Out[4]= {F[h], 314432 + F[68]}
```

а также целый ряд им подобных 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[397]:= Gs[x_List | x_String] := x
In[398]:= {Gs[avzransian], Gs["avzransian"], Gs[{a, v, z, r, a, n, s, i, a, n, 7, 2}]}
Out[398]= {Gs[avzransian], "avzransian", {a, v, z, r, a, n, s, i, a, n, 7, 2}}
In[399]:= Sv[x_] := Module[{}, If[ListQ[x], Length[x], If[StringQ[x], StringLength[x],
                                                                    HoldForm[Sv[x]]]]]
In[400]:= {Sv[avzransian], Sv["avzransian"], Sv[{a, v, z, r, a, n, s, i, a, n, 7, 2}]}
Out[400]= {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 **"__"**, например:

```
In[102]:= G[x_, y__] := a*x + b + y
In[103]:= {G[x, y], G[x, y, z, h, g, v, d, s, w, r, t]}
Out[103]= {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 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[6]:= {G[43], G[42.68], G[14/21], G[], F[h]}
Out[6]= {1849, G[42.68], G[2/3], 4624, F[h]}
In[7]:= SV[x_Integer: 43, y_List: {68, 63, 43}] := (x + First[y])^2
In[8]:= {SV[4, 6], SV[7, {3}], SV[], SV[{14, 21, 43, 63, 68}], SV[Null, {14, 21, 43, 63, 68}]}
Out[8]= {SV[4, 6], 100, 12321, 3249, SV[Null, {14, 21, 43, 63, 68}]}
In[11]:= Art[x_String: "rans", y_Integer: 72] := (420 + StringLength[x])*y^2
In[12]:= {Art[42, 68], Art["IAN_2011", 14], Art["AVZ"], Art[420]}
Out[12]= {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.

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, наиболее часто используемые *ПФ* можно сохранять в специальном *mx*-файле по функции *Save*, например, в "D:\\Math_myLib\\UserLib72.mx", а при необходимости использования его функций достаточно данный файл загрузить в текущий сеанс по функции *Get*, обеспечив все *ПФ* из данного файла доступными *любому* документу на весь период текущего сеанса работы с пакетом. Несколько детальнее вопрос создания библиотек, содержащих пользовательские процедуры и функции, наряду с доступом к ним рассматривается в конце настоящей книги. В качестве стандартного механизма *CAS Mathematica* для организации пользовательских средств является сохранение их в пакетах, рассматриваемых ниже. Данный механизм *Mathematica* является довольно эффективным, широко используемым и самим пакетом, и пользователями. Довольно широко использовался он и нами. Однако, в ряде случаев предпочтительнее подход на основе библиотечной организации, рассматриваемой несколько ниже.

Глава 7. 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[942]:= << NumericalCalculus`
In[943]:= {NLimit[Sin[x]/x, x -> 0], NLimit[Sin[x]/(x/Cosh[x] + Log[x]), x -> -Infinity]}
Out[943]= {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[33]:= $Packages
Out[33]= {"NumericalCalculus`", "NumericalMath`NLimit`", "NumericalMath`", "Global`",
  "PrimalityProving`", "NumberTheory`", "ResourceLocator`", "DocumentationSearch`",
  "GetFEKernelInit`", "JLink`", "PacletManager`", "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[221]:= << PrimalityProving`
In[222]:= ProvablePrimeQ[2011]
Out[222]= True
In[223]:= $Packages
Out[223]= {"PrimalityProving`", "ResourceLocator`", "DocumentationSearch`", "System`",
  "WebServices`", "GetFEKernelInit`", "JLink`", "PacletManager`", "Global`"}
```

```
In[224]:= $ContextPath
Out[224]= {"PrimalityProving`", "PacletManager`", "WebServices`", "System`", "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]:= Df[x_, y_] := Module[{}, x + y]; F[x_] := x^2; G[x_, y] := 14*x + 21*y &
In[313]:= Map[Context, {Df, F, G}]
Out[313]= {"AladjevProceduresAndFunctions", "Global", "Global"}
```

т.е., имена F и G носят в текущем сеансе глобальный характер, тогда как Df определено в пакете с указанным контекстом. При этом, следует иметь ввиду, вполне возможны и конфликтные ситуации при вызовах функций, чьи идентичные имена определены в разных пакетах. В любом случае, при вызове подобных функций *Mathematica* выводит сообщения о 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 последующем не распознаваемы.

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 системы. Вполне уместо 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 | tx}"]
```

или с предварительным переопределением в качестве текущего каталога, содержащего пакеты в форматах $\{«.m», «.tx»\}$, например:

```
SetDirectory["<Путь к подкаталогу с пакетом>"]; << Имя_пакета`
```

После загрузки пакета в текущий сеанс обращение к функциям/процедурам, которые

содержатся в нем, можно обращаться 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[196]:= BeginPackage["Test`"]
ArtKr::usage = "The call ArtKr[x, y] returns  $x^2 + y^2 + (x + y)/z^3$ ."
Begin["`Private`"]
GS[x_, y_, z_] :=  $x^2 + y^2 + Kr[x, y, z]$ 
Kr[x_, y_, z_] :=  $(x + y)/z^3$ 
End[]
Begin["`ArtKr`"]
ArtKr[a_, b_, c_] := Test`Private`GS[a, b, c]
End[]
EndPackage[]

Out[196]= "Test`"
Out[197]= "The call ArtKr[x, y] returns  $x^2 + y^2$ ."
Out[198]= "Test`Private`"
Out[201]= "Test`Private`"
Out[202]= "Test`ArtKr`"
Out[204]= "Test`ArtKr`"

In[206]:= 6*ArtKr[14, 22, 6]
Out[206]= 4081

In[207]:= {GS[x, y, z], Kr[x, y, z]}
Out[207]= {GS[x, y, z], Kr[x, y, z]}

In[208]:= $ContextPath
Out[208]= {"Test`", "MyFunctions`", "PacletManager`", "WebServices`", "System`", "Global`"}

In[209]:= {Test`Private`GS[x, y, z], Test`Private`Kr[x, y, z]}
Out[209]= { $x^2 + y^2 + (x + y)/z^3$ ,  $(x + y)/z^3$ }
```

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 пример весьма наглядно иллюстрирует вышесказанное.

```
In[331]:= SetDirectory["D:\\Math_myLib"]
```

```
Out[331]= "D:\\Math_myLib"
```

```
In[332]:= Needs["MyFunctions`"]
```

```
Needs::nocont: Context MyFunctions` was not created when Needs was evaluated. >>
```

```
Out[332]= {Temporary}
```

```
In[333]:= Df[(Sin[x] + Cos[y])/(a + 1/x^3), 1/x^3]
```

```
Out[333]= -((Cos[y] + Sin[x])/(a + 1/x^3)^2)
```

```
In[334]:= $ContextPath
```

```
Out[334]= {"PacletManager`", "WebServices`", "System`", "Global`"}
```

Из примера следует, загрузив файл *"MyFunctions.m"* из каталога *"D:\\Math_myLib"*, 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."
=====
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 переменной **\$Help** и расширенная по вызову **Help[]**.

```

In[217]:= BeginPackage["MyFunctions`"]

UprocQ::usage = "UprocQ[x] returns the False if  $x$  is not a procedure; otherwise, two-
element list of the format {True, {Module | Block}} is returned."
Uprocs::usage = "Uprocs[] returns a list of procedures names activated in the current
session. In the absence of such procedures the empty list is returned."
ProcQ::usage = "ProcQ[x] returns the True if  $x$  is a procedure and the False otherwise."
Subs::usage = "Subs[x, y, z] returns the result of substitution  $y \rightarrow z$  for an expression  $x$ . The
procedure expands standard function Replace."
BlockQ::usage = "BlockQ[x] returns the True if  $x$  is a Block and the False otherwise."
StringEnd::usage = "StringEnd[x, y] returns the True if  $y$  finishes a string  $x$  and the False
otherwise."
Locals::usage = "Locals[x] returns the False if  $x$  is not a Mathematica-procedure; otherwise,
the set of local variables of  $x$  in string format is returned."
Spos::usage = "Spos[x, y, p, d] returns position number of the first occurrence of an one-letter
string  $y$  into a string  $x$  to the left ( $d=0$ ) or to the right ( $d=1$ ) from the given position  $p$ ."
SortNL::usage = "SortNL[L, p, b] sorts a listlist  $L$  on elements in a  $p$ -position of its sublists;
b={Greater | Less}."
Df::usage = "Df[x, y] returns the result of differentiation of  $x$  on  $y$ . The procedure expands
standard function Df."
NamesProc::usage = "NamesProc[] returns the list of names of the user procedures
activated in the current session."
FunctionQ::usage = "FunctionQ[x] returns the True if  $x$  is a function and the False
otherwise."
Mapp::usage = "Mapp[F, L] generalizes standard function Map on number of arguments
the greater than 1."
FunCompose::usage = "FunComposep[L, x] returns the nested function from the given list
of functions  $L$  from a variable (expression)  $x$ ."

Begin["`FunCompose`"]
FunCompose[L_List, 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]
End[]

Begin["`UprocQ`"]
UprocQ[x_] := Module[{a, b, c}, If[ProcQ[x], a := ToString[Definition[x]];
{b, c} = {StringPosition[a, " := Module[["], StringPosition[a, " := Block[["];
{b, c} = {Quiet[Check[First[Flatten[First[b]]], 0]], Quiet[Check[First[Flatten[First[c]]], 0]]];
{True, If[c == 0, Module, If[b == 0, Block, If[b < c, Module, Block]]], False}]
End[]

Begin["`Mapp`"]
Mapp[F_, L_]:=ToExpression[StringReplace[ToString[Map[F,L,{1}]],{"[" -> "[", "]" -> "]"})]
End[]

```

```

Begin["`FunctionQ`"]
FunctionQ[x_] := Module[{a, b}, a = ToString[InputForm[x]]; If[StringFreeQ[a, "#1"] ==
    False && StringTake[a, {-3, -1}] == " & ", True,
    a = Quiet[Check[ToString[InputForm[Definition[x]]], Return[False]]];
    b = x <> " := Function[{""];
    Quiet[Check[If[b == StringTake[a, StringLength[b]] || StringTake[a, -2] == "& " &&
        x <> " := " == StringTake[a, StringLength[x] + 4], True, False], False]]]
End[]

Begin["`NamesProc`"]
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]]
End[]

Begin["`Df`"]
Df[x_, y_] := Module[{a}, Simplify[Subs[D[Subs[x, y, a], a], a, y]]
End[]

Begin["`SortNL`"]
SortNL[L_List, p_Integer, 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 3-rd argument, should be {Less | Greater}, but has received ", {b}]],
            Print[SortNL::"incorrect 2-nd argument, should lay in interval ", {1, Length[First[L]}]],
            Print[SortNL::"incorrect first argument, should be numeric listlist, but has received ", {L}]]]
End[]

Begin["`Spos`"]
Spos[x_String, y_String, p_Integer, dir_Integer] := Module[{a, b, c},
    If[StringLength[y] > 1 || dir != 0 && dir != 1, Return[False], b = StringLength[x];
    If[p < 1 || p > b, False, If[p === 1 && dir === 0, c = 0, If[p === b && dir === 1, c = 0,
        If[dir === 0, For[a = p, a >= 1, a -- 1, If[StringTake[x, {a}] === y, Return[a, c]],
        For[a = p, a <= b, a ++ 1, If[StringTake[x, {a}] === y, Return[a, c]]]]];
    If[a === 0 || a === b + 1, 0, a]]
End[]

Begin["`Locals`"]
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]]}]]]
End[]
Begin["`StringEnd`"]
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]]]
End[]
Begin["`Subs`"]
Subs[x_, y_, z_] := Module[{a, b, c, h, t}, If[! HowAct[y], x/. y -> z,
c = If[MemberQ[{Plus}, Head[z]], "(" <> ToString[InputForm[z]] <> ")",
If[MemberQ[{Power, Times}, Head[z]], "(" <> ToString[InputForm[z]]
<> ")", ToString[z]]; {a, b, h} = First[{Map[ToString, Map[InputForm,
{x, y, 1/y}]]]; t = Simplify[ToExpression[StringReplace[
StringReplace[a, b -> c], h -> "1/" <> c]]]; If[t === x, x/. y -> z, t]]]
End[]
Begin["`BlockQ`"]
BlockQ[x_// Definition] := Module[{a}, If[Evaluate[x] == ToExpression[StringTake[
ToString[Definition[x]], {1, First[First[StringPosition[ToString[Definition[x]], " :=
Block[{"] - 1}]]], True, False]]]
End[]
Begin["`Uprocs`"]
Uprocs[] := Module[{a, b, c, d, h, g, k, t1, t2}, a := "_$Art21_Kr14$.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 = Symbol[StringTake[d, First[First[StringPosition[d, "["] - 1]]];
t1 = If[StringCount[d, " := Module[{"] != 0, Module, Block];
t2 = StringTake[d, Last[First[StringPosition[d, "]]]];
If[ProcQ[h], g = Append[g, {h, t2, t1}], Null]; k = k + 1]; DeleteFile[a]; Return[g]]
End[]
Begin["`ProcQ`"]
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[SameQ[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[SameQ[ToExpression[c], x], True, False]]; d || h ]
End[]
EndPackage[]

```

Этот пакет для возможности последующего доступа к функциям (*определения которых со справками находятся в нем*) как в текущем сеансе, так и с целью сохранения в файле, предварительно требуется *активировать*; иными словами *вычислить*, в результате чего возвращается следующая информация, а именно: (1) имя пакета **"MyFunctions"**, (2) по всем содержащимся в пакете функциям/процедурам справки (*если таковы определены в пакете*) и (3) парные *полные* имена функций в формате **"MyFunctions`Имя_функции"**.
Следующий фрагмент представляет результат активизации пакета **MyFunctions**.

```
Out[217]= "MyFunctions`"
Out[218]= "Uprocs[] returns a list of procedures names activated in the current session. In the absence of such procedures the empty list is returned."
Out[219]= "UprocQ[x] returns the False if x is not a procedure; otherwise, two-element list of the format {True, {Module | Block}} is returned."
Out[220]= "ProcQ[x] returns the True if x is a procedure and the False otherwise."
Out[221]= "Subs[x, y, z] returns the result of substitution y -> z for an expression x. The procedure expands standard function Replace."
Out[222]= "BlockQ[x] returns the True if x is a Block and the False otherwise."
Out[223]= "StringEnd[x, y] returns the True if y finishes a string x and the False otherwise."
Out[224]= "Locals[x] returns the False if x is not a Mathematica-procedure; otherwise, the set of local variables of x in string format is returned."
Out[225]= "Spos[x, y, p, d] returns position number of the first occurrence of an one-letter string y into a string x to the left (d = 0) or to the right (d = 1) from the given position p."
Out[226]= "SortNL[L, p, b] sorts a listlist L on elements in a p-position of its sublists; b={Greater | Less}."
Out[227]= "Df[x, y] returns the result of differentiation of x on y. The procedure expands standard function Df."
Out[228]= "NamesProc[]" returns the list of names of the user procedures activated in the current session."
Out[229]= "FunctionQ[x] returns the True if x is a function and the False otherwise."
Out[230]= "Mapp[F, L] generalizes standard function Map on number of arguments the greater than 1."
Out[231]= "FunComposep[L, x] returns the nested function from the given list of functions L from a variable (expression) x."
Out[232]= "MyFunctions`FunCompose`"
Out[234]= "MyFunctions`FunCompose`"
Out[235]= "MyFunctions`Mapp`"
Out[237]= "MyFunctions`Mapp`"
Out[238]= "MyFunctions`FunctionQ`"
Out[240]= "MyFunctions`FunctionQ`"
Out[241]= "MyFunctions`NamesProc`"
Out[243]= "MyFunctions`NamesProc`"
Out[244]= "MyFunctions`Df`"
```



```

Out[246]= "MyFunctions`Df`"
Out[247]= "MyFunctions`SortNL`"
Out[249]= "MyFunctions`SortNL`"
Out[250]= "MyFunctions`Spos`"
Out[252]= "MyFunctions`Spos`"
Out[253]= "MyFunctions`Locals`"
Out[255]= "MyFunctions`Locals`"
Out[256]= "MyFunctions`StringEnd`"
Out[258]= "MyFunctions`StringEnd`"
Out[259]= "MyFunctions`Subs`"
Out[261]= "MyFunctions`Subs`"
Out[262]= "MyFunctions`BlockQ`"
Out[264]= "MyFunctions`BlockQ`"
Out[265]= "MyFunctions`Uprocs`"
Out[267]= "MyFunctions`Uprocs`"
Out[268]= "MyFunctions`ProcQ`"
Out[270]= "MyFunctions`ProcQ`"
In[271]:= $Packages
Out[271]= {"MyFunctions`", "ResourceLocator", "DocumentationSearch", "JLink",
           "GetFEKernelInit", "PacletManager", "WebServices", "System", "Global"}

```

Завершает фрагмент вызов системной переменной **\$Packages**, возвращающей список контекстов, отвечающих пакетам, загруженным в текущий сеанс *Mathematica*; *первым* в нем указан контекст нашего пакета **"MyFunctions`"**. В результате активации пакета в текущем сеансе становятся *доступными* все определенные в нем функции/процедуры 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 возможности *Mathematica*. Достаточно простой фрагмент иллюстрирует *сохранение* пакета **"MyFunctions"** в файле формата *«.mx»* с последующей загрузкой его в новую сессию с проверкой доступности в новой сессии функций/процедур, чьи определения содержатся в пакете, и справок по ним.

```

In[272]:= Directory[]
Out[272]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[273]:= SetDirectory["D:\\Math_myLib"]
Out[273]= "D:\\Math_myLib"
In[274]:= Save["D:\\Math_myLib\\MyFunctions.mx", "MyFunctions`"]
In[275]:= Save["D:\\Math_myLib\\MyFunctions.m", "MyFunctions`"]
In[276]:= DumpSave["D:\\Math_myLib\\MyFunctions.mx", "MyFunctions`"]
Out[276]= {"MyFunctions`"}

```

Предварительно проверяется текущий каталог, где по умолчанию будет произведено

сохранение пакета, затем для удобства текущий каталог переопределяется на каталог "D:\\Math_myLib", где по функциям **Save** и **DumpSave** сохраняется искомый пакет в файлах сразу в двух форматах «.m» и «.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. Сохранение по **Save** и **DumpSave** принципиально различно.

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.

```
In[772]:= SetDirectory["D:\\Math_myLib"]
Out[772]= "D:\\Math_myLib"
In[773]:= << MyFunctions`
In[774]:= $Packages
Out[4]= {"MyFunctions`", "ResourceLocator`", "DocumentationSearch`", "JLink`",
        "GetFEKernelInit`", "PacletManager`", "WebServices`", "System`", "Global`"}
In[775]:= $ContextPath
Out[775]= {"MyFunctions`", "PacletManager`", "WebServices`", "System`", "Global`"}
In[776]:= Context[FunCompose]
Out[776]= "MyFunctions`"
In[777]:= FunCompose[{F, G, H}, x]
Out[777]= F[G[H[x]]]
In[778]:= ?FunCompose
FunComposep[L, x] returns the nested function from the given list of functions L from a
variable (expression) x.
In[779]:= ?Df
Df[x, y] returns the result of differentiation of x on y. The procedure expands standard
function Df.
In[780]:= Df[(a + Sin[x])/Cos[x], Cos[x]]
Out[780]= -Sec[x]^2 (a + Sin[x])
In[781]:= NamesProc[]
Out[781]= {}
In[782]:= ?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.

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.м», 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.{м | mx}"]**; при этом, файл с пакетом должен располагаться в одном из каталогов, определенном переменной **\$Path**; при наличии в каталоге файлов с расширениями «.м» 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 текст процедуры представлен следующим достаточно простым и прозрачным фрагментом.

```
In[1]:= Need[x_] := Module[{}, If[StringQ[x] && StringTake[x, -1] === "",
                               If[MemberQ[$Packages, x], Needs[x], Needs[x, FindFile[x]]], False]]
In[2]:= $Packages
Out[2]= {"ResourceLocator", "DocumentationSearch", "GetFEKernelInit", "JLink",
         "PacletManager", "WebServices", "System", "Global"}
In[3]:= Need["MyFunctions`"]
Out[3]= {Temporary}
In[4]:= Map[ProcQ, {BlockQ, Df, FunctionQ, Locals, NamesProc, ProcQ, UprocQ, Uprocs}]
Out[4]= {True, True, True, True, True, True, True, True}
```

Предложенные выше методы загрузки пакета в текущий сеанс посредством функций **Get["Пакет`"]** (<< Пакет`) и **Needs["Пакет", "Пакет.{м | mx}"]** являются явными, но в целом ряде случаев целесообразно определять режим автоматической загрузки некоего пакета, что 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[551]:= DeclarePackage["MyFunctions", {"ProcQ", "Df", "Locals", "Uprocs", "SortNL"}]
Out[551]= "MyFunctions`"
In[552]:= $Packages
Out[552]= {"ResourceLocator", "DocumentationSearch", "GetFEKernelInit", "JLink",
         "PacletManager", "WebServices", "System", "Global"}
In[553]:= Df[(Sin[x] + Cos[x])/Log[x]^2, Log[x]^2]
Out[553]= -((Cos[x] + Sin[x])/Log[x]^4)
In[554]:= Map[ProcQ, {Locals, Avz_68, Df, Mapp, SortNL}]
```

```
Out[554]= {True, False, True, False, True}
```

```
In[555]:= ?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 текст процедуры и пример ее применения для пакета "**MyFunctions`**".

```
In[48]:= SetDirectory["D:\\Math_myLib"]
```

```
Out[48]= "D:\\Math_myLib"
```

```
In[49]:= 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 have mx-format", Null];
```

```
  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[50]:= PackNames["MyFunctions`"]
```

```
Out[50]= {ProcQ, Spos, BlockQ, FunctionQ, Subs, StringEnd, Locals, NamesProc, Mapp, Df,
  Uprocs, SortNL, FunCompose, UprocQ}
```

До сих пор рассматривалась загрузка пакета как явная по функциям *Get* (<<) и *Needs*, так и в автоматическом режиме по функции **DeclarePackage**. Между тем, в целом ряде случаев целесообразно загружать пакеты в момент начальной загрузки *Mathematica*, как это имеет место для целого ряда стандартных пакетов *Mathematica*. С этой целью можно использовать т.н. инициализационные файлы с именами «*init.m*», располагаемые в ряде подкаталогов системы каталогов *Mathematica*. Данные файлы содержат коды, выполняемые каждый раз при загрузке ядра либо препроцессора *Mathematica*.

Для допустимого расположения инициализационных файлов в целях обеспечения ими требуемых можно использовать следующие подкаталоги *Mathematica*, а именно:

\$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 8.0.1.0* данные подкаталоги принимают вид:

```
In[1]:= $BaseDirectory <> "\\Kernel"
```

```
Out[1]= "C:\Documents and Settings\All Users\Application Data\Mathematica\Kernel"
```

```
In[2]:= $UserBaseDirectory <> "\\Kernel"
```

```
Out[2]= "C:\Documents and Settings\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 текст процедуры и пример ее применения даны ниже.

```
In[1]:= AutoLoadPack[x_, y_] := Module[{}, SetDirectory[y]; Quiet[Needs[x]];
ResetDirectory[]; Null]

In[2]:= AutoLoadPack["MyFunctions", "D:\\Math_myLib"]

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, False, 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 - 31.10.2011**)

AutoLoadPack[x_, y_] := Module[{}, SetDirectory[y]; Quiet[Needs[x]]; ResetDirectory[]; Null]
AutoLoadPack["MyFunctions", "D:\\Math_myLib"]
```

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;
- 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*. Следующий фрагмент представляет исходный текст процедуры и пример обновления системной переменной **\$Path** списком трех новых подкаталогов.

```

In[1]:= UpdatePath[x_List] := Module[{a}, a = Flatten[Append[$Path, x]];
                                         $Path = DeleteDuplicates[a]; Null]

In[2]:= UpdatePath[{"D:\\Math_myLib", "D:\\UserLib6789", "D:\\Book"}]
In[3]:= $Path
Out[3]= {"C:\\Program Files\\Wolfram Research\\Mathematica\\8.0\\SystemFiles\\Links",
"C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica\\Kernel",
"C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica\\Autoload",
"C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica\\Applications",
"C:\\Documents and Settings\\All Users\\Application Data\\Mathematica\\Kernel",
"C:\\Documents and Settings\\All Users\\Application Data\\Mathematica\\Autoload",
"C:\\Documents and Settings\\All Users\\Application Data\\Mathematica\\Applications", ".",
"C:\\Documents and Settings\\Aladjev", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\AddOns\\Packages", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\AddOns\\LegacyPackages", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\SystemFiles\\Autoload", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\AddOns\\Autoload", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\AddOns\\Applications", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\AddOns\\ExtraPackages", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\SystemFiles\\Kernel\\Packages", "C:\\Program Files\\Wolfram
Research\\Mathematica\\8.0\\Documentation\\English\\System", "D:\\Math_myLib",
"D:\\UserLib6789", "D:\\Book"}
In[4]:= << MyFunctions`
Out[4]= {Temporary}
In[5]:= Map[ProcQ, {ProcQ, Spos, BlockQ, FunctionQ, Subs, StringEnd, Locals, SortNL,
NamesProc, Mapp, Df, Uprocs, FunCompose, UprocQ}]
Out[5]= {True, True, True, True, True, True, True, True, True, True, False, True, True, True, True}

```

Пример обновления 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 - 31.10.2011**)
UpdatePath[x_] := Module[{}, $Path = DeleteDuplicates[Flatten[ReleaseHold
                                                         [Append[$Path, x]]]]; Null]
UpdatePath[{"D:\\Math_myLib", "D:\\UserLib6789", "D:\\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, True, False, True, True, True, True}
In[2]:= a = Length[$Path]; {$Path[[a - 2]], $Path[[a - 1]], $Path[[a]]}
Out[2]= {"D:\\Math_myLib", "D:\\UserLib6789", "D:\\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[17]:= UpdatePackages[x_] := Module[{}, ClearAttributes[$Packages, Protected];
        $Packages = DeleteDuplicates[Flatten[ReleaseHold[Append[$Packages, x]]]];
        SetAttributes[$Packages, Protected]; Null]
In[18]:= UpdatePackages["MyFunctions"]
In[19]:= $Packages
Out[19]= {"ResourceLocator", "DocumentationSearch", "GetFEKernelInit", "JLink",
        "PacletManager", "WebServices", "System", "Global", "MyFunctions"}
In[20]:= Attributes[$Packages]
Out[20]= {Protected}
In[21]:= UpdateContextPaths[x_] := Module[{}, $ContextPath =
        DeleteDuplicates[Flatten[ReleaseHold[Append[$ContextPath, x]]]]; Null]
In[22]:= UpdateContextPaths["MyFunctions"]; $ContextPath
Out[22]= {"PacletManager", "WebServices", "System", "Global", "MyFunctions"}

```

Данная процедура вполне может сочетаться с ранее представленными средствами по работе с *пакетами* пользователя, упрощая их программирование и применение. Итак, в принципе, 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*) мы с полным основанием можем отождествлять с документами *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 высокую унификацию пакетных и пользовательских средств, пакеты *Mathematica* располагаются отдельно, в общем случае, по различным каталогам, поддерживая *контекстную* взаимосвязь как с системой *Mathematica*, так и с другими пакетами как системными, так и пользовательскими. В ряде случаев данная организация не представляется нам достаточно удобной.

Глава 8. Средства ввода/вывода пакетов Maple и Mathematica

Будучи *встроенными* языками программирования *Maple* и *Mathematica*, прежде всего ориентированными на символьные вычисления (*компьютерная алгебра*) и обработку, они располагают относительно ограниченными возможностями по работе с данными, находящимися во внешней памяти *ПК*. В этом отношении они существенно уступают традиционным языкам программирования *C*, *Basic*, *Fortran*, *Cobol*, *PL/1*, *ADA*, *Pascal* и др. Вместе с тем, ориентируясь, прежде всего, на решение задач в символьном виде, данные языки предоставляют набор средств для доступа к файлам данных, который вполне может удовлетворить довольно широкий круг пользователей математических приложений пакета. В данной главе средства доступа к файлам данных рассмотрены достаточно поверхностно ввиду ограниченного объема, обширности проблематики и цели настоящей книги. Однако заинтересованный читатель, например, по средствам доступа пакета *Maple* довольно детально по полноте изложения, в значительной мере перекрывая поставляемую с пакетом документацию, может ознакомиться в книге [42]. С целью развития *способов* доступа к файловой системе *ПК*, например, для *Maple* был создан ряд эффективных средств, представленных в нашей Библиотеке [45]. Начиная с *Maple 9*, пользователю предоставляются некоторые *дополнительные* средства работы с файлами данных, поддерживаемые процедурами модуля **FileTools**; между тем, они по целому ряду важных показателей достаточно существенно уступают средствам из нашей Библиотеки [45]. Ниже акцентируется внимание как на базовых средствах, так и на различиях средств доступа рассматриваемых пакетов *Maple* и *Mathematica*.

8.1. Средства пакета Maple для работы с внутренними файлами

Средства *Maple*-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 – 4* различными типами внутренних файлов, из которых отметим лишь наиболее важные на *первых* этапах работы с пакетом. Однако, перед дальнейшим рассмотрением целесообразно ввести понятие *спецификатора файла (СФ)*, который определяет полный путь к искомому файлу в файловой системе *ЭВМ*. Как правило, он кодируется в форме значения *{string, symbol}*-типа и для платформы *{DOS | Windows}* имеет следующий простой формат кодирования, а именно:

<УВВ>:\ \ | / <Подкаталог_1> \ \ | / ... \ \ | / <Подкаталог_N> \ \ | / <Файл>

где *УВВ* определяет логическое имя устройства (*например, жесткий диск, CD, flash и т. д.*), *Подкаталог_K* – подкаталог файловой системы *ПК* и *Файл* – имя искомого файла в формате *<Основное имя>{. <Расширение имени>}*. При этом, символ *\ \ | /* служит в качестве разделителя при определении пути к файлу *любого* типа; понятие *СФ* имеет силу для обоих пакетов. Естественным образом понятие *СФ* распространяется также на цепочку подкаталогов. Под *текущей* понимается цепочка подкаталогов, с которой работают средства доступа пакета по умолчанию, т.е. в таком случае к файлу можно обращаться лишь по его имени. В пакете *Maple* по функции *currentdir()* возвращается значение текущей цепочки, тогда как по вызову функции *currentdir(<Цепочка>)* будет

установлена указанная *цепочка* в качестве пути к текущему подкаталогу, например:

```
> currentdir(); ⇒ "C:\\Program Files\\Maple 11"
> currentdir("D:/UserLib6789\\UserLib11/Users"): currentdir();
"D:/UserLib6789\\UserLib11\\Users"
```

Как иллюстрирует этот пример, при установке новой текущей цепочки подкаталогов функция *currentdir* возвращает *предыдущее* значение текущей цепочки, что позволяет при необходимости весьма легко возвращаться к предыдущему значению. При этом, следует иметь в виду, что установка текущей цепочки отменяется только *перезагрузкой* пакета либо ее *переопределением* (т.е. на нее не действуют предложения **restart**, **done**, **quit**, **stop**). Более того, установленный по *currentdir*-функции каталог становится текущим лишь для *внешних* файлов пакета, т.е. эта функция используется с рассматриваемыми ниже функциями доступа к *внешним* файлам данных пакета. При этом, при попытке определить текущим *несуществующий* каталог вызов функции *currentdir* инициирует ошибочную ситуацию, как это наглядно иллюстрирует следующий пример:

```
> currentdir("C:\\Tallinn\\Academy\\Grodno\\Vilnius");
Error, (in currentdir) file or directory, C:\\Tallinn\\Academy\\Grodno\\Vilnius, does not exist
```

Посему, с целью расширения функциональных возможностей стандартного средства нами была создана процедура *Currentdir*, исходный текст которой приведен ниже.

```
> Currentdir := proc() local a, b, c; assign67(Release(a), c = currentdir()), `if`(nargs = 0,
RETURN(Currentdir(a)), `if`(map(type, {args}, {'string', 'symbol'}) = {true}, assign(b =
CF(cat("", args[1])), ERROR("arguments %1 should be strings or symbols", [args])),
`if`(1 < nargs, ERROR("%1 quantity of actual arguments more than 1", [args]),
`if`(type(b,dir),op([currentdir(b),c][-1]), `if`(type(b,file),Currentdir(cld(CFF(b)[1..-2],"\\")),
op([MkDir(b), currentdir(b), c][-1])))
end proc:
> Currentdir("C:\\Tallinn\\Noosphere_Academy\\Grodno\\Vilnius"): currentdir();
"c:\\tallinn\\noosphere_academy\\grodno\\vilnius"
```

Вызов *Currentdir(W)* делает текущим каталог, определенный полным путем к нему *W*; при этом, если путь *W* не существует, он создается. В любом случае вызов процедуры с аргументом возвращает предыдущий текущий каталог, тогда как вызов процедуры без аргументов восстанавливает в качестве текущего каталог, который был определен текущим при инсталляции *Maple*. В целом ряде случаев работы с файловой системой *ПК* процедура *Currentdir* предпочтительнее стандартной. Процедура, в свою очередь, существенно использует нашу процедуру *MkDir* [45], которая довольно существенно расширяет стандартную функцию *mkdir*, предназначенную для создания каталогов, как весьма наглядно иллюстрирует следующий достаточно простой пример:

```
> mkdir("C:\\Tallinn\\Noosphere\\Grodno\\Vilnius");
Error, (in mkdir) file or directory does not exist
> MkDir("C:\\Tallinn\\Noosphere\\Grodno\\Vilnius");
"c:\\tallinn\\noosphere\\grodno\\vilnius"
```

В случае кодирования *несуществующего* каталога он создается. Если вызов процедуры использует в качестве второго необязательного аргумента **1**, по заданному пути будет создан пустой закрытый файл. В *любом* из указанных случаев вызов процедуры **MkDir** возвращает созданный *полный* путь к каталогу или файлу. Процедура имеет и другие полезные особенности, например, позволяя создавать в каталогах файлы при *наличии* в них *одноименных* подкаталогов. В целом же, процедура **MkDir** расширяет не только стандартную функцию **mkdir**, но и *одноименную* команду DOS [9-14,28-37,42-45]. Нами в дополнение к базовой процедуре **MkDir** был создан ряд ее полезных модификаций, включая обобщение на случай, когда делается попытка создания каталога или файла на устройстве или в каталоге, защищенном от записи. Так, вызов процедуры **MkDir4** обеспечивает создание требуемого подкаталога либо файла на дисковом устройстве с возвратом полного пути к нему. Если устройство либо какой-то элемент создаваемого пути защищен от записи, процедура выбирает *первое* доступное устройство (*по выбору может быть и устройство «А:»*). При отсутствии доступных устройств иницируется ошибочная ситуация с диагностикой «*Error, in (MkDir4) accessible disk drives do not exist*». С отмеченными средствами можно весьма детально ознакомиться в [9-14,28-37,42-45].

