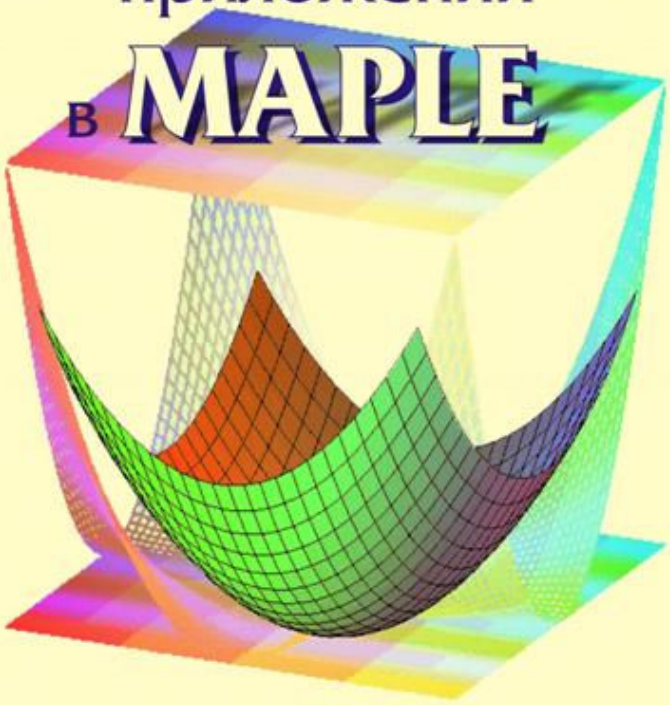
IAN
International Academy of Noosphere, Tallinn Branch

В.З. Аладьев, В.К. Бойко, Е.А. Ровба

Программирование и разработка приложений в MAPLE



YKSUG
Yanka Kupala State University of Grodno

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 И РАЗРАБОТКА ПРИЛОЖЕНИЙ В MAPLE

М о н о г р а ф и я

Гродно – Таллинн 2007

Аладьев В.З. Программирование и разработка приложений в *Maple*: монография / В.З. Аладьев, В.К. Бойко, Е.А. Ровба.– Гродно: ГрГУ; Таллинн: Межд. Акад. Ноосферы, Балт. отд.– 2007, 458 с., ISBN 978-985-417-891-2 (*ГрГУ*), ISBN 978-9985-9508-2-1 (*МАНУР*)

Монография вводит в программную среду известного математического пакета ***Maple***, представляющего собой одну из наиболее развитых *современных* систем компьютерной алгебры. Данное исследование – достаточно детальное введение в среду встроенного ***Maple***-языка программирования, позволяющего пользователю не только четко представить все возможности пакета, но и разрабатывать в его среде сложные приложения для многих разделов техники, математики, физики, химии и других естественнонаучных дисциплин, для решения которых пакет не имеет стандартных средств. При этом, язык ***Maple*** может оказаться весьма эффективным средством в системе преподавания указанных дисциплин. Именно в данном направлении он может получить свое не меньшее признание, чем у многочисленных исследователей естественнонаучных дисциплин, использующих математические методы.

Представленный в монографии материал покрывает практически все основные функциональные возможности ***Maple***-языка с иллюстрацией целого ряда как их наиболее массовых приложений при решении широкого круга математических задач, так и наиболее интересных *особенностей*, позволяющих использовать их нестандартным образом, расширяя тем самым потенциал встроенного ***Maple***-языка; целый ряд рассмотренных приемов может оказаться полезным при формировании эффективной *концепции* программирования в его среде.

Все это делает книгу полезным пособием по пакету *Maple* как для студентов, так и для профессионалов из различных фундаментальных и прикладных областей современного естествознания. В свете вышеизложенного ***Maple*** можно рассматривать в качестве достаточно хорошо сбалансированной интегрированной среды для выполнения разнообразных вычислений, работы с графическими объектами и для программирования на высокоуровневом процедурном языке, прежде всего, задач, носящих математический характер с акцентом на символьных (*алгебраических*) вычислениях.

Монография является *одним* из немногих изданий по программированию в среде пакета ***Maple***, что и определяет ее место *среди* исследований по программным средствам для ПК, использующих операционную среду *Windows*. Вместе с тем мобильность пакета позволяет использовать его многими другими популярными платформами. Монография рассчитана на достаточно широкий круг специалистов, использующих в своей профессиональной деятельности ПК для решения задач математического характера, а также на студентов и учащихся, изучающих курс *«Основы информатики и вычислительной техники»* физико-математических и других естественнонаучных специальностей соответствующих университетов и колледжей.

Настоящая монография представляет собой авторский оригинал-макет мастер-классов по программированию в среде *Maple*, проведенных в ряде университетов Прибалтики и СНГ. Данное издание может быть полезно пользователям *Maple*, чьи задачи требуют непосредственного программирования приложений в среде пакета.

Рецензент: доктор физико-математических наук, профессор *А.Н. Дудин*

ISBN 978-985-417-891-2 (*ГрГУ*)
ISBN 978-9985-9508-2-1 (*МАНУР*)

© Аладьев В.З. (*Таллинн*)
Бойко В.К., Ровба Е.А. (*Гродно*), 2007

Maple V, Maple 6, Maple 7, Maple 8, Maple 9, Maple 10 – торговые марки MapleSoft Inc.

Содержание

Предисловие 5
Глава 1. Базовые сведения по *Maple*-языку пакета 21
1.1. Базовые элементы *Maple*-языка пакета 23
1.2. Идентификаторы, предложения присвоения и выделения *Maple*-языка 30
1.3. Средства *Maple*-языка для определения свойств переменных 42
1.4. Типы числовых и символьных данных *Maple*-языка пакета 45
1.5. Базовые типы структур данных *Maple*-языка 51
1.6. Средства тестирования типов данных, структур данных и выражений 80
1.7. Конвертация *Maple*-выражений из одного типа в другой 89
1.8. Функции математической логики и средства тестирования пакета 93
Глава 2. Базовые управляющие структуры *Maple*-языка пакета 106
2.1. Предварительные сведения общего характера 106
2.2. Управляющие структуры ветвления *Maple*-языка (if-*предложение*) 108
2.3. Циклические управляющие структуры *Maple*-языка (while_do-*предложение*) 111
2.4. Специальные типы циклических управляющих структур *Maple*-языка 115
Глава 3. Организация механизма процедур в *Maple*-языке пакета 118
3.1. Определения процедур в *Maple*-языке и их типы 119
3.2. Формальные и фактические аргументы *Maple*-процедуры 126
3.3. Локальные и глобальные переменные *Maple*-процедуры 131
3.4. Определяющие параметры и описания *Maple*-процедур 138
3.5. Механизмы возврата *Maple*-процедурой результата ее вызова 150
3.6. Средства обработки ошибочных и особых ситуаций в *Maple*-языке 156
3.7. Расширенные средства *Maple*-языка для работы с процедурами 167
3.8. Расширение функциональных средств *Maple*-языка пакета 176
3.9. Иллюстративные примеры оформления *Maple*-процедур 184
3.10. Элементы отладки *Maple*-процедур и функций 205
Глава 4. Организация программных модулей в *Maple*-языке 213
4.1. Вводная часть 213
4.2. Организация программных модулей *Maple*-языка 216
4.3. Сохранение процедур и программных модулей в файлах 227
Глава 5. Средства *Maple*-языка для работы с данными и структурами строчного, символьного, списочного, множественного и табличного типов 235
5.1. Средства работы *Maple*-языка с выражениями строчного и символьного типов 235
5.2. Средства работы *Maple*-языка с множествами, списками и таблицами 249
5.3. Алгебраические правила подстановок для символьных вычислений 265
5.4. Средства *Maple*-языка для обработки алгебраических выражений 273
5.5. Управление форматом вывода результатов вычисления выражений 305
Глава 6. Средства ввода/вывода *Maple*-языка 311
6.1. Средства *Maple*-языка для работы с внутренними файлами пакета 311
6.2. Средства *Maple*-языка для работы с внешними файлами данных 321

6.2.1. Открытие, закрытие и удаление внешних файлов любого типа 327
6.2.2. Средства обработки особой ситуации «*конец файла*» доступа к файлам 331
6.2.3. Обработка особых и ошибочных ситуаций процедур доступа к файлам 333
6.3. Базовые средства *Maple*-языка для обеспечения доступа к внешним файлам данных *TEXT*-типа 335
6.3.1. Базовые средства доступа к файлам данных на уровне *Maple*-выражений 340
6.4. Средства *Maple*-языка для обеспечения доступа к внешним файлам данных BINARY-типа 343
6.5. Обеспечение форматированного доступа к внешним файлам данных 345
Глава 7. Графические средства *Maple*-языка пакета 352
7.1. Графическая интерпретация алгебраических выражений и уравнений в среде *Maple*-языка 352
7.2. Двухмерное представление функциональных зависимостей и данных в среде *Maple*-языка 354
7.3. Трехмерное представление функциональных зависимостей и данных в среде *Maple*-языка 373
7.4. Создание графических объектов на основе базовых примитивов 383
Глава 8. Создание и работа с библиотеками пользователя 392
8.1. Классический способ создания *Maple*-библиотек 393
8.2. Специальные способы создания библиотек пользователя в среде *Maple* 415
8.3. Создание пакетных модулей пользователя 426
8.4. Статистический анализ *Maple*-библиотек 436
9. Средства *Maple*, не рассматриваемые в настоящей книге, но полезные для приложений 444
Заключение 443
Литература 448
Справка по авторам 453

Предисловие

Системы компьютерной алгебры (**СКА**) находят все более широкое применение во многих областях науки таких как математика, физика, химия, информатика и т.д., техники, технологии, образовании и т.д. **СКА** типа ***Maple, Mathematica, MuPAD, Macsyma, Reduce, Axiom*** и ***Magma*** становятся все более популярными для решения задач преподавания математически ориентированных дисциплин, как в научных исследованиях, так и в промышленности. Данные системы являются мощными инструментами для ученых, инженеров и педагогов. Исследования на основе **СКА**-технологии, как правило, 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 символьных (*алгебраических*) вычислений и обработки на компьютерах, так и на *создании* языков программирования и программной *среды* для реализации подобных алгоритмов и базирующихся на них проблем различного назначения.

В серии наших работ [1-20,22-33,39,41-46,47,49,50,91,103] довольно детально рассмотрены такие математические пакеты как ***Maple, Reduce, MathCAD, Mathematica***. При этом, *особое* внимание нами уделелялось особенностям каждого из пакетов, его преимуществам, а также недостаткам, эффективным приемам и методам программирования в его среде, созданию набора средств, расширяющих его возможности, а также выработке системы предложений по его *дальнейшему* развитию. Наш опыт апробации и использования четырех математических пакетов ***Mathematica, Reduce, Maple, MathCAD*** в различных математических и физических приложениях позволяет нам рассматривать *именно* пакеты ***Maple*** и ***Mathematica*** в качестве бесспорных лидеров (*на основе специального обобщенного индекса*) среди всех известных на сегодня современных **СКА**. При этом, мы предпочитаем именно пакет ***Maple*** (*несмотря на все его недостатки и недоработки*) из-за целого ряда преимуществ, среди которых особо следует выделить такие как развитые графические средства, достаточно эффективные средства решения систем дифференциальных уравнений, средства создания графических интерфейсов пользователя, мощная библиотека математических функций, большой набор сопутствующих пакетных модулей для различных приложений, современный встроенный язык программирования (**4GL**) *интерпретирующего* типа, интерфейс с рядом других *Windows*-приложений, перспективная концептуальная поддержка.

Исследователи используют пакет ***Maple*** как важный инструмент при решении задач, связанных с их исследованиями. Пакет идеален (*по нынешним понятиям*) для формулировки, решения и исследования различных математических моделей. Его *алгебраические* средства существенно расширяют диапазон проблем, которые могут быть решены на *качественном* уровне. Педагоги в средних школах, колледжах и университетах обновляют *традиционные* учебные планы, вводя задачи и упражнения, которые используют диалоговую математику и физику ***Maple***. Тогда как студенты могут сконцентрироваться на важных *концепциях*, а не на *утомительных* алгебраических вычислениях и преобразованиях. Наконец, инженеры и специалисты в промышленности используют ***Maple*** как эффективный инструмент, заменяющий много традиционных ресурсов типа справочников, калькуляторов, редакторов, крупноформатных таблиц и языков программирования. Эти пользователи легко решают весьма широкий диапазон математически ориентированных задач, разрабатывая проекты и объединяя результаты (*как числовые, так и графические*) их вычислений в профессиональные отчеты довольно высокого качества.

Между тем, наш эксплуатационный опыт в *течение* **1997–2006** г.г. с пакетом ***Maple*** релизов **4 – 10** позволил нам не только оценить его преимущества по сравнению с другими подобными пакетами, но также выявил ряд ошибок и недостатков, устраненных нами. Кроме того, пакет ***Maple*** не поддерживал ряд достаточно важных процедур обработки информации, *алгебраических* и *численных* вычислений, включая средства доступа к файлам данных. Ввиду сказанного, в процессе работы с пакетом ***Maple*** мы создали достаточно много эффективного программного обеспечения (*процедуры и программные модули*), целым рядом характеристик расширяющих базовые и по выбору возможности пакета. Данное программное обеспечение было организовано в виде *Библиотеки*, которая является структурно подобной главной библиотеке ***Maple*** и обеспечена развитой справочной системой, аналогичной подобной системе пакета ***Maple*** и весьма органично с ней связанной. Комментированное *описание* данной *Библиотеки* представлено в нашей книге [103]. К ней же прилагается данная *Библиотека* версии **2.1810,** тогда как обновленную версию данной *Библиотеки* можно бесплатно загружать с любого из адресов, указанных в [109]. Демонстрационная же версия *Библиотеки* находится по адресу [108].

При этом, программные средства, составляющие *Библиотеку*, в своем большинстве имеют дело именно с базовой средой ***Maple,*** что *пролонгирует* их актуальность как на текущие релизы, начиная с *шестого*, так и на *последующие* релизы пакета. В этой связи здесь уместно обратить внимание на один весьма существенный момент. При достаточно частом объявлении о новой продукции ***MapleSoft,*** между тем, уделяет недостаточно внимания устранению имеющихся ошибок и дефектов, переходящих от релиза к релизу. Некоторые из них являются достаточно существенными. Мы отмечали данное обстоятельство в наших книгах неоднократно, этому вопросу посвящен целый ряд замечаний и членов **MUG** (*Maple Users Group*). Более того, расширению инструментальных средств основной среды пакета также уделяется недостаточное внимание, что *особенно* заметно в режиме продвинутого программирования в его среде. Представленная в [103] *Библиотека* содержит расширения инструментальных средств, прежде всего, *базовой* среды пакета, что пролонгирует их актуальность и на *последующие* релизы пакета, а также весьма существенно упрощает программирование целого ряда задач в его среде и обеспечивает более высокий уровень совместимости релизов **6 - 10**. Выявленная нами несовместимость пакета как на уровне релизов, так и на уровне базовых операционных платформ ***– Windows 98SE*** и ниже, с одной стороны, и ***Windows ME/2000/XP*** и выше, с другой стороны, потребовала решения проблемы совместимости и для средств нашей *Библиотеки* относительно релизов **6 - 10**.