Наряду с представленными нами создан ряд других процедур *общего* назначения для работы с файловой системой **ПК**. В частности, нами введены **3 новых** типа выражений, а именно: **dir**, **file** и **path**, определяющие соответственно *каталог*, *файл* и *полный путь*. Вызов **type(W, 'dir')** возвращает **true**-значение, если аргумент **W** определяет каталог, и **false**-значение в противном случае [28-37,42-45], как иллюстрирует пример:

```
> type( "C:/Temp", 'dir'), type( "C:\\", 'dir'), type("C:/Program files/Maple 11/lib", 'dir');
      true, true, true
```

При этом, следует иметь в виду, что возврат **false**-значения говорит только о том, что тестируемый каталог **F** (*точнее последний элемент F-цепочки*) отсутствует в: (1) цепочке, определенной **F**, (2) цепочках, определенных предопределенной переменной пакета **libname**, (3) цепочке, определенной вызовом **currentdir()** и (4) цепочках, определенных открытыми файлами данных текущего сеанса. Тогда как, вообще говоря, этот каталог **F** может быть элементом *файловой* системы **ПК**. Детально с представленными и целым рядом других средств работы с файловой системой **ПК** можно ознакомиться в [42-45].

Предложения **Maple**-языка сохраняются во *внутренних* файлах пакета в одном из *двух* форматов (*входном Maple-формате и внутреннем m-формате; оба файла являются ASCII-файлами*) на основе **save**-предложения одного из следующих двух форматов:

save Id1, Id2, ..., Idn, <CФ> или **save(Id1, Id2, ..., Idn, <CФ>)**

По предложению **save** любого из этих форматов в указанный **СФ** файл выгружаются только те определения идентификаторов, чьи имена закодированы в **Id1, Id2, ..., Idn**, перед **СФ**-аргументом. При этом, если в **СФ** указано **m**-расширение имени, то файл сохраняется во внутреннем **m**-формате, иначе используется входной формат **Maple**-языка (*в данном формате производится ввод информации в Maple-документ с консоли*). В случае указания в предложении **save** *неопределенных* **Id**-идентификаторов, они будут сохраняться в *файле* в виде **Id:='Id'** с выводом соответствующих сообщений. Успешное завершение **save**-предложения возвращает **NULL**-значение. Фрагмент иллюстрирует

использование **save**-предложения для создания файлов обоих указанных форматов:

```
> A:= 68: B:= [68,63,43,48,14,21]: P:= () -> `(args)/nargs: M:= module () export sr; sr:= () ->
    `(args)/nargs end module: S:= 2011:
> save(A, B, P, M, S, "C:/Temp/Tallinn"); save(A, B, P, M, S, "C:/Temp/Tallinn.m");
> restart; read("C:/Temp/tallinn"): %;    => 2011
> A, B, 6*P(68,63, 43, 48, 14 ,21), 6*M:- sr(1,2,3,4,5,6); => 68, [68, 63, 43, 48, 14, 21], 233, 21
> restart; read "C:/Temp/tallinn.m"; %;    => -1 (A)
> A, B, 6*P(68, 63, 43, 48, 14, 21), 6*M:- sr(1,2,3,4,5);    => 68, [68, 63, 43, 48, 14, 21], 257, 18
```

В данном фрагменте определяются переменные и по **save**-предложению сохраняются в *Tallin*-файлах в форматах *входного Maple*-языка и *внутреннем*. На небольшого объема файлах не ощущается преимущества того или иного формата, однако в случае роста объема файла предпочтение *внутреннего m*-формата становится все ощутимее, давая возможность создавать более компактные файлы и быстрее загружать их в память.

Загрузка сохраненного по **save** файла производится по предложению **read** форматов:

read <CФ> или **read(<CФ>)**

однако, результат выполнения **read** зависит от формата загружаемого файла. Общим является лишь тот факт, что после загрузки файла содержащиеся в нем определения вычисляются, становясь доступными в текущем сеансе пакета, если впоследствии не определяется противного. Отметим лишь, что при загрузке *m*-файла информации не выводится и возвращается *NULL*-значение. Так, пример (A) предыдущего фрагмента иллюстрирует применение предложения **read** для загрузки *m*-файла. Однако, в этом месте может возникнуть недопонимание в связи с возвратом %-вызовом значения **-1**. А дело здесь обстоит следующим образом. Подобный %-вызов возвращает значение выполнения предыдущего ему предложения. Но так как *перед* предложением **read** мы выполнили **restart**, а выполнение предложения **read** возвратило *NULL*-значение (*т.е. ничего*), то %-вызов после **restart** либо после загрузки пакета в зависимости от релиза и его клона возвращает следующие значения: результат вызова последней процедуры, определенной в инициализационном файле «*Maple.ini*» (*Maple 6/7* и *Maple 9-11* клона *standard*), **10** (*Maple 8* и *Maple 9/9.5* клона *standard*) и **-1** (*Maple 10/11* клона *standard*).

На первых порах работы с *Maple* средства доступа к внутренним *m*-файлам наиболее полезны при необходимости создания различного рода библиотек пользовательских функций и/или процедур, а также часто используемых *Maple*-конструкций. Однако, здесь имеется одна весьма существенная проблема, а именно. Программные модули в такого типа файлах для релизов, младших *10-го*, сохраняются некорректно (*точнее, не сохраняются их тела*). Поэтому нами был создан ряд процедур как обобщающих, так и расширяющих стандартные предложения **save** и **read** относительно работы с файлами внутреннего *Maple*-формата [9-14,28-37,42-45]. Так, в частности, пара процедур **Save2** и **Read1** обеспечивает корректное сохранение всех типов *Maple*-объектов в файлах *m*-формата с последующим чтением в текущий сеанс. Так, **Save2** довольно существенно расширяет предложение **save** с полным наследованием синтаксиса последнего, однако в отличие от него позволяет корректно сохранять в *m*-файлах переменные, процедуры и программные модули. Вызов процедуры возвращает полный путь к файлу с объектами

Maple, сохраненными в файле. В свою очередь, процедура *Read1(F)* используется для загрузки в память файлов, сохраненных по вызову *Save2*, наряду с получением имен объектов, сохраненных в них, в разрезе их типов. С другими нашими средствами *Save*-группы можно достаточно детально ознакомиться в наших книгах [9-14,28-37,42-44].

В ряде задач продвинутого программирования возникает вопрос определения *типов* и *имен* объектов, содержащихся в файлах внутреннего *Maple*-формата (*m-файлы*), без прямой загрузки таких файлов предложением *read*. Эта задача решается процедурой *cmf(F)*, возвращающей последовательность *списков*, описывающих объекты *m-файла F* в следующих *четырёх* аспектах: *процедуры* ("*Proc*"), *неопределённые* ("*unassigned*"), *модули* ("*Mod*") и *другие* ("*Others*"). Каждый список первым элементом имеет описатель *типа*, другие представляют *имена* объектов, соответствующих ему. При отсутствии объектов какого-нибудь типа соответствующий *список* не возвращается. Следующий фрагмент представляет исходный текст процедуры и типичный пример ее использования.

```
> cmf := proc(F::file) local a, b, c, d, t, r, u;
  if not (Ftype(F) = ".m") then error "datafile must be m-file but has type <%1>", Ftype(F)
  else a := Release1();
    if a <> SD(readbytes(F, 'TEXT', 2)[2]) then error "datafile <%1> is noncompatible with
      the current release", F else
      filepos(F, 0), assign(r = ["Others"], b = ["6", "Mod"], d = [if (a = 6, {"R6", "RF"},
        {"f*"}), "Proc"], u = ["GF", "unassigned"]);
      do a := readline(F); if a = 0 then close(F); break elif a[1] = "I" then
        assign('c' = Iddn1(a)), search(a, c, 't'), assign('t' = t + length(c));
        if a[t .. t + 1] = b[1] then b := [op(b), c] elif member(a[t .. t + 1], d[1]) then
          d := [op(d), c]
          elif a[t .. t + 1] = u[1] then u := [op(u), c] else r := [op(r), c]
        end if
      end if
    end do
  end if;
  seq('if (nops(a) = 1, NULL, a), a = [b[2 .. -1], d[2 .. -1], r, u[2 .. -1]]);
end proc;

> Tab:=table(): T1:=table([a=b, c=d]): To:=table(): Kr:=proc() avz end proc: M:=module()
end module: module M1 () end module: P1 := () -> `+`(args)/nargs: Art := proc() local a;
a:=68 end proc: Sv := {14, 21, 68, 63, 43, 48}: Gs := [42, 47, 67, 89, 96]:
> save(Kr, Tab, T1, P, M, To, V, G, P1, S, M1, Art, Kr, R, Sv, Gs, Q, "C:\\temp\\test.m");
> cmf("C:\\temp\\test.m");
["Mod",M,M1],["Proc",Kr,P1,Art],["Others",Tab,T1,To,Kr,Sv,Gs],["unassigned",P,V,G,S,R,Q]
```

Под *внутренними* файлами пакета мы понимаем здесь файлы, чья структура известна пакету. К этим файлам относятся файлы с документами (*worksheets*), которые в одном из 9 форматов сохраняются по цепочке команд "*File -> {Save As | Save}*" *GUI* (наиболее используемые форматы "*.mws*", "*.mw*", "*.mpl*" и "*.maplet*"), файлы с *Maple*-объектами,

сохраняемыми по предложению **save** (входной и внутренней форматы), а также файлы пакетных и пользовательских библиотек, структурно подобных главной библиотеке пакета (наиболее используемые форматы *“.ind”*, *“.hdb”*, *“.lib”* и *“.mla”*). Как правило, эти файлы могут представить вполне определенный интерес при решении многих задач, требующих продвинутых приемов программирования. Для стандартной поддержки работы с ними пакет располагает рядом средств, некоторые из которых упоминались выше, тогда как для обеспечения расширенной работы с подобными файлами создан комплекс средств, некоторые из которых рассматриваются в настоящей книге, в также в полном объеме включены в нашу библиотеку [45]. Наш и других пользователей опыт использования этих средств показал их вполне приемлемую эффективность в работе с внутренними файлами *Maple* при программировании многих нестандартных задач. По нашим наработкам в данном направлении пакет *Maple* пополнился целым рядом средств работы с файлами данных в рамках пакетного модуля *FileTools*; между тем, в их числе до сих пор отсутствует целый ряд важных средств – аналогов нашим [42-45].

8.2. 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 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 простой фрагмент иллюстрирует сказанное.

```
In[81]:= Directory[]
Out[81]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[82]:= SetDirectory["D:\\Math_myLib"]
Out[82]= "D:\\Math_myLib"
In[83]:= SetDirectory[$UserDocumentsDirectory]
Out[83]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[84]:= SetDirectory[]
Out[84]= "C:\\Documents and Settings\\Aladjev"
In[85]:= SetDirectory["D:\\Math_myLib\\rans"]
SetDirectory::cdir: Cannot set current directory to D:/Math_myLib/rans. >>
Out[85]= $Failed
In[86]:= SetDir[x_String] := 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[87]:= SetDir["C:\\111\\222\\333\\444\\555\\666\\777\\888\\999"]
Out[87]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999"
In[88]:= SetDir["H:\\111\\222\\333\\444\\555\\666\\777\\888\\999"]
Out[88]= $Failed
In[89]:= Directory[]
Out[89]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999"
In[90]:= SetDir["rans\\ian\\avz"]
Out[90]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999\\rans\\ian\\avz"
In[91]:= 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]]]; b]
In[92]:= Adrive[]
Out[92]= {"C", "D", "G", "F"}
In[93]:= Directory[]
Out[93]= "C:\\111\\222\\333\\444\\555\\666\\777\\888\\999\\rans\\ian\\avz"
> Adrive(); => "C", "D", "G", "F"
> currentdir(); => "C:\\Program Files\\Maple 11"
```

Между тем, при попытке определения в качестве текущего *несуществующего* каталога вполне реально возникновение ситуации, когда в качестве **УВВ** 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*, правда, возвращающей вместо списка последовательность имен *YBB*, доступных в текущий момент. Последняя часть фрагмента представляет исходный текст 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, в 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[221]:= Art1 := #^2 &; Art2 := #^3 &; Art3 := #^4 &; Art4 := #^5 &
In[222]:= Save["D:/Math_myLib\\Art1", Art1]; Save["D:/Math_myLib\\Art2.m", Art2];
          Save["D:/Math_myLib\\Art3.mx", Art3]; Save["D:/Math_myLib\\Art4.nb", Art4]
          ===== Новый текущий сеанс =====
In[121]:= SetDirectory["D:\\Math_myLib"]
Out[121]= "D:\\Math_myLib"
In[122]:= << "Art1"; << "Art2.m"; << "Art3.mx"; << "Art4.nb"
In[123]:= {Art1[68], Art2[68], Art3[68], Art4[68]}
Out[123]= {4624, 314432, 21381376, 1453933568}
```

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]*, чей вызов возвращает список имен объектов в строчном формате, ранее сохраненных в файле *x* по функции *Save*. Более того, может возникнуть необходимость не загружать по функции *Get* (<<) весь файл, ранее созданный по *Save*, в текущий сеанс с активизацией *всех* содержащихся в нем объектов, а загружать содержащиеся в файле объекты выборочно, т.е. создавать в некотором роде библиотеки пользовательских средств. Эту задачу может выполнять процедура *Aobj[w, y]*, чей вызов активизирует в текущем сеансе объект *y* из файла *w*, ранее созданного по функции *Save*. Следующий фрагмент иллюстрирует сказанное.


```

In[20]:= Art1 := #^2 &; Art2 = #^3 &; Art3 := #^4 &; Art4 = #^5 &; Kr = 14;
In[21]:= Save["D:/Math_myLib/Obj.m", {Adrive, SetDir, Art1, Art2, Art3, Art4, Nobj, Kr}]
In[22]:= Nobj[x_String] := Module[{a, b, c, d, p}, If[FileExistsQ[x] &&
MemberQ[{"Table", "Package"}, Quiet[FileFormat[x]]], {a, b, d} = {OpenRead[x], {}, "68"};
Label[c]; d = Read[a, String]; If[d === EndOfFile, Return[b], If[StringTake[d, 1] === " ",
Goto[c, Null]]; p = StringTake[d, First[First[StringPosition[d, {" := ", " = "}] - 1];
b = Append[b, If[StringFreeQ[p, "["], p, StringTake[p, StringPosition[p, "["] [[1]][[1]] - 1]]];
Goto[c, $Failed]]
In[23]:= Nobj["D:\\Math_myLib\\Obj.m"]
Out[23]= {"Adrive", "SetDir", "Art1", "Art2", "Art3", "Art4", "Nobj", "Kr"}
===== Новый текущий сеанс =====
In[881]:= Aobj[x_String, y_String] := Module[{a, b, c, d, p},
If[FileExistsQ[x], a = Nobj[x], Return[$Failed]];
If[! MemberQ[a, y], Return[$Failed], {a, b} = {"", OpenRead[x]}; Label[c];
d = Read[b, String]; If[StringPosition[d, y] != {} && StringPosition[d, y] [[1]][[1]] == 1,
a = a <> d, Goto[c]]; Label[p]; d = Read[b, String];
If[d === EndOfFile || d === " ", Return[ToExpression[a]], a = a <> d; Goto[p]]
In[882]:= Aobj["D:\\Math_myLib\\Obj.m", "Adrive"]
In[883]:= Adrive[]
Out[883]= {"C", "D", "G", "F"}

```

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* текст процедуры *Nobj* и пример ее применения. Тогда как во второй части фрагмента, отражающего работу в новом сеансе, приведен исходный текст процедуры *Aobj[x, y]* и пример 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 ругих интересных процедур для обеспечения работы с файлами входного формата *Mathematica*, чьи имена имеют расширения {"*.nb*", "*.m*", "*.txt*"} и др. 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*.

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 *mх*-файла с последующим тестированием результата его загрузки в новом сеансе пакета.

```
In[748]:= DumpSave["D:\\Math_myLib\\Objects.mx", {Adrive, SetDir, Art1, Art2, Art3,
      Art4, Kr, Nobj, Aobj}]
Out[748]= {Adrive, SetDir, #1^2 &, #1^3 &, #1^4 &, #1^5 &, 14, Nobj, Aobj}
===== Новый текущий сеанс =====
In[81]:= Get["D:\\Math_myLib\\Obj1.mx"]
In[82]:= Nobj["D:\\Math_myLib\\Objects.m"]
Out[82]= {"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.

8.3. Средства Maple для работы с внешними файлами данных

Средства доступа к файлам включают ряд функций; при этом, под понятием «*файл*» подразумевается не только организованная совокупность данных во внешней памяти *ПК*, но и канал, а также непосредственный интерфейс с пользователем. В отношении такого понимания все составляющие понятие *файл* компоненты эквивалентны, посему за весьма редким исключением любой файл обрабатывается одним и тем же набором функций доступа. Для обеспечения доступа к *файлам* пакет *Maple* располагает рядом довольно эффективных для *ПС* данного типа средств. Подобно случаю большинства систем программирования для инициации операций обмена с файлом требуется его предварительное открытие, а после завершения работы с файлом – его закрытие; при этом, если открытие файла должно быть явным (кроме стандартных и ряда специальных *файлов*), закрытие его в общем случае не обязательно должно быть явным, т.к. в конце работы с прикладной программой все используемые ею ресурсы, включая также и ее файлы различных типов, закрываются автоматически. Однако, для большинства *ПС* число одновременно открытых файлов ограничено и их максимальное число лежит в пределах (8–24). Для случая *Maple* количество открытых пользовательских файлов не должно превышать 7. С целью устранения этого (в ряде случаев довольно существенного) ограничения нами была создана процедура *holdof* [45], которая использует довольно простой механизм расширения числа доступных каналов *ввода/вывода* (*в/в*), а именно.

Если число открытых каналов *в/в* меньше 7, вызов процедуры *holdof(hold)* возвращает список логических номеров открытых каналов *в/в*; в противном случае возвращается логический номер открытого канала *в/в* с сохранением состояния файла данных, ранее открытого по данному каналу. Успешный вызов процедуры позволяет использовать дополнительный канал для работы с файлами. После завершения работы с выделенным

каналом вызовом процедуры *holdof(restore)* восстанавливается отложенное состояние файла данных, ранее открытого по этому каналу с возвратом *NULL*-значения. Таким образом, принципиальная схема применения *holdof* имеет следующий вид:

holdof(hold) => обработка файла данных на выделенном канале в/в => holdof(restore)

Следующий фрагмент представляет исходный текст процедуры *holdof*.

```
> holdof := proc(C::symbol) local a, b, k, psi, j, omega; global _avzagnartkrarn68;
  psi := (x) -> {k $ (k = 0 .. 6)} minus {seq(x[j][1], j = 4 .. nops(x))};
  `if (member(C, {'hold', 'restore'}), assign(a = iostatus()), ERROR("argument should be
    'hold' or 'restore' but has received <%1>", C));
  if C = 'hold' then `if (nops(a) < 10, RETURN(psi(a)), NULL);
    for k from 4 to nops(a) do
      if not member(a[k][2], {"default", "terminal"}) and member(a[k][3], {STREAM, RAW})
      then _avzagnartkrarn68 := filepos(a[k][2]), a[k]; close(a[k][1]), RETURN(a[k][1])
      end if
    end do;
    if type(_avzagnartkrarn68, 'assignable1') then ERROR("too many files already open")
    end if
  else
    try `if (not typeseq(_avzagnartkrarn68, 'seqn'), RETURN(),
      assign67(omega = _avzagnartkrarn68));
      unassign('_avzagnartkrarn68'), close(omega[2][1])
    catch "file descriptor not in use": null("processing of an open I/O channel")
    end try;
    b := [`if (omega[2][3] = 'RAW', 'open', 'fopen'),
      omega[2][2], op(omega[2][5] .. `if (omega[2][3] = 'RAW', 5, 6))];
    null(OpenLN(omega[2][1], op(b)), filepos(omega[2][1], omega[1])),
    `if (1 < nargs, NULL, WARNING("file <%1> had been restored on I/O channel
      <%2>", omega[2][2], omega[2][1]))
  end if
end proc;
```

В целом ряде случаев, имеющих дело с доступом к *файлам* данных, нужно обеспечить открытие файла по *конкретному* каналу *в/в*. Для этой цели представляется достаточно полезной наша процедура *OpenLN*. Успешный вызов процедуры *OpenLN(N, O, F, par)* возвращает 2-элементную последовательность, чей *первый* элемент определяет номер логического канала *в/в*, т.е. *N*, тогда как *второй* – множество всех остающихся номеров открытых каналов. Третий *F*-аргумент определяет *имя* или *полный* путь открываемого файла данных. Четвертый аргумент *par* определяет остальные аргументы процедуры *OpenLN*, которые зависят от функции открытия *O* {*open*, *fopen*}. С целым рядом наших других полезных процедур, относящихся к открытию файлов данных в *Maple*, можно ознакомиться в [9-14, 28-37, 42-45]. Следующий фрагмент представляет исходный текст процедуры *OpenLN* и пример ее конкретного использования для открытия файла.

```

> OpenLN := proc(N::digit, O::symbol, F::{string, symbol}) local a, k, h, p, L, t, K;
  assign(K = cat("", F)), `if`(6 < N, ERROR("the first argument should be less than 6, but has
  received <%1>", N), `if`(member(O, {'fopen', 'open'}), assign(h = iostatus(), L = {},
  a=DoF1(K)),ERROR("mode of opening should be {open, fopen}, but has received %1",O)));
  if a = 2 then error "file <%1> is already open", F elif member(a, {3, 4}) then
    error "<%1> is a directory", K end if;
  if nargs = 3 then L := {k $ (k = 0 .. 6)} else
    seq(assign('L' = {op(L), h[k][1]}), k = 4 .. nops(h)), assign(p = {k $ (k = 0 .. 6)} minus L)
  end if;
  if member(N, L) then error "logical I/O channel <%1> is busy", N end if;
  member(N, p, 't'), seq(open(cat(currentdir(), "/$Art17_Kr9$.", k), 'WRITE'), k = 1 .. t - 1);
  try O(K, seq(args[k], k = 4 .. nargs)), p minus {N}, seq(fremove(cat(currentdir(),
  "$Art17_Kr9$.", k)), k = 1 .. t - 1)
  catch "too many files already open": seq(fremove(cat(currentdir(), "/$Art17_Kr9$.", k)),
  k = 1 .. t - 1)
  catch "file I/O error": seq(fremove(cat(currentdir(), "/$Art17_Kr9$.", k)), k = 1 .. t - 1);
    error "access error to datafile %1; possibly, file has attribute
    {readonly, hidden, or/and system}", F
  end try
end proc:
> OpenLN(3, open, "D:\\Math_myLib\\Module", 'READ'), [iostatus()];
3, {0, 1, 2, 4, 5, 6}, [5, 0, 7, [3, "D:\\Math_myLib\\Module", RAW, FD = 11, READ, BINARY]]

```

В дальнейшем нам понадобится ряд понятий, довольно хорошо известных читателю, имеющему опыт программирования. Это относится к таким понятиям как *тип файла: буферизированные (STREAM) и небуферизированные (RAW)*. Какой-то особой разницы между ними основные средства доступа *Maple* не делают, однако в их реализации первые во временном отношении более реактивны, т.к. используют механизм *буферизации* [8,11]. Небуферизированный тип файлов используется, как правило, в случае необходимости оперативного запроса состояния того либо другого ресурса *ЭВМ*. Тогда как, в общем случае следует использовать *буферизированный* тип файла, используемый *большинством* функций доступа *Maple* по умолчанию. Функции языка (*например, iostatus*), которые возвращают состояние системы *в/в*, для идентификации файлов буферизированных и небуферизированных используют идентификаторы *STREAM* и *RAW* соответственно.

Большинство операционных сред (*MS-DOS, Windows, MAC и др.*) различают файлы, рассматривая их организацию в виде последовательности символов (*текстовые*) либо последовательности байтов (*бинарные*). Основой различия является выделение в них специальных управляющих байтов, идентифицирующих состояние «*перевод строки и возврат каретки*». Это состояние исторически восходит к принципам работы *пишущих машинок* по созданию текстовых документов и в консоли *ЭВМ* реализуется клавишей *Enter*. В среде некоторых других операционных систем различия между файлами этих типов могут носить несколько иной характер, невидимый на уровне рассматриваемых нами средств доступа пакета *Maple*. Средства доступа пакета поддерживают работу с

текстовыми и с бинарными файлами. Функции доступа *Maple*, различающие оба типа файлов, используют для них идентификаторы *TEXT* (текстовый) и *BINARY* (бинарный) соответственно. На внутреннем уровне пакет средствами внутренней *I/O*-библиотеки (*iolib*-библиотеки) обеспечивает доступ к файлам данных следующих пяти видов:

STREAM – буферизированный файл, соответствующий стандарту *I/O*-библиотеки *C*-языка;

RAW – небуферизированный файл, соответствующий стандарту *Unix* и ряда других сред;

PIPE – двухконцевой канал, соответствующий *Unix*-стандарту и поддерживаемый только операционной *Unix*-платформой;

PROCESS – канал, который одним концом связан с другим процессом по *Unix*-стандарту;

DIRECT – прямой доступ к текущему (по умолчанию) либо высшего уровня потоку вывода/ввода (терминал) {в частности, одновременный ввод запросов на обслуживание и получение возвращаемых результатов их обработки в интерактивном режиме работы с ядром *Maple*}.

Для обеспечения доступа к файлу любого из упомянутых пяти видов он должен быть предварительно открыт в одном из двух режимов доступа *READ* (чтение) или *WRITE* (запись). При этом, режим открытия определяется или явно в момент вызова функций открытия: *foren*, *open*, *pipe* или *ropen*, либо неявно в результате вызова ряда функций *в/в* внутренней *I/O*-библиотеки пакета. Если же файл вида {*STREAM* | *RAW* | *DIRECT*} открыт в *READ*-режиме доступа, попытка записи в него информации автоматически закрывает его с последующим переоткрытием в *WRITE*-режиме доступа к последней сканированной до этого позиции файла. Эта особенность позволяет перлюстрировать файл до нужной позиции, а затем переоткрывать его на запись с данной позиции. Но другие виды файлов данных, открытые явно, не могут изменять режим доступа без их предварительного явного закрытия. Попытка записи в файл, отмеченный как «только для чтения (*readonly*)» вызывает ошибочную ситуацию. Детальнее реакция пакета на использование указанных режимов доступа рассмотрена в [9,28-37]. Функции доступа, использующие в своих форматах *типы* режима, используют для них соответственно специальные идентификаторы *READ* (чтение) и *WRITE* (запись).

Для файлов *DIRECT*-вида дополнительно определяются два предопределенных типа: *default* и *terminal*. Подобно развитым программным средствам функции доступа *Maple* интерпретируют интерфейс с пользователем как файлы с именами *двух* типов *default* и *terminal*. Первый из них определяет текущий входной поток, в рамках которого пакет читает и обрабатывает запросы пользователя, тогда как второй файл определяет входной поток высшего уровня, который в начале загрузки пакета являлся текущим. Первый файл определяет ввод/вывод пакета по умолчанию; например, в интерактивном режиме под *default*-файлом понимается ввод/вывод с консоли (клавиатура+дисплей). Тогда как второй определяет ввод/вывод высшего уровня текущего сеанса работы с пакетом. В режиме интерактивного функционирования пакета оба типа пакетных файлов будут совпадать. А именно, при отсутствии *read*-предложения *terminal*-ввод эквивалентен *default*-файлу, а при отсутствии процедур *writeto* или *appendto terminal*-вывод также эквивалентен *default*-файлу. Различия возникают лишь в случае чтения/записи *Maple*-предложений посредством указанных функций доступа к файлам.

Тут же уместно напомнить, что для идентификации в функциях доступа файла пакет использует два подхода: на основе его спецификатора (*СФ*), определяющего полный

путь к нему, и номера логического канала \mathcal{V}/\mathcal{V} , детально рассматриваемого в [28-37]. В принципе, оба способа определения файла эквивалентны, однако в конкретном случае предпочтение отдается одному из них. Так, предпочтение второму способу отдается при программировании задач доступа, характеризующихся частыми обращениями к файлу. Наряду с этим, второй подход делает программы более гибкими к возможным модификациям. С ростом опыта программирования задач доступа сам пользователь практически начинает ощущать всю целесообразность использования того или иного способа в каждом конкретном случае. Большинство функций доступа в пакете *Maple* допускает использование как первого, так и второго способа идентификации файлов.

Между тем, *Maple* не отождествляет **СФ**, закодированные хотя и одинаковыми, но на различных регистрах буквами и/или символами разделителей каталогов {"/", "\\\"}. В этой связи появляется возможность открывать один и тот же файл на разных каналах и в различных режимах доступа. Так, наша процедура *open2* обеспечивает открытие файла сразу на двух логических каналах \mathcal{V}/\mathcal{V} в различных режимах доступа [45]. Вызов процедуры *open2(F, L, P, N)* обеспечивает открытие файла со **СФ** F на двух логических каналах \mathcal{V}/\mathcal{V} , чьи номера возвращаются через аргументы L, P , а через N – размер файла в байтах соответственно. Файл F открывается посредством функции *open* на канале L и посредством функции *fopen* на логическом канале P . Успешный вызов *open2(F, L, P, N)* процедуры возвращает *NULL*-значение, т.е. *ничего*. Дополнительно, вызов процедуры допускает до трех аргументов, определяющих режимы доступа на каналах L, P и тип файла по каналу P соответственно. Детально с процедурой можно ознакомиться в [9]. С другими нашими средствами, относящимися к открытию файлов данных, а также с некоторыми их особенностями, полезными для программирования задач обработки файлов, можно ознакомиться в наших книгах и Библиотеке [9-14, 28-37, 42-45].