В заключение данной *преамбулы* вкратце изложим (*адресуясь, прежде всего, к нашим достаточно многочисленным читателям как настоящим, так и будущим*) наше *личное* мнение по сравнительной оценке пакетов ***Maple*** и ***Mathematica***. Как *один*, так и *другой* пакеты изобилуют многочисленными ошибками (*в целом ряде случаев недопустимыми для систем подобного рода*), устранению которых разработчиками как ***MapleSoft Inc.,*** так и ***Wolfram Research*** уделяется сравнительно небольшое внимание. Из коммерческих соображений часто весьма необоснованно выпускаются новые релизы, сохраняющие старые ошибки и привнося в ряде случаев как новые ошибки, так и различного рода *экзотические* излишества. Данный вопрос неоднократно поднимался как в наших изданиях, так и перед разработчиками. Однако, если разработчики ***Maple*** в режиме открытого диалога с пользователями в какой-то мере пытаются решить данную проблему, то ***Wolfram Research*** достаточно болезненно воспринимает любую (*в подавляющем большинстве обоснованную*) критику в свой адрес. При этом, ***Wolfram Research*** ведет весьма агрессивную маркетинговую политику, не вполне *адекватную* качеству ее продукции. Именно это, прежде всего, объясняет ее временные количественные преимущества, которые достаточно быст-

ро уменьшаются. Сравнивая отклики пользователей пакетов ***Maple*** и ***Mathematica***, а также в свете нашего многолетнего опыта работы с обоими пакетами, можно вполне *однозначно* констатировать, что *вторые* при использовании пакета имеют значительно больше проблем.

Из нашего опыта достаточно глубокого использования и апробирования обоих пакетов отметим, что ***Maple –*** существенно более дружелюбная и открытая система, использующая достаточно развитый встроенный язык **4**-го поколения интерпретирующего типа, что упрощает освоение пакета пользователю, имеющему опыт современного программирования. Тогда как пакет ***Mathematica*** имеет несколько архаичный и не столь изящный язык, в целом ряде отношений отличный от современных языков программирования. Наконец, ***Maple*** имеет по ряду показателей *более* развитые *инструментальные* средства (*например*, *для решения диффуравнений в частных производных*, *предоставления используемого алгоритма решения задачи*, *настройки графического интерфейса пользователя на конкретные приложения и др.*), а также весьма широкий спектр бесплатных приложений во многих фундаментальных и прикладных областях современного естествознания.

Пакет ***Maple*** воплощает *новейшую* технологию *символьных* вычислений, *числовых* вычислений с произвольной точностью, наличие инновационных ***Web-****компонент*, расширяемой технологии пользовательского интерфейса (***Maplets***), и весьма развитых математических алгоритмов для решения сложных математических задач. В настоящее время пакет использует более **5** *миллионов* студентов, ученых, исследователей и специалистов из различных областей. Практически каждый ведущий университет и научно-исследовательский институт в мире, включая такие, как *MIT*, *Cambridge*, *Stanford*, *Oxford*, *Waterloo* и др., используют пакет для учебных и исследовательских целей. В промышленных целях пакет используется такими ведущими корпорациями как *Boeing*, *Bosch*, *Canon*, *Motorola*, *NASA*, *Toyota*, *Hewlett Packard*, *Sun, Microsystems*, *Ford*, *General Electric*, *Daimler-Chrysler* и др.

Резюмируя сказанное (*более детальный сравнительный анализ обоих пакетов можно найти в серии наших работ* [1-20,22-33,39,41-46,47,49,50,91,103]), начинающему пользователю систем компьютерной алгебры *рекомендуем* все же пакет ***Maple,*** как наиболее *перспективное* средство в данной области компьютерной математики. Этому существенно способствует и творческий *альянс* ***MapleSoft*** с всемирно известным разработчиком математического **ПО –** ***NAG Ltd***. *И* это при том, что *последний* имеет и свою довольно приличную **СКА – *AXIOM,*** являющуюся на сегодня *лидером* среди **СКА** на *европейском* уровне. При этом, ***Maple*** постоянно *отвоевывает* позиции у ***Mathematica*** и начинает доминировать в образовании, что весьма существенно с ориентацией на перспективу; используемая ***Maple-***идеология занимает все более существенное место при создании различных электронных материалов математического характера.

Вместе с тем, современное развитие пакета ***Maple*** вызывает и ряд серьезных опасений, которые в общих чертах можно определить следующим образом. Качество любого программного обеспечения определяется в соответствии с большим количеством характеристик, среди которых можно отметить такие существенные как: (**1**) *совместимость* программных средств «*снизу-вверх*», (**2**) *устойчивость функционирования* относительно операционных платформ, наряду с качественной поддержкой и сопровождением, и т. д. Данным критериям последние релизы пакета ***Maple,*** начиная с **7**-го, удовлетворяют все меньше и меньше, а именно.

Достаточно существенные ошибки и недоработки (*многие из них неоднократно отражались в наших книгах и статьях*, *а также в ряде других источниках*, *включая многочисленные форумы по* ***Maple***) *переходят* от релиза к релизу. Отсутствует совместимость релизов пакета ***Maple*** «*снизу-вверх*». О несовместимости релизов ***Maple*** мы неоднократно отмечали

в своих книгах и статьях. Кое-что для усовершенствования совместимости нами было сделано (*в частности, посредством нашей Библиотеки, представленной в* [103,109]), однако не все. Тем временем, для ***Maple*** релизов **9** и **10** была обнаружена несовместимость уже среди их клонов. Как хорошо известно, ***Maple* 9** и **10** поддерживают два режима **– *классический*** (*например, для* ***Maple 9*** *ядро "**cwMaple9.exe**"и для* ***Maple 10*** *ядро "**cwMaple.exe**"*) и ***стандартный*** (*например, для* ***Maple 9*** *ядро `**Maplew9.exe**` и для* ***Maple 10*** *ядро* ***Maplew.exe***). Оказывается, что данные клоны несовместимы даже на уровне встроенных функций.

В частности, если в *классическом* режиме встроенная функция ***system*** выполняется корректно, то в *стандартном* режиме, возвращая *код завершения* **0,** она некорректно выполняет некоторые команды (*программы*) **MS DOS**. По этой причине процедуры нашей *Библиотеки*, использующие данную функцию и отлаженные в ***Maple*** релизов **8** и ниже, а также в *классическом* режиме ***Maple* 9-10,** в *стандартном* режиме ***Maple* 9-10** выполняются некорректно, часто вызывая *непредсказуемые* ошибочные ситуации. В целях устранения подобных ситуаций нами была определена процедура ***System,*** заменяющая указанную стандартную функцию ***system*** и устраняющая ее основные недостатки [103]. Естественно, подобные нарушения требований к качественному программному обеспечению не допустимы для программных средств подобного типа и могут вести к нежелательным для ***Maple*** последствиям**.** Более того, нам кажется, что их действие уже начинает сказываться.

Ввиду сказанного, упомянутая наша *Библиотека* корректно работает с ***Maple*** релизов **6 - 8** и ***Maple* 9 - 10** (*классический режим*), тогда как для ***Maple* 9 - 10** (*стандартный режим*) некоторые библитечные средства, использующие стандартную функцию ***system,*** будут выпоняться некорректно или вызывать ошибочные ситуации, в ряде случаев непредсказуемые. В этой связи заинтересованный читатель в качестве достаточно полезного упражнения имеет хорошую возможность использовать процедуру ***System*** для *обновления* упомянутых процедур *Библиотеки* на предмет расширения сферы их применимости и на стандартный режим ***Maple.*** В целях большей информативности приведем краткую характеристику нашей *Библиотеки* [103,108,109].

Характеристика нашей библиотеки программных средств*.** Упоминаемая здесь *Библиотека* расширяет диапазон и эффективность *использования* пакета ***Maple на платформе ***Windows*** благодаря содержащимся в ней средствам в трех основных направлениях: (**1**) *устранение ряда основных дефектов и недостатков*, (**2**) *расширение возможностей ряда стандартных средств*, и (**3**) *пополнение пакета новыми средствами, расширяющими возможности его программной среды*, включая средства, повышающие уровень совместимости релизов **6 - 10** пакета, о которой говорилось выше. Основное внимание было уделено дополнительным средствам, созданным нами в процессе использования пакета ***Maple*** релизов **4-10,** которые по целому ряду параметров существенно расширяют возможности пакета и облегчают работу с ним. Значительное внимание уделено также средствам, обеспечивающим повышение уровня совместимости пакета релизов **6-10**. Большой и всесторонний опыт использования данного программного обеспечения подтвердил его высокие эксплуатационные характеристики при *использовании* пакета ***Maple*** в многочисленных приложениях, потребовавших не только стандартных средств, но и программирования своих собственных, ориентированных на конкретные приложения.