В общем случае процедура доступа к файлам имеет вложенную структуру, в которой ядро, состоящее из функций собственно обработки файла (*запись, чтение, дописывание и др.*), обрамляется оболочкой, содержащей функции *открытия* и *закрытия* файла. При этом, следует иметь в виду, процедура *закрытия* файла в общем случае *необязательна*, ибо в результате завершения работы с пакетом (*Exit, quit, done, stop*) или выполнения **restart**-предложения *все* открытые в текущем сеансе пакета файлы будут закрыты. Но во избежание потери информации (*в ряде случаев из пакетного буфера обмена в файлы не могут быть переданы все данные*) желательно выполнять явное закрытие всех открытых файлов пользователя. Более того, ввиду возможности одновременно открывать лишь относительно *небольшое* число файлов (*не более 7*) следует довольно четко отслеживать режим своевременного закрытия не требующих доступа файлов, давая возможность открывать для доступа другие файлы, не вызывая ошибочных ситуаций типа «... too many files already open». С целью устранения данного ограничения и была разработана упомянутая выше процедура *holdof*, сохраняющая свою актуальность до **10**-го релиза *Maple* включительно, и только, начиная с релиза **11**, данное ограничение было снято.

Для обеспечения доступа к *буферизированным (Б)* и *небуферизированным (нБ)* файлам *Maple* располагает парами функций открытия/закрытия файлов соответственно, а именно:

(Б)	<i>fopen</i> (<СФ> , { READ WRITE APPEND {, TEXT , BINARY }))	<i>fclose</i> (<СФ>)
(нБ)	<i>open</i> (<СФ> , { READ WRITE })	<i>close</i> (<СФ1> , ..., <СФn>)

Следует отметить, что *Maple* имеет соответствующее функциональное обеспечение и для оболочки процедур доступа к файлам видов *PIPE* и *PROCESS* (модульные функции *propen* и *pclose* из *process*-модуля пакета). Однако здесь эти средства не обсуждаются, т. к. файлы *PIPE*-вида поддерживаются только для *Unix*-платформы, а *PROCESS*-вида лишь для платформ, обеспечивающих многопроцессовый режим функционирования (например, *Unix*-систем). Средства оболочки функций доступа пакета *Maple* здесь не рассматриваются, т.к. достаточно детально рассмотрены в наших уже цитированных книгах. Еще раз следует отметить, что среди отечественных изданий по пакету *Maple* именно в наших книгах средства доступа к файлам рассмотрены наиболее детально.

Здесь отметим лишь одно полезное соображение, вытекающее из того факта, что при кодировании имени файла или его **СФ** хотя и одинаковыми, но на разных регистрах буквами и/или символами разделителей каталогов {"/", "\""} появляется возможность открывать один и тот же файл на разных каналах и в различных режимах доступа. В наших ранее упомянутых книгах приведен целый ряд весьма интересных примеров, иллюстрирующих использование этой возможности. Ниже приведен весьма простой фрагмент, иллюстрирующий использование данной возможности для тестирования файла на предмет быть палиндромом. С этой целью файл открывается на различных логических каналах *б/б* и сравнение символов производится с начала и конца файла.

```
> RevPath := proc(F::{string, symbol}) local a, k;
    a := convert(cat("", F), 'bytes');
    for k to nops(a) do
        a[k] := `if` (belong(a[k], 65 .. 90), a[k] + 32, `if` (belong(a[k], 97 .. 122), a[k] - 32,
            `if` (a[k] = 47, 92, `if` (a[k] = 92, 47, a[k])))
    end do;
    convert(a, 'bytes')
end proc;
> RevPath("D:/Math_myLib/TestFile_68"); => "d:\\mATH_MYLIB\\tESTfile_68"
> FilePal := proc(F::file) local a, b, c, d, k, x, y, n, m;
    a := filepos(F, infinity), close(F);
    b, c, d := fopen(F, READ, TEXT), fopen(RevPath(F), READ, TEXT), trunc(1/2*a);
    for k from 0 to d - 1 do assign('x' = filepos(b, k), 'y' = filepos(c, a - k - 1));
        assign('n' = readbytes(b, TEXT), 'm' = readbytes(c, TEXT));
    if n = m then next else return false, [x, n], [y, m], Close() end if
    end do;
    true, Close()
end proc;
> H:= "Mathematical Theory of Homogeneous Structures": H:= cat(H, Rlss(H));
> F:= "D:\\Math_myLib\\testfile": writebytes(F, H): Close(); FilePal(F); => true
> FilePal("D:\\Math_myLib\\testfile_68"); => false, [5, "Z"], [85, "Q"]
```

Первая часть фрагмента представляет простую процедуру *RevPath*, обеспечивающую возврат модифицированного пути к файлу, заданному ее фактическим аргументом, в

соответствии с простым алгоритмом – буквы заменяются на те же *самые*, но на другом регистре, а разделители подкаталогов "/" заменяются на "\\" и наоборот. Вторая часть фрагмента представляет процедуру *FilePal(F)*, которая тестирует текстовый файл *F* на предмет быть палиндромом. При обнаружении палиндрома *вызов* процедуры *FilePal* возвращает *true*, иначе возвращается 3-х элементная последовательность, чьим *первым* элементом является *false*, остальные *два* представляют списки с номерами позиций, по которым обнаружены различия и сами *различающиеся* символы. Итак, открывая файл по двум различным логическим каналам, получаем возможность вести его обработку с *двух* направлений одновременно – с начала файла и с его конца до его середины. Но отметим, что приведенный пример носит несколько искусственный характер, т.к. для этой цели существует *одноименная* процедура в библиотеке [45], предназначенная для тестирования произвольного *текстового файла*. Однако, этот пример весьма наглядно иллюстрирует возможность использования файла, открытого по разным логическим каналам *в/в*. Между тем, имеются намного более серьезные примеры такого подхода.

Для контроля за режимами открытия и закрытия файлов служит функция *iostatus()*, чей вызов позволяет получать информацию по состояниям всех открытых в текущем сеансе файлов. При этом, учитываются как явно открытые по функциям *{fopen, open, pipe}*, так и *неявно* другими функциями доступа (*например, fprintf-функцией*). Функция возвращает результат своего вызова в виде списка, содержащего по меньшей мере три элемента, первые из которых соответственно определяют:

iostatus()[1] – общее число открытых функциями доступа файлов в текущем сеансе;

iostatus()[2] – общее число текущих активных вложенных операций чтения файлов;

iostatus()[3] – верхнее значение для *iostatus()*[1]+*iostatus()*[2], т.е. максимально допустимое число одновременно открываемых в текущем сеансе внешних файлов.

Но, начиная с четвертого (*при наличии открытых файлов*), элементы списка, который возвращается *iostatus*-функцией, представляют собой списки (числом соответствуют числу открытых в текущий момент файлов). Каждый из 6-ти элементов такого списка, описывающего соответствующий открытый файл, соответственно определяет:

1 – номер логического канала *в/в* (определяется функциями доступа, открывшими файл);

2 – спецификатор файла (полный путь к файлу в системе каталогов *Windows-среды*);

3 – вид файла данных (*STREAM, RAW, PIPE, PROCESS, DIRECT*);

4 – *FP* = адрес указателя файла *STREAM* или *PROCESS* и *FD* = номер файлов другого вида;

5 – текущий режим доступа к файлу данных (*READ, WRITE*);

6 – тип файла данных (*TEXT, BINARY*).

При этом, второй элемент списка-описателя файла представляет непосредственный спецификатор файла, если файл имеет *{STREAM | RAW}*-вид либо *{default | terminal}*-идентификатор для файлов *DIRECT*-вида. Четвертый элемент списка-описателя дает внутреннее значение или для указателя данного файла (*FP*), или номера логического канала *в/в* (*FD*). Эта информация представляет особый интерес при работе с файлами *PIPE*-вида, ибо значения номеров внутреннего логического канала и возвращаемого функциями доступа *{fopen, open, popen}* в общем случае не совпадают. При отсутствии открытых файлов вызов *iostatus*-функции возвращает список [0, 0, 7]. Отметим, что в

отличие от предыдущих релизов, начиная с 11-го релиза, пакет допускает открытие более 7 логических каналов *б/б*, тогда как для предыдущих релизов такая возможность обеспечивалась исключительно нашей процедурой *holdof* [9-14,45].

Заккрытие всех открытых в текущем сеансе файлов выполняется как по предложению или функции {*done* | *stop* | *quit*}, так и по клавишам (*Alt+F4*) выхода из сеанса, а также по **restart**-предложению, которое восстанавливает состояние пакета *Maple* на момент его начальной загрузки. Тогда как *выборочное* закрытие файлов выполняется по {*close* | *fclose* | *pclose*}–функции для ранее открытых по функции {*fopen* | *open* | *pipe*, *popen*} файлов. Все эти функции имеют единый формат кодирования следующего простого вида:

$$\{fclose | close | pclose\}(<C\Phi 1>, <C\Phi 2>, ..., <C\Phi n>)$$

где в качестве единственного фактического аргумента выступает последовательность *спецификаторов закрываемых* файлов и/или *логических каналов б/б*. При этом, успешное завершение каждой из данных функций доступа закрывает соответствующие файлы и возвращает значение *NULL*. Явное закрытие файлов по рассмотренным функциям доступа освобождает занимаемые ими *логические каналы б/б* и обеспечивает *реальную* запись информации в файлы. При этом, все функции закрытия файлов *эквивалентны* и *взаимозаменяемы*. Попытка закрыть несуществующий файл или незадействованный логический канал *б/б* вызывает ошибочную ситуацию для пакета *Maple 6/7*, тогда как для *Maple 8–11* возвращается *NULL*-значение, как и в случае существующих закрытых файлов во всех релизах. Это одна из довольно досадных *недоработок*, способствующая несовместимости релизов. Более того, попытка закрыть файл {*default* | *terminal*}–вида вызывает ошибочную ситуацию, тогда как после закрытия логического канала *б/б* его номер может вновь использоваться в текущем сеансе пакета.

Наконец, следующие две функции доступа позволяют определять режим *обязательной записи* информации в файл и *удалять* как закрытые, так и открытые файлы файловой системы компьютера соответственно. Например, по функции *fflush*(<CΦ1>, ..., <CΦn>) обеспечивается режим *обязательной записи* в файл при выполнении каждой подобной операции. Это распространяется как на файлы, открываемые *неявно*, так и на файлы, открываемые *явно* по {*fopen* | *popen*}–функции. При этом, эта процедура выполняется автоматически при *явном* закрытии файла. Применение *fflush*-функции видится нам целесообразным, например, при режиме работы с *данными*, когда в течение текущего сеанса производится *длительное* формирование результатов вычислений при условии периодического сохранения их в файлах в режиме дописывания (*APPEND*), позволяя достаточно существенно повышать надежность *сохранности* полученных результатов, перед реальной записью в файл накапливающихся в пакетном буфере обмена.

Тогда как по функции *fremove*(<CΦ1>, ..., <CΦn>) производится удаление из файловой системы компьютера указанных своими *спецификаторами* и/или номерами *логических каналов б/б* файлов. В случае применения *fremove*-функции к *открытому* файлу, он предварительно закрывается, а затем удаляется. Данная функция часто используется для обеспечения *пустоты* файла, в который должна производиться запись. Это вовсе не является лишним, т.к., например, по *writeline*-функции будет производиться запись поверх уже существующей в файле в его начало, не изменяя *остального* содержимого.

Это может привести к ситуации, когда начало файла содержит новую информацию, а его конец – старую, т.е. не производится полного обновления содержимого файла. Использование *remove*-функции к отсутствующему файлу инициирует ошибочную ситуацию, которую можно обрабатывать программно. Успешное завершение вызова функции *remove* возвращает *NULL*-значение. В наших книгах [9-14,28-37,42] можно найти довольно интересные примеры, иллюстрирующие применение рассмотренных функциональных средств, образующих оболочку средств доступа к *внешним* файлам: открытие и закрытие файлов разных типов, проверка состояния открытых файлов, а также удаление файлов наряду с анализом основных особых ситуаций, и обсуждение ряда наших средств, *расширяющих* функциональные возможности, поддерживающие оболочку функциональных средств доступа к файлам данных [42-45].

При работе с файлами достаточно существенную ее компоненту составляет *обработка* особой ситуации «конец файла», которая особенно актуальна при применении средств доступа, поддерживаемых *Maple*, для чего *Maple* предлагает тестирующую функцию *feof(<CФ>)*, возвращающую *true*-значение только тогда, когда в процессе применения функций доступа *readline*, *fscanf* или *readbytes* к файлу *STREAM*-вида, заданному его фактическим аргументом (*спецификатор либо номер логического канала*), была реально обнаружена ситуация «конец файла», в противном случае возвращается *false*-значение. При этом, если в качестве аргумента *feof*-функции был указан спецификатор и файл ранее не открывался, то он открывается в *READ*-режиме доступа как *BINARY*-файл.

По функции *filepos(<CФ> {, p})* возвращается номер текущей сканируемой позиции в файле, определенном первым аргументом (*спецификатор либо номер приписанного ему логического канала*). Под сканируемой понимается позиция файла, в которую установлен указатель, т.е. в которую будет писаться или из которой будет читаться информация при очередной операции доступа к файлу. Более того, под информацией (*логической записью*) понимается порция информации, записываемая/читаемая *в(из) файл(а)* за одну операцию обмена. Так, для *{writeline | readline}*-функции логической записью является строка, а для *{writebytes | readbytes}*-функции – заданное число байтов (*символов*). Если указан второй необязательный аргумент функции, определяющий номер позиции в файле, то производится установка указателя в эту позицию с возвратом ее номера. В качестве номера позиции допускаются целочисленные значения из диапазона *[0 .. ∞]*; нулевое значение *filepos*-функция возвращает для *пустого* файла или по достижении конца файла. Тогда как по вызову функции *filepos(<CФ>, ∞)* производится установка указателя в конец файла с возвратом *количества* байтов, образующих файл, заданный ее первым фактическим аргументом. Согласно представленным в книгах [9-14,28-37] замечаниям *feof*-функция не может применяться для надежного тестирования особой ситуации «конец файла», поэтому в *READ*-режиме к *TEXT*-файлам следует применять непосредственно *readline*-функцию или *filepos*-функцию. В указанных книгах можно найти примеры использования *filepos*-функции для некоторых вариантов обработки ситуации «конец файла», которые могут оказаться довольно полезными на практике.

Довольно существенное различие между файлами типов *TEXT* и *BINARY* имеет место относительно обработки ситуации «конец файла». Для *BINARY*-файлов у пакета нет средств для обработки такой особой ситуации и в каждом конкретном случае данная

задача возлагается на пользователя, что во многих случаях весьма нежелательно. Этот вопрос довольно детально рассматривался в наших книгах [9-14,28-37,42-44]. С целью решения данной проблемы нами были предложены две процедуры *Fend* и *Find* [45].

Следующий фрагмент представляет исходный текст процедуры *Find* и результаты ее применения для обработки ситуации «*конец файла*» по сравнению с использованием для этой же цели стандартной процедуры пакета *feof*.

```
> Find := proc(F::nonnegint) global __filesize;
  [if (type(eval(cat(__filesize, F)), 'symbol'), [assign(cat(__filesize, F) = filepos(F, infinity)),
  filepos(F, 0)], NULL), if (eval(cat(__filesize, F)) <= filepos(F),
  [true, unassign(cat(__filesize, F))][1], false)][-1]
end proc;

> S:=NULL: p:=open("D:\\Math_myLib\\Test.txt", READ): while not Find(p) do S:=S,
  readline(p) end do: close(p), S;
  "string1", "string2", "string3", "string4", "string5"
> S := NULL: p:=open("D:\\Math_myLib\\Test.txt", READ): while not feof(p) do S:=S,
  readline(p) end do: close(p), S;
  Error, (in feof) operation only supported for STREAM files
> S := NULL: p:=open("D:\\Math_myLib\\Test.txt", READ): while not Find(p) do
  S:=cat(S, readbytes(p, TEXT)) end do: close(p), S;
  "string1"
  string2
  string3
  string4
  string5"
> S := NULL: p:=open("D:\\Math_myLib\\Test.txt", READ): while not feof(p) do
  S:=cat(S, readbytes(p, TEXT)) end do: close(p), S;
  Error, (in feof) operation only supported for STREAM files
```

Вызов *Fend*(*F* [, 'h']) возвращает *true* при обнаружении у файла *F*, заданного номером логического канала *h/h* либо *СФ*, ситуации «*конец файла*», и *false* в противном случае. Через второй необязательный аргумент в случае *false*-значения возвращается список формы [*сканируемая позиция файла, длина остатка файла*]. При этом необходимо иметь в виду, процедура *Fend* имеет смысл лишь для открытого файла. Тогда как в отличие от *Fend*, процедура *Find* имеет только один аргумент – номер логического канала, по которому был открыт тестируемый файл данных.

В процессе выполнения *процедур* доступа к файлам возможны различного рода *особые* и *ошибочные* ситуации, наиболее часто встречающиеся из которых рассмотрены в [28-37]. Обработку этих ситуаций можно производить на *основе* соответствующих средств пакета и, прежде всего, на основе предложений *error* и *try*, из которых предложение *try* является наиболее развитым средством обработки ошибочных ситуаций в целом. Следующий фрагмент иллюстрирует применение *try*-предложения при реализации процедуры *BootDrive*, чей вызов *BootDrive()* возвращает 2-элементный список, *первый*

элемент которого определяет имя устройства начальной загрузки системы, тогда как второй элемент определяет путь к главному системному каталогу компьютера.

```

> BootDrive := proc() local a, b, c, v, k, t1, t2, f, D, x, y, w, G;
  D := (x, y) -> op({seq(`if`(search(x, k), RETURN(k), false), k = y)});
  G := proc(k, w) local n, h, z;
    z := cat(k, ":\Boot.ini"); n := fopen(z, 'READ', 'TEXT');
    while `not`(Find(n)) do h := CF2(readline(n));
    if Search2(h, {"signature(", "multi(") <> [] and search(h, cat(" ", w)) then
      return Close(z), map(Case, [k, cat(k, ":", FNS(sextr(h, "\\", ssign)[1], " ", 3))], 'upper')
    end if
    end do;
    Close(z), ERROR("system file BOOT.INI is corrupted")
  end proc;
  assign(a = map(CF2, {Adrive()}), t1 = {"95", "98", "me"}, t2 = {"2003", "2000", "nt", "xp"});
  assign(f = cat([libname][1][1 .. 2], "\\_$_Art21_Kr14$_"), system(cat("Ver > ", f));
  assign(v = CF2(readbytes(f, 'TEXT', infinity)), delf1(f, `*`), assign(w = D(v, t1 union t2));
  if member(w, t2) then for k in a do f := cat(k, ":\PageFile.sys");
    try open(f, 'READ'); close(f)
    catch "file or directory does not exist": next
    catch "file or directory, %1, does not exist": next
    catch "file I/O error": return G(k, w)
    catch "permission denied": return G(k, w)
  end try
  end do
  elif member(w, t1) then
    for k in a do f := cat(k, ":\MsDos.sys");
      if not type(f, 'file') then next end if;
      do b := readline(f);
      if b = 0 then WARNING("file MSDOS.SYS on <%1> is corrupted", k); close(f); break
      end if;
      if search(CF2(b), "windir") then close(f); b := FNS(b[Search(b, "=")[1] + 1 .. -1], " ", 3);
        break end if
      end do;
      if b = 0 then next end if;
      try open(f, 'READ'); close(f)
      catch "file or directory does not exist": next
      catch "file or directory, %1, does not exist": next
      catch "file I/O error": return map(Case, [b[1], b], 'upper')
      catch "permission denied": return map(Case, [b[1], b], 'upper')
    end try
  end do;
end do;

```

```

error "fatal system error: correct file MSDOS.SYS has not been found on %1",
    map(Case, a, 'upper')
else error "System error: unknown host operating system <%1>", v
end if
end proc:
> BootDrive(); ⇒ ["C", "C:\\WINDOWS"]

```

Процедура *BootDrive* успешно функционирует в среде пакета *Maple* релизов 7–11 на платформе *Windows* {95 | 98 | ME | 2000 | 2003 | NT | XP}. Детальнее с основными особыми и ошибочными ситуациями *в/в Maple* можно ознакомиться по ?IO_errors-конструкции.

Не отвлекаясь теперь на детали, принципиальную схему организации средств доступа к внешним файлам можно с учетом вышесказанного представить в форме следующей простой схемы, позволяющей систематизировать изложение, хотя она в значительной степени и носит довольно субъективный характер, а именно:

Оболочка средств доступа: <i>fopen, open, popen</i> (функции открытия файлов)		
TEXT-файлы	BINARY-файлы	FORMAT-файлы
<i>writeline, readline</i>	<i>writebytes, readbytes</i>	<i>writedata, readdata</i>
<i>writestat, readstat</i>		<i>fprintf</i>
<i>writeto, appendto</i>		<i>fscanf</i>
Оболочка средств доступа: <i>fclose, close, pclose</i> (функции закрытия файлов)		
Общие функции доступа: <i>feof, filepos, fremove, fflush, iostatus, system</i>		

В данной схеме выделяются *три* уровня: оболочка, функции непосредственного доступа к файлам и общие обслуживающие функции. Оболочка средств доступа включает функции явного открытия и закрытия файлов; при этом, в режиме открытия можно определять как тип открываемого файла (*TEXT, BINARY*), так и режим доступа к нему (*APPEND, READ, WRITE*). Хотя средства данного уровня в общем случае вовсе не обязательны, т.к. сами функции непосредственного доступа (*второй уровень*) обеспечивают открытие файлов, а по завершении работы с *Maple* все открытые в сеансе файлы автоматически закрываются, в целом ряде случаев использование средств оболочки не только крайне желательно, но и обязательно, позволяя обеспечивать более гибкое управление общим механизмом доступа к внешним файлам пакета. Функциональные средства оболочки на вполне достаточном уровне были рассмотрены выше.

Третий уровень составляют общие средства доступа, обеспечивающие ряд сервисных процедур по обслуживанию файлов данных. Набор их невелик и поддерживаемые ими процедуры сводятся к удалению файлов (*remove*) из файловой системы компьютера, тестированию состояния открытых файлов (*iostatus*), а также идентификации особой ситуации «конец файла» (*feof*) и текущей позиции сканирования в файле (*filepos*) вместе с обеспечением гарантированной записи данных в файл (*fflush*). Посредством *system*-функции пользователь имеет возможность непосредственно в среде пакета выполнять команды системы *DOS*, в частности, команды обслуживания файловой системы. Это обстоятельство позволяет вполне эффективно обходиться имеющимися у ядра *Maple* собственными средствами обслуживания файлов данных.

Наконец, *второй* уровень схемы представляет собственно функциональные средства непосредственного доступа к внешним файлам пакета, обеспечивая операции записи и чтения данных. Эти средства как в плане их основной *функциональной* нагрузки, так и в плане освоения целесообразно разделить на *три* группы, обеспечивающие работу с файлами соответственно типов **TEXT**, **BINARY**, **FORMAT**. *Первые две* группы файлов стандартно определяются **Maple** и для непосредственного доступа к ним имеется ряд встроенных функций. Тогда как третий тип определен нами условно и пакетом явно не поддерживается, однако он введен нами для более четкого представления общей *схемы* организации доступа к *внешним* файлам **Maple**. При этом, в целом ряде случаев классифицированные нами по *трем* группам средства могут взаимно перекрываться функционально, однако это вопрос отдельного рассмотрения [28-37]. Суть каждого из *указанных* типов доступа к файлам наряду с поддерживающими их средствами пакета весьма детально рассматриваются в наших работах [9-12,28-37,42-44] и каких-нибудь затруднений не вызывают. Более того, рассмотрение данной тематики в упомянутых книгах является наиболее детальным в отечественной литературе по пакету **Maple**. К тому же для *цели* настоящего рассмотрения более детальное обсуждение этих средств не является достаточно важным. Сделаем только некоторые полезные замечания.

С целью *расширения* стандартных функциональных средств доступа к *файлам* данных пакет, начиная с релиза **9**, снабжен пакетным модулем **FileTools**, содержащим набор средств, расширяющих стандартные средства доступа к файлам. Модуль включает **2** подмодуля **Text** и **Binary**, соответственно включающие средства обработки *текстовых* и *бинарных* файлов наряду со средствами, ориентированными на любой тип файлов.

Текстовые файлы содержат некоторые форматирующие понятия, не применимые к бинарным данным (*например, строки и символы перевода строк*). Далее, данные в таком текстовом файле представляются символами. Внутреннее представление данных для такого файла конвертируется в строку *перед* записью в файл. Тем временем *бинарные* файлы содержат данные в бинарном формате. На текущий момент это означает, что бинарные файлы содержат данные только типов *{integer, float}* и данные аппаратных средств компьютера, которые могут быть обработаны средствами модуля **Binary**. При этом, **FileTools** содержит **24** средства общего назначения для работы с файлами.

Наша библиотека также содержит целый ряд средств работы с файлами данных типа *{TEXT, BINARY, FORMAT}*, а также с файлами произвольного типа и каталогами [45], которые не только достаточно существенно расширяют стандартные средства пакета, послужили *прообразом* ряду средств для модуля **FileTools**; при этом, некоторые из них и на сегодня являются уникальными. В частности, до сих пор в пакете нет средств для создания произвольной цепочки подкаталогов; такую задачу успешно решает **MkDir**-процедура. Наряду с этим, нами был создан ряд других полезных средств для работы с файлами данных **TEXT**-типа. В частности, процедуры *{writedata1, readdata1}* весьма существенно расширяют возможности стандартных процедур *{writedata, readdata}* и обеспечивают сохранение в файле данных любого типа *{list, listlist, vector, matrix, Matrix}*. Процедуры **DAopen**, **DAread** и **DAClose** [45] обеспечивают поддержку *прямого* доступа к файлам **TEXT**-типа (*и даже более общего rlb-типа*), точнее к его записям. Так, наряду с представленными, для файлов данных **TEXT**-типа нами было создано **36** достаточно

полезных средств различного назначения, с которыми можно ознакомиться в [28–37]. Многие из них представляют не только чисто практический интерес, но и достаточно полезны с учебной точки зрения, используя эффективные приемы программирования алгоритмов, специфических для задач доступа к текстовым файлам данных.

Для обеспечения доступа к файлам на уровне произвольных выражений *Maple*-язык располагает функциями *writestat* и *readstat*, которые обеспечивают запись и чтение логических записей, в качестве которых выступают последовательности выражений, разделенные запятой и завершающиеся управляющими *hex(0D0A)*-символами «перевод строки и возврат каретки». В этом отношении указанные средства также имеют дело с внешними файлами *TEXT*-типа. Следует иметь в виду, что файлы данных, созданные посредством *writestat*-функции, не читаются предложением *read*. Поэтому, для этой цели можно использовать нашу достаточно несложную процедуру *wsread* [45].

В отличие от функциональных средств, обеспечивающих доступ к внешним файлам данных на уровне логических записей, в качестве которых могут выступать строки или последовательности *Maple*-выражений, а также целые *Maple*-предложения, средства *{writebytes, readbytes}* обеспечивают доступ к файлам данных *BINARY*-типа на уровне отдельных символов (байтов). Более того, в отличие от предыдущих средств доступа, данные средства рассматривают все символы, составляющие файл, однородными, не различая в них управляющие символы *hex(0D0A)* «перевода строки и возврата каретки». В качестве расширения средств *Maple* для обеспечения доступа к файлам данных типа *BINARY* нами был создан ряд процедур, довольно полезных во многих приложениях, имеющих дело с файлами произвольного типа. Как уже отмечалось выше, довольно существенное различие между файлами типов *TEXT* и *BINARY* имеет место в первую очередь относительно обработки ситуации «конец файла». Так как у *Maple* нет средств обработки такой особой ситуации для файлов *BINARY*-типа, то для решения данной проблемы нами были представлены две процедуры *Fend* и *Find*. Тогда как процедура *statf* производит анализ открытых файлов относительно типов *TEXT* и *BINARY* [45].

Помимо средств работы с внешними файлами типов *TEXT* и *BINARY*, чья структурная организация базируется соответственно на логических записях строчного и символьного типов, *Maple* обеспечивает средства для работы с файлами, чья структура имеет более сложную форматизацию логических записей. Для этих целей *Maple* располагает 2-мя парами процедур *{writedata, readdata}* и встроенных функций *{fprintf, fscanf}*. Данные средства наряду с особенностями их применения и недостатками рассмотрены нами весьма детально в цитированных книгах. В частности, функции *writedata* и *readdata* имеют существенные ограничения по типам данных, обрабатываемых ими, а именно допускаются только типы *{integer, float, string}*. Между тем, во многих случаях имеется необходимость сохранения в *ASCII*-файлах данных произвольного типа. В частности, данные могут содержать значения *fraction*-типа и/или произвольные выражения. Эта проблема легко решается процедурой *writedata1* [45] и сопутствующей ей *readdata1*-процедурой. Процедура *writedata1(F, D)* пишет произвольные данные из выражений типов *{matrix, Matrix, list, listlist, vector}* в текстовый файл *F*. Если данные *D* представлены вектором либо списком значений, то значения выводятся в отдельной строке, иначе значения выводятся построчно с разделением их пробелом в строке. Вообще говоря, в

качестве данных могут выступать произвольные выражения, достаточно существенно расширяя сферу приложений данных средств при работе с файлами данных.

Еще на одном моменте следует акцентировать наше внимание. Выше уже отмечалось обстоятельство, при кодировании *спецификатора* файла в различных представлениях (*в частности, использование одинаковых букв на разных регистрах либо разных разделителей каталогов*) имеется возможность открытия одного и того же файла не только на разных логических каналах, но и для разных *режимов* доступа. Естественно, предоставляемой возможностью следует пользоваться весьма осмотрительно, однако выгоды она имеет немалые. В качестве примера приводится процедура *utfile(F, S)*, которая обеспечивает обновление «на месте» текстового файла *F* на основе *системы подстановок S*, обе части которых имеют одинаковые длины. Подстановки *S* задаются списком уравнений, чьи левые части по длине идентичны правым, и производятся во все строки текстового *F*-файла. Успешный вызов *utfile(F, S)* процедуры возвращает *NULL*-значение с выводом соответствующего сообщения. Следующий фрагмент иллюстрирует сказанное.

```
> utfile := proc(F::file, S::list(equation)) local a, b, d, k;
    seq(if (length(lhs(k)) = length(rhs(k)), NULL, ERROR("substitutions should have parts
        of identical length, but have received <%1>", k)), k = S);
    assign(a = fopen(F, 'READ'), b = RevPath(F));
    while not Fend(a) do d:=filepos(a); filepos(b,d); writeline(b,SUB_S(S,readline(a))) end do;
    close(a, b), WARNING("datafile <%1> had been updated in situ", F)
end proc;
> utfile("D:\\Math_myLib\\RANS_IAN.txt", ["avz"="ian", pub=420, "bk"="72"]);
Warning, datafile <D:\\Math_myLib\\RANS_IAN.txt> had been updated in situ
```

В данном разделе на содержательном уровне рассмотрены элементы использования и программирования функциональных средств доступа к внешним файлам, которые составляют основу прикладного программирования в среде *Maple* задач, связанных с обработкой данных, находящихся на *внешних* носителях. При этом, следует отметить, что искушенный пользователь вполне может ограничиться только представленными средствами доступа, сочетая их с иными функциональными средствами для создания собственных более развитых средств доступа. И для целого ряда приложений данная задача является достаточно актуальной, т.к. существующая в *Maple* система доступа к внешним файлам, на наш взгляд, – одно из наиболее слабых мест пакета, требующее существенной доработки и развития в ряде направлений. Именно в таком контексте и разрабатывались средства доступа, отраженные, например, в нашей Библиотеке [45] и в книгах [9-14,28-37,42-44]. Между тем, не взирая на вполне достаточный уровень (*с точки зрения целей данного издания*) представления средств доступа к *внешним* файлам, за более детальной информацией следует обращаться к упомянутым нашим книгам и Библиотеке, посвященным детальной проработке широкого круга аспектов данной проблематики. Там же можно познакомиться и с целым рядом специальных вопросов системы *Maple*, не нашедших отражения в литературе и документации по пакету. В целом, средства *Maple* доступа к внешним файлам данных, на наш взгляд, в значительной степени уступают аналогичным средствам пакета *Mathematica*.