Со всей *определенностью* следует констатировать, что серия наших книг по ***Maple*** [29-33, 39,41-46,91,103], представляющая разработанные нами средства и содержащая предложения по дальнейшему развитию пакета, в значительной степени стимулировала появление таких *полезных* приложений как пакетные модули ***ListTools, FileTools, StringTools*** и ***LibraryTools*.** Между тем, и в свете данных *приложений* средства нашей *Библиотеки* су-

щественно расширяют возможности пакета, во многих случаях *перекрывая* средства указанных пакетных модулей. Текущая версия *Библиотеки* содержит набор средств (*более* ***730*** *процедур и программных модулей*), ориентируемых на следующие основные виды обработки информации и вычисления [103,108,109]:

1. Программные средства общего назначения
2. Программные средства для работы с процедурными и модульными объектами
3. Программные средства для работы с числовыми выражениями
4. Программныее средства для работы со строчными и символьными выражениями
5. Программные средства для работы со списками, множествами и таблицами
6. Программное обеспечение поддержки структур данных специального типа
7. Программное обеспечение для по-битной обработки информации
8. Программные средства, расширяющие графические возможности пакета
9. Расширение и исправление стандартного программного обеспечения ***Maple***
10. Программное обеспечение для работы с файлами данных
10.1. Программное обеспечение общего назначения
10.2. Программное обеспечение для работы с текстовыми файлами
10.3. Программное обеспечение для работы с бинарными файлами
10.4. Программное обеспечение для работы с файлами ***Maple***
10.5. Специальное программное обеспечение для работы с файлами данных
11. Программное обеспечение для решения задач математического анализа
12. Программное обеспечение для решения задач линейной алгебры
12.1. Программное обеспечение общего назначения
12.2. Программное обеспечение для работы с ***rtable***-объектами
13. Программное обеспечение для решения задач простой статистики
13.1. Программное обеспечение для решения задач описательной статистики
13.2. Программное обеспечение для решения задач регрессионного анализа
13.3. Программное обеспечение для проверки статистических гипотез
13.4. Элементы анализа временных (*динамических*) и вариационных рядов
14. Программное обеспечение для работы с библиотеками пользователя

Основные новации нашей *Библиотеки* с привязкой к вышеперечисленным разделам, тематически классифицирующим средства *Библиотеки*, кратко охарактеризованы в Предисловии к нашей книге [103] и на *web*-страницах ***www.aladjev.narod.ru/MapleBook.htm*** и ***www.exponenta.ru/educat/news/aladjev/book2.asp.*** Исходя же из нашего многолетнего опыта использования пакета ***Maple*** релизов **4 - 10** и опыта наших коллег из университетов и академических институтов России, Эстонии, Белоруссии, Литвы, Латвии, Украины, а также ряда других стран, следует отметить, что многие из средств (*или их аналоги*) нашей *Библиотеки* весьма целесообразно включить в стандартные поставки последующих релизов пакета ***Maple.*** Соответствующие предложения были нами представлены разработчикам пакета. При этом, можно констатировать, что *ряд* наших книг по ***Maple***-проблематике, которые представляют средства, разработанные нами, и содержат полезные рекомендации по дальнейшему развитию пакета, стимулировали появление модулей ***FileTools, ListTools, LibraryTools*** и ***StringTools***. Однако, в этом отношении средства, представленные нами, существенно расширяют возможности пакета, во многих случаях превышая таковые из указанных пакетных модулей. В настоящее время они доступны пользователям ***Maple*** в виде предлагаемой *Библиотеки*, функционирующей на платформах ***Windows*** и поддерживающей релизы **6-10** пакета. Данная *Библиотека* прилагается к нашей книге [103], а также может быть получена по адресу [109]. Средства *Библиотеки* в целом ряде случаев позволяют существенно упрощать программирование раз-

личных прикладных задач в среде пакета ***Maple*** релизов **6 - 10**. Настоящая *Библиотека* была отмечена в **2004** г. наградой ***Smart Award*** от *Smart DownLoads Network*.

Программные средства, предоставляемые данной *Библиотекой*, снимают *целый* ряд вопросов, возникших в дискуссиях членов *группы пользователей* ***Maple*** (**MUG**) на целом ряде форумов по ***Maple,*** и существенно расширяют функциональные возможности пакета, облегчая его использование и расширяя сферу приложений. *Библиотека* предназначена для достаточно широкой аудитории ученых, специалистов, преподавателей, аспирантов и студентов естественно научных специальностей, которые в своей профессиональной работе используют пакет ***Maple*** релизов **6 - 10** на платформе ***Windows.*** *Библиотека* содержит оптимально разработанное, интуитивное программное обеспечение (*набор процедур и программных модулей*), которое достаточно хорошо дополняет уже *доступное* программное обеспечение пакета с ориентацией на самый широкий круг пользователей, в целом ряде случаев расширяя сферу применения пакета и его эффективность.

Библиотека структурно подобна главной библиотеке ***Maple,*** снабжена развитой справочной системой по средствам, расположенным в ней, и логически связана с главной библиотекой пакета, обеспечивая доступ к средствам, содержащимся в ней, подобно стандартным средствам пакета. Простое руководство описывает установку *Библиотеки* при наличии на компьютере с платформой ***Windows*** инсталлированного пакета ***Maple*** релизов **6, 7, 8, 9, 9.5** и/или **10**. Для полной установки данной *Библиотеки* требуется **17** MB свободного пространства на жестком диске. Сопутствующие материалы содержат немало дополнительной *полезной* информации, которая по тем либо иным причинам не была включена в основной текст книги.

Все исходные тексты средств, содержащихся в *Библиотеке*, доступны пользователю, что позволяет использовать их в качестве хорошего иллюстративного материала при освоении программирования в среде пакета. В них представлено использование различных полезных методов и приемов программирования, включая и *нестандартные*, которые во многих случаях позволяют существенно упрощать программирование задач в среде пакета ***Maple,*** делая их более прозрачными и изящными с математической точки зрения. При этом, следует отметить, что в ряде случаев тексты процедур оставляют достаточно широкое поле для их оптимизации (*в нынешнем виде большинство из них по эффективности, практически, не уступает оптимальным*), однако это было сделано вполне умышленно с целью иллюстрации ряда особенностей и возможностей языка программирования ***Maple.*** Это будет весьма полезно при освоении *практического* программирования в среде пакета ***Maple.***

Следует отметить, что поставляемые с *Библиотекой* файлы "**ProcUser.txt**" (*для* ***Maple*** **6–9**) и "**ProcUser10.txt**" (*для* ***Maple*** **9.5/10**), содержащие *исходные* тексты программных средств, составляющих *Библиотеку*, а также полный набор ***mws***-файлов с ***help***-страницами, составляющими справочную базу *Библиотеки*, наряду с большим набором различного назначения примеров, позволяют достаточно легко адаптировать *Библиотеку* на базовые платформы, отличные от ***Windows***-платформы. Более того, в виду наследования встроенными языками математических пакетов целого ряда общих черт, имеется хорошая возможность адаптации ряда процедур нашей ***Maple***-библиотеки к программной среде других пакетов. В частности, целый ряд процедур *Библиотеки* достаточно легко был нами адаптирован к среде пакета ***Mathematica*** и некоторых других математических пакетов, тогда как предложенный нами метод "*дисковых транзитов*", существенно расширяющий возможности программирования, эффективен не только для математических пакетов. При этом, следует иметь в виду, что *исходные* тексты программных средств, представленные в книге [103], и их представления в нашей *Библиотеке* (*будучи функционально*

эквивалентными) могут в определенной степени различаться, что обусловлено широким использованием *Библиотеки* также и в учебных целях.