8.4. 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

Детальнее с форматами, поддерживаемыми пакетом, можно ознакомиться в справке.

Mathematica обеспечивает достаточно эффективный системно-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 *в/в*>]

По вызову **Streams[]** возвращается список *Stream*-объектов файлов данных, открытых в текущем сеансе, включая системные файлы. Для получения списка *Stream*-объектов файлов, отличных от системных, можно использовать вызов процедуры **StreamsU[]**. В следующем фрагменте приведены исходные тексты упомянутых процедур, а также и примеры применения рассмотренных функций доступа к файлам данных.

```
In[1]:= Streams[]
Out[1]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[2]:= S1 = OpenRead["D:/Math_myLib/testfile"]
Out[2]= InputStream["D:/Math_myLib/testfile", 33]
```

```

In[3]:= S2 = OpenWrite["D:/Math_myLib/testfile.txt"]
Out[3]= OutputStream["D:/Math_myLib/testfile.txt", 34]
In[4]:= S3 = OpenAppend["D:/Math_myLib/testfile1.txt"]
Out[4]= OutputStream["D:/Math_myLib/testfile1.txt", 35]
In[5]:= Streams[]
Out[5]= {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[6]:= Close[S3]
Out[6]= "D:/Math_myLib/testfile1.txt"
In[7]:= Streams[]
Out[7]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
InputStream["D:/Math_myLib/testfile", 33], OutputStream["D:/Math_myLib/testfile.txt", 34]}
In[8]:= OpenWrite[]
Out[8]= OutputStream["C:\\Documents and Settings\\Aladjev\\Local Settings\\Temp
\\m-0a9fa8c8-b787-4e47-958b-c392c61f793c", 36]
In[9]:= Close["D:/Math_myLib/testfile1.txt"]
General::openx: D:/Math_myLib/testfile1.txt is not open. >>
Out[9]= Close["D:/Math_myLib/testfile1.txt"]
In[10]:= Close["D:/Math_myLib/test72.txt"]
General::openx: D:/Math_myLib/test72.txt is not open. >>
Out[10]= Close["D:/Math_myLib/test72.txt"]
In[11]:= Closes[x_] := Quiet[Check[Close[x], Null]]
In[12]:= Closes["D:/Math_myLib/test72.txt"]
In[13]:= Closes["D:/Math_myLib/testfile1.txt"]
In[14]:= SP = Streams[]
Out[14]= {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[15]:= StreamsU[] := Select[Streams[], !MemberQ[{"[stdout", "[stderr"],
StringTake[ToString[#1], {13, 19}]] &]
In[16]:= StreamsU[]
Out[16]= {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[17]:= close["D:/Math_myLib/testfile"]
Out[17]= "D:/Math_myLib/testfile"
In[18]:= StreamsU[]

```

```
Out[18]= {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_j ($j=1..n$), 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. По умолчанию вывод в файл выражений выполняется в строчном формате отдельными записями (*строками*). Следующий фрагмент вполне наглядно иллюстрирует применение функций **OpenWrite**, **Write**, **StreamsU** и **CloseAll**.

```
In[86]:= S = OpenWrite["D:/Math_myLib/file72"]
Out[86]= OutputStream["D:/Math_myLib/file72", 36]
In[87]:= For[k = 1, k <= 25, k++, Write[S, "RANS_" <> ToString[k]]]
In[88]:= {Streams[], Close[S], StreamsU[]}
Out[88]= {{OutputStream["stdout", 1], OutputStream["stderr", 2],
OutputStream["D:/Math_myLib/file72", 36]}, "D:/Math_myLib/file72", {}}
In[89]:= Write["D:/Math_myLib/file72", "International Academy of Noosphere"]
In[90]:= StreamsU[]
Out[90]= {OutputStream["D:/Math_myLib/file72", 40]}
In[91]:= Close[OutputStream["D:/Math_myLib/file72", 40]]
Out[91]= "D:/Math_myLib/file72"
In[92]:= StreamsU[]
Out[92]= {}
In[93]:= Write["D:/Math_myLib/file420", "International Academy of Noosphere"]
In[94]:= StreamsU[]
Out[94]= {OutputStream["D:/Math_myLib/file420", 42]}
In[95]:= {StreamsU[], Close[OutputStream["D:/Math_myLib/file420", 42]], StreamsU[]}
Out[95]= {{OutputStream["D:/Math_myLib/file420", 42]}, "D:/Math_myLib/file420", {}}
```

```

In[96]:= CloseAll[] := Map[Close, StreamsU[]]
In[97]:= CloseAll[]
Out[97]= {}
In[98]:= For[k = 1, k <= 3, k++, OpenWrite["C:/1" <> ToString[k]]]
In[98]:= StreamsU[]
Out[99]= {OutputStream["C:/1", 88], OutputStream["C:/2", 89], OutputStream["C:/3", 90]}
In[100]:= CloseAll[]
Out[100]= {"C:/1", "C:/2", "C:/3"}

```

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, завершенную символом перевода строки и возврата каретки;

Word – считывает последовательность символов, завершенную разделителем слов.

Смысл данных типов достаточно прозрачен и может быть детализирован в справке по пакету. Подобно пакету *Maple* пакет *Mathematica* также дает 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[1]:= F = "D:/Math_myLib/file72"; {S, S1} = {OpenRead[F],
      OpenRead[If[UpperCaseQ[StringTake[F, 1]], ToLowerCase[F], ToUpperCase[F]]]}
Out[1]= {InputStream["D:/Math_myLib/file72", 44], InputStream["d:/math_mylib/file72", 45]}
In[2]:= R = {}; For[k = 1, k <= 7, k++, R = Append[R, {Read[S], Read[S1]}]]; Flatten[R]
Out[2]= {"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[136]:= Read["D:\\Math_myLib\\file72"]
Out[136]= "RANS_1"
In[137]:= StreamsU[]
Out[137]= {InputStream["D:\\Math_myLib\\file72", 59]}
In[138]:= Read["D:\\Math_myLib\\file720"]
      Read::openx: D:\\Math_myLib\\file720 is not open. >>
Out[138]= Read["D:\\Math_myLib\\file720"]
In[146]:= Read["D:\\Math_myLib/file72", {Byte, Byte, Byte, Byte, Byte, Byte}]
Out[146]= {34, 82, 65, 78, 83, 95}
In[147]:= {h, S} = {"", {}}; While[h != EndOfFile, h = Read["D:\\Math_myLib\\file420",
      Word]; S = Append[S, h]]; CloseAll[]; Select[S, #1 != EndOfFile &]
Out[147]= {"International", "Academy", "of", "Noosphere", "RANS", "Tallinn", "April_2011"}
```

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.

Принимая во внимание внутреннюю структуру файла *m*-формата с пакетом, можно предложить довольно простую процедуру **ExtrPackName** для обработки такого файла.

```
In[111]:= ExtrPackName[F_String, N_String] := 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[112]:= ExtrPackName["D:\\Math_myLib\\AVZ_Package.m", "ArtKr"]
Out[112]= "Definition of ArtKr is absent in file <D:\\Math_myLib\\AVZ_Package.m>"
In[113]:= ExtrPackName["D:\\Math_myLib\\AVZ_Package.m", "Subs"]
In[114]:= ExtrPackName["D:\\Math_myLib\\AVZ_Package.m", "Df"]
In[115]:= Df[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2]
Out[115]= ((-1 + x^6) Cos[1/x^2] - (1 + x^6) Sin[1/x^2])/x^4

```

Успешный вызов **ExtrPackName**[*F*, *N*] процедуры возвращает определение объекта *N* с его вычислением, которое содержится в *m*-файле *F* с пакетом, делая это определение доступным в текущем сеансе. Если формат файла *F* отличен от *m*-формата, то вызов возвращает значение **\$Failed**, тогда как при отсутствии в файле запрошенного объекта *N* вызов процедуры возвращает соответствующее сообщение.

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 крайней мере одну из подстрок *h1*, ..., *hN*; при этом, **Find** работает и с закрытыми файлами.

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 текст с примером применения представляет следующий фрагмент вместе с примером использования рассмотренной стандартной функции **Skip** пакета *Mathematica*.

```

In[334]:= R = OpenRead["D:\\Math_myLib\\file72"]; {h, S} = {"", {}};
          While[h != EndOfFile, h = Read[R, String]; S = Append[S, h];
          Skip[R, String, 2]]; CloseAll[]; Select[S, #1 != EndOfFile &]
Out[334]= {"RANS_1", "RANS_4", "RANS_7", "RANS_10", "RANS_13", "RANS_16"}
In[335]:= FileOpenQ[F_] := Module[{a, b, c, d}, a = FileType[F];
          If[a === Directory, Return[Directory], If[a === None, Return[$Failed],
          c = Map[ToString, StreamsU[]]; d = StringLength[F] + 13;

```

```

a = Select[c, StringTake[#, d] == "InputStream[" <> F <> "," &];
b = Select[c, StringTake[#, d + 1] == "OutputStream[" <> F <> "," &];
If[a != {}, {True, "Input"}, If[b != {}, {True, "Output"}, False]]]]
In[336]:= Map[FileOpenQ, {"D:\\Math_myLib\\file72", "D:\\Math_myLib\\file420",
"C:\\temp", "C:\\Temp\\Academy\\RANS.IAN", "C:\\http.htm"}]
Out[336]= {{True, "Input"}, {True, "Output"}, Directory, $Failed, False}

```

Вызов функции **FileOpenQ[F]** возвращает список {True, "Input"}, если файл *F* *открыт* на чтение; {True, "Output"}, если файл *F* *открыт* на запись; Directory, если – *F* каталог; \$Failed, если *F* не определяет ни файла, ни каталога, и False в противном случае. Все 4 представленные функции доступа обеспечивают довольно эффективные средства для достаточно тонкого *манипулирования* с файлами и в сочетании с другими функциями *доступа* обеспечивают пользователя как стандартным набором средств для обработки файлов данных, так и дают возможность на их *основе* создавать собственные средства, которые позволяют как решать специфические задачи работы с файлами данных, так и в определенной степени расширять стандартные возможности пакета. Ряд *подобных* средств представлен и в настоящей книге, и в 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[6]:= FileNames["*", "d:/Book"]
Out[6]= {"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[557]:= 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[637]:= d = FileDate["C:\\Book\\MapleMathem.doc"]; DateString[d]
Out[637]= "Wed 9 Feb 2011 18:05:37"

```


SetFileDate[F] – изменяет даты обновления и доступа к файлу *F* на текущие, например:

```
In[640]:= d = FileDate["C:\\Http.htm"]; DateString[d]
```

```
Out[640]= "Tue 25 Jan 2011 16:00:44"
```

```
In[641]:= d = SetFileDate["C:\\Http.htm"]; p = FileDate["C:\\Http.htm"]; DateString[p]
```

```
Out[641]= "Wed 9 Feb 2011 19:14:16"
```

FileByteCount[F] – возвращает размер файла *F* в байтах, например:

```
In[638]:= FileByteCount["C:\\Book\\MapleMathem.doc"]
```

```
Out[638]= 6136832
```

FileType[F] – возвращает тип файла *F* в разрезах: **File**, **Directory**, **None** (отсутствует):

```
In[646]:= Map[FileType, {"C:\\Book\\Addition.doc", "D:/Math_myLib", "RANS_IAN"}]
```

```
Out[646]= {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; для устранения данной ситуации была разработана довольно простая процедура **FileFormat1**, которая корректно обрабатывает главные каталоги, а также верно обрабатывает недоступные либо несуществующие устройства внешней памяти, например:

```
In[650]:= Map[FileFormat, {"C:\\Book\\Addition.doc", "C:/", "D:/Math_myLib/Art1"}]
```

```
Out[650]= {"DOC", "Directory", "Table"}
```

```
In[675]:= FileFormat1[F_String] := Module[{a, b}, {a, b} = {StringTake[F, 1],
```

```
MemberQ[ToUpperCase[Adrive[]], ToUpperCase[StringTake[F, 1]]];
```

```
If[StringLength[F] == 3 && MemberQ[{"/", ":\\"}, StringTake[F, {2, 3}] ] &&  
LetterQ[a] && b, "Directory", If[b == False, "$Failed", FileFormat[F]]]
```

```
In[676]:= Map[FileFormat1, {"C:\\Book\\Addition.doc", "C:\\", "D:/Math/Art", "H:/"}]
```

```
Out[676]= {"DOC", "Directory", "Table", "$Failed"}
```

```
In[677]:= Map[FileFormat, {"C:\\Book\\Addition.doc", "C:\\", "D:/Math_myLib/Art"}]
```

```
General::cdir: Cannot set current directory to CMap. >>
```

```
General::cdir: Cannot set current directory to System Volume Information. >>
```

```
Out[677]= $Aborted
```

Вызов процедуры **FileFormat1** на главных каталогах несуществующих или неактивных устройств внешней памяти возвращает значение "\$Failed". Кроме перечисленных для работы с файлами пакет **Mathematica** располагает целым рядом достаточно полезных функций, здесь не рассматриваемых, с которыми заинтересованный читатель сможет ознакомиться в справочной системе либо в документации по пакету.

Выше были кратко рассмотрены функции низшего уровня доступа к файлам данных: **OpenRead**, **OpenWrite**, **Read**, **Write**, **Skip** и **Streams**; между тем, не менее важными для доступа к файлам представляются также функции **Get**, **Put**, **Export**, **Import**, **ReadList** и **BinaryReadList**, которые поддерживают операции чтения и записи данных требуемого формата. Вкратце представим здесь указанные функции доступа к файлам данных.

Export["F.ex", D] – экспортирует данные *D* в файл "F. ex" в формате, который определяется расширением «.ex» файла данных;

Export[F, D, "ft"] – экспортирует данные *D* в файл *F* в заданном формате *ft*;

Export[F, D, E] – экспортирует данные *D* в файл *F*, трактуя *D* как *E*.

К сожалению, в имеющемся у нас релизе 8.0.1.0 пакета *Mathematica* данные функции доступа в полной мере не поддерживаются, о чем свидетельствует простой пример:

```
In[93]:= Export["D:/Math_myLib/Example.txt", "Academy of Noosphere, October 2011"]
```

```
General::unavail: Export is not available in this version of Mathematica.
```

```
Out[93]= Export["D:/Math_myLib/Example.txt", "Academy of Noosphere, October 2011"]
```

```
In[94]:= Export["D:\\Aladjev.jpg", <Портрет>]
```

```
General::unavail: Export is not available in this version of Mathematica.
```

```
Out[94]= 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. довольно существенные ограничения, например:

```
In[160]:= Import["D:\\UserLib6789\\CopyRight_Info\\Miscellaneous\\Aladjev1.jpg"]
```

```
Out[160]= <Портрет>
```

```
In[161]:= Import["http://www.aladjev.narod.ru/"]
```

```
General::unavail: Export is not available in this version of Mathematica.
```

```
In[162]:= Import["D:/Math_myLib/ArtKr.txt"]
```

```
Out[162]= "ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]]"
```

```
In[163]:= ArtKr[42.68, 47.63]
```

```
Out[163]= 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 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 *F*;

Put[W1, W2, ..., Wn, F] – записывает последовательность выражений W_j ($j=1..n$) 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[150]:= F = "D:\\Math_myLib\\IAN"; {Put[F], StreamsU[], FileByteCount[F]}
Out[150]= {Null, {}, 0}
In[151]:= DeleteFile[F]
In[152]:= {OpenWrite[F], Close[F], StreamsU[], FileByteCount[F]}
Out[152]= {OutputStream["D:\\Math_myLib\\IAN", 43], "D:\\Math_myLib\\IAN", {}, 0}
In[170]:= R[x_] := Sin[x]*Cos[x]; Put[R, "D:/Math_myLib/R"]; StreamsU[]
Out[170]= {}
In[171]:= Get["D:/Math_myLib/R"]; {StreamsU[], Definition[R]}
Out[171]= {{}, Definition[R]}
In[172]:= R[19.42]
Out[172]= 0.454502
```

Из представленного фрагмента, в частности, следует, что вызов как функции **Put**, так и функции **Get** по завершению операции записи/чтения сразу *закрывает* файл, что не позволяет производить дозапись в файл, а лишь в режиме обновления. Итак, за один вызов функции **Put**[F] можно записать несколько выражений в файл F, тогда как один вызов функции **Get**[F] активизирует в текущем сеансе все выражения, находящиеся в F.

Достаточно полезными оказываются функции доступа **ReadList** и **BinaryReadList**, чьи краткие характеристики с примерами применения представлены ниже. Так, **ReadList** имеет следующие четыре формата кодирования, обеспечивающие чтение данных из заданного файла в формате списка либо вложенного списка, а именно:

ReadList[F] – читает все оставшиеся выражения файла F, возвращая их в формате списка;

ReadList[F, T] – читает выражения типа T из файла F, возвращая их в формате списка;

ReadList[F, {T1, T2, ..., Tn}] – читает выражения типов T_j ($j=1..n$) из файла F, возвращая их в формате вложенного списка;

ReadList[F, {T1, T2, ..., Tn}, m] – читает только первые m групп выражений типов T_j ($j=1..n$) из файла F, возвращая их в формате вложенного списка.

Следующий фрагмент иллюстрирует использование функции доступа **ReadList**:

```
In[177]:= ReadList["D:\\Math_myLib\\exp.txt", {Word, Number}, 7]
Out[177]= {"rans", 2011}, {"ian", 420}, {"tru", 72}, {"avz", 68}, {"agn", 63}, {"art", 21}, {"kr", 14}
In[178]:= ReadList["D:\\Math_myLib\\exp.txt", {Word, Number}]
Out[178]= {"rans", 2011}, {"ian", 420}, {"tru", 72}, {"avz", 68}, {"agn", 63}, {"art", 21}, {"kr", 14}
In[179]:= ReadList["D:\\Math_myLib\\exp.txt", String]
Out[179]= {"rans 2011 ian 420 10 2 rags 72 avz 68 agn 63 art 21 kr 14"}
In[180]:= StreamsU[]
Out[180]= {}
```

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[185]:= BinaryReadList["D:\\Math_myLib\\exp.txt"]
Out[185]= {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[186]:= StreamsU[]
Out[186]= {}
```

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*, чья максимальная длина определяется 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}

S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\exp.txt", "Integer16"];
                                If[h === EndOfFile, S, S = Append[S, h]; Goto[A]]
In[7]:= For[k = 42, k <= 65, k++, BinaryWrite["D:\\Math_myLib\\w.txt", k]];
```

```

{StreamsU[], CloseAll[]}
Out[7]= {{OutputStream["D:\\Math_myLib\\w.txt", 46]}, {"D:\\Math_myLib\\w.txt"}}
In[8]:= S = {}; Label[A]; h = BinaryRead["D:\\Math_myLib\\w.txt"]; If[h === EndOfFile,
S, S = Append[S, h]; Goto[A]]
Out[8]= {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[9]:= {StreamsU[], CloseAll[]}
Out[9]= {{InputStream["D:\\Math_myLib\\w.txt", 47]}, {"D:\\Math_myLib\\w.txt"}}
In[10]:= BinaryWrite["D:\\Math_myLib\\w.txt", Sin[x]*Cos[x], "Character8"];
{StreamsU[], CloseAll[]}
Out[10]= {{OutputStream["D:\\Math_myLib\\w.txt", 44]}, {"D:\\Math_myLib\\w.txt"}}
In[11]:= Read["D:\\Math_myLib\\w.txt"]
Out[11]= Cos[x] Sin[x]
In[12]:= BinaryWrite["D:\\Math_myLib\\Z.txt", "426847636743624889148922"]
Out[12]= "D:\\Math_myLib\\Z.txt"
In[13]:= {StreamsU[], CloseAll[]}
Out[13]= {{OutputStream["D:\\Math_myLib\\Z.txt", 53]}, {"D:\\Math_myLib\\Z.txt"}}
In[14]:= Read["D:\\Math_myLib\\Z.txt", Record]
Out[14]= "426847636743624889148922"
In[15]:= {StreamsU[], CloseAll[]}
Out[15]= {{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 *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[1]:= FindFile["init.m"]
Out[1]= "C:\Documents and Settings\Aladjev\Application Data\Mathematica\Kernel\F"
In[2]:= FileExistsQ["D:\\Math_myLib\\exp.txt"]
Out[2]= True
In[3]:= FileNameSplit["D:\\Math_myLib\\AVZ_Package.nb"]
Out[3]= {"D:", "Math_myLib", "AVZ_Package.nb"}
In[4]:= FileNameJoin[%]
Out[4]= "D:\\Math_myLib\\AVZ_Package.nb"
In[5]:= FileBaseName["D:\\Math_myLib\\AVZ_Package.nb"]
Out[5]= "AVZ_Package"
In[6]:= ExpandFileName["AVZ_Package.nb"]
Out[6]= "C:\\Documents and Settings\\Aladjev\\My Documents\\AVZ_Package.nb"
In[7]:= FileExistsQ[%]
Out[7]= False
In[8]:= FileNameDepth["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb"]
Out[8]= 5
In[9]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb"]
Out[9]= "AVZ_Package.nb"
In[10]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", 3]
Out[10]= "D:\\Book\\Aladjev"
In[11]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", -2]
Out[11]= "Math_myLib\\AVZ_Package.nb"
In[12]:= FileNameTake["D:\\Book\\Aladjev\\Math_myLib\\AVZ_Package.nb", {2, 4}]
Out[12]= "Book\\Aladjev\\Math_myLib"
```

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 этих 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[1]:= DirectoryQ["C:/Documents and Settings/Aladjev/My Documents/My Videos/"]
Out[1]= False
In[2]:= DirQ[d_String] := DirectoryQ[If[StringTake[d, -1] == "/", StringTake[d, {1, -2}], d]]
In[3]:= DirQ["C:/Documents and Settings/Aladjev/My Documents/My Videos/"]
```

```
Out[3]= True
In[4]:= DirQ["C:/Documents and Settings/Aladjev\\My Documents/My Videos\\"]
Out[4]= 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:

```
In[5]:= Map[DirectoryName, {"D:/Math_myLib/AVZ_Package.nb", "D:\\Math_myLib"}]
Out[5]= {"D:\\Math_myLib\\", "D:\\"}
In[6]:= DirName[F_String] := If[DirQ[F], None, If[! FileExistsQ[F], $Failed,
    Quiet[Check[FileNameJoin[FileNameSplit[F][[1; -2]]], None]]]]
In[6]:= Map[DirName, {"D:/Math_myLib/AVZ_Package.nb", "D:\\Math_myLib", "H:/"}]
Out[6]= {"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[525]:= CDir[d_String] := Module[{a}, 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[526]:= CreateDirectory["H:\\Temp\\GSV"]
CreateDirectory::nffil: File not found during CreateDirectory[H:\\Temp\\GSV]. >>
Out[526]= $Failed
In[527]:= CDir["H:\\Temp\\AVZ\\RANS\\IAN"]
Out[527]= "D:\\Temp\\AVZ\\RANS\\IAN"
In[528]:= CDir["G:/Temp\\AVZ\\Tallinn\\IAN\\Groldno"]
Out[528]= "D:\\Temp\\AVZ\\Tallinn\\IAN\\Groldno"
```

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:/Math_myLib", "C:/Temp/Temp"]
CopyDirectory::file: Cannot overwrite existing file C:/Temp/Temp. >>
Out[539]= $Failed
In[540]:= CopyDir[d1_String, d2_String] := CopyDirectory[d1, If[DirQ[d2], d2 <> "\\\"
<> FileNameSplit[d2][[-1]], d2]]
In[541]:= CopyDir["D:/Math_myLib", "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[68]:= Attrib[F_String, A_] := Module[{a, b, c, d, h}, If[DirectoryQ[F] | |
FileExistsQ[F], {a, b, d} = {"$Art22_Kr14$", "attrib ", " > "}; If[! ListQ[A], Null,
If[A == {}, Run["attrib -A -H -S -R " <> F], Run["attrib -A -H -S -R " <> F];
Run["attrib " <> StringReplace[StringTake[ToString[Map["+" <> # &, A]], {2, -2}],
", " -> " "] <> " " <> F]], Return[F <> " is not a directory or a datafile"]];
c = Run[b <> F <> d <> a]; If[c != 0, Return[$Failed], 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, h]]
In[69]:= Map[Attrib, {"D:/Math_myLib/ArtKr", "C:/Windows", "C:/Program Files", "C:/"}]
Out[69]= {"A", "S", "H", "R"}, {}, {"Parameter", "form"}, "Drive C:/"}
In[70]:= Map[Attrib, {"C:/System Volume Information", "C:/Boot.ini", "C:/ntldr"}]
Out[70]= {"Pa"}, {"S", "H"}, {"A", "S", "H", "R"}
In[71]:= Map[Attrib, {"D:/", "C:\\", "H:/", "G:\\"}]
Out[71]= {"Drive D:/", "Drive C:\\", "H:/ is not a directory or a datafile", "G:\\ is not a
directory or a datafile"}
In[72]:= Attrib["D:/Math_myLib\\R", Null]
Out[72]= {"A", "H", "R"}
In[73]:= Attrib["D:/Math_myLib\\R", {"S", "H"}]
Out[73]= {"S", "H"}
In[74]:= Attrib["D:/Math_myLib\\R", {"A", "H", "R", "S"}]
Out[74]= {"A", "S", "H", "R"}
```


Успешный вызов **Attrib**[*F*,*Null*] процедуры возвращает список атрибутов заданного *F*-файла либо 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*". Если вызов процедуры **Attrib**[*F*, *A*] содержит два аргумента, из которых *второй* определяет список, то в случае пустого списка для *F* отменяются все указанные атрибуты, иначе *F* будут назначены определяемые *списком* атрибуты. Данная процедура представляется достаточно полезной при работе с файловой системой **ПК**. Более того, использование процедурой функции **Run** иллюстрирует один из очень полезных способов обеспечения пакетом интерфейса с базовой операционной платформой. Представим в этой связи данную функцию, имеющую следующий формат кодирования:

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 затененных областях представлено содержимое соответствующих текстовых файлов.

```
In[676]:= Run["Dir ", "C:\\", " > ", "C:\\Dir"]
```

```
Out[676]= 0
```

```
Volume in drive C is Aladjev_VZ
```

```
Volume Serial Number is CCA9-402C
```

```
Directory of C:\
```

```
01/05/2011  02:38 PM  <DIR>      ARCHIVE
02/12/2011  09:10 PM  <DIR>      Book
01/30/2011  09:09 PM  <DIR>      Documents and Settings
01/30/2011  11:14 AM  <DIR>      Program Files
02/12/2011  04:04 PM  <DIR>      TEMP
02/09/2011  12:44 PM  <DIR>      WINDOWS
               0 File(s)        0 bytes
               6 Dir(s)  99,884,134,400 bytes free
```

```
In[677]:= Run["Attrib ", "D:/Math_myLib\\ArtKr", " > ", "C:\\Attrib"]
```

```
Out[677]= 0
```

```
A  SHR  D:\Math_myLib\ArtKr
```

Используя данную функцию, возможно выполнять в среде пакета **Mathematica** целый ряд весьма полезных {*com* | *exe*}-программ самого различного назначения, тем самым достаточно существенно расширяя функциональные возможности самого пакета.

В качестве довольно полезного примера можно привести процедуру **SearchFile**[*F*], чей вызов возвращает список путей к файлу *F*, найденному в рамках файловой системы; в случае отсутствия искомого файла вызов процедуры **SearchFile**[*F*] возвращает пустой список. Процедура существенно использует рассмотренную выше **Run**-функцию.

```

In[243]:= SearchFile[F_String] := Module[{a, b, f, dir, k},
    {a, b, f} = {Map[ToUpperCase[#] <> ":\" &, ADrive[]], {}, "$$$$Dir$$$txt"};
    dir[y_String] := Module[{a, b, c}, Run["Dir " <> "/A/B/S " <> y <> " " <> f];
    c = {}; Label[b]; a = Read[f, String]; If [a === EndOfFile, Close[f];
    DeleteFile[f]; Return[c], If[StringEnd[a, F], c = Append[c, If[FileExistsQ[a],
    a, Null]]; Goto[b], Goto[b]]];
    For[k = 1, k <= Length[a], k++, b = Append[b, dir[a[[k]]]]; Flatten[b]]
In[244]:= SearchFile["AVZ_Package.nb"]
Out[244]= {"C:\\Book\\AVZ_Package.nb", "D:\\Book\\AVZ_Package.nb",
    "D:\\Math_myLib\\AVZ_Package.nb", "G:\\Book\\AVZ_Package.nb"}
In[245]:= SearchFile["init.m"]
Out[245]= {"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[246]:= Length[%]
Out[246]= 91
In[247]:= SearchFile["MapleMathem.doc"]
Out[248]= {"C:\\Book\\MapleMathem.doc", "D:\\Book\\MapleMathem.doc",
    "G:\\Book\\MapleMathem.doc"}

```

В целом ряде случаев возникает необходимость загрузки в текущий сеанс файлов *{m, nb, mx}*-типов, расположенных в одном из каталогов файловой системы компьютера; при этом, располагая полным именем файла, мы не располагаем информацией о его местоположении в файловой системе компьютера. Данную задачу решает процедура **LoadFile**, чей исходный текст и примеры применения представляет фрагмент.

```

In[1245]:= LoadFile[F_String] := Module[{a, b}, If[! MemberQ[{"nb", "m", "mx"},
    FileExtension[F]], Return["File <" <> F <> "> has an inadmissible type"],
    {a, b} = {SearchFile[F], StreamsU[]];
    If[a == {}, Return["File <" <> F <> "> has not been found"], Null];
    $Load$Files$ = Select[a, MemberQ[{"nb", "mx"}, FileExtension[#]] | |
    Read[#, String] === "(* ::Package:: *)" &];
    If[$Load$Files$ != {}, Map[Close, Select[StreamsU[], ! MemberQ[b, #] &]];
    Get[$Load$Files$[[1]]], Return["File <" <> F <> "> has not been found"];
    "File <" <> $Load$Files$[[1]] <> "> has been loaded;\n$Load$Files$
    defines the list with full paths to the found files."]
In[1246]:= LoadFile["AVZ_Package.m"]
Out[1246]= "File <D:\\Math_myLib\\AVZ_Package.m> has been loaded;
    $Load$Files$ defines the list with full paths to the found files."
In[1247]:= $Load$Files$
Out[1247]= {"C:\\Book\\AVZ_Package.m", "D:\\Math_myLib\\AVZ_Package.m"}

```

Вызов **LoadFile[F]** процедуры загружает в текущий сеанс файл, заданный его полным именем *F* и имеющий расширение $\{m, nb, mx\}$. При этом, в случае нахождения списка файлов с идентичным именем *F* производится загрузка *первого* из списка с возвратом соответствующего сообщения. Через *глобальную* переменную **\$Load\$Files\$** процедура возвращает список всех файлов *F*, найденных в процессе поиска.