Наш и опыт наших коллег показывает, что использование *Библиотеки* в целом ряде случаев существенно расширяет возможности пакета ***Maple*** релизов **6 – 10** и последующих релизов, упрощая программирование различных прикладных задач в его среде. Данная *Библиотека* представит особый интерес прежде всего для тех, кто использует пакет ***Maple*** не только как высоко интеллектуальный калькулятор, но также и как *среду* программирования различных задач математического характера в своей профессиональной деятельности.

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. Ряд часто используемых процедур *Библиотеки*, ориентированных на массовое применение при программировании различных приложений, оптимизирован. Тогда как многие, обладая функциональной полнотой, на которую они и были ориентированы, между тем, в полной мере не оптимизированы, что предоставляет пользователю (*прежде всего серьезно осваивающему программирование в* ***Maple***) достаточно широкое *поле* для его творчества как по оптимизиции процедуры, так и по созданию собственных аналогов, постоянно контролируя себя готовым, отлаженным и корректно функционирующим прообразом. Более того, используемые в процедурах полезные, эффективные (*а в целом ряде случаев и нестандартные*) *приемы* программирования позволяют более глубоко и за более короткий срок освоить программную среду пакета. Использование же во многих процедурах обработки особых и ошибочных ситуаций дает возможность акцентировать уже на ранней стадии *внимание* на таких важных компонентах создания программных средств, как их надежность, мобильность и ошибкоустойчивость. Наконец, работая с *Библиотекой*, пользователь не только имеет прекрасную возможность освоить многие из ее средств для своей текущей и последующей работы с пакетом, но и проникается *концепцией* эффективной организации своих собственных ***Maple***-библиотек, содержащих средства, обеспечивающие его профессиональные интересы и потребности. Есть надежда, что и читатель книги найдет среди средств *Библиотеки* полезные для своего творчества.

В предлагаемой книге рассматриваются принципы работы в программной среде ***Maple,*** основу которого составляет язык *программирования* ***Maple,*** что является непосредственным продолжением наших книг упомянутой серии [1-20,22-33,39,41-46,47,49,50,91,103], в которых обсуждаются **ПС** того же типа, что и *рассматриваемое* в настоящей книге. Придерживаясь выработанных методики и методологии подготовки указанных книг, наш подход делает основной акцент на изложении материала на основе развернутой *апробации* описываемой предметной области при решении задач как сугубо теоретических, так и прикладных. В предлагаемой книге представлены *базовые* сведения по языку программирования ***Maple –*** составляющему *основу* программной *среды* пакета, в которой пользователь имеет достаточно широкие возможности по разработке собственных ***Maple***-приложений.

Пакет ***Maple*** способен решать большое число, прежде всего, *математически ориентированных* задач вообще без программирования в общепринятом смысле. Вполне можно ограничиться лишь описанием алгоритма решения своей задачи, разбитого на отдельные *последовательные* этапы, для которых ***Maple*** имеет уже готовые решения. При этом, пакет ***Maple*** располагает большим набором процедур и функций, непосредственно решающих совсем не тривиальные задачи как то 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 довольно развитый язык программирования (*в дальнейшем* ***Maple****-язык*), пакет позволяет программировать в своей среде самые разнообразные задачи из различных приложений. Несколько поясним данный аспект, которому в отечественной литературе уделяется недостаточно внимания.

Относительно проблематики, рассматриваемой в настоящей книге, *вполне* уместно сделать несколько существенных замечаний. К большому сожалению, у *многих* пользователей современных математических пакетов, включая и *системы компьютерной алгебры* **–** *основной темы нашей книги* **–** бытует достаточно распространенное мнение, что использование подобных средств не требует знания программирования, ибо все, что нужно для решения их задач, якобы уже имеется в этих средствах и задача сводится лишь к выбору нужного средства (*процедуры*, *модуля*, *функции и т.д.*). Подобный подход к данным средствам носит в значительной степени дилетантский характер, причины которого достаточно детально рассмотрены в нашей книге [103].

Двухуровневая лингвистическая поддержка пакета ***Maple*** обеспечивается такими языками программирования как ***C*** и ***Maple.*** В ряде публикаций встречается иная (*не вполне обоснованная на наш взгляд*) классификацию, когда выделялись *три* языка **–** *реализации*, *входной* и *программирования*. Суть же состоит в следующем. Действительно, *ядро пакета* ***Maple*** содержит набор высокоэффективных программ, написанных на ***C****-*языке. Более того, библиотека функций доступа к компонентам файловой системы компьютера непосредственно заимствована из соответствующей библиотеки ***C.*** По нашим оценкам доля программных средств пакета, написанных на ***C,*** не превышает **15%.** Остальная масса программных средств пакета (*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* язык 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 определенными допущениями можно говорить о *входном* ***Maple****-*языке и *языке программирования* ***Maple***. В основе своей *входной* ***Maple****-*язык пакета базируется на встроенном языке программирования, являясь его подмножеством, обеспечивающим интерактивный режим работы с пакетом. Именно на входном ***Maple****-*языке в этом режиме пишутся и выполняются ***Maple****-*документы.

Входной язык ориентирован на решение математически ориентированных задач практически любой сложности в интерактивном режиме. Он обеспечивает диалог пользователя со своей вычислительной компонентой (*вычислителем*), принимая запросы пользователя на обработку данных с их последующей обработкой и возвратом результатов в символьном, числовом и/или графическом видах. *Входной* язык является языком *интерпретирующего* типа и идеологически подобен языкам этого типа. Он располагает большим числом математических и графических функций и процедур, и другими средствами из обширных библиотек пакета. Интерактивный характер языка позволяет легко реализовывать на его основе интуитивный принцип решения своих задач, при котором ход решения можно пошагово верифицировать, получая в конце концов требуемое решение. Уже введя первые предложения в текущий сеанс пакета, вы начинаете работать со *входным* ***Maple****-*языком. В настоящей же книге рассматривается наиболее полная лингвистическая компонента пакета **–** *встроенный* ***Maple****-*язык программирования (*или просто* ***Maple****-язык*). Вместе с тем, все примеры применения представленных в книге средств являются типичными предложениями *входного* ***Maple****-*языка пакета.

Среда программирования пакета обеспечивается *встроенным* ***Maple***-языком, являющимся 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, в той либо иной мере знакомому как с этими языками, так и с программированием вообще, не составит особого труда освоить и ***Maple***-язык.

Maple-язык имеет вполне традиционные стредства структурирования программ, включает в себя все команды и функции *входного* языка, ему доступны все специальные операторы и функции пакета. Многие из них являются достаточно серьезными программами, например, алгебраическое дифференцирование и интегрирование, задачи линейной алгебры, графика и анимация сложных трехмерных объектов и т. д. Являясь *проблемно-ориентированным* языком программирования, ***Maple***-*язык* характеризуется довольно развитыми средствами для описания задач математического характера, возникающих в различных прикладных областях. В соответствии с языками данного класса структуры *управляющей логики* и *данных* ***Maple***-языка в существенной *мере* отражают характеристику средств, прежде всего, именно для математических приложений. Наследуя многие черты ***C***-языка, на котором написан компилятор интерпретирующего типа, ***Maple***-язык позволяет обеспечивать как численные вычисления с любой точностью, так и символьные вычисления, поддерживая все основные операции традиционной математики. Однако, здесь следует привести одно весьма существенное пояснение.

Хорошо известно, что *далеко* не все задачи поддаются решению в аналитическом 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 весьма неэффективно. Именно поэтому пакет предоставляет как средства перекодировки программ с ***Maple***-языка на ***C, Java, FORTRAN, MatLab*** и ***VisualBasic,*** так и поддержку достаточно эффективного интерфейса с известным пакетом ***MatLab***.

Средства ***Maple***-языка позволяют пользователю работать в среде пакета в двух режимах: (**1**) на основе функциональных средств языка с использованием правил оформления и работы с ***Maple***-документом предоставляется возможность на интерактивном уровне формировать и выполнять требуемый алгоритм вашей задачи без сколько-нибудь серьезного знания даже основ программирования, а подобно конструктору собирать из готовых функциональных компонент *входного* языка на основе его синтаксиса требуемый вам алгоритм, включая его выполнение, отображение результатов на экране (*в обычном и/или графическом виде*), в файле и в твердой копии, и (**2**) использовать всю мощь ***Maple***-языка для создания развитых систем конкретного назначения, так и средств, расширяющих собственно саму *среду* ***Maple,*** чьи возможности определяются только *вашими* собственными умением и навыками. При этом, первоначальное освоение языка не предполагает предварительного серьезного знакомства с *основами* программирования, хотя их знание и весьма предпочтительно.

Программирование в среде ***Maple***-языка в большинстве случаев не требует какого-либо особого программистского навыка (*хотя его наличие и весьма нелишне*), т. к. в отличие от *других* языков универсального назначения и многих проблемно-ориентированных язык ***Maple*** включает большое число математически *ориентированных* функций и процедур,

позволяя только одним вызовом решать достаточно сложные самостоятельные задачи, например: решать системы дифференциальных или алгебраических уравнений, находить минимакс выражения, вычислять производные и интегралы, выводить графики сложных функций и т.д. Интерактивность языка обеспечивает простоту его освоения и *удобство* редактирования и отладки прикладных ***Maple***-документов и программ. Реальная же мощь ***Maple***-языка обеспечивается не только его управляющими структурами и структурами данных, но и всем богатством его *функциональных* (*встроенных*, *библиотечных*, *модульных*) и *прикладных* (***Maple***-*документов*) сред-тв, созданных к настоящему времени пользователями из различных прикладных областей, прежде всего, математических. Важнейшим преимуществом ***Maple*** является открытость его архитектуры, что позволило в кратчайшие сроки создать широким кругом пользователей из многих областей науки, образования, техники и т.д. обширных наборов *процедур* и *модулей*, которые значительно расширили как его возможности, так и сферу приложений. К их числу можно с полным основанием отнести и представленную в [103] *Библиотеку*, содержащую более **730** средств, дополняющих средства пакета, устраняющих ряд его недоработок, расширяющих ряд его стандартных средств и повышающих уровень совместимости релизов пакета. Представленные в [103] средства используются достаточно широко как при работе с пакетом ***Maple*** в интерактивном режиме, так и при программировании различных задач в его среде. Они представляют несомненный интерес при программировании различных задач в среде ***Maple*****,** как упрощая собственно сам процесс программирования, так и делая его более прозрачным с формальной точки зрения.

Таким образом, пакет ***Maple*** **–** не просто *высоко* интеллектуальный калькулятор, способный аналитически решать многие задачи, 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 много специальных задач, которые оказались за пределами интересов ее разработчиков. *Освоив* относительно простой, но весьма эффективный ***Maple***-язык, пользователь может изменять уже существующие процедуры либо расширять пакет новыми, ориентированными на решение нужных ему задач. Эти процедуры можно включать в одну или несколько пользовательских библиотек, снабдить справочной базой, логически сцепить с главной библиотекой пакета, так что их средства на *логическом* уровне будут неотличимы от стандартных средств пакета. Именно таким образом и организована наша *Библиотека* [103,108,109]. И последнее, ***Maple***-язык **–** наименее подверженная изменениям компонента пакета, поэтому ее освоение позволит вам весьма существенно *пролонгировать* эффективное использование пакета для решения тех задач, которые прямо не поддерживаются пакетом.

Поскольку ***Maple***-язык является одновременно и языком *реализации* пакета, то его освоение и практическое программирование в его среде позволят не только существенно повысить ваш *уровень* использования предоставляемых пакетом возможностей (*уровень владения пакетом*), но и глубже понять как идеологию, так и внутреннюю кухню самого пакета. Учитывая же ведущие позиции ***Maple*** в современной компьютерной алгебре и во многом распространенную его идеологию в этой области, вы получаете прекрасную и пролонгированную возможность весьма эффективного использования подобных ему средств для своей *профессиональной* деятельности, прежде всего, в математике, физике, информатике и др.

В настоящей книге рассматриваются основы ***Maple***-языка программирования в предположении, что читатель в определенной мере имеет представление о работе в ***Windows***-среде и с самим пакетом в режиме его главного меню (**GUI**) в пределах, например, книг [8-14,29-33,39,42-46,54-62,103] и подобных им изданий. Представленные ниже сведения по ***Maple***-языку в значительной мере обеспечат вас тем необходимым минимумом зна-

ний, который позволит знакомиться со средствами главной библиотеки пакета и нашей *Библиотеки* [103]. Данная работа, в свою очередь, даст вам *определенный* навык программирования, а также обеспечит вас набором полезных приемов и методов программирования (*включая и нестандартные*) в среде ***Maple***-языка наряду с практически полезными средствами, упрощающими решение многих прикладных задач, составляющих ваши приложения. Кратко охарактеризуем содержание предлагаемой книги.

Первая глава книги представляет *базовые* сведения по ***Maple***-языку, включая рассмотрение таких вопросов, как базовые элементы языка, идентификаторы, предложения присвоения и выделения, средства языка для определения свойств переменных, типы числовых и символьных данных, базовые типы структур данных, средства тестирования типов данных, структур данных и выражений, конвертация выражений из одного типа в другой и др.

Вторая глава представляет *базовые* управляющие структуры ***Maple***-языка: *ветвления* (**if**-*предложение*), организации *циклических* вычислений (**while_do**-*предложение*), а также специальные типы циклических управляющих структур ***Maple***-языка.

Третья глава достаточно детально рассматривает одну из ключевых *образующих модульного* программирования **–** *процедуры* и средства их организации в среде ***Maple***-языка. В ней представлены как базовые так и расширенные средства наряду с вопросами расширения функциональных средств ***Maple***-языка. Довольно подробно рассмотрены такие вопросы как определения процедур и их типы, формальные и фактические аргументы, локальные и глобальные переменные, определяющие параметры и *описания* процедур, механизмы возврата результата вызова процедуры, средства обработки ошибочных ситуаций, расширенные средства для работы с процедурами, расширение функциональных средств ***Maple***-языка и др.

Четвертая глава рассматривает вопросы организация *программных модулей* ***Maple***-языка и сохранения процедур и программных модулей в файлах входного и внутреннего форматов.

Пятая глава представляет средства ***Maple***-языка для работы с данными и структурами символьного, строчного, списочного, множественного и табличного типов, в том числе и средства работы с выражениями строчного и символьного типов, множествами, списками и таблицами. Рассматриваются алгебраические правила *подстановок* для символьных вычислений и др.

Шестая глава представляет средства ***Maple***-языка для обеспечения доступа к внешним файлам данных. Полнота изложения в значительной степени перекрывает как поставляемую с пакетом документацию, так и другие издания. С целью развития системы доступа к файловой системе компьютера нами был разработан целый ряд эффективных средств, представленных как в настоящей главе, так и в нашей *Библиотеке* [103,108,109].

Седьмая глава представляет основные средства ***Maple***-языка для обеспечения работы с графическими объектами. Пакет располагает развитым набором функций различных уровней, обеспечивающих формирование графических структур данных и вывод соответствующих им графических объектов как в двух, так и в трех измерениях, а также в широком диапазоне систем координат. Здесь же рассматриваются и такие вопросы, как расширение функциональных возможностей графических средств пакета.

Восьмая заключительная глава знакомит с вопросами создания и ведения *библиотек* пользователя. Рассмотрены такие вопросы как *классический* способ создания ***Maple***-библиотек, *специальные* способы создания библиотек в среде ***Maple,*** создание пакетных модулей и др.