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 ЭВМ, что и подтверждает их применение для решения задач подобного типа.

```
In[360]:= SearchDir[D_String] := Module[{a, b, c, d, f, h, dir, k},
  {a, b, f} = {Map[ToUpperCase[#] <> ":\\" &, Adrive[]], {}, "$$$$14Dir22$$$$.txt"};
  dir[y_String] := Module[{a, b, c, Q}, Run["Dir " <> "/B/S " <> y <> " > " <> f];
  Q[x_String] := 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[361]:= SearchDir["book"]
Out[361]= {"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[362]:= SearchDir["Old Grodno"]
Out[362]= {"C:\\ARCHIVE", "C:\\ARCHIVE\\OLD GRODNO", "G:\\ARCHIVE",
"G:\\ARCHIVE\\OLD GRODNO", "G:\\RANS_IAN_2011",
"G:\\RANS_IAN_2011\\OLD GRODNO"}

In[481]:= SearchDir["Photo"]
Out[481]= {"C:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS",
"C:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS\\PHOTO",
"G:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS",
"G:\\ARCHIVE\\MISCELLANY\\A_DOCUMENTS\\PHOTO"}
```

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 текст процедуры и ее применение.

```
In[226]:= 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 = "Art22####Kr14####.txt"; c = Run[b <> " > ", f];
  If[c != 0, Return[$Failed], a = OpenRead[f];
    h[x_String] := 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[227]:= ActiveProcess[]
Out[227]= {"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",
"cwmaple.exe", "explorer.exe", "FreeCommander.exe", "javaw.exe", "jqs.exe", "lsass.exe",
"LvAgent.exe", "maplew8.exe", "Mathematica.exe", "MathKernel.exe", "mserver.exe",
"services.exe", "Skype.exe", "skypePM.exe", "smss.exe", "soundman.exe", "spoolsv.exe",
"svchost.exe", "tlist.exe", "vsmon.exe", "winlogon.exe", "WINWORD.EXE", "zlclient.exe"}
```

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:

```
In[645]:= {$System, $SystemID, $OperatingSystem}
Out[645]= {"Microsoft Windows (32-bit)", "Windows", "Windows"}

In[646]:= Ver[] := Block[{a, b}, a = "Art####Kr####.txt"; Run["Ver > " <> a];
  b = Read[a, String]; Close[a]; DeleteFile[a]; b]

In[647]:= Ver[]
Out[647]= "Microsoft Windows XP [Version 5.1.2600]"
```

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 среде *Maple* следует существенно ограничить.

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 отличие от *Maple* в *Mathematica* в общем случае различаются в качестве разделителей прямой "/" и обратный "\" слэши, а именно: если обратный слэш является *стандартным* разделителем элементов *файловой* системы, то *прямой* слэш также вполне может выполнять эту функцию, исключая случай, когда он кодируется в *конце* цепочки каталогов. Для устранения данного неудобства нами создана простая функция *DirQ*, рассмотренная выше. Простой фрагмент иллюстрирует сказанное:

```
In[1069]:= $PathnameSeparator
Out[1069]= "\"
In[1070]:= DirectoryQ["C:/Program Files/Wolfram Research/Mathematica/8.0"]
Out[1070]= True
In[1071]:= DirectoryQ["C:\\Program Files\\Wolfram Research\\Mathematica/8.0"]
Out[1071]= True
In[1072]:= DirectoryQ["C:\\Program Files\\Maple 11\\"]
Out[1072]= True
In[1073]:= DirectoryQ["C:\\Program Files\\Maple 11/"]
Out[1073]= False
In[1074]:= Map[DirQ, {"C:\\Program Files/Maple 11/", "C:/Program Files/Maple 11\\"}]
Out[1074]= {True, True}
```

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.

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 уже были представлены выше и, прежде всего, это касалось средств доступа, однако можно привести и несколько более наглядные примеры. В частности, 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, чья разработка потребовала нестандартного подхода. Тогда как разработка подобного средства для Mathematica оказалась намного более простой, не потребовав какого-либо нестандартного подхода, что прекрасно иллюстрирует достаточно простая процедура NbName.

```
In[942]:= 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[943]:= NbName[]
Out[943]= {"Ver.nb", "Files$Directories.nb", "AVZ_Package.nb", "SearchFile.nb"}
In[1487]:= NbName[]
Out[1487]= {"SearchFile.nb", "LoadFile.nb", "ActiveProcess.nb", "Integral.nb", "Ver.nb"}
In[1493]:= AcNb[] := StringSplit[NotebookFileName[], {"\\", "/" }][[-1]]
In[1494]:= AcNb[]
Out[1494]= "Files$Directories.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 всех средств, находящихся в файле F nb-формата с пакетом, снабженных краткими справками (usages) по их использованию в среде пакета Mathematica.

```
In[244]:= NamesNbPackage[F_String] := Module[{Res = {}, Tr}, Tr[x_String] :=
    Module[{a = OpenRead[x], c, d, h, s = " RowBox[{", g = "::", p = "usage", v = "=", Label[c];
        d = Read[a, String]; If[d === EndOfFile, Close[a]; Return[Res],
        If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False},
        If[StringTake[d, {1, 10}] == s, h = Flatten[StringPosition[d, {s, g}]]];
        Res = Append[Res, StringTake[d, {h[[2]] + 2, h[[3]] - 5}]]; Goto[c]],
        Goto[c]]]; Tr[F]; Closes[F]; Map[ToExpression, Sort[Res]]]
In[245]:= NamesNbPackage["D:\\Math_myLib\\AVZ_Package.nb"]
Out[245]= {AcNb, ActiveProcess, Adrive, Aobj, Attrb, AutoLoadPack, BlockQ, CALL,
CDir, CloseAll, Closes, CopyDir, Df, DirName, DirQ, ExtrPackName, FileFormat1,
FileOpenQ, FunCompose, FunctionQ, GV, Int, ListListQ, LoadFile, Locals, Map1, Map2,
Map3, Mapp, MixCaseQ, NamesNbPackage, NamesProc, NbName, Need, Nobj, Nproc,
PackNames, ProcQ, SearchDir, SearchFile, SetDir, SortNL, Spos, StreamsU, StringEnd,
SubDelStr, Subs, UpdateContextPaths, UpdatePackages, UpdatePath, UprocQ, Uprocs, Ver}
```

8.5. Сравнение средств доступа к файлам Maple и Mathematica

Исходя из нашего опыта работы с пакетами *Maple* и *Mathematica* не только для ряда задач, имеющих дело с обработкой файлов данных различного типа, но и в процессе создания собственных средств обработки файловой системы компьютера в среде этих пакетов, которые по ряду показателей не только улучшают стандартные, но и, порой, существенно их расширяют, можно вкратце охарактеризовать стандартные средства доступа обоих пакетов на предмет их предпочтительности в решении таких задач.

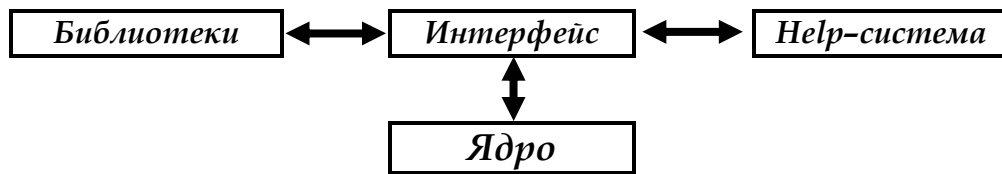
При работе с *внутренними файлами* пакетов ситуация представляется нам следующим образом, а именно. Если объекты пакета сохраняются во входном формате, т.е. имеют текстовый формат, то средства обоих пакетов можно с определенной достоверностью рассматривать эквивалентными, ибо оба пакета имеют достаточно развитые средства для работы с файлами такого формата. Для обоих пакетов пользователь может и сам относительно несложно создавать собственные средства обработки таких внутренних файлов, что и иллюстрируется целым рядом примеров средств, приведенных в этой книге. В частности, для пакета *Maple* немало средств такого типа находится в нашей Библиотеке [45], тогда как для *Mathematica* представлено здесь и в нашем пакете [90].

Тогда как совершенно иная ситуация имеет место для *внутренних* файлов *внутренних* форматов пакетов. Если в *Maple* *внутренний* формат (*m*, *mws*, *lib*)-*файлы*) структурно вполне просматриваем и пользователь даже без X-документации относительно легко может его обработать программно, то в *Mathematica* *внутренний* формат (*mx*, *nb*, *dll*)-*файлы*) для пользователя без специальной документации, практически, недоступен. Но так как обычный пользователь не работает с внутренними форматами на уровне их *структурной* организации, то особых преимуществ *Maple* здесь не имеет, тогда как средства для работы с *внешними* файлами и файловой системой ПК в целом наряду с большим набором *глобальных* переменных, ориентированных в том числе и на работу с файлами, более предпочтительны, на наш взгляд, именно у пакета *Mathematica*.

Резюмируя сказанное, можно констатировать, пакет *Maple* более предпочтителен для работы с *внутренними* файлами *внутреннего* формата (*m*-*файлы* и *lib*-*файлы*), тогда как пакету *Mathematica* следует отдавать предпочтение при работе с *внешними* файлами пакета и файловой системой компьютера в целом. Между тем, оба пакета не лишены целого ряда существенных недостатков относительно средств доступа к файлам, что вполне можно объяснить основными задачами пакетов. Однако, если *Mathematica* и предоставляет довольно развитый *базовый* набор средств данного типа, позволяющий пользователю с той либо иной эффективностью создавать средства доступа для своих задач, то в случае с *Maple* такая задача представляется нам более сложной. Именно по этой причине нами было создано немало средств доступа к файлам данных в *Maple* и именно в процессе такой работы были выявлены недостатки базовых средств доступа пакета *Maple* в целом. Тогда как подобная работа для *Mathematica* оказывается более простой, позволяя создавать средства доступа на основе базовых, для которых средств доступа пакета *Maple* недостаточно. Естественно, можно создавать такие средства на основе внешних по отношению к пакету средств, однако тогда уж лучше применять в целом и другие программные средства для обработки файлов данных.

Глава 9. Организация программного обеспечения пользователя

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 *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; при этом, все библиотеки пишутся, как правило, на встроенном *Maple*-языке.

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.

9.1. Организация программного обеспечения в среде Maple

Пакет *Maple* располагает целым рядом средств для создания довольно эффективных механизмов работы с библиотеками пользователя, которые структурно аналогичны главной библиотеке пакета. Эти библиотеки позволяют использовать содержащиеся в них средства на уровне доступа, аналогичного встроенным средствам пакета. Итак, в данном разделе мы представим *три* достаточно эффективных уровня организации пользовательских библиотек *процедур*, *модулей* и *функций*. При этом, наши средства, представленные в [32-35,37,38,42,45], позволяют существенно упрощать и расширять набор функций по работе с библиотеками пользователя. Как показывает наш опыт и опыт наших коллег, эти средства расширяют возможности пользователя по созданию и организации библиотек своего программного обеспечения в среде пакета *Maple*.

Перед дальнейшим изложением сделаем следующее существенное замечание. Работа с библиотеками любой организации – это работа, прежде всего, с файлами данных различного типа. В виду этого необходимо быть знакомыми со средствами доступа к файловой системе компьютера и с основными типами файлов, поддерживаемыми в среде *Maple*. Будучи *встроенным* языком программирования пакета *Maple*, в первую очередь, ориентированного, на *алгебраические* вычисления и обработку, *Maple*-язык располагает относительно ограниченными возможностями для работы с данными, находящимися во внешней памяти компьютера. И в данном отношении *Maple*-язык существенно уступает таким традиционным языкам программирования как *Fortran*, *ADA*, *C*, *Cobol*, *PL/1*, *Pascal*, *Basic* и др. Вместе с тем, ориентируясь, в первую очередь, на решение задач математического характера, *Maple*-язык предоставляет некоторый набор средств для доступа к файлам данных, который вполне может удовлетворить довольно широкий круг пользователей физико-математических приложений. Так, в наших книгах [12,34-38,42-44] средства *Maple* для доступа к файлам различных типов рассмотрены достаточно детально, по полноте изложения перекрывая поставляемую с пакетом документацию наряду с известной нам отечественной литературой [46-60]. Для расширения пакетных средств доступа к файлам данных нами был создан целый ряд средств, с которыми можно ознакомиться в вышеупомянутых наших книгах и в Библиотеке [45], ориентированной на пакет *Maple* релизов 6 – 11. С этими средствами можно ознакомиться и по демоверсии нашей библиотеки [45]. Между тем, начиная с релиза 9, *Maple* включает пакетный модуль *FileTools*, содержащий набор из не менее 26 средств для работы с файлами двух основных типов, с которыми имеет дело пакет, а именно: *бинарными (BINARY)* и *текстовыми (TEXT)*. Наши средства, в массе своей, не пересекаются со средствами данного модуля, существенно расширяя возможности пакета по работе с файлами данных. За описанием наших средств доступа к файлам

данных рекомендуется обратиться к нашим книгам [42-44] либо бесплатно загрузить исходные тексты наших книг по *Maple*-тематике со следующих *web*-адресов:

<http://www.aladjev-maple.narod.ru>

<http://www.aladjev-maple-book.narod.ru>

<http://www.lib.grsu.by/library/data/resources/catalog/107235-172717.pdf>

Эти средства, прежде всего, относятся к базовым средствам системы ввода/вывода, но их использование позволяет не только эффективнее организовать работу с файловой системой компьютера, но и создавать специальные библиотеки своих процедур.

Отметим, что со всей определенностью можно констатировать, что пакетные модули *FileTools* и *LibraryTools* были вдохновлены рядом наших книг по *Maple*-тематике [29-33,39,42-46], с которыми разработчики пакета были ознакомлены. Однако, наш набор подобных процедур является существенно более представительным и они, при этом, сосредоточены на более широком практическом использовании при решении задач, имеющих дело с обработкой файлов данных различного назначения. Более того, нам не известно более обстоятельного рассмотрения системы доступа, обеспечиваемой пакетом *Maple*, чем в наших предыдущих книгах [11-14,28-38,42-44]. Именно в этом отношении они рекомендуются читателю, имеющему дело с подобными задачами.

Еще на одном важном моменте следует акцентировать внимание. Разработка *Maple*-приложений базируется на программной среде пакета. Здесь можно выделить ряд уровней разработки, которые представляются нам под следующим углом, а именно:

(1) *Maple*-документы, решающие отдельные прикладные задачи (типичным примером такого подхода может служить набор *Maple*-документов, решающих типичные инженерно-физические задачи методом конечных элементов [11,13,14,28,33,36-38]);

(2) отдельные процедуры, программные модули и их наборы, сохраненные в файлах и решающие конкретные прикладные задачи (типичными примерами могут служить средства, созданные многочисленными пользователями пакета, а также пакетные модули);

(3) библиотеки процедур и модулей, организованные аналогично главной библиотеке пакета и решающие достаточно широкий круг как задач из различных приложений, так и системных, расширяющих функциональную среду самого пакета (достаточно типичным примером может служить наша Библиотека для *Maple* релизов 6 - 11[45]).

Естественно, представленная классификация в значительной степени субъективна, но при отсутствии другой и она может служить в качестве некоторой отправной точки. Средства *первого* уровня представляют собой законченные *Maple*-документы, т.е. они имеют {*mws*, *mw*}-формат и решают конкретные прикладные задачи из различных приложений. Документы могут содержать описательную часть, постановку задачи, алгоритм, описанный входным *Maple*-языком, результаты выполнения документа и другую необходимую информацию. Как правило, сложная задача либо их комплекс реализуются в виде набора {*mws*, *mw*}-файлов, сохраняемых в отдельном каталоге и доступных для выполнения путем загрузки в среду *Maple* соответствующего релиза либо поочередно, либо в требуемом порядке. Для создания этих документов в целом ряде случаев вполне можно обходиться только входным *Maple*-языком, а не полным набором средств, предоставляемых его расширением – *встроенным* языком. В целом,

данный уровень допускает и диспетчирование документов, например, следующим образом. У традиционного *Maple*-документа после его отладки удаляются все *Output*-параграфы и он сохраняется в текстовом файле в рамках своих *Input*-параграфов. В этом случае получаем *Maple*-документы, допускающие выполнять загрузку в нужном порядке в текущий сеанс пакета по *read*-предложению, созданные выше описанным образом *txt*-файлы, обеспечивая необходимое диспетчирование таких документов.

Второй уровень представляет отдельные процедуры или программные модули, либо их наборы, решающие конкретные как прикладные, так и системные задачи. Данные средства могут сохраняться в файлах как *входного*, так и *внутреннего Maple*-формата. Для них могут создаваться и библиотечные организации, например, архивного типа. В данном случае пользователю уже в значительной степени приходится использовать встроенный *Maple*-язык пакета. И от уровня владения этим языком в значительной мере зависит возможность пользователя создавать достаточно сложные приложения.

Наконец, *третий* уровень организации *пользовательского* программного обеспечения характеризуется использованием для разработки достаточно сложных приложений в полной мере средств встроенного *Maple*-языка, формированием библиотек, которые подобны главной библиотеке пакета, и пакетными модулями, ориентированными на относительно узкие области приложений, а также и на самое массовое использование при разработке приложений в среде пакета. Довольно типичным примером средств третьего уровня является и наша Библиотека [45], рассматриваемая в книгах [32-37].

Библиотека *UserLib6789* расширяет диапазон и эффективность использования пакета *Maple* релизов *6-11* на платформе *Windows* благодаря содержащимся в ней средствам в *трех* основных направлениях, а именно: (1) устранение ряда основных дефектов и недостатков, (2) расширение возможностей целого ряда стандартных средств пакета, и (3) пополнение пакета новыми средствами, довольно существенно расширяющими возможности его программной среды, включая также средства, улучшающие уровень совместимости релизов *6 - 11* пакета. Основное внимание уделено дополнительным средствам, созданным в процессе использования пакета *Maple* релизов *4 - 11*, которые по целому ряду параметров существенно расширяют возможности пакета в целом и облегчают работу с ним. Текущая версия *2.2215* Библиотеки содержит средства (*более 850 процедур и программных модулей*), ориентируемые на все основные виды обработки информации и вычисления. Многие средства Библиотеки подверглись оптимизации. Большой опыт использования данной Библиотеки наряду с опытом многочисленных пользователей подтвердили ее довольно высокие эксплуатационные характеристики при программировании разнообразных приложений в среде пакета *Maple*. Многие из средств Библиотеки неоднократно обсуждались на различных форумах, позволяя решать задачи либо трудно решаемые в среде пакета, либо и вовсе не решаемые.

Библиотека в совокупности с главной *Maple*-библиотекой обладает полнотой в том отношении, что любое ее средство использует или средства главной библиотеки и/или средства самой Библиотеки. В этом плане она полностью самодостаточна. Более того, используемые в процедурах Библиотеки полезные, эффективные (*а в целом ряде случаев и нестандартные*) приемы программирования позволяют и более глубоко, и за более короткий срок освоить программную среду пакета. Тогда как использование в

большинстве процедур обработки особых и ошибочных ситуаций дает возможность акцентировать уже на ранней стадии освоения программирования свое внимание на таких важных аспектах создания программных средств в *Maple*, как их мобильность, надежность и ошибкоустойчивость. Наконец, работая с Библиотекой, пользователь не только имеет прекрасную возможность освоить многие из ее средств для текущей и последующей работы с пакетом, но также проникается концепцией эффективной организации своих собственных *Maple*-библиотек, содержащих средства, которые обеспечивают его профессиональные интересы и потребности. Есть надежда, что и читатель книги найдет среди средств Библиотеки полезные для своего творчества. В целях большей информативности с краткой характеристикой нашей Библиотеки [45] читатель может ознакомиться в работах [12,28,29,32-44].

9.1.1. Классический способ создания Maple-библиотек

В настоящем подразделе вкратце остановимся на организации главной библиотеки пакета, которая в зависимости от релиза пакета может быть не единственной. Однако главная библиотека, содержащая все массовые средства, обеспечивающие важнейшие функции пакета, является единственной. Тогда как остальные включают средства по поддержке дополнительных возможностей пакета, например, обеспечения работы с системами измерений, решения задач из конкретных приложений и т.д. Такой обзор в последующем позволит более адекватно оценить возможности пакетов *Mathematica* и *Maple* относительно предоставляемых ими средств по эффективной организации в их среде пользовательского программного обеспечения.

Главная библиотека *Lib* пакета содержит наиболее часто используемые процедуры и модули (которые не включены в ядро пакета). Данная библиотека находится в каталоге *Lib* пакета и содержит набор файлов, определяемый релизом пакета. Например, для релизов не выше 8-го главная библиотека содержит три основных файла «*Maple.hdb*», «*Maple.ind*» и «*Maple.lib*», тогда как в релизах 6, 7 присутствует дополнительно файл «*Maple.rep*». Учитывая, что релизы 9 и 9.5 были в определенной мере переходными и в довольно краткой истории своего существования оказались в эксплуатации весьма ненадежными, имеет смысл говорить о качественных изменениях пакета, начиная с релиза 10. В *Maple 10* главная и другие библиотеки пакета организационно устроены несколько иначе, а именно: вместо трех файлов «*maple.hdb*», «*maple.ind*» и «*maple.lib*» (библиотека *mlib*-типа) они состоят из 2 файлов «*maple.hdb*», «*maple.mla*» (библиотека *mla*-типа), где первый (в общем случае необязательный) файл «*maple.hdb*» структурно остался без изменения, тогда как файл «*maple.mla*» представляет собой, по сути дела, слияние 2 файлов «*maple.ind*» и «*maple.lib*» прежней организации с соответствующей корректировкой входов в начальной индексной части файла «*maple.mla*». На данном аспекте (как не принципиальном) внимания не акцентируется, принимая во внимание то обстоятельство, что библиотека *mlib*-типа легко конвертируется в эквивалентную ей библиотеку *mla*-типа, и наоборот.

С учетом сказанного организация библиотеки пользователя, аналогичной пакетной библиотеке, схематично может быть представлена нижеследующим образом.

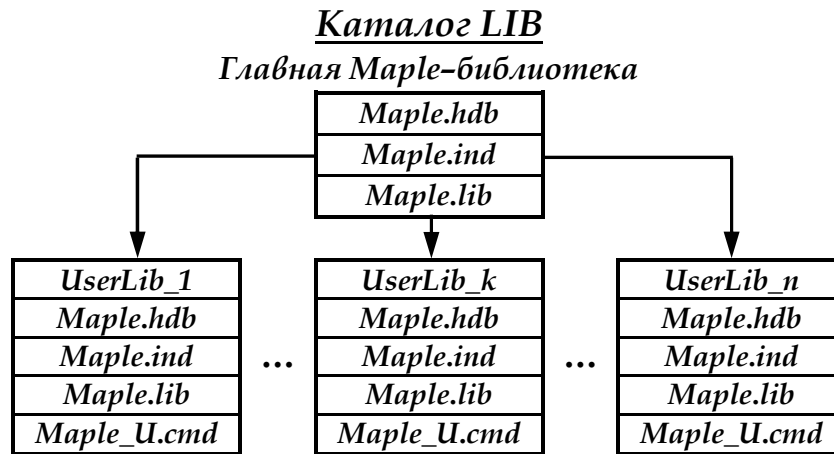


Рис. 2. Принципиальная организация главной Maple-библиотеки и пользовательских библиотек, организационно аналогичных главной библиотеке пакета.

На первом уровне находится главная библиотека *Maple*, располагаемая в каталоге *LIB* и содержащая 3 файла. Тогда как на втором уровне такой библиотечной организации находятся собственно пользовательские библиотеки, расположенные в подкаталогах *UserLib_k* ($k=1..n$) каталога *LIB*, содержащего главную *Maple*-библиотеку, стандартно поставляемую с пакетом. В данном случае файловая организация пользовательских библиотек принимает следующий простой вид, наследуя структурную организацию главной библиотеки (рис. 2). При этом, каждая библиотека пользователя располагается в отдельном подкаталоге каталога *LIB*. Первые три файла библиотеки пользователя полностью аналогичны одноименным файлам главной *Maple*-библиотеки, тогда как отдельный файл «*Maple_U.cmd*» содержит список имен процедур, расположенных в библиотеке и историю работы с библиотекой. При этом, в зависимости от текущего релиза в процессе работы с библиотекой пользователя в каталоге могут появиться и 3 дополнительных файла «*Maple.rep*», «*elpam.ind*» и «*elpam.lib*», чье описание может быть найдено в наших предыдущих книгах [12,28,32-44]. Ниже под термином «*Maple-библиотека*» будет пониматься библиотека, структурно и организационно подобная главной *Maple*-библиотеке пакета *Maple*.

Организация пользовательских библиотек, представленная выше, является довольно удобной и четко локализует их расположение, обеспечивая их достаточно удобную программную обработку. Наряду с этим, такая организация позволяет использовать для поддержки пользовательских библиотек средства пакета, точнее его встроенную функцию *march*. Процедуры, которые представлены в [45], обеспечивают создание и обновление библиотек пользователя согласно упомянутой файловой организации, а также их логическое соединение с главной *Maple*-библиотекой, обеспечивая доступ к средствам, находящимся в них, на уровне стандартных средств. Итак, использование библиотечной организации, аналогичной пакетной, существенно упрощает работу с пользовательскими библиотеками. Кратко представим этапы создания библиотеки пользователя, организационно аналогичной главной библиотеке пакета.

Этап 1. Прежде всего, предполагается, что у пользователя имеется набор готовых и отлаженных процедур и/или программных модулей, которым он желает наполнить

вновь создаваемую библиотеку, подобную *Maple*-библиотеке пакета. На первом этапе создается пустая библиотека, используя встроенную *iolib*-функцию (ранее она была на уровне утилиты) *march*, с форматом вызова которой можно познакомиться в справке по пакету либо в [28,29,32-44], где дополнительно можно получить информацию и по особенностям кодирования полного пути к файлу, что обеспечивает ряд интересных возможностей для работы с файловой системой компьютера.

Перед созданием библиотеки для нее по *march*-функции предварительно создается каталог (если он не был создан ранее), в котором она будет размещена. Создавать каталог можно либо средствами *DOS*, *Windows*, либо в среде самого пакета *Maple*, используя *iolib*-функцию *mkdir*, либо наши процедуры *MkDir*, *MkDir1*, позволяющие создавать вложенные цепочки каталогов любой глубины вложенности.

Создав по встроенной функции *mkdir* или по нашей процедуре *MkDir1* каталог под создаваемую библиотеку (например, подкаталог "C:/Program Files/Maple 11/UserLib9"), теперь мы можем в нем создать «пустую» библиотеку с именем *UserLib*. Делается это по вызову встроенной функции *march* следующего формата кодирования, а именно:

```
> march('create', "C:\\Program Files\\Maple 11\\UserLib9", <Размер>)
```

где аргумент <Размер> определяет размер создаваемой библиотеки. При этом, размер библиотеки определяется числом содержащихся в ней *m*-файлов с *Maple*-объектами (процедуры, модули, таблицы и т.д.); практически, имеется возможность сохранения в такой библиотеке числа *m*-файлов, примерно равного удвоенному числу, заданному аргументом <Размер>. Таким образом, по вызову *march* следующего формата:

```
> march('create', "C:\\Program Files\\Maple 11\\UserLib9", 420)
```

создается пустая библиотека *UserLib9*, предназначенная для сохранения порядка 800 различных *Maple*-объектов. В библиотеке создаются два макетных файла «*Maple.ind*» (индексный) и «*Maple.lib*» (с *m*-файлами *Maple*-объектов), впоследствии обновляемые в случае помещения в библиотеку новых *Maple*-объектов либо при ее реорганизации средствами той же *march*-функции. Как результат выполнения указанных процедур создается «пустая» библиотека, которая готова к наполнению средствами: модулями, процедурами и другими *Maple*-объектами, например, таблицами. Следующий этап сводится к созданию будущих элементов библиотеки, если они отсутствуют.

Этап 2. На данном этапе необходимо в текущем сеансе вычислить определения всех *Maple*-объектов, помещаемых в библиотеку, как иллюстрирует простой пример:

```
> Sr:= () -> `+`(args)/nargs; Ds:= () -> sqrt(sum((args[k] - Sr(args))^2, k = 1..nargs)/nargs);  
> 6*Sr(64, 59, 39, 10, 17, 44), 6*Ds(64, 59, 39, 10, 17, 44);
```

Теперь процедуры *Sr* и *Ds* готовы для включения в созданную библиотеку *UserLib9*.

Этап 3. Для определения пути к библиотеке, принимающей *Maple*-объекты, служит глобальная переменная *savelibname*, которая первоначально имеет неопределенное значение. Для указания пути к библиотеке этой переменной присваивается полный путь к ней. Можно ограничиться и просто именем библиотеки, если она находится в текущем каталоге, однако первый способ более универсален, например, для нашего случая определяем переменную *savelibname* следующим образом, а именно:

> **savelibname** := "C:\\Program Files\\Maple 11\\UserLib9":

Этап 4. На данном этапе производится непосредственное сохранение в библиотеке подготовленных выше процедур. Делается это вызовом процедуры *savelib*, имеющей формат вызова следующего общего вида, а именно:

> **savelib**(*N1*, *N2*, ...);

где *Nj* – имена сохраняемых в библиотеке *Maple*-объектов. При рассмотрении нашего примера вызов процедуры принимает следующий вид, а именно:

> **savelib**(*Sr*, *Ds*);

Вызов процедуры *savelib* ничего не возвращает, поэтому для проверки результата ее выполнения рекомендуется использовать еще один формат вызова *march*-функции:

> **march**('list', <Путь к библиотеке>)

по которому возвращается список всех сохраненных в библиотеке, указанной вторым аргументом, *Maple*-объектов, точнее соответствующих им *m*-файлов, например:

> **march**('list', "C:\\Program Files\\Maple 11\\UserLib9");

[[*"Ds.m"*, [2006, 10, 2, 10, 24, 18], 1090, 115], [*"Sr.m"*, [2006, 10, 2, 10, 24, 18], 1024, 66]]

Возвращаемый функцией *march* вложенный список содержит по одному подсписку для каждого содержащегося в библиотеке *UserLib9* *m*-файла (первый элемент в виде «*File_Name.m*»), второй содержит список с датой и временем создания файла, тогда как третий и четвертый указывает начальную позицию данного файла (а точнее его смещение) в библиотечном файле «*Maple.lib*», и его размер в байтах соответственно. Результат проверки показывает, что наши две процедуры были сохранены успешно.

Этап 5. На данном этапе производится логическое сцепление созданной библиотеки с главной *Maple*-библиотекой, что позволит впредь обращаться к находящимся в ней объектам подобно стандартным средствам пакета *Maple*. Для данных целей служит предопределенная переменная *libname*, определяющая последовательность путей к библиотекам пакета, в которых будут отыскиваться вызываемые средства, если они не были определены и вычислены непосредственно в текущем сеансе. В нашем случае исходное состояние *libname*-переменной имеет следующий вид, а именно:

> **libname**; ⇒ "C:\\Program Files\\Maple 11/lib"

которое определяет, что пакет располагает главной *Maple*-библиотекой «*C:/program files/maple 11/lib*», с которой предполагается логически соединить нашу библиотеку «*C:/program files/maple 11/lib/userlib9*» при условии, что она будет иметь наименьший приоритет. Приоритет определяется положением библиотеки в *libname*-переменной – чем ближе библиотека находится к началу цепочки, тем выше будет ее приоритет, определяющий порядок поиска библиотечных средств при вызове, а именно: поиск вызываемого средства начинается с библиотеки с максимальным приоритетом. Для подключения созданной нами библиотеки *Userlib9* нужно переопределить значение *libname*-переменной пакета, а именно:

> **libname**:= **libname**, "C:\\Program Files\\Maple 11\\UserLib9": **libname**;

"C:\\Program Files\\Maple 11/lib", "C:\\Program Files\\Maple 11\\Lib/UserLib9"

В результате переопределения *libname*-переменной наша библиотека получит самый маленький приоритет. Именно так следует поступать при логическом подключении новых библиотек к главной библиотеке пока не проведена их детальная апробация в совокупности с другими сцепленными библиотеками пакета. Между тем, при такой организации мы должны будем каждый раз после загрузки пакета перед обращением к средствам из библиотеки *UserLib9* выполнять указанное переопределение *libname*-переменной, что, естественно, неудобно.

Во избежание этого рекомендуется поступить следующим образом. *Maple* допускает ряд инициализационных файлов (*ini-файлов*), из которых *txt*-файл файл «*MapleF.ini*» создается уже при инсталляции пакета в его *Users*-подкаталоге, где *F* - номер релиза пакета. В этот же подкаталог рекомендуется любым доступным средством (например, по *Notepad*) записать файл «*Maple.ini*» с единственной строкой следующего вида:

libname := libname, "C:/Program Files/Maple 11/UserLib9":

или дополнить файл этой строкой, если он уже существовал. Такой подход обеспечит вам автоматическое сцепление вашей библиотеки с главной *Maple*-библиотекой уже после каждой загрузки пакета. В результате выполнения описанных этапов создается библиотека пользователя *UserLib9*, средства которой становятся доступными наравне со стандартными средствами при каждой загрузке *Maple*. При этом, подкаталог *Users* предполагается по умолчанию, однако установка для инициации *libname* может быть пересмотрена при инсталляции. В [42-44] представлена таблица, которая определяет наиболее целесообразное расположение файла «*Maple.ini*» в каталогах пакета: *Users*, *Bin.win* и *Lib* наряду с правилами обработки пакетом таких файлов в зависимости от релиза пакета. Данные *ini*-файлы могут содержать любую полезную информацию по инициализации как специального, так и общего характера [9], включая определения процедур, модулей либо их вызовов. В конкретном случае Библиотеки данный файл расположен в *Users*-подкаталоге и используется для организации логической связи с главной *Maple*-библиотекой пакета. На основе файла «*Maple.ini*» имеется хорошая возможность поддерживать достаточно эффективные и простые механизмы связей пользовательских библиотек. Рекомендуется располагать инициализационный файл «*Maple.ini*» именно в подкаталоге *Users* пакета *Maple*.

Этап 6. Обновление созданной библиотеки новыми средствами возможно выполнять согласно этапам 3–4, представленным выше. Однако для этих целей вполне подойдет и созданная нами [43,45] простая процедура *uplib*, которая обеспечивает расширение библиотеки, заданной полным путем, *Maple*-объектами, чьи имена представляются в виде одного имени, их списка или множества. Для обеспечения эффективной работы с библиотекой пользователя разработан набор процедур различного назначения [45], с которыми можно ознакомиться в книгах [9,28-30,32-38,42-44]. При этом, ряд важных процедур обработки библиотек, подобных главной библиотеке пакета, обеспечивает и его *iolib*-функция *march*. Тогда как наши средства обеспечивают не только базовые функции с библиотеками, но и такие процедуры как восстановление поврежденных библиотек, сбор статистики по использованию средств библиотек, поддержка работы с библиотеками, организационно отличными от *Maple*-библиотек, и ряд других.

Этап 7. Создав *Maple*-библиотеку выше описанным способом и имея средства для ее обновления, уже вполне возможно использовать ее средства наравне с пакетными для программирования своих приложений и дальнейшего развития данной и других ей подобных библиотек. Однако для придания вашей библиотеке статуса законченного программного продукта весьма желательно обеспечить ее собственной справочной базой, описывающей все содержащиеся в библиотеке средства. Вполне разумно взять за прообраз такой базы справочную базу самого пакета, которая представляется нам (за исключением ряда не очень существенных огрехов) вполне прилично организованной.

Прежде всего, нам потребуется создать саму справочную базу библиотеки (для нашего конкретного случая – библиотеки "*C:/Program Files/Maple 11\\UserLib9*"). Однако здесь ситуация несколько отлична от традиционной, а именно. Справочные страницы по средствам *Maple*-библиотеки находятся в справочной базе в виде файла «*Maple.hdb*», расположенного в том же каталоге, что и сама библиотека. Каждый такой файл базы данных содержит одну либо несколько страниц справки и служебную информацию, необходимую для обеспечения работы броузера как со справочной базой пакета, так и справочной базой пользователя. На наш взгляд, справочная система пакета *Maple* организована достаточно эффективно и удобна для практического использования в работе с пакетом и его приложениями в различных областях.

Справочная система пакета расположена в *Graphic User Interface (GUI, Iris)*, поэтому она непосредственно не может обрабатываться программными средствами пакета. Таким образом, обращение к справочной системе обеспечивается на основе функциональных запросов *INTERFACE_HELP(...)*-формата, с детальными описанием и возможностями которых можно ознакомиться в [38,42-44]. С целью упрощения работы со справочной системой пользовательской библиотеки нами была создана процедура *helpman*; при этом, процедура обрабатывает основные ошибочные и особые ситуации, иницилируя ошибки или выводя соответствующие информационные сообщения [45]. Между тем, в среде пакета имеются собственные средства поддержки справочной системы, но эти средства в ряде случаев имеют недостатки. Так, способ обновления справочной базы (дополнение/удаление страниц) через *GUI* недостаточно надежен и в некоторых случаях не дает результата, прежде всего при попытке обновления существующих страниц. В данном контексте процедура *helpman* автоматизирует функции поддержки ведения справочной базы библиотеки пользователя, аналогичной главной *Maple*-библиотеке пакета. Так как наша библиотека отражена в предопределенной *libname*-переменной пакета, то и созданная для нее справочная база логически сцепляется с аналогичной базой пакета, обеспечивая стандартную для *Maple* технологию работы со справочной информацией. Таким образом, процедура *helpman* представляет достаточно удобное средство для обновления справочной базы библиотек пользователя на основе заранее подготовленных *mws*-файлов со справкой, обеспечивая возможность создания одним вызовом несколько справочных разделов, однако просмотр и удаление производится по одному справочному разделу для каждого вызова процедуры; при этом, *helpman* лишена ряда недостатков, присущих стандартным средствам пакета. В ряде случаев она помогала решать задачу обновления справочных баз *Maple*-библиотек там, где стандартные средства *GUI* пакета оказывались бессильными [28,32-38,42-44]. Именно

данной процедуре, зарекомендовавшей свои эксплуатационные качества, мы отдаем предпочтение при работе со справочными базами своих *Maple*-библиотек.

Таким образом, нами представлена довольно простая технология создания и базовых функций ведения пользовательских библиотек, аналогичных главной библиотеке. В [28-30,32-38,42-44] можно найти целый ряд примеров создания и ведения библиотек пользователя на основе описанной технологии с обсуждением целого ряда довольно важных особенностей, которые встречаются на данном пути. Как уже отмечалось, для поддержки разнообразных процедур работы с *Maple*-библиотеками нами был создан целый ряд полезных средств, представленных в книгах [32-38,42-44] и в прилагаемой к ним библиотеке программных средств для пакета *Maple* релизов 6-11. Эти средства были созданы еще для *Maple 6*, тогда как некий их аналог в лице модуля *LibraryTools* появился только в 8-м релизе пакета. Первоначально это был набор из 5 процедур, в *Maple 9* – 7 процедур, в *Maple 10, 11* – 16 процедур. Однако, целый ряд важных средств для работы с библиотеками так и не был реализован. Тогда как наш набор средств по работе с библиотеками предоставляет, например, также средства для восстановления поврежденной библиотеки, средства автоматизации ведения библиотек, средства их оптимизации. В частности, процедура *User_pflm(F, U {, S {, R})* обеспечивает создание или обновление библиотек пользователя, аналогичных главной *Maple*-библиотеке, с их логической связью с главной библиотекой, обеспечивая доступ к содержащимся в них средствам на уровне стандартных библиотечных средств пакета. Более того, 3-й и 4-й необязательный аргумент процедуры позволяют устанавливать приоритет для создаваемой библиотеки: максимальный или минимальный (по умолчанию полагается минимальный приоритет). Успешный вызов *User_pflm* возвращает текущее состояние *libname*-переменной и выводит сообщение по созданной/обновленной библиотеке. После этого пользователь получает возможность работать со средствами библиотеки аналогично стандартным средствам пакета. Детальнее с использованием процедуры *User_pflm* и ее модификациями можно ознакомиться в [9,32-38,42-44]. Между тем, ее обобщение процедуру *User_pflmH* можно рассматривать в качестве наиболее общего и универсального инструмента создания пользовательских библиотек, аналогичных главной библиотеке пакета *Maple* и логически связанных с ней. В целом же процедура *User_pflmH* [45] выполняет следующие дополнительные функции, а именно:

- * регистрация в системном файле «*Win.ini*» текущего релиза пакета, если ранее этого не было сделано;
- * создание/обновление инициализационного файла «*Maple.ini*» с целью обеспечения логической связи создаваемой библиотеки пользователя с главной библиотекой (при этом, последняя операция обеспечивает возможность создания справочной базы библиотеки согласно пользовательскому предложению).

Процедура *User_pflmH* выводит соответствующие сообщения о выполняемой работе. Таким образом, при создании/обновлении библиотеки пользователь должен только определить ее имя (если библиотека располагается в каталоге *LIB* пакета) либо полный путь к ней, наряду с множеством или списком имен *Maple*-объектов, сохраняемых в библиотеке, и чьи определения были вычислены в текущем сеансе. Дополнительно, пользователь может определить размер вновь создаваемой библиотеки, справочную

базу данных и режим логической связи библиотеки с главной библиотекой пакета. В этом смысле процедура *User_pflMH* и другие наши средства работы с библиотеками в определенном отношении поддерживают развитые функциональные возможности автоматизации работы с библиотеками подобно случаю хорошо известной утилиты *sed* для операционной системы **UNIX (LINUX)**. Таким образом, пакет предоставляет средства создания библиотек пользователя, аналогичных главной **Maple**-библиотеке, позволяя унифицировать средства пользователя под технологию, поддерживаемую пакетом и существенно облегчает организацию его программных средств. Важность библиотечной организации пользователя с приоритетами выше или ниже главной библиотеки пакета обеспечивают решение 2-х таких практически важных задач, как:

1. удобство работы со средствами на уровне средств пакета;
2. возможность простой замены неэффективных либо некорректных средств пакета аналогичными пользовательскими средствами.

Между тем, пользователь располагает возможностью не только создавать собственные библиотеки, аналогичные главной **Maple**-библиотеке, либо средствами собственно самого пакета, либо вышеуказанными нашими средствами, но также возможностью обновлять и саму главную библиотеку пакета. Для обеспечения такой возможности требуется снять защиту с главной **Maple**-библиотеки, что выполняет наша процедура *AtrRW*, вызов *AtrRW(L)* которой для **Maple** версий 8-10 по принципу «переключателя» изменяет для библиотеки атрибут *READONLY* или *WRITABLE* (в **Maple**-концепции) на противоположный, полный путь к которой определяется аргументом *L*. Тогда как для **Maple** версий 6, 7 вызов процедуры *AtrRW(L)* по принципу «переключателя» изменяет атрибут *readonly* или *writable* (в **DOS**-концепции) на противоположный для библиотеки, полный путь к которой определяется аргументом *L*. Между тем, обновление главной библиотеки пакета **Maple** следует производить достаточно осмотрительно.

Для обеспечения обновления главной **Maple**-библиотеки таблицами, процедурами и программными модулями предназначены две довольно полезные процедуры *UpLib* и *MapleLib*, которые существенно используют процедуру *AtrRW* и позволяют также обновлять библиотеки пользователя, аналогичные главной библиотеке пакета. Так, в частности, именно процедура *UpLib* используется нами наиболее активно для задачи обновления **Maple**-библиотек пакета. Успешный вызов *MapleLib* обновляет главную **Maple**-библиотеку процедурами, таблицами и/или модулями, чьи имена определены ее первым фактическим аргументом (множество или список имен). При этом, если был закодирован второй необязательный аргумент, то он будет определять полный путь к библиотеке пользователя, аналогичной главной библиотеке. Процедура *MapleLib* выводит соответствующее сообщение о сделанной работе и обрабатывает основные ошибочные ситуации, связанные с отсутствием либо повреждением обрабатываемой библиотеки, или с отсутствием сохраняемых объектов. При возникновении данных ситуаций выводится необходимая диагностика. Процедура *MapleLib* обеспечивает вышеупомянутые функции для библиотек указанного типа для пакета релизов 6 - 11. Более того, библиотеки в **Maple** релизов 6 и 7 получают *readonly*-атрибут (в концепции **DOS**) после обновления, тогда как библиотеки в **Maple** версий 8-11 после обновления получают *READONLY*-атрибут (в концепции пакета **Maple**).

Однако, обновление главной библиотеки не рекомендуется выполнять пользователю недостаточной квалификации во избежание возможных ее повреждений. Между тем, квалифицированный пользователь в определенных случаях вполне может проводить данную процедуру, например, при замене стандартных средств, имеющих ошибки или недостаточно эффективных. В библиотеку можно помещать часто используемые средства пользователя. Так, мы в процессе апробации и эксплуатации пакета *Maple* версий 4 – 11 обновляли главную библиотеку пакета как из-за обнаруженных ошибок составляющих ее средств, так и при необходимости ее расширения эффективными средствами частого использования. Наряду с вышеуказанным стандартным подходом *Maple* располагает средствами по созданию библиотек пользователя с организаций, отличных от стандартной [42-44], ряд из них включен и в нашу Библиотеку [45].

9.1.2. Специальные способы создания библиотек пользователя

Прежде всего, созданные и отлаженные процедуры и программные модули можно сохранять в текстовых файлах во входном формате *Maple*-языка. В этом случае они впоследствии читаются *read*-предложением, которое корректно загружает в текущий сеанс определения как процедур, так и программных модулей (*стандартные средства пакета для модулей не могут обеспечить их корректного сохранения в файлах внутреннего m-формата*). Сохранение процедур и модулей производится по *save*-предложению:

$$\text{save}(N1, N2, \dots, Nk, \langle \text{CF} \rangle)$$

где фактические аргументы N_j определяют идентификаторы сохраняемых процедур и/или модулей, а *CF* – спецификатор принимающего файла (*имя либо полный путь к файлу*). Загрузка сохраненных процедур и модулей производится по предложению *read(<CF>)*, в результате чего определения процедур и модулей, находящихся в *CF*-файле, вычисляются и они становятся доступными текущему сеансу пакета. Таким образом, по *save*-предложению возможно создавать файлы формата входного языка, которые содержат корректные определения сохраненных в них объектов (*процедуры, модули и др.*). В то же время программные модули в файлах внутреннего формата (*m-файлы*) сохраняются некорректно, точнее без их тела. Данная проблематика детально рассмотрена в наших книгах [28-30,32-38,42-44]; там же представлен целый ряд наших средств по устранению такой ситуации. В частности, в целях устранения указанного недостатка нами была создана процедура *SaveMP*, обеспечивающая для пакета *Maple* релизов 6–11 корректное выполнение данной операции. Успешный вызов процедуры *SaveMP(F, M)* обеспечивает сохранение в файле *F* средств (*процедур и/или программных модулей*), чьи имена задаются списком либо множеством *M*. Сохранение таких средств производится в режиме дописывания (*APPEND*) с выводом поясняющего сообщения. Тогда как вызов *SaveMP(F, "list")* обеспечивает возврат имен содержащихся в файле *F* средств с указанием их типов. Наконец, вызов *SaveMP(F, M, "load")* обеспечит загрузку в текущий сеанс средств файла *F*, чьи имена определены списком/множеством *M*. С данной процедурой детальнее можно ознакомиться в [38,42-44]. Как показывает опыт, процедура *SaveMP* может вполне успешно использоваться для создания достаточно простых и эффективных нестандартных (*архивного типа*) библиотек пользователя.

Сохраняя процедуры и модули по *save*-предложению в файлах *входного* формата, мы, тем самым, создаем своего рода простейшие библиотеки, чьи средства загружаются в текущий сеанс по *read*-предложению и сразу же становятся доступными аналогично стандартным средствам пакета. Недостатком этой библиотечной организации (*наряду с некоторыми другими*) является то, что каждое новое обновление входящего в данную библиотеку средства требует обновления всей библиотеки. Поэтому в рамках такого подхода была создана процедура *simplel*, полезная в целом ряде приложений. Вызов процедуры *simplel(L {, N1,...,Nk})* допускает один или более фактических аргументов, где первый *L*-аргумент определяет полный путь к файлу входного *Maple*-формата с сохраняемыми или сохраненными процедурами и/или модулями. Вызов процедуры *simplel(L,N1,N2,...,Nk)* ничего не возвращает, сохраняя в файле *L* объекты, чьи имена определены фактическими аргументами, начиная со второго. Сохранение объектов производится в режиме дописывания (*APPEND*), позволяя сохранять (*архивировать*) все версии средств с указанием дат их сохранения. При этом, последующая загрузка файла *L* активирует в текущем сеансе лишь *последние* сохраненные версии объектов. Процедура *simplel* позволяет создавать библиотечный файл *L* в цепочке каталогов любого уровня вложенности, обрабатывая основные особые и ошибочные ситуации. Данная процедура может оказаться полезным средством при организации простых библиотек пользователя, несущих и архивные черты программных средств. Данная процедура может быть расширена новыми функциональными возможностями, что оставляем читателю в качестве достаточно полезного упражнения. С более общими средствами поддержки ведения простых библиотек пользователя, которые отличны от *Maple*-библиотек, можно познакомиться в наших работах [28-30,32-38,42-45].

Для организации простых библиотек пользователя возможно использовать еще один в ряде случаев полезный прием. В основу его ложится табличная структура, входами которой являются имена процедур и программных модулей, тогда как выходами – их определения. При этом, если для процедуры определение кодируется в чистом виде, то для программного модуля оно кодируется в следующем простом формате:

'parse("module <Имя> () export ... end module")'

где «*Имя*» определяет имя программного модуля. Затем созданная данным способом таблица сохраняется посредством *save*-предложения в файле любого допустимого формата (*внутреннем или входном*). После чего последующие загрузки этого файла по предложению *read* активируют в текущем сеансе сохраненные в нем средства, доступ к которым аналогичен доступу к средствам пакетного модуля. Но, если обращение к процедуре обеспечивается стандартным для таблиц способом, т.е. по конструкции:

<Имя таблицы>[<Имя процедуры>](...)

то к программному модулю требуется обращение по конструкции формата:

<Имя модуля> :- <Имя экспортной переменной>](...)

В свете представленного выше подхода к организации пользовательских библиотек на основе табличной организации может оказаться достаточно полезной процедура *SoftTab* [42-45]. Детальнее с такого типа организацией библиотек в среде *Maple* и ее особенностями можно ознакомиться в наших книгах [28-30,32-38,42-45]. Интересные

примеры, представленные там же, весьма наглядно иллюстрируют вышесказанное. С некоторыми другими нашими средствами ведения простых библиотек пользователя, которые организационно отличны от *Maple*-библиотек пакета, можно познакомиться в цитированных книгах. В книгах достаточно детально описана сущность алгоритмов реализованных средств, довольно полезных при работе с библиотеками пользователя нестандартной организации, наряду с файлами данных, содержащими определения *Maple*-объектов, прежде всего процедур и программных модулей. Там же возможно найти целый ряд других полезных средств для работы с библиотеками пользователя как аналогичными главной *Maple*-библиотеке, так и нестандартными.

Здесь приведем еще один подход к организации программных средств пользователя, который достаточно интересен с ряда точек зрения. Суть его состоит в следующем. В файлах внутреннего *m*-формата по одной сохраняются разработанные и отлаженные процедуры пользователя; все файлы размещаются в заданном подкаталоге главного каталога пакета. Следующий весьма простой пример иллюстрирует сказанное:

```
> HHH := (x, y, z) -> x + y^2 + z^3: GGG := (x, y, z) -> sqrt(x + y^2 + z^3) + sin(x*y + z):
> SSS := (x, y) -> (x*ln(2 + y) - sqrt(x + y))^2/(x + y + sin(x)*cos(y)):
> save(HHH, "C:/Program Files/Maple 10/User_Library/hhh.m"):
> save(GGG, "C:/Program Files/Maple 10/User_Library/ggg.m"):
> save(SSS, "C:/Program Files/Maple 10/User_Library/sss.m"):
```

На следующем этапе создается таблица, входами которой являются *имена* созданных процедур, а выходами полные пути к файлам внутреннего *m*-формата, содержащим определения этих процедур. При этом, имена файлов имеют вид «*имя.m*», где «*имя*» совпадает с именем процедуры, чье определение содержится в данном файле.

```
> TabProc := table([HHH = "C:/Program Files/Maple 10/User_Library/hhh.m",
                    SSS = "C:/Program Files/Maple 10/User_Library/sss.m",
                    GGG = "C:/Program Files/Maple 10/User_Library/ggg.m"]):
```

На следующем этапе такая таблица *TabProc* сохраняется в любой *Maple*-библиотеке, логически сцепленной с главной библиотекой пакета. При этом, допускается также сохранение данной таблицы и в главной библиотеке пакета. Ниже приведен пример сохранения данной таблицы в главной библиотеке пакета *Maple*.

```
> MapleLib({TabProc});
Warning, library <C:\\Program Files\\Maple 10\\lib> has received the
`READONLY`-attribute
Warning, tools <{TabProc}> have been saved/updated in <C:\\Program Files\\
Maple 10\\lib>
> restart; eval(TabProc);
table([HHH = "C:/Program Files/Maple 10/User_Library/hhh.m",
        SSS = "C:/Program Files/Maple 10/User_Library/sss.m",
        GGG = "C:/Program Files/Maple 10/User_Library/ggg.m"])
```

Пример иллюстрирует успешное сохранение таблицы *TabProc* в главной библиотеке с последующей проверкой процедуры сохранения. Далее, обеспечение возможности

вызовов процедур из файлов, содержащих определения требуемых процедур, будет обеспечиваться специальной процедурой **CALL(W)**, чей единственный фактический аргумент **W** определяет вызов требуемой процедуры. Первый вызов **CALL(H(x,y,...,z))** процедуры не только загружает в текущий сеанс *вычисленное* определение процедуры **H**, но и возвращает значение **H(x,y,...,z)**. При этом, все последующие **CALL(H(x,y,...,z))** также возвращают значение **H(x,y,...,z)**; при этом, все последующие вызовы требуемой процедуры в текущем сеансе пакета могут иметь стандартный вид **H(x,y,...,z)**. Между тем, исходный текст процедуры **CALL** имеет следующий простой вид; тогда как суть реализованного им алгоритма достаточно несложно усматривается после знакомстве с пакетом *Maple* в рамках, например, наших книг [42-44].

```
> CALL := proc(W::anything)
  local a, b, c;
  if type(W, 'function') then
    a:=convert('procname(args)', 'string'); b:=Search2(a, {"("}); c:=parse(a[b[1] + 1..b[2] - 1]);
    try ParProc(c) catch: try read TabProc[c] catch: error "file with definition of procedure
                                     %1 was not found", c end try end try;
    (proc() writeline("$$$$$$", cat(convert(W, 'string'), ";")); close("$$$$$$");
      read "$$$$$$"; % end proc)(), remove("$$$$$$")
    else W end if
  end proc;
```

> CALL(SSS(Pi, 2, 1)); $\Rightarrow \frac{(2 \pi \ln(2) - \sqrt{\pi + 2})^2}{\pi + 2}$

> SSS(1, 2, Pi); $\Rightarrow \frac{(2 \ln(2) - \sqrt{3})^2}{3 + \sin(1) \cos(2)}$

> CALL(ArtKr(42, 47, 67));
Error, (in CALL) file with definition of procedure **ArtKr** was not found

```
> CALL := proc(W::anything)
  local a, b, c;
  if type(W, 'function') then
    a:=convert('procname(args)', 'string'); b:=Search2(a, {"("}); c:=parse(a[b[1] + 1..b[2] - 1]);
    try ParProc(c) catch: try read cat("C:/Program Files/Maple 10/User_Library/", c, ".m")
      catch: error "file with definition of procedure %1 was not found", c end try end try;
    (proc() writeline("$$$$$$", cat(convert(W, 'string'), ";")); close("$$$$$$");
      read "$$$$$$"; % end proc)(), remove("$$$$$$")
    else W end if
  end proc;
```

> simplify(CALL(SSS(Pi/4, 3, 5))); $\Rightarrow \frac{1}{4} \frac{(\pi \ln(5) - 2 \sqrt{\pi + 12})^2}{\pi + 12 + 2 \sqrt{2} \cos(3)}$

> GGG(-1, Pi/2, Pi/2); $\Rightarrow \frac{\sqrt{-16 + 4 \pi^2 + 2 \pi^3}}{4}$

После отладки процедуры *CALL* она также сохраняется либо в главной библиотеке, либо в *Maple*-библиотеке пакета, логически связанной с ней, например:

> **MapleLib**(**{CALL}**);

Warning, library <C:\\Program Files\\Maple 10\\lib> has received the
`READONLY`-attribute

Warning, tools <{CALL}> have been saved/updated in <C:\\Program Files\\Maple
10\\lib>

Между тем, таблица *TabProc* и процедура *CALL* могут не сохраняться в библиотеках, логически сцепленных с главной библиотекой, а помещаться в инициализационный файл, например, «*Maple.ini*», находящийся в подкаталоге *Users* пакета. При загрузке пакета в текущий сеанс операционной системы все находящиеся в нем определения вычисляются и становятся доступными в текущем сеансе пакета. Но этот подход не представляется нам достаточно эффективным, ибо в рабочую область пакета будут загружены средства, которые, вообще говоря, могут оказаться не востребуемыми в текущем сеансе. Хотя при таком подходе модификация этих средств и, прежде всего, таблицы *TabProc* представляется нам несколько более простой и удобной.

При обнаружении вызовом *CALL(WWW(x,y,...,z))* отсутствия *m*-файла в подкаталоге *User_Library* с определением искомой процедуры *WWW* возникает особая ситуация, легко обрабатываемая программно. При необходимости модификации какой-либо процедуры пользователя заменяется только соответствующий ей *m*-файл, тогда как в случае необходимости расширить пользовательскую библиотеку такой организации новыми процедурами в подкаталоге *User_Library* сохраняются соответствующие им *m*-файлы, после чего соответствующим образом обновляется сама таблица *TabProc* с последующим обновлением ее в главной библиотеке пакета или в *Maple*-библиотеке, логически связанной с первой библиотекой.

Между тем, процедуру *CALL* можно легко модифицировать таким образом, чтобы не использовать таблицу *TabProc*, а именно: заранее зная путь к каталогу, содержащему файлы с определениями процедур пользователя, непосредственно в процедуре *CALL* вычислять путь к файлу с искомой пользовательской процедурой. В принципе, такой подход на первый взгляд представляется более простым, однако мы ограничиваемся конкретным каталогом с процедурами пользователя. Если это приемлемо, то вполне можно обойтись и без таблицы *TabProc*, которая позволяет использовать для разных процедур пользователя различные подкаталоги, содержащие их определения. Более того, второй пример фрагмента представляет исходный текст процедуры *CALL*, не требующей использования таблицы *TabProc*, что в ряде случаев представляется нам достаточно целесообразным.

Между тем, следует иметь ввиду, что *m*-файлы могут содержать также определения нескольких процедур, связанных логически или сгруппированных по определенной теме. В принципе, допустим довольно широкий диапазон различных сочетаний по разбиению пользовательских процедур по *m*-файлам библиотеки, организованной описанным образом. Однако в данном случае во избежание усложнения алгоритма обработки имен *m*-файлов, содержащих более одного определения, целесообразно

использовать именно таблицу *TabProc*. При этом, вид таблицы *TabProc* в этом случае несколько отличается относительно ее первоначального вида, а именно:

```
> TabProc := table([HHH = "C:/Program Files/Maple 10/User_Library/hhh.m",
                    [SSS, GGG] = "C:/Program Files/Maple 10/User_Library/sss.m",
                    VVV = "C:/Program Files/Maple 10/User_Library/vvv.m"]):
```

Если *m*-файл содержит определение только одной процедуры, то элемент таблицы кодируется подобно элементам для процедур *HHH* и *VVV*, тогда как для случая более одной процедуры соответствующий вход таблицы *TabProc* кодируется в виде списка, элементами которого являются имена процедур, чьи определения находятся в одном и том же *m*-файле; путь к этому файлу определяется выходом данного элемента как это проиллюстрировано для процедур с именами *SSS* и *GGG*.

С учетом подобной организации таблицы *TabProc* алгоритм процедуры *CALL* может быть представлен исходным текстом нижеследующей модификации процедуры:

```
> CALL := proc(W::anything)
    local a, b, c;
    if type(W, 'function') then
        a:=convert('procname(args)', 'string'); b:=Search2(a, {"("}); c:=parse(a[b[1] + 1..b[2] - 1]);
        try ParProc(c) catch: try read TabProc[seq(`if` (belong(c,k),k,ERROR("file with definition
        of procedure %1 was not found", c)), k=op([map(op, [indices(TabProc)])])]
        end try
    end try;
    (proc() writeline("$$$$$$", cat(convert(W, 'string'), ";")); close("$$$$$$");
        read "$$$$$$"; % end proc(), remove("$$$$$$")
    else W end if
end proc;
```

$$> \text{CALL}(\text{SSS}(\text{Pi}, 2, 1)), \text{SSS}(1, 2, \text{Pi}); \quad \Rightarrow \quad \frac{(\pi \ln(4) - \sqrt{\pi + 2})^2}{\pi + 2}, \frac{(\ln(4) - \sqrt{3})^2}{3 + \sin(1) \cos(2)}$$

Между тем, вместо инициации *особой* ситуации можно определять возврат вызова несуществующей процедуры *невычисленным* с выводом диагностического сообщения, что весьма просто обеспечивается заменой в первых двух представленных вариантах процедуры *CALL* следующей конструкции, а именно:

«error "file with definition of procedure %1 was not found", c» на конструкцию вида
«WARNING("file with definition of procedure %1 was not found", c): W»

тогда как исходный текст *третьего* варианта процедуры заменяется следующим:

```
> CALL := proc(W::anything)
    local a, b, c, k;
    if type(W, 'function') then a := convert('procname(args)', 'string');
        b := Search2(a, {"("); c := parse(a[b[1] + 1 .. b[2] - 1]);
        try ParProc(c)
        catch: try read TabProc[(proc() for k in op([map(op, [indices(TabProc)])]) do
            if belong(c, k) then return k end if end do;

```

```

WARNING("file with definition of procedure %1 was not found", c)
end proc)(c)]
catch "read must have a file name": W end try
end try;
(proc) writeline("$$$$$$", cat(convert(W, 'string'), ";")); close("$$$$$$");
read "$$$$$$"; % end proc(), remove("$$$$$$")
else W
end if
end proc:

```

Схематично предложенный выше подход к организации алгоритма, реализованного первым вариантом процедуры *CALL*, можно представить нижеследующим образом:

Вызов процедуры <i>CALL</i> (<i>W</i> (<i>x</i> , ..., <i>z</i>))	
Если <i>W</i> (<i>x</i> , ..., <i>z</i>) не вызов процедуры, то возвращается вычисленное <i>W</i> (<i>x</i> , ..., <i>z</i>)	
Иначе:	(a) вычисляется имя искомой процедуры <i>W</i>
	(b) определяется наличие процедуры в текущем сеансе пакета или в его библиотеках, логически сцепленных с главной <i>Maple</i> -библиотекой
	(c) если процедура идентифицирована в (b), то возвращается <i>W</i> (<i>x</i> , ..., <i>z</i>), иначе через <i>TabProc</i> [<i>W</i>] загружается в текущий сеанс ее определение из соответствующего файла и вычисляется, становясь доступным на период текущего сеанса, затем возвращается значение <i>W</i> (<i>x</i> , ..., <i>z</i>)
	(d) если запрошенный файл с определением не найден в подкаталоге библиотеки пользователя, то идентифицируется особая ситуация с возвратом соответствующего диагностического сообщения либо возвращается вызов несуществующей процедуры невычисленным с выводом соответствующего диагностического сообщения
	(e) создание справочной базы для библиотеки пользователя, имеющей организацию, подобную пакетной, либо отличную от нее

Представленная схема алгоритма, реализуемого *первым* типом процедуры *CALL*, дает возможность достаточно эффективно организовывать пользовательские библиотеки, содержащие отдельные файлы с определениями процедур и логически не связанные с главной библиотекой пакета *Maple*. Данная организация библиотеки (*скорее архива*) позволяет весьма легко обновлять библиотеку и довольно несложно ее использовать, зная лишь *имена* процедур, содержащихся в ней. Вышесказанное просто переносится на второй и третий типы процедуры *CALL* с довольно тривиальной модификацией. Такая организация пользовательских библиотек представляется нам весьма простой для использования и в среде целого ряда других программных средств, не имеющих достаточно приемлемой поддержки библиотек пользовательских средств.

Выше реализации процедуры *CALL* охватывали лишь объекты типа «*процедур*», тогда как нижеследующий вариант такой процедуры дополнительно охватывает и модули как стандартной структуры, так и заданные табличной структурой. Так, в качестве типичного примера модуля с табличной структурой может служить модуль *linalg*.

```

> CALL := proc(W::anything)
local a, b, c, d, h, t;
  t := x -> WARNING("file with definition of procedure or module %1 was not found", c);
  d := proc(x) try read TabProc[x] catch: h := 1 end try;
    (proc() writeline("$$$$$$", cat(convert(W, 'string'), ";")); close("$$$$$$");
      read "$$$$$$"; % end proc)(), remove("$$$$$$")
    end proc;
  if type(W, 'function') then a := convert('procname(args)', 'string');
    b := Search2(a, {"(", "["}); c := parse(a[b[1] + 1 .. b[2] - 1]);
    try ParProc(c) catch: d(c) end try;
    if h = 1 then t(c) else W end if
  end if;
  if type(c, `module`) or type(c, 'table') then d(c); if h = 1 then t(c) end if else W
  end if
end proc:

> CALL(SSS(Pi, 2, 1)), SSS(1, 2, Pi);            $\Rightarrow$   $\frac{(\pi \ln(4) - \sqrt{\pi + 2})^2}{\pi + 2}, \frac{(\ln(4) - \sqrt{3})^2}{3 + \sin(1) \cos(2)}$ 

> CALL(ArtKr(42, 47, 67));
Warning, file with definition of procedure or module ArtKr was not found
ArtKr(42, 47, 67)
> CALL(linalg[matrix](3, 3, [V, G, S, 42, 47, 67, Art, Kr, Arn]));

$$\begin{bmatrix} V & G & S \\ 42 & 47 & 67 \\ Art & Kr & Arn \end{bmatrix}$$

> CALL(ZZZ[matrix](3, 3, [V, G, S, 42, 47, 67, Art, Kr, Arn]));
Warning, file with definition of procedure or module ZZZ was not found
ZZZmatrix(3, 3, [V, G, S, 42, 47, 67, Art, Kr, Arn])

```

Еще раз акцентируем внимание на том моменте, что *доступность* средств в текущем сеансе пакета определяется *наличием* их определений в ядре пакета (а значит в рабочей области пакета), главной библиотеке и в *Maple*-библиотеках, логически сцепленных с главной библиотекой пакета и организационно подобных ей.

Следует отметить, что вместо *m*-файлов внутреннего формата *Maple*-языка вполне возможно в качестве файлов библиотеки, содержащих определения *Maple*-объектов, использовать и весьма просто редактируемые *txt*-файлы с определениями, например, файл «*ArtK.txt*» содержит определения процедур и программного модуля, а именно:

```

ArtKr := (x) -> sqrt(sin(x) + cos(x)):
ArtKr1 := proc(x, y) if evalf(x*y) >= 0 then sin(x)+cos(y) else sin(x)+cos(y) end if end proc:
SveGal := module () export x, y; x:=(a, b) -> a+b mod 7; y:=(a, b) -> a*b mod 7 end module:

```

Загрузив данный файл по предложению **read**, обеспечиваем доступность в текущем сеансе пакета средств, чьи определения находятся в файле *txt*-формата.

```
> read("C:\\Program Files\\Maple 11\\Users\\ArtKr.txt"):
> ArtKr1(Pi/4, Pi/4), ArtKr(Pi/2);           ⇒      2^(1/2), 1
> SveGal:- x(42, 68), SveGal:- y(42, 47);    ⇒      5, 0
```

Данный подход позволяет сохранять в *txt*-файлах библиотеки пользователя наборы определений любых *Maple*-объектов, включая программные модули; пакет *Maple* для корректного сохранения в файлах *m*-формата определений программных модулей не располагает стандартными средствами. Именно поэтому, нами были разработаны процедура *savem* и некоторые другие, которые обеспечивают корректное сохранение определений программных модулей в файлах *m*-формата, достаточно существенно расширяя возможности программной среды пакета *Maple* версий 6 – 10 [9,37,38,44,45]. И только, начиная с *Maple 11*, данная весьма существенная недоработка устранена.

Довольно важной компонентой пользовательской библиотеки является ее справочная база. Принцип ее организации для пользовательской библиотеки, подобной главной библиотеке пакета, достаточно детально рассмотрен в наших книгах [9,42-44]. Между тем, этот же принцип пригоден и для библиотек пользователя организации, которая отлична от главной библиотеки пакета. Для этого достаточно использовать средства справочной системы пакета. Однако поскольку справочная система находится *внутри* графического интерфейса пакета, то она не может обрабатываться непосредственно процедурами пакета. Вместо этого доступ к справочной системе обеспечивается через функциональный вызов *INTERFACE_HELP(...)*, поддерживающий базовые процедуры работы со справочной базой – создание базы справок, ее обновление и визуализация. Страницы справки сохраняются в одном или нескольких файлах *help*-базы с именем «*maple.hdb*». Каждая справочная база содержит одну или несколько страниц справки в зависимости от количества средств, содержащихся в библиотеке пользователя. Для обеспечения доступа к *help*-базе через *графический* интерфейс (т.е. подобно справочной базе пакета) пути к каталогам, содержащим эти файлы, должны быть закодированы в переменной *libname* пакета. Так, следующий весьма простой пример иллюстрирует создание справочная база для процедуры *HHH*, *m*-файл которой с ее определением находится в подкаталоге «C:/Program Files/Maple 10/User_Library». Так, следующий пример иллюстрирует создание справочной базы для 3 процедур *SSS*, *GGG* и *HHH*, рассмотренных несколько выше. Между тем, если первый функциональный вызов *INTERFACE_HELP* создает справочную базу с единственной справочной страницей для *HHH*-процедуры, то все последующие вызовы *INTERFACE_HELP* такого же типа пополняют базу новыми справочными страницами. Последняя строка этого примера иллюстрирует подключение библиотеки пользователя с созданной справочной базой к цепочке библиотек в предопределенной *libname*-переменной пакета *Maple*.

```
> INTERFACE_HELP(insert, topic="HHH", text = TEXT("Справка по HHH процедуре"),
  library = "C:/Program Files/Maple 10/User_Library");
> INTERFACE_HELP(insert, topic="GGG", text = TEXT("Справка по GGG процедуре"),
  library = "C:/Program Files/Maple 10/User_Library");
> INTERFACE_HELP(insert, topic="SSS", text = TEXT("Справка по SSS процедуре"),
  library = "C:/Program Files/Maple 10/User_Library");
> libname := libname, "C:/Program Files/Maple 10/User_Library":
```

Первый аргумент функции *INTERFACE_HELP* определяет следующие процедуры по работе со справочной базой, а именно: *display* – визуализация заданной справочной страницы на экране, *insert* – размещение в базе «*maple.hdb*» справочной страницы с заданным содержанием, *delete* – удаление из базы «*maple.hdb*» справочной страницы с указанным именем. После создания справочной базы пользователя ее очень просто модифицировать средствами графического интерфейса пакета. При создании нашей библиотеки средств для Maple [45] нами использовался именно данный подход.

Из сказанного следует, что пакет Maple не только располагает достаточно развитыми средствами по организации программных средств пользователя в виде библиотек, но и позволяет пользователю создавать библиотеки средств, организационно отличные от стандартной Maple-библиотеки пакета. При этом, организационное разнесение библиотек на двух уровнях – справочных баз на уровне графического интерфейса и собственно библиотек на уровне программной среды – является весьма существенной предпосылкой для создания *общей* справочной базы как для главной библиотеки, так и для библиотек пользователя совершенно независимо от их организации. Рассмотрим теперь возможности организации средств пользователя в среде пакета Mathematica.

9.2. 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 содержанием, тогда как оператор перенаправления «>>>» дописывает информацию в конец принимающего файла, сохраняя его предыдущее содержание.

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 ввиду, что определяя объект в *Mathematica*, чье определение зависит от других объектов, необходимо загружать в текущую сессию и такие объекты, чтобы обеспечить корректный доступ к основному объекту. Поэтому, сохранять следует все связанные определения. Функция **Save** решает эту задачу, выявляя определения всех объектов, от которых зависит сохраняемый объект и автоматически их сохраняет, но для эффективности исключает из этого списка определения объектов, имена которых имеют *Protected*-атрибут, предполагая доступность таких объектов в текущем сеансе. О других способах сохранения определений *Mathematica*-объектов в файлах разных форматов более говорить нецелесообразно, как не имеющих отношения к проблеме организации программного обеспечения пользователя, для решения которой в виде, определенном выше, вполне достаточно сохранения определений во входном *Input*-формате. Между тем, в случае достаточно сложных и объемных определений вполне приемлемым и эффективным может оказаться и сохранение по функции **DumpSave**, которая обеспечивает сохранение во *внутреннем* формате пакета (*бинарный mх-файл*), наиболее оптимальный для считывания пакетом. Однако, в качестве недостатка этой функции является то, что сохраненные по ней файлы успешно считываются 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 (*m. n. 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[418]:= Import["D:\\Math_myLib\\ArtKr.m"]
Import::nfil: File not found during Import. >>
Out[418]= $Failed
In[419]:= Off[Import::nfil]
In[420]:= Import["D:\\Math_myLib\\ArtKr.m"]
Out[420]= $Failed
In[421]:= On[Import::nfil]
In[422]:= F[x_] := Module[{a}, Off[Import::nfil]; a := Import[x]; On[Import::nfil]; a]
In[423]:= F["D:\\Math_myLib\\ArtKr.m"]
Import::nfil: File not found during Import. >>
Out[423]= $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 предыдущем разделе относительно случая 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[342]:= NF[x_] := Sin[x]*Cos[x]
In[343]:= 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[442]:= Save["D:\\Math_myLib\\NF.txt", NF]
In[443]:= 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 функций.

```
In[771]:= 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[772]:= CALL[42.68]
Out[772]= 42.68
In[773]:= CALL[Sin[Pi/2]]
Out[773]= 1
In[774]:= Directory[]
Out[774]= "C:\\Documents and Settings\\Aladjev\\My Documents"
In[775]:= CALL[AVZ[42, 47, 67, 89, 96]]
Out[775]= AVZ[42, 47, 67, 89, 96]
In[776]:= CALL[Log[68]*Cos[42.47]]
Out[776]= 0.246697
In[777]:= CALL[NF[68.42]]
Out[777]= -0.491857
In[778]:= CALL[ArtKr[68.42, 63.47]]
Out[778]= 7.81774
In[779]:= CALL[KrArt[68.42, 63.47]]
Out[779]= KrArt[68.42, 63.47]
In[780]:= CALL[NF[z]]
Out[780]= Cos[z]*Sin[z]
In[781]:= CALL[ArtKr[x, y]]
Out[781]= Sqrt[47*Cos[x] + 42*Sin[y]]
In[782]:= CALL[ArtKr[42.68, 47.63]]
Out[782]= 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[111]:= $UserBaseDirectory
```

```
Out[111]= "C:\\Documents and Settings\\Aladjev\\Application Data\\Mathematica"
```

```
In[112]:= $BaseDirectory
```

```
Out[112]= "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 отлаженной 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[1347]:= AGN = Sqrt[CALL[NF[42.47]]^2 + CALL[ArtKr[19.89, 19.96]]^3]
```

```
Out[1347]= 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_1, N_2, \dots, N_k }] 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, 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[659]:= EmptyFileQ[F_]; FileExistsQ[F] := If[Read[F, Byte] === EndOfFile, Close[F];
      True, Close[F]; False]]

In[660]:= {EmptyFileQ["D:\\G1"], EmptyFileQ["D:\\G2"], EmptyFileQ["D:\\GS"]}
Out[660]= {False, True, EmptyFileQ["D:\\GS"]}

In[822]:= UserLib[L_;/; FileExistsQ[L], f_;/; ListQ[f]] := Module[{a, b = "", c, d = 0},
  If[f[[1]] === "print" && f[[2]] === "all" && ! EmptyFileQ[L], FilePrint[L],
  If[f[[1]] === "print" && f[[2]] != "all" && ! EmptyFileQ[L],
    a = OpenRead[L]; While[b != "EndOfFile", b = Read[a, String];
    If[SuffPref[b, f[[2]]] <> "[", 1], Print[b]; d = 1; Continue[],
    If[d == 1, If[b === "\"$$$$$\"", Close[a]; Break[], Print[b];
    Continue[], Continue[]]; If[d == 1, Null,
    Print[f[[2]] <> " is absent in Library " <> StrStr[L]],
  If[f[[1]] === "add", PutAppend[ToExpression["Definition[" <> f[[2]] <> "], L];
    PutAppend["$$$$$", L],
  If[f[[1]] === "load" && f[[2]] === "all" && ! EmptyFileQ[L], Get[L];,
  If[f[[1]] === "load" && f[[2]] != "all" && ! EmptyFileQ[L],
    a = OpenRead[L]; c = ""; While[b != "EndOfFile", b = Read[a,
    String]; If[SuffPref[b, f[[2]]] <> "[", 1], c = c <> b; d = 1; Continue[],
    If[d == 1, If[b === "\"$$$$$\"", Close[a]; Break[], c = c <> b;
    Continue[], Continue[]]; If[d == 1, ToExpression[c],
    Print[f[[2]] <> " is absent in Library " <> StrStr[L]],
  If[f[[1]] === "names" && f[[2]] === "list" && ! EmptyFileQ[L],
    a = OpenRead[L]; c = {}; While[b != "EndOfFile", b = Read[a, String];
    If[Quiet[StringTake[b, {1, 1}]] != " " && b != "\"$$$$$\"",
    c = Append[c, StringTake[b, {1, Flatten[StringPosition[b, "["]][1] - 1}]],
    Continue[]]; Close[a]; Return[c, Defer[UserLib[L, f]]]]]]]]

In[823]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "Lib"}]
Out[823]= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "Lib"}]
In[824]:= UserLib["D:\\Math_MyLib\\Library.txt", {"names", "list"}]
Out[824]= UserLib["D:\\Math_MyLib\\Library.txt", {"names", "list"}]
In[825]:= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "all"}]
Out[825]= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "all"}]
In[826]:= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "Lib"}]
Out[826]= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "Lib"}]
In[827]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "all"}]
Out[827]= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "all"}]
In[828]:= UserLib["D:\\Math_MyLib\\Library.txt", {"add", "Lib1"}]
=====
In[835]:= UserLib["D:\\Math_MyLib\\Library.txt", {"add", "Lib8"}]
In[836]:= UserLib["D:\\Math_MyLib\\Library.txt", {"names", "list"}]
Out[829]= {"Lib1", "Lib2", "Lib3", "Lib4", "Lib5", "Lib6", "Lib7", "Lib8"}

```

```

In[830]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "Lib7"}]
In[831]:= UserLib["D:\\Math_MyLib\\Library.txt", {"Names", "List"}]
Out[831]= UserLib["D:\\Math_MyLib\\Library.txt", {"Names", "List"}]
In[832]:= UserLib["D:\\Math_MyLib\\Library.txt", {"load", "ArtKr"}]
          ArtKr is absent in Library "D:\\Math_MyLib\\Library.txt"
In[833]:= UserLib["D:\\Math_MyLib\\Library.txt", {"print", "ArtKr"}]
          ArtKr is absent in Library "D:\\Math_MyLib\\Library.txt"

```

Прежде всего, вводим простую функцию **EmptyFileQ[F]**, возвращающую True, если *F* является пустым файлом, и False в противном случае; если файл *F* отсутствует, вызов возвращается невычисленным. Тогда как достаточно простая процедура **UserLib[L, f]** поддерживает ряд массовых функций по ведению простой библиотеки пользователя, располагающейся в файле *L txt*-формата. В качестве второго *фактического* аргумента *f* выступает двух-элементный список, для которого на сегодня допустимыми парами значений элементов являются нижеследующие, а именно:

{"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 с именем *ProcName*. Исходный текст такой простой **Help**-процедуры и пример ее типичного применения приводятся ниже, а именно:

```
In[25]:= Help[x_] := Module[{}, FilePrint["D:\\Math_myLib\\" <> ToString[x] <> ".hb"]]  
In[26]:= 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 обеих 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.

9.3. Оценка эффективности пакетов Maple и Mathematica по организации пользовательского программного обеспечения

Рассмотрев на принципиальном уровне возможности пакетов *Maple* и *Mathematica* в плане предоставляемых ими средств организации пользовательского программного обеспечения, попытаемся представить свою точку зрения на эти два весьма важных аспекта. Однако, прежде сделаем одно весьма существенное замечание, поясняющее суть наших доводов. И пакет *Maple*, и пакет *Mathematica* обеспечивают возможность создания в их среде приложений в виде документов, содержащих и оформительские, и вычисляемые элементы, т.е. такие документы можно сохранять в файлах формата, допускаемого пакетом, и затем, загружая в текущий сеанс и вычисляя, обеспечивать все содержащиеся в них определения доступными в текущем сеансе. Как правило, к числу таких документов относятся решения достаточно сложных прикладных задач, которые не имеет смысла тесно ассоциировать с пакетом ввиду их востребованности 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 форумах по вышеупомянутым пакетам 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). Так, для дифференцирования выражений в пакетах *Maple* и *Mathematica* имеются процедура *diff* и функция *D* соответственно, как это иллюстрирует следующий достаточно простой фрагмент, а именно:

Maple:

```
> diff(x*sin(x) + cos(x), x);           ⇒      x cos(x)
> diff((x*sin(x) + cos(x))/x^3, x^3);
Error, wrong number (or type) of parameters in function diff
> diff(x*sin(x) + cos(x), x);           ⇒      x cos(x)
> diff((x*sin(x) + cos(x))/x^3, x^3);    ⇒       $\frac{-x \sin(x) - \cos(x)}{x^6}$ 
```

Mathematica:

```
In[46]:= D[x*Sin[x] + Cos[x], x]
Out[46]= x Cos[x]
In[47]:= D[(x*Sin[x] + Cos[x])/x^3, x^3]
General::ivar: x^3 is not a valid variable. >>
Out[47]=  $\partial_{x^3} (\cos[x] + x \sin[x]) / x^3$ 
In[48]:= Df[(x*Sin[x] + Cos[x])/x^3, x^3]
Out[48]= - (Cos[x] + x Sin[x]) / x^6
```

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 нами были созданы две достаточно простые процедуры, решающие задачу дифференцирования выражения по любому из входящих в него подвыражений. Ниже представлены исходные тексты данных процедур, именованные *diff* и *Df* соответственно для *Maple* и *Mathematica*, с некоторыми типичными примерами их применения для дифференцирования.

```

> Subs := proc(s::{'=' , set('='), list('=')), Z::{set, algebraic, procedure, list, `module`, string})
local k, Kr, Res;
Kr := proc(s, Z)
local a, b, c, d, h, Res;
a, b := map(convert, [lhs(s), rhs(s)], 'string'), cat("##", convert(eval(Z), 'string'), "##");
c := Search2(b, {a[1]});
if c = [] then return convert(eval(Z), 'string') end if;
h := [seq([c[k], c[k] + length(a[1]) - 1], k = 1 .. nops(c))];
d := [1 .. h[1][1] - 1, seq(h[k][2] + 1 .. h[k + 1][1] - 1, k = 1 .. nops(h) - 1),
h[-1][2] + 1 .. length(b)];
Res := cat(seq(op([b[d[k]], `if` (member(op(0, parse(a[2])), {`+`, `*`, `^`}), cat("(", a[2], ")"),
convert(a[2], string))), k = 1 .. nops(d) - 1), b[d[-1]][3 .. -3] end proc; Res:=Kr(s, Z);
if member(whattype(Z), {string, symbol}) then convert(Res,whattype(Z))
else simplify(parse(Res))
end if;
StringTools[SubstituteAll](StringTools[SubstituteAll](Res, "1/1/", "1/"), "1*1/", "1/");
if member(whattype(Z), {string, symbol}) then convert(Res, whattype(Z))
else simplify(parse(Res)) end if
end proc;
> diff := proc(x::algebraic, y::algebraic)
local h, Res;
if type(x, 'numeric') or type(y, 'numeric') then return x else Res := x end if;
for h in [args[2] .. -1] do
if type(h, 'symbol') then Res := diff(Res, h) else Res := Subs(k = h, diff(Subs(h = k, Res), k))
end if
end do;
simplify(Res)
end proc;
> diff(x*sin(x)*42*cos(x)^2/x^3 - (a+b)/x^2 + ln(x)/x^2, x^2);

$$\frac{-42 \sin(x) \cos(x)^2 + a + b - \ln(x)}{x^4}$$

> diff(x*sin(x)*42*cos(x)^2/x^3 - (a+b)/x^2 + ln(x)/x^2, x^2);
Error, wrong number (or type) of parameters in function diff
> diff(x*sin(x)*42*cos(x)^2/x^3 - (a+b)/x^2 + ln(x)/x^2, x^2, x^2, x^4, x^8);

```


$$\frac{-42 \sin(x) \cos(x)^2 + a + b - \ln(x)}{x^{16}}$$

> diff(2010, x^2), diff(x^4, 47), diff(1/(a+x)^2, (a+x), 68), diff(42, 68); \Rightarrow 2010, x^4, 0, 42

Описание работы с процедурой *diff* можно найти в справке по ней в Библиотеке [45]. Дополнительно приведена и процедура *Subs*, которая существенно используется *diff* и расширяет возможности стандартной процедуры *subs*, имеющей ряд недостатков; в этом отношении созданная процедура *Subs* свободна от ряда недостатков процедуры *subs*. Представленная ниже процедура *Df* менее функциональна, чем процедура *diff* для пакета *Maple*, ввиду того, что она приведена с целью иллюстрации сказанного, а выполняемая ею функция достаточно хорошо просматривается, исходя из ее текста. Заинтересованный читатель вполне сможет ее функционально расширить по своему собственному усмотрению, например, возможностью кратного дифференцирования по различным подвыражениям дифференцируемого выражения.

```
In[247]:= 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[248]:= D[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]]
Out[248]= Sin[x]/Sqrt[Cos[x] + Sin[x]^2]
In[249]:= Df[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]]
Out[249]= Sin[x]/Sqrt[Cos[x] + Sin[x]^2]
In[250]:= D[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]^2]
General::ivar: Sin[x]^2 is not a valid variable. >>
Out[250]=  $\partial_{\sin[x]^2} \sqrt{\cos[x] + \sin[x]^2}$ 
In[251]:= Df[Sqrt[Sin[x]^2 + Cos[x]], Sin[x]^2]
Out[251]= 1/(2 Sqrt[Cos[x] + Sin[x]^2])
In[252]:= Df[(Cos[x]^2 + x*Sin[x]^2)/Cos[x]*Sin[x]^2, Sin[x]^2]
Out[252]= x*Sin[x]*Tan[x]
In[253]:= 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[253]=  $\partial_{\sin[x]^2} (\sin[x] (\cos[x]^2 + x \sin[x]^2) \tan[x])$ 
In[254]:= D[x, 5]
General::ivar: 5 is not a valid variable. >>
Out[254]=  $\partial_5 x$ 
In[255]:= Df[x, 5]
Df::incorrect second argument {5}
In[256]:= Df[5, x]
```

```

Out[256]= 0
In[257]:= D[5, x]
Out[257]= 0
In[258]:= Df[(x^2*cos[x]^2 + x*sin[x]^2)/(Log[x^2] - x^2*cos[x]*sin[x]^2), x^2]
Out[258]= 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[259]:= 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[260]:= Df[(x^2*cos[x]^2 + x*sin[x]^2)/(Log[x^2] - x^2*cos[x]*sin[x]^2), x^2]
Out[260]= (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[261]:= 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[261]= (x (x Cos[x]^2 + Log[x^2]))/(Log[x^2] - Sqrt[x^2 + Sin[x]]^2
In[262]:= {Df[2010, x^2], Df[x^2, x^2], Df[x^2, 420], Df[72, 420]}
Out[262]= {0, 1, 0, 0}
In[263]:= Df[(a*Sqrt[x + Cos[x]] + b*cos[x])/(d + Cos[x]), Cos[x]]
Out[263]= (a (d - 2 x) - a Cos[x] + 2 b d Sqrt[x + Cos[x]])/(2 (d + Cos[x])^2 Sqrt[x + Cos[x]])
In[264]:= Simplify[D[(a*Sqrt[x + Cos[x]] + b*cos[x])/(d + Cos[x]), Cos[x]]]
Out[264]= (a (d - 2 x) - a Cos[x] + 2 b d Sqrt[x + Cos[x]])/(2 (d + Cos[x])^2 Sqrt[x + Cos[x]])
In[265]:= Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y],
Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]

```

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. Однако не только *повышение* эффективности пакетных средств или расширения их функциональности обуславливают необходимость их доработки с замещением их пользовательскими (*так мы поступали в случае Maple*). К сожалению, далеко не всегда такой подход оказывается продуктивным, хотя и необходимым. Так, например, для стандартной операции `^`` (`**`) пакета, являющейся одной из базовых в математике, крайне настоятельно требуется модификация, так как результат `0^0 = 1` некорректен, но в данном случае наш подход не дает результата, т.к. эта операция не только защищена от обновления, что легко обходится, но и находится в ядре пакета, имея *максимальный* приоритет относительно средств, определения которых находятся в главной библиотеке пакета *Maple*. В этом отношении *Mathematica* дает корректный результат, как весьма наглядно иллюстрирует следующий пример, а именно:

```

In[2522]:= 0^0
Power::indet: Indeterminate expression 0^0 encountered. >>
Out[2522]= Indeterminate

```

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 *diff* и *Df* или некоторые их функциональные аналоги.

Естественно, оформлять такие средства можно довольно просто, сохраняя, например, в файлах *tex*-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.

Пакет *Maple* предоставляет весьма удобный и простой механизм для решения задачи организации пользовательского программного обеспечения. Данный механизм выше описан достаточно детально для пользователей данного пакета и вполне понятен для пользователей *Mathematica*. Пользователю довольно просто средствами пакета либо разработанными нами [43–45] создать собственную библиотеку со справочной базой и по мере готовности программных средств (*при этом, в качестве них допустимы любые объекты пакета: процедуры, модули, таблицы и т.д.*) помещать их в библиотеку, которая структурна аналогична главной библиотеке пакета. Справочная база по библиотеке также создается весьма просто средствами пакета или нашими и обновляется даже на уровне *графического интерфейса пользователя (GUI)* справочными страницами по всем библиотечным средствам, которые легко оформлять аналогичными записям справки по средствам самого пакета. При этом, имеется возможность [44] включения наиболее часто используемых (*массовых*) средств пользователя непосредственно в библиотеку пакета, а справки по ним в справочную базу пакета. Правда, в более старших версиях пакета такая возможность прямо не поддерживается, но посредством наших средств это вполне возможно. Конечно, данное решение должно быть весьма взвешенным, но с учетом персонального характера использования пакета в целом ряде случаев такое решение вполне допустимо. Данный подход позволяет организовывать программное обеспечение пользователя полностью унифицированным с библиотекой пакета, что весьма существенно упрощает его использование при решении различных задач. В частности, пакет предоставляет простую возможность определения уровней доступа к пакетным и пользовательским средствам в текущем сеансе. Данное важное свойство обеспечивается простым изменением порядка путей к библиотекам пользователя и пакета в предопределенной *libname*-переменной пакета.

Наконец, следует остановиться на средствах *восстановления* поврежденных библиотек и статистической обработке используемых библиотеками средств. Для данных целей нами был создан ряд процедур [45], обеспечивающих как восстановление библиотек, имеющих достаточно серьезные повреждения, так и предоставляющих статистику по

функционированию библиотеки (точнее, использованию ее средств) в среде пакета, что позволяет решать важные вопросы оптимизации пользовательских библиотек.

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 «необъять 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 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.

9.4. Программирование больших систем в среде математических пакетов *Mathematica* и *Maple*

Построение достаточно больших систем программного обеспечения в обоих пакетах и *Maple*, и *Mathematica* предполагает следование общим принципам, используемым при построении, практически, любой большой системы программного обеспечения. Детали могут быть специфичны для программной среды (*Maple*, *Mathematica* 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 больше можно и использовать их. Принципы в определенной степени выведены из нашего опыта разработки достаточно больших проектов в программной среде обоих пакетов – среде языков *Maple* и *Mathematica*.

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.

В любом случае вопросу разработки архитектуры проекта следует уделить внимания даже больше, чем собственно программированию. Наш опыт разработки достаточно сложных программных систем со всей очевидностью подтверждает факт, разработка качественной архитектуры, учитывающей все основные особенности системы, много важнее и сложнее, чем этап программирования. При наличии хорошей архитектуры с достаточно продуманной детализацией составляющих ее компонент уже довольно несложно программировать систему коллективом программистов различного уровня под руководством системных аналитика и программиста. Немало найдется довольно сносных программистов для программирования большинства компонент системы, да и исправлять их программы намного проще, чем изменять архитектуру системы. При этом, весьма значительную роль следует отвести разработке документации, удобной как для пользователя, так и для последующего ведения системы, например, в случае ее модификации под изменившиеся условия эксплуатации.

И *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. Имеются 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 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 малопригодной 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 *Math*-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 (APM)* пакета, представляющее собой интегрированную среду разработки для *Mathematica*. *APM* базируется на широко используемой платформе *IDE* по имени *Eclipse*; при этом, *APM* можно использовать непосредственно или можно установить *Mathematica* инструменты в *Eclipse*. *APM* 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, *APM* 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 *APM* поддерживает интеграцию с *CVS* и может быть достаточно легко настроен на работу с рядом других систем ведения исходных кодов. Более того,

APM обеспечивает поддержку разработок в среде других языков типа *Java*, *Python*, *C* и *C++*, позволяя интегрировать их в среду *Mathematica*. В качестве последней важной особенности *APM* 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 *APM* 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;** *APM* 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; *APM* 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 клиента к *CVS*, расширяется на многие другие виды управления кодом;
- **Ошибки и сообщения;** обеспечивает нахождение и отчет по многим классам ошибок во всех файлах с исходным кодом проекта независимо от их режима открытия, что во многом облегчает отладку проекта, содержащего много файлов.

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[793]:= MemoryInUse[]
Out[793]= 20320708
In[794]:= 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[795]:= MemoryInUse[]
Out[795]= 20339892
In[796]:= Share[]
Out[796]= 1702208
In[797]:= MemoryInUse[]
Out[797]= 18639260
```

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.

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 версий.

Глава 10. 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 этом качестве могут быть:

- средства встроенного языка программирования {*Maple, Mathematica (Math)*};
- библиотеки различного назначения и организации;
- стандартные процедуры вычислений и обработки информации;
- стандартные средства отладки программных средств пользователя;
- пользовательские средства различного назначения 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 *Maple* и *Mathematica*, 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 ряд полезных для него средств.

В данном контексте представляется достаточно интересным и проект по сбору *ошибок* в пакете *Maple*, которыми он изобилует подобно *Mathematica* [100]. 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, инициируемых 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 квалификации 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 модерируется, в то время как аналогичная конференция *comp.soft-sys.math.mathematica* не только модерируется, но также и цензурируется в том отношении, что фильтруются посты, с относительно резкой критикой 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 некоторое действие.

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 <" <> 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.

Наряду с механизмами типирования *формальных* аргументов оба пакета располагают 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 первая часть фрагмента.