Подобно другим **ПС,** ориентированным на решение задач *математического* характера, пакет ***Maple*** располагает достаточно развитыми средствами, обеспечивающими решение широкого круга задач *математического анализа* и *линейной алгебры*. Эти две дисциплины представляют собой основную *базовую* компоненту современного математического образования естественно-научных специальностей и особых пояснений не требуют. Вместе с тем, следует сразу же отметить, что в дальнейшем вопросы решения задач математического анализа, линейной алгебры и других математически-ориентированных дисциплин в среде ***Maple*** (*учитывая специфику настоящей книги*) не рассматриваются. В принципе, сложности при освоении средств данной группы особой возникать не должно и они достаточно хорошо изложены в многочисленной литературе как зарубежной, так и отечественной [8-14,54-62]. В книгах [41-43,103] мы представили средства, функционирующие в среде пакета релизов **6–10** и ориентированные, прежде всего, на решение задач *математического анализа*. Там же представлены средства, расширяющие возможности пакета при решении задач *линейной алгебры*. Данные средства классифицируются относительно стандартных средств *двух основных* пакетных модулей, ориентированных, прежде всего, на задачи *линейной алгебры*, а именно: традиционный ***Maple-***модуль ***linalg*** и имплантированный модуль ***LinearAlgebra*** фирмы ***NAG***. Представлены дополнительные средства, обеспечивающие решение ряда массовых задач *линейной алгебры*.

Более того, не рассматриваются детально здесь и такие важные средства пакета как *графические* и *обеспечения доступа* к файлам данных. Мотивировано данное решение тем, что эти средства довольно хорошо изложены в наших предыдущих книгах, книгах других авторов и в справочной системе по пакету. В частности, в книге [12] достаточно детально рассмотрена система *ввода/вывода* пакета, которая не претерпела каких-либо существенных изменений вот уже на протяжении **6** релизов. Основной акцент здесь сделан на тех аспектах средств доступа, которые не нашли отражения в имеющихся на сегодня изданиях по ***Maple***-языку, включая и фирменную литературу, стандартно поставляемую с пакетом. В целом же система *ввода/вывода* пакета может быть охарактеризована следующим образом.

Будучи языком программирования в среде пакета *компьютерной алгебры*, ориентированного, прежде всего, на задачи алгебраических вычислений, ***Maple***-язык располагает относительно ограниченными возможностями при работе с данными, которые расположены во внешней памяти компьютера. Более того, в этом отношении ***Maple***-язык существенно уступает таким традиционным языкам программирования как ***C, Cobol, PL/1, FORTRAN, Pascal, Ada, Basic*** и т. д. В то же самое время ***Maple***-язык, ориентированный, прежде всего, на решение задач математического характера, предоставляет тот набор средств для доступа к файлам данных, которые могут вполне удовлетворить довольно широкую аудиторию пользователей пакета и его физических и математических приложений. В наших книгах [41-42,103] представлен целый ряд дополнительных средств *доступа* к файлам данных, *существенно* расширяющих пакет в данном направлении. Многие из них упрощают программирование целого ряда задач, имеющих дело с доступом к файлам данных различной организации, содержания и назначения.

Со всей определенностью можно констатировать, что *новые* 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 значительно *более* представительным и они сосредоточены на более широком практическом использовании при решении задач, имеющих дело с обработкой файлов данных. Более того, нам не известно более обстоятельное рассмотрение системы доступа к файлам, обеспечиваемой пакетом, чем в наших предыдущих книгах [8-14,

29-33,39,41-43,45,46,103]. Именно в этом отношении они рекомендуются читателю, имеющему дело с подобными задачами.

В определенной мере вышесказанное можно отнести и к *графическим* средствам пакета. Так, в вышеуказанных книгах представлены средства, расширяющие стандартный набор графического инструментария пакета ***Maple*** релизов **6 – 10**. Предлагаемые средства являются довольно полезными процедурами, представляющими определенный прикладной интерес, что подтверждает эффективность их использования при решении целого ряда прикладных задач с использованием графических объектов. В частности, на сегодня, стандартные средства пакета, ориентированные на работу с анимируемыми графическими объектами, имеют целый ряд ограничений на *динамическое* обновление их характеристик. Между тем, многие задачи, имеющие дело с *графическими анимируемыми* объектами, предполагают возможность *динамического* обновления их характеристик. Представлен ряд средств в этом направлении. Заинтересованный читатель отсылается к книгам [8-14,32,78,84,86,88,55,59-62], а также к [91] с адресом сайта (*Локальная копия сайта*), с которого можно бесплатно загружать некоторые из наших предыдущих книг по данной тематике.

Настоящая книга носит вводный характер, представляя собой *основы* языка программирования ***Maple,*** что позволит читателю, не знакомому в достаточной мере с программной средой пакета, более осознанно воспринимать информацию по программным средствам пакета. По этой причине она лишь скелетно представляет ***Maple***-язык пакета, не отвлекаясь на его тонкости, особенности и другие частности в свете основной цели настоящей книги. Следует отметить, что к сфере ***Maple***-языка пакета мы отнесли не только синтаксис, семантику, структуры данных, управляющие структуры и т. д., но и функции и процедуры различных уровней, которые доступны собственно ***Maple***-языку в процессе программирования в его среде программ и их последующего выполнения.

Читатель, имеющий опыт программирования в среде ***Maple,*** может *вполне* (*быть может, за редким исключением*) безболезненно опустить данный материал. Тогда как начинающий на этом поприще получит необходимый *минимум* сведений по ***Maple***-языку, который позволит значительно проще совершенствоваться в этом направлении. Для более глубокого ознакомления с ***Maple***-языком нами приведены полезные ссылки как на отечественные, так и на зарубежные издания. Более того, читатель, заинтересованный в освоении программной среды пакета ***Maple,*** может найти немало полезной информации как в *справочной* системе по пакету, литературе, поставляемой с пакетом, так и на ведущих ***Web***-*серверах* и форумах, посвященных ***Maple***-тематике. В век почти массовой компьютеризации и интернетизации лишь ленивый может аппелировать к недостатку нужной информации по какому бы то ни было разделу современного знания.

По ходу настоящей книги будет приведено немало ссылок на наши издания, что следует совершенно непредвзято трактовать и вот почему. Прежде всего, в настояшей книге рассматриваются вопросы программирования в среде пакета ***Maple,*** которое является основой разработки в его среде эффективного программного обеспечения, ориентированного на те или иные приложения. В отечественной же литературе других изданий, рассматривающих столь скрупулезно данную проблематику (*как*, *впрочем*, *и в англоязычной*), на мой взгляд, просто нет (*и это не только мое субъективное мнение*). Наконец, вполне это уместно и в том потребительском контексте, что основная масса цитируемых наших изданий по ***Maple***-тематике может быть совершенно бесплатно получена с указанного в [91] *Internet*-адреса Гродненского университета. Конечно, там представлены издания, ориентированные на ***Maple*** **5 – 7,** однако ввиду полной приемственности по ос-

новным аспектам программной среды пакета они (*с некоторыми оговорками*) *вполне* пригодны и для последующих релизов пакета.

Заключение представляет краткий экскурс в историю создания нашей ***Maple****-библиотеки*, которая будет неоднократно цитироваться в настоящей книге. Здесь лишь хочу предупредить, что во многом она создавалась еще в ***Maple*** релизов **5–7,** когда нас не совсем устраивали как целый ряд базовых средств пакета, так и возможности пакета. Поэтому создавались средства как заполняющие подобный *ваккуум*, так и устраняющие замеченные недостатки пакета. С появлением новых релизов *базовые* средства пакета и его средства в целом расширялись и улучшались, поэтому *ряд* средств *библиотеки* могут, на первый взгляд, показаться устаревшими (*obsolete*), однако они все еще представляют интерес с *учебной* точки зрения, предлагая немало эффективных и полезных приемов программирования, не утративших своей актуальности и поныне. Многие же из библиотечных средств и на сегодня не имеют аналогов. В последние пару лет ***Maple***-тематике в ее прямом смысле мы уделяли относительно немного внимания, используя, между тем, пакет в целом ряде физико-математических и инженерных приложений, а также в преподавании математических курсов для докторантов, различных курсов и мастер-классов по ***Maple***. В будущем данная тематика не планируется в числе наших активных интересов, поэтому наши публикации в этом направлении будут носить в значительной мере *спорадический* характер. Вот и данное издание, посвященное вопросам *сугубо* программирования в ***Maple,*** также в значительной мере специально не планировалось, поэтому оно не было подвержено тщательному редактированию. Однако смеем надеяться, что и в таком виде оно представит определенный интерес как для начинающих, так и даже имеющих определенный опыт программирования в среде ***Maple,*** да и не только.