```
> GS := proc(a::integer := 42, b::integer := 47) a + b end proc;
> GS(), GS(14, 22), GS(72); => 89, 36, 119
> GS := proc(a::integer := 42, b::integer := 47.42) a + b end proc;
> GS(), GS(14, 22), GS(72); => 89.42, 36, 119.42
> GS := proc(a::integer := 42, b::integer := 47.42 + sin(x)) a + b end proc;
> GS(), GS(14, 22), GS(72); => 89.42 + sin(x), 36, 119.42 + sin(x)
> GS := proc(a::integer := 42, b::expects(integer) := 47.42 + sin(x)) a + b end proc;
> GS(); => 89.42 + sin(x)
> GS(7.42);
Error, invalid input: GS expects its 2nd argument, b, to be of type integer, but received 7.42
In[643]:= Default[G, 1] = 64; Default[G, 2] = 69; G[x_, y_] := Module[{}, x^2 + y^2]
In[644]:= {G[47, 42], G[], G[67]}
Out[644]= {3973, 8857, 9250}
In[1537]:= Defaults[x_/:ProcQ[x], y_List/:DeleteDuplicates[Map[IntegerQ, y]] == {True},
z_List] := Module[{a = Arity[x], k = 1}, For[k, k <= a, k++, ToExpression["Default["
<> ToString[x] <> "," <> ToString[y[[k]]] <> "]" <> "=" <> ToString1[z[[k]]]]]]
In[1538]:= Defaults[GS, {1, 2}, {63 + Sin[x], 68 + Cos[y]}]
In[1539]:= GS[x_, y_] := Module[{}, x^2 + y^2 + 2011]
In[1540]:= {GS[47, 42], GS[], GS[67]}
Out[1540]= {5984, 2011 + (68 + Cos[y])^2 + (63 + Sin[x])^2, 6500 + (68 + Cos[y])^2}
In[1541]:= Defaults[GS2, {1, 2}, {63 + Sin[x], 68 + Cos[y]}]
In[1542]:= GS2[x_ Integer, y_] := Module[{}, x^2 + y^2 + 2011]
In[1543]:= {GS2[47, 42], GS2[], GS2[67]}
Out[1543]= {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 .

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 изошренных 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 использованием единого механизма *синхронизации* использования *глобальных* переменных процедур с одноименными переменными вне процедур. Этот несложный механизм описан на конкретных примерах в настоящей книге выше.

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. Итак, в противовес большей *читабельности* *Maple*-процедур наряду с лучшей процедурной технологичностью *Mathematica*-процедуры обладают большей эффективностью как по временным показателям, так и по размеру исходного кода. А в целом, *Mathematica*

располагает более развитой функциональной поддержкой программирования часто используемых средств и средств, расширяющих средства собственно самого пакета.

В контексте работы с *внешними* файлами данных, без чего немислимо 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.

```
In[2727]:= A[x_;/ StringQ[x], y_;/ If[y >= 75 && y <= 420, True, False]] :=
Module[{a, b = 75}, a = 400; 4*(Length[x]*b + y)/a]
In[2728]:= A["RANS_IAN", 300]
Out[2728]= 3

> with(MmaTranslator); => [FromMma, FromMmaNotebook, Mma, MmaToMaple]
> `A[x_;/ StringQ[x], y_;/ If[y >= 75 && y <= 420, True, False]] := Module[{a, b = 75},
a = 400; 4*(Length[x]*b + y)/a];
> FromMma(%);

A(x_ and StringQ(x),y_ and `if(75 <= y and y <= 420,true,false)) := Module([ a, b := 75], a := 400,  $\frac{4 (nops(x) b + y)}{a}$ )
```

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 *{mws, mw}*-файлы, для *Mathematica* – *nb*-файлы. Для получения доступа к таким объектам, чьи определения содержатся в таких файлах, такие файлы с документами должны быть загружены в текущий сеанс и выполнены (*вычислены*), активизировав все содержащиеся в них определения объектов и позволив обращаться к ним наравне со стандартными 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.

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, содержат 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[725]:= Log[{a, k} = {0, 1}; For[k, k <= 200, k++, a = a + If[PrimeQ[k], k, 0]]; a + 420.75]
Out[725]= 8.44414
> ln((proc(x) local a, k; a := 0; for k to 200 do a := a + `if`(type(k, prime), k, 0) end do;
      a + x end proc)(420.75)); => 8.444138511
> evalf(ln(add(`if`(type(k, prime), k, 0), k=1..200) + 420.75)); => 8.444138511
In[733]:= Log[Sum[If[PrimeQ[k], k, 0], {k, 200}] + 420.75]
Out[733]= 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 *преференции* 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* проводится, в основном, на основе их релизов 8 – 11 и 7.0 – 8.0.4.0 соответственно).

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 состоит в следующем, а именно:

– *CAS Maple*, поддерживая довольно развитый процедурный язык программирования, наилучшим образом отвечает задачам образовательного характера и, в частности, совершенствования преподавания математически-ориентированных дисциплин для университетов, освоения систем компьютерной математики, а также применения в задачах автоматизации аналитических и численных преобразований, и вычислений в относительно несложных научно-технических проектах;

– *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 рекомендацию.

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. С другой стороны, *Maple*, в свою очередь, обладает вполне определенными преимуществами при использовании циклических конструкций высокого уровня вложенности, о чем уже говорилось выше.

Здесь лишь приведем простой пример циклической конструкции, реализованной как в *Maple*, так и в *Mathematica*. Результаты говорят сами за себя – если в *Maple* требуется для выполнения 21.2 и 25.2 сек. соответственно для версий 8 и 11, то для *Mathematica* последней версии 8 для этой же конструкции требуется уже 253.9 сек., т.е. на порядок больше. Более того, с ростом глубины вложенности и диапазона переменной цикла у циклических конструкций данная разница довольно существенно растет.

<pre>> t := time(): for k1 to 10 do for k2 to 10 do for k3 to 10 do for k4 to 10 do for k5 to 10 do for k6 to 10 do for k7 to 10 do for k8 to 10 do 75 end do end do end do end do end do end do do end do end do: time() - t; => 21.156 (Maple 8) & 25.203 (Maple 11) In[1]:= 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[1]= 253.859</pre>

Между тем, из приведенного примера не только следует, что *Maple* использует более эффективные алгоритмы для реализации циклических конструкций, но и выявляет с логической точки зрения не вполне корректную ситуацию, когда время выполнения одной и той же конструкции в более младшем релизе пакета меньше, чем в старшем. Однако это не самый худший вариант, к большому сожалению, оба пакета страдают обилием неустраняемых из релиза в релиз ошибок, а также дополнением новых 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*.

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 обеими 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.

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., Гершгорн Н. Вычислительные задачи на персональном компьютере.– Киев: Изд-во Техника, 1991, 248 с.
5. Аладьев В.З., Тупало В.Г. Алгебраические вычисления на компьютере.– Москва: Изд-во Минтопэнерго, 1993, 251 с., ISBN 5-942-00456-8.
6. Аладьев В.З., Хунт Ю., Шишаков М.Л. Математика на персональном компьютере.– Гомель: Изд-во ФОРТ, 1996, 498 с.
7. Аладьев В.З., Шишаков М.Л. Введение в среду пакета *Mathematica 2.2.*– Москва: Изд-во ФилинЪ, 1997, ISBN 5-89568-004-6.
8. Аладьев В.З., Ваганов В.А. и др. Введение в среду математического пакета *Maple V.*– Минск: Изд-во IAN Press, 1998, 452 с., ISBN 14-064256-98.
9. Аладьев В.З. др. Программирование в среде математического пакета *Maple V.*– Гомель: TRG & Salcombe Press, 1999, 470 с.
10. Аладьев В., Ваганов В., Хунт Ю., Шишаков М. Рабочее место для математика.– Гомель–Таллинн: International Academy of Noosphere, 1999, 605 с.
11. Аладьев В.З., Богдэвичус М.А. Решение математических и физико-технических задач с пакетом *Maple V.*– Вильнюс: Technics Press, 1999, 686 с., ISBN 9986-05-398-6.
12. Аладьев В.З., Шишаков М.Л. АРМ математика.– М.: Изд-во БИНОМ, 2000 + CD.
13. Аладьев В., Богдэвичус М. *Maple 6:* Решение математических, статистических и инженерно-физических задач.– М.: Изд-во БИНОМ, 2001, ISBN 5-93308-085-X.
14. Aladjev V.Z., Bogdevicius M.A. *Interactive Maple: Solution of Mathematical, Engineering, Statistical and Physical Problems.*– Tallinn-Vilnius: Academy of Noosphere, 2002, CD.
15. Aladjev V.Z., Bogdevicius M.A. Use of package *Maple V* for solution of physical and engineering problems // Int. Conf. TRANSBALTICA- 99.– Vilnius: Technics Press, 1999.
16. Aladjev V.Z., Hunt U.J. Workstation for mathematicians // Int. Conf. TRANSBALTICA- 99.– Vilnius: Technics Press, April 1999.
17. Aladjev V.Z., Hunt U.J. Workstation for mathematicians // Internat. Conf. «Perfection of Mechanisms of Management», Institute of Modern Knowledge, 1999, Grodno, Byelorussia.
18. Aladjev V.Z., Shishakov M.L. Programming in Package *Maple V* // 2nd Internat. Conf. «Computer Algebra in Fundamental and Applied Researches and Education».– Minsk, 1999.
19. Aladjev V.Z., Shishakov M.L. A Workstation for mathematicians // 2nd 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. 8th 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 // 3rd Internat. Conf. «Differential Equations and Applications».- Sant-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.- Гродно: Изд-во Гродненский госуниверситет, Беларусь, 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.- Gomel: State University, Gomel, Byelorussia.
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., Луно 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. 2nd ed.- Tallinn: International Academy of Noosphere, 2004.
39. *Aladjev V.Z., Bogdevicius M.A.* Computer algebra system Maple: A new software toolbox // 4th 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, Таллинн: Международная Академия Ноосферы, 2006, ISBN 9985-9508-1-X.
 44. Аладьев В.З., Бойко В.К., Ровба Е. Программирование и разработка приложений в Maple.- Гродно: Изд-во ГрГУ, <http://www.aladjev-maple-book.narod.ru/>, 2007, 456 с.
 45. Аладьев В.З. Библиотека версии 2.2215 для пакета Maple.- www.aladjev.narod.ru, www.aladjev-maple-book.narod.ru, 2011; <http://depositfiles.com/files/iwjtv8xuz>.
 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.- Минск: Электронная книга БГУ, 2001.
 61. www.ecsocman.edu.ru/text/16089443, www.aladjev.narod.ru, 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
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 – 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, 2011; The package can be freely downloaded from 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: PaloAlto: Fultus Books, 2009, 535 p., ISBN 1-59682-137-X.
93. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Вопросы математической теории классических однородных структур.– Гомель: BELGUT, 1996, ISBN 5-063-56078-5
94. Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. Математическая теория классических однородных структур.– Таллинн-Гомель: TRG & VASCO & Salcombe Eesti Ltd., 1998
95. Aladjev V.Z., Haritonov V.N. *General Theory of Statistics*.– СА: Palo Alto: Fultus Books, 2004, 256 p., ISBN 1-59682-012-8.
96. Аладьев В.З., Харитонов В.Н. Общая теория статистики.– Palo Alto: Fultus Books, 2006, 256 с., ISBN 1-59682-086-1, Adobe Acrobat eBook (pdf), ISBN 1-59682-081-0.
97. Mangano S. *Mathematica Cookbook*.– СА: Sebastopol: O'Reilly Media, Inc., 2010, 828 p.
98. Kendrick D. et al. *Computational Economics*.– Princeton University Press, 2006, 406 p.
99. Hasbrouck J. *Empirical Market Microstructure: The institutions, economics, and econometrics of securities trading*.– Oxford University Press, 2007, 198 p., ISBN 9780195301649.
100. <http://maple.bug-list.org/> – Обновляемая база данных ошибок в системе Maple.
101. Aladjev V.Z., Vaganov V.A. *Modular Programming: Mathematica vs Maple, and vice versa*.– СА: USA, Palo Alto, Fultus Corporation, 2011, ISBN 978-1-59682-268-9, 418 p.
102. 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.
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, 3rd ed.*– Cambridge University Press, 2005, 550 p., ISBN 0521846781.
105. Sisson P. *College Algebra, 2nd 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 978-1460938386.
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. Annong Xu. *Introduction to Scientific Computing: Numerical Analysis With Mathematica*.– China Machine Press, 2010, ISBN 9787111310914.
116. Hazrat R. *Mathematica: A Problem-Centered Approach*.– Springer Press, 2010, 188 p.
117. Hastings K.J. *Introduction to Probability with Mathematica*.– CRC Press, 2010, 451 p.
118. Geveci T. *Calculus I - III*.– Cognella Press, 2011, ISBN 9781935551423, 538 p.

Перечень пользовательских процедур для пакетов Mathematica и Maple, рассматриваемых и упоминаемых в настоящей книге

Пакет Maple:

&Shift, &ma, AllMatrices, ActRemObj, ArtMod, AtomicQ, AtrRW, CALL, ChkPnt, Commonest, Composition, ComposeList, Currentdir, Daclose, Daopen, Daread, DecLevLS, DefOpt, DelSubStr, DifNestLevels, DoF, E_mail, ExpGlobals, ExpLocals, ExprOfStr, FOR_DO, Fend, FilePal, Find, FixedPointList, Flatten, FlattenFul, FoldList, Fremove, Globals, Hasfun, Index, InsDelList, ListOp, ListListGroup, ListPack, Lprot, M_Type, MapAt, MapIndexed, MapInSitu, MapInSitu1, MapleLib, MapleLib, MkDir, MkDir, MkDir, MkDir1, MkDir4, MkDir4, MlsNest, mapt, NF, Nest, NestCycles, NestList, NestWhile, Nresults, OpenLN, PadList, ParProc, ParProc1, Plib, Predecessors, Pref, Pref1, ProtectedQ, ProtectedQ1, partproc, Read1, Red_n, Remember_T, RevPath, Save2, SaveMP, SoftTab, Split, Sproc, StatLib, StatLib, StrExprQ, SymbolToList, StrOfSymbLQ, SubProcs, SubsProcQ, Subs, TestOnNCF, Timetest, Tuples, UpLib, UpLib, UpLib, WARNING, With, With6, WithM, _SL, assign67, belong, belong, cmf, com_exe1, com_exe2, `convert/proc`, dcemod, deltab, difff, elib, email, extexp, extrcalls, helpman, holdof, insert, insertN, intproc, inttt, isDir, isFile, isplabel, istablerev, lop3, map7, member1, member2, mnames, mod21, mwsname, nvalue, open2, pusers, readdata1, readm1, revtabl, save1, saveall, savem, savem1, simple1, statf, swmpat, swmpat1, tabar, tpacmod, tuples, `type/Type`, `type/boolproc`, `type/color`, `type/complex1`, `type/dir`, `type/file`, `type/file`, `type/file1`, `type/fpath`, `type/parse`, `type/path`, `type/path`, Unwith, uglocal, uplib, usertype, utfile, writedata1, wsread

Пакет Mathematica:

\$HeadProc, \$ProcName, AcNb, ActRemObj, ActiveProcess, Adrive, Affiliate, AllMatrices, Aobj, ArrayInd, Args, Args0, Args1, Arity, AssignToList, AtomicQ, Attrib, AutoLoadPack, BinaryListQ, BitGet1, BitSet1, Bits, BlockQ, Border, CALL, CatN, CDir, CallsInProc, CharacterQ, CloseAll, Closes, Contexts1, CopyDir, DeCod, DefFunc, DefFunc1, DefFunc2, Defaults, DelEl, DelSubStr, Df, Df1, Df2, DirName, DirQ, QFunction, Email, Email1, EmptyFileQ, ExpLocals, ExprQ, ExprOfStr, ExtrPackName, FileFormat1, FileOpenQ, FreeSpaceVol, FunCompose, FunctionQ, FunctionQ1, GC, GV, Gather1, Globals, HeadingQ, HowAct, Iff, InsertN, Ind, Int, Int1, IsFileOpen, IsMonotonic, IsPermutation, LeftFold, ListListGroup, ListListQ, ListOp, ListToSeq, ListToString, LoadFile, Locals, Locals1, LongestCommonSubSequence, LongestCommonSubString, Map1, Map2, Map3, Map4, Map5, Map6, MapInSitu, MapInSitu1, MapInSitu2, Mapp, MaximalPalindromicSubstring, MaxNestLevel, MdP, MemberLN, MemberQ1, MemberQ2, MemberT, MinusLis, MixCaseQ, ModLibraryPath, ModuleQ, NamesCS, NamesNbPackage, NamesNbPackage1, NamesProc, NamesProc1, NbName, Need, NestCycles, NestQL, Nobj, Npackage, Nproc, Nvalue, OP, OpenFiles, OverLap, PackNames, PalindromeQ, PartialSums, Predecessors, PredecessorsL, PredecessorsR, PrefixQ, Prev, ProcCall, ProcCalls, ProcQ, ProcQ1, ProcsAct, ProtectedQ, ProtectedQ1, PureFuncQ, Range1, Range2, Range3, ReadFullFile, RemProcOnHead, Rename, ReplaceAll1, RightFold, RhsLhs, RETURN, ScanLikeProcs, SearchDir, SearchFile, Seq, SeqDel, SeqIns, SeqQ, SeqToList, SeqUnion, SequenceQ, SetDir,

SortNL, SortNL1, SortString, Spos, StrDelEnds, StrExprQ, SymbolToList, StrOfSymbQ, StrStr, StreamsU, StringEnd, StringPosition1, StringReplace1, StringTake1, StringTake2, SubDelStr, SubProcs, SubsProcQ, SubStr, Subs, SuffixQ, SuffPref, SymbolQ, Tbl, TestArgsTypes, TestOnNCF, TestProcCalls, ToList, ToString1, Tuples1, Type, TypeActObj, TwoHandQ, UpdateContextPaths, UpdatePackages, UpdatePath, UprocQ, Uprocs, UserLib, VarExch, VarExch1, Ver, WhatObj, WhatType

Детальное описание как процедур представленного списка, так и целого ряда других пользовательских процедур для пакетов *Maple* и *Mathematica* могут быть найдены в библиотеке процедур для *Maple* [45] и в небольшом пакете для *Mathematica* [90]. При этом, следует отметить, что исходные тексты процедур обоих пакетов, приведенные в данной книге, в целом ряде случаев содержат вызовы *нестандартных* средств, которые не нашли отражения в настоящей книге, однако представлены в [45,90]. Поэтому, для детального их анализа требуется знакомство с этими средствами, по меньшей мере, на уровне справок по ним. Между тем, основной алгоритм большинства процедур книги достаточно хорошо просматривается и без знакомства с такими средствами. Тогда как реальное использование этих средств возможно лишь после инсталляции библиотеки [45] (для работы с *Maple*-процедурами) и загрузки в текущий сеанс пакета, в частности, файла «AVZ_Package.nb» с последующим его вычислением (для работы с процедурами *Mathematica* [90]). Наряду с иллюстративными целями указанные средства для обоих пакетов могут быть использованы в качестве средств, расширяющих их возможности и достаточно существенно облегчающих программирование весьма широкого круга задач, прежде всего, носящих системный характер.

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.

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 воспринимается при минимуме комментирования, в случае с *Mathematica* эта задача представляется нам несколько более сложной. В любом случае навык работы с исходным кодом в среде обоих 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 пакетов 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.

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.

Таким образом, читателю, для которого более приемлем вариант работы с *исходными* текстами программ в форме листинга, получают исходные коды процедур/функций, тогда как для читателей, для которых более привычен режим работы *непосредственно* на ПК, получают в свое распоряжение нашу библиотеку для Maple последней версии 2.2215 [45], содержащей более 850 *процедур* и *модулей* различного назначения, которые снабжены как справочной базой, так и их исходными текстами с многочисленными и разнообразными примерами применения. Библиотека располагает двумя режимами инсталляции, позволяя проводить ее инсталляцию независимо от версий BIOS базовой операционной системы DOS, так как успешность программной инсталляции зависит от конкретной системы, используя DOS-команды, тогда как второй способ позволяет инсталлировать библиотеку в *файловом* режиме путем простого копирования файлов библиотеки. Детальная инструкция по инсталляции в обоих режимах представлена в файле «ReadMe.html». Для Mathematica исходные коды более 200 процедур/функций находятся в двух файлах AVZ_Package.nb и AVZ_Package.m; при этом, первый файл с Mathematica-документом вполне можно непосредственно загружать в текущий сеанс пакета по цепочке GUI Evaluation → Evaluation Notebook и получать доступ ко всем содержащимся в нем средствам, краткую справку по ним, а также средствами пакета получать *исходный* код на экран ПК. Такой подход позволяет удовлетворить читателя с разными пристрастиями. Наш опыт проведения разного уровня мастер-классов по обоим пакетам подтверждает целесообразность такого подхода. Надеемся, что наша методика представления материала данной книги найдет понимание у читателя. На затронутый вопрос, впрочем, имеют место право быть и иные точки зрения.

АЛАДЬЕВ Виктор Захарович

Аладьев В.З. родился **14.06.1942** в г. Гродно (*Западная Беларусь*). После успешного окончания 2-й средней школы в 1959 г. поступил на 1-й курс физико-математического факультета Гродненского университета, а в 1962 г. был переведен на отделение «Математики» Тартуского университета (ЭССР). В 1966 г. он успешно закончил Тартуский госуниверситет по специальности «Математика». В 1969 г. поступил в аспирантуру Академии Наук ЭССР по специальности «Теория вероятностей и математическая статистика», которую успешно закончил в 1972 сразу по двум специальностям «Теоретическая кибернетика» и «Техническая кибернетика». В 1972 г. ему была присвоена докторская степень по математике (DSc) у проф. **Р. Белмана** (США) за работу «*Mathematical Theory of Homogeneous Structures and Their Applications*». С 1969 **Аладьев В.З.** – Президент созданной им *Таллиннской творческой группы (ТТГ)*, научные результаты которой получили международное признание, прежде всего, в области исследований по математической теории однородных структур (*Cellular Automata*). С 1972 по 1990 г.г. он занимал ответственные посты (*главный инженер, зам. директора по науке*) в ряде проектно-технологических и исследовательских организаций г. Таллинна. Деятельность **Аладьева В.** на этих постах была неоднократно отмечена наградами и премиями Совета министров СССР, ЦСУ СССР, ВГПТИ ЦСУ СССР и др.

Аладьев В. является автором более **450** научных и научно-технических работ (*включая 75 монографий, книг и сборников статей*), опубликованных в СССР, России, ФРГ, Литве, Беларуси, Эстонии, Украине, ГДР, Чехословакии, Венгрии, Японии, Болгарии, США, Голландии и Великобритании. С 1972 г. является референтом и членом редколлегии международного математического журнала «*Zentralblatt für Mathematik*» и с 1980 г. – он член **IAMM** (*International Association on Mathematical Modelling, USA*). Им создана Эстонская школа по математической теории однородных структур, чьи фундаментальные результаты получили международное признание и легли в основу *нового* раздела современной математической кибернетики – теории сложных систем.

Им введена ныне общепринятая русскоязычная терминология и получен целый ряд фундаментальных результатов по математической теории однородных структур и ее приложениям, прежде всего, в математической биологии развития. Работы **Аладьева В.З.** отражены в математических энциклопедиях как советской, так и зарубежных, целом ряде монографий, статей и докладов в журналах и на международных конференциях, часто цитируются ведущими исследователями в данной области.

Немало прикладных работ **Аладьева В.З.** относится к информатике, среди которых, в первую очередь, можно отметить широко известные книги по *системам компьютерной алгебры и компьютерной математики* (**MathCAD, Reduce, Mathematica, Maple**). Наряду с данными оригинальными изданиями им создана *большая* библиотека новых программных средств для системы **Maple**, отмеченная сетевой наградой *Smart Award* от *Smart Downloads Network* и в настоящее время достаточно широко используемая в СНГ и за его пределами. Многие его работы в данном направлении представлены в интернете для свободного доступа и включены в списки обязательной либо дополнительной ли-

тературы в программы университетов. Довольно широко известны его *мастер-классы* по математическим пакетам, даваемые в университетах СНГ и в других странах.

В 1993 г. **Аладьев В.З.** по результатам своей многолетней научной активности избран членом рабочей группы *IFIP (International Federation for Information Processing, USA)* по математической теории *однородных структур* и ее приложениям. На целом ряде международных научных форумов по математике и кибернетике **Аладьев В.З.** участвовал в качестве члена оргкомитета или приглашенного докладчика. В апреле 1994 **Аладьев В.З.** по совокупности научных работ в области кибернетики избран академиком Российской Академии Космонавтики по отделению «*Фундаментальных исследований*», а в сентябре 1994 г. он избирается академиком Российской Академии Ноосферы по отделению «*Информатики*». В сентябре 1995 г. **Аладьев В.З.** избирается действительным членом Российской Академии Естественных Наук (**РАЕН**) по отделению «*Ноосферные знания и технологии*», а в 1998 г. – академиком Российской Экологической Академии.

В ноябре 1997 г. **Аладьев В.З.** избран академик-секретарем *Балтийского отделения* Российской Академии Ноосферы, объединяющего ученых и специалистов стран Балтии и Беларуси, работающих в области комплекса научных дисциплин, входящих в проблематику ноосферы и смежных с нею областей научной деятельности, включая теоретические и прикладные вопросы по проблематике *однородных структур*. В результате реорганизации Российской Академии Ноосферы в Международную, в декабре 1998 г. **Аладьев В.З.** избирается ее Первым вице-президентом. В конце 1999 г. **Аладьев В.З.** по совокупности научных работ в области кибернетики и информатики избирается иностранным членом **РАЕН** по отделению «*Информатики и кибернетики*».

Наиболее значительные научные результаты **Аладьева В.З.** относятся к математической теории однородных структур и ее приложениям. Сфера его научных интересов включает математику, информатику, кибернетику, вычислительные науки, физику, космонавтику и другие естественно-научные направления.

БОЙКО Валерий Константинович

Бойко В.К. родился **10.09.1950** в городе Гродно (Беларусь). В 1967 г. закончил среднюю школу и поступил на физико-математический факультет Гродненского государственного педагогического института им. Я. Купалы, который с отличием окончил в 1971 г. В период 1971–1976 учился в аспирантуре и служил в армии. В 1978 г. защитил кандидатскую диссертацию на тему «*Краевая задача с управлением для системы интегро-дифференциальных уравнений Фредгольма*». В период 1976–1983 работал ассистентом, старшим преподавателем, доцентом кафедры дифференциальных уравнений Гродненского государственного университета им. Я. Купалы.

В 1983 г. **Бойко В.К.** закончил курсы французского языка и с 1984 г. по 1987 г. по направлению Министерства Образования СССР работал доцентом кафедры математики строительного института в городе Эль-Аснам (Алжир). После возвращения **Бойко В.К.** продолжает работать доцентом на кафедре дифференциальных уравнений Гродненского государственного университета им. Я. Купалы. Он активно занимается научной

работой, является ответственным исполнителем ряда научных тем, работает заместителем декана по научной работе. С 1998 г. по 2007 г. являлся деканом факультета математики и информатики Гродненского государственного университета им. Я. Купалы. В настоящее время заведует кафедрой высшей математики в этом же университете.

Основные научные результаты **Бойко В.К.** относятся к задачам *управления* для различных классов *линейных* систем. Им получено решение задачи о *минимальном* числе входов, предложены алгоритмы построения оптимальных управлений и фильтров в различных задачах, рассмотрен ряд других интересных свойств управляемых систем. Полученные им результаты используются как в теоретических исследованиях, так и при решении конкретных прикладных задач для управляемых объектов.

Бойко В.К. является активным сторонником развития математического образования в Гродненской области и Беларуси в целом. В течение многих лет он возглавляет жюри областных математических олимпиад. По его инициативе при факультете с 2000 года возобновила свою работу *Школа точных наук* для талантливых детей. Им ведется большая работа по внедрению современных информационных технологий в научные исследования, учебный процесс и другие сферы деятельности университета. Как признание заслуг факультета в этой области в 2005 математический факультет университета был переименован в факультет *математики и информатики*.

Бойко В.К. является автором более **80** научных и научно-методических работ (*включая 3 монографии и 5 методических указаний*), опубликованных в СССР, Беларуси, Украине, России и Польше, редактором ряда научных изданий. По результатам многолетней научно-педагогической деятельности и активного участия в академической деятельности Международной Академии Ноосферы (**МАНУР**) **Бойко В.К.** в мае 2007 г. избран членом-корреспондентом **МАНУР**.

РОВБА Евгений Алексеевич

Ровба Е.А. родился **01.04.1949** в деревне *Андрошевицина* Лидского района Гродненской области. В 1971 г. с отличием окончил *Белорусский государственный университет (БГУ)* по специальности «*математика*». В 1975 защитил кандидатскую диссертацию на тему «*Некоторые вопросы рациональной аппроксимации*». В период с 1974 г. по 1979 г. работал старшим преподавателем, доцентом, затем заведующим кафедрой математического факультета Гомельского государственного университета. В 1979 г. **Ровба Е.А.** переехал на работу в Гродненский государственный университет им. Я. Купалы, в котором работал сначала доцентом, а с 1993 заведовал кафедрой теории функций, функционального анализа, вероятностей и прикладной математики. В 1999 г. защитил докторскую диссертацию на тему «*Интерполяция и ряды Фурье в рациональной аппроксимации*», которая в 2001 г. легла в основу одноименной монографии. Работал заместителем декана и деканом математического факультета, проректором по учебной работе **ГрГУ**.

С 1986 по 1988 г. – консультант Министерства образования Республики Куба. С 1996 г. по 1998 г. работал деканом Гродненского филиала специального факультета бизнеса и информационных технологий **БГУ**. С 1998 по 2000 г.г. – ректор Института повыше-

ния квалификации и переподготовки *руководящих* работников Гродненского университета. С 2000 г. **Ровба Е.А.** является профессором и первым проректором университета, в 2005 г. назначен ректором *ГрГУ*, в качестве которого работает и по сей день.

Основные научные исследования **Е.А. Ровба** относятся к теории рациональных приближений функций. Им построены интерполяционные и интегральные *рациональные* операторы, обладающие *классическими* с точки зрения конструктивной теории функций аппроксимационными характеристиками. Им введены обобщения *квадратурных* формул Гаусса на базе *ортogonalных* систем *рациональных* функций и интерполяции. Получены оценки уклонений рациональных операторов Фурье и Валле-Пуссена на классах функций, имеющих *дробные* производные ограниченной вариации. Впервые интерполирование по специальным системам узлов использовано для рациональной аппроксимации различных классов функций, отображающих ее особенности. Полученные методы и результаты используются в теоретических исследованиях в теории аппроксимации функций, а также при решении конкретных задач вычислительной математики, в конструктивной теории функций и численных методах.

Созидательность организационной деятельности **Е.А. Ровба** оказала весьма большое влияние на развитие образовательной среды Гродненской области (*Беларусь*). **Ровба Е.** инициировал создание в Гродно полнокровной самофинансируемой образовательной структуры в сфере последишломного образования Гродненского филиала Специального факультета бизнеса и информационных технологий. Полученный им в данном направлении опыт был успешен и стал основой для создания новой инновационной образовательной структуры – ныне Институт повышения квалификации и переподготовки кадров *ГрГУ*. Уже будучи первым проректором университета, **Е.А. Ровба** способствовал открытию и становлению лицеев, гимназий, созданию новых учебно-методических объединений в Гродненской области.

Е.А. Ровба – автор более 120 научных и научно-методических работ (*включая 4 монографии и 4 учебных пособия*), опубликованных в СССР, Беларуси, Болгарии, России, Кубе, Украине и Великобритании. **Е.А. Ровба** вносит серьезный вклад в воспитание и подготовку научных кадров. Им подготовлено 2 кандидата физико-математических наук.

По итогам многолетней научно-педагогической деятельности и активного участия в академической деятельности *Международной академии ноосферы (МАНУР)* **Е.А. Ровба** в мае 2007 г. избран действительным членом (*академиком*) *МАНУР*. С марта 2000 избран академиком Белорусской Инженерной Академии. С 2009 г. он – Почетный профессор Тульского государственного педагогического университета, а с 2010 – Почетный профессор национального педагогического университета (*Киев*). **Е. Ровба** – председатель редакционного совета журнала «*Веснік Гродзенскага Дзяржаўнага ўніверсітэта імя Янкі Купалы*», член редколлегий журнала «*Вестник БГУ*» Серия I, научно-педагогического журнала «*Европейские педагогические студии*» (*национальный педагогический университет имени М. Драгоманова, Киев*). В 2008 г. **Е.А. Ровба** награжден высокой государственной наградой – медалью Франциска Скорины, в 2009 – нагрудным знаком Министерства образования Беларуси «*Отличник образования*», в 2010 г. **Е. Ровба** был удостоен звания «*Почетный гражданин Гродно*».