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* [11,13,14,43,110]);

(**2**) отдельные *процедуры*, *программные модули* либо их наборы, сохраненные в файлах и решающие конкретные прикладные задачи (*типичными примерами служат 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 (*типичным примером служит наша Библиотека* [41,103,108,109]).

Естественно, представленная *классификация* в значительной степени субъективна, однако при отсутствии иной и она может служить в качестве *отправной* точки. Средства *первого* уровня представляют собой законченные ***Maple***-документы, т.е. {***mws***, ***mw***}-файлы, решающие конкретные прикладные залачи из различных приложений. Они могут содержать описательную часть, постановку задачи, алгоритм, описанный *входным* ***Maple***-языком, результаты выполнения документа и другую необходимую информацию. Как правило, сложная задача либо комплекс задач реализуются *набором* {***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 таких документов вполне можно обходиться лишь входным ***Maple***-языком, а не полным набором средств, предоставляемых его расширением – встроенным языком

программирования. В целом, данный уровень допускает и *диспетчирование* документов, например, соедующим образом.

У традиционного ***Maple***-документа после его отладки удаляются все ***Output***-параграфы и он сохраняется в текстовом файле в рамках только своих ***Input***-параграфов. Создается ***Maple***-документ, обеспечивающий *загрузку* в нужном порядке в текущий сеанс по **read**-предложению созданные описанным образом ***txt***-*файлы*, обеспечивая необходимое диспетчирование документов.

Второй уровень представляет отдельные процедуры либо 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 этом случае пользователь уже в значительной мере использует встроенный ***Maple***-язык.

Наконец, третий уровень характеризуется использованием для разработки достаточно сложных приложений в полной мере средств встроенного ***Maple***-языка, созданием библиотек, подобных *главной* библиотеке пакета, наряду с пакетными модулями, ориентированных как на относительно узкие области приложений, так и на самое массовое использование при разработке приложений в среде пакета. Типичным примером средств третьего уровня является и наша *Библиотека*, рассматриваемая в настоящей книге.

Литература содержит интересные и *полезные* как зарубежные, так и отечественные источники по ***Maple***-тематике, которые являются *вполне* доступными. При этом многие из наших изданий (*точнее их авторские оригинал-макеты*) можно бесплатно получать с указанного в [91] *Internet*-адреса Гродненского государственного университета им. Я. Купалы. Там же можно найти и массу полезных примеров по ***Maple***-тематике.

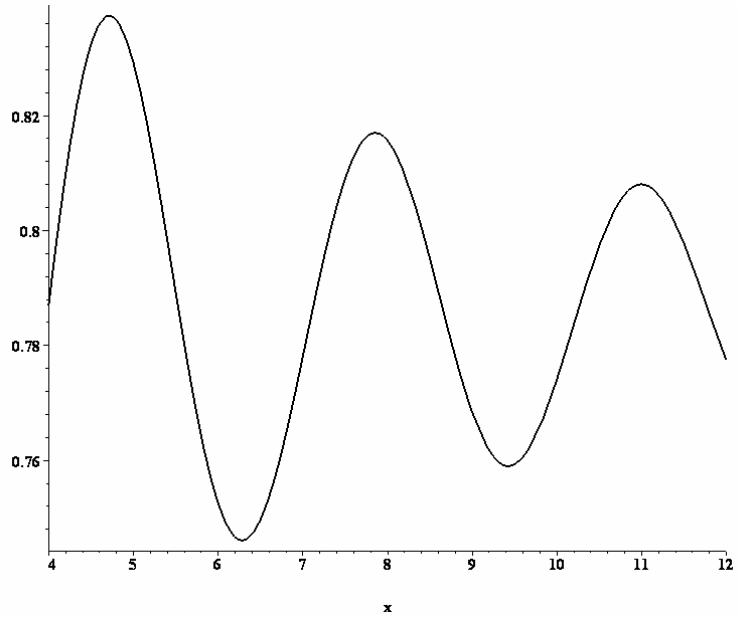
В дальнейшем изложении нами используются следующие *основные* соглашения и обозначения, а также основные используемые в тексте сокращения:

- **ПС** – программное средство;
- **ПО** – программное обеспечение;
- **ПК** – персональный компьютер;
- альтернативные элементы разделяются символом " | " и заключаются в фигурные скобки, например, {**A** | **B**} – **A** или **B**;
- смысл конструкции указывается в угловых скобках "**<** ... **>**";
- под выражением *"по умолчанию"* понимается значение той или иной характеристики операционной среды или пакета, которое используется, если пользователем не определено для нее иное значение;
- под «*регистро-зависимым* (*независимым*)» режимом будем понимать такой режим *поиска*, сравнения и т.д. символов и строк, когда принимаются (*не принимаются*) в расчет *различия* между одинаковыми буквами *верхнего* и *нижнего* регистра клавиатуры. В частности, все идентификаторы в программной среде ***Maple*** регистро-зависимы;
- понятия "*имя УВВ*, *имя файла*, *каталог*, *подкаталог*, *цепочка каталогов*, *путь*" соответствуют полностью соглашениям **MS DOS;** при кодировании пути к файлу в качестве разделителя используется, как правило, *сдвоенный* обратный слэш (\\);
- **СФ** - спецификатор файла, определяющий полный путь к нему либо имя;
- **СБО** (***Clipboard***) – системный буфер обмена;
- **ГМП** (**GUI**) - *главное меню пакета;* содержит группы функций его оболочки;
- **hhh**-файл - файл, имеющий "hhh" в качестве расширения его имени;
- под ***Input***-*параграфом* в дальнейшем понимается вводимая в интерактивном режиме в текущий сеанс информация, слева идентифицируемая символом ввода

«**>**». По умолчанию, ввод оформляется красным цветом. Тогда как под ***Output***-*параграфом* понимается результат выполнения предшествующего ему ***Input***-параграфа (*выражения*, *тексты процедур и модулей*, *графические объекты и т.д.*). Следующий простой пример поясняет вышесказанное:

> A, V:= 2, 12: assign('G' = Int(1/x*sin(x)*cos(x), x)), G = value(G); plot(rhs(%), x = 4 .. V, thickness=A, color=green, axesfont=[TIMES, BOLD, V], labelfont=[TIMES, BOLD, V]);

$$\int \frac{\sin(x)\cos(x)}{x}\,dx = \frac{1}{2}\,\mathrm{Si}(2x)$$



> MkDir("D:/Temp/RANS/IAN/RAC/REA/Test", 1); ⇒ "d:\temp\rans\ian\rac\rea\test"
> type(%, 'dir'), type(%, 'file'); ⇒ *false, true*

В дальнейшем ради удобства и компактности там, где это возможно, блок из **2**-х параграфов {***Input, Output***} будем представлять в строчном формате, а именно:

> Input-параграф; ⇒ *Output-параграф*

Остальные необходимые обозначения, понятия, сокращения и соглашения будут нами вводится по мере надобности по тексту книги. Тогда как с нашими последующими разработками и изданиями по данной проблематике можно периодически знакомиться на ***Web***-страницах:

www.aladjev.newmail.ru, www.aladjev.narod.ru, www.aladjev-maple.narod.ru

На этих же страницах находятся ***email***-адреса, в которые можно отправлять все замечания, пожелания и предложения, относящиеся к материалу настоящей книги, а также в целом по ***Maple***-тематике. Все они будут приняты с благодарностью и без нашего внимания не оставлены.

Гродно – Таллинн, февраль, 2007

Глава 1. *Базовые сведения по Maple-языку пакета*

Пакет ***Maple*** способен решать большое число, прежде всего, математически ориентированных задач вообще без программирования в общепринятом смысле. Вполне можно ограничиться лишь описанием алгоритма решения своей задачи, разбитого на отдельные *последовательные* этапы, для которых ***Maple*** имеет уже готовые решения. При этом, ***Maple*** располагает довольно большим набором процедур и функций, непосредственно решающих совсем не тривиальные задачи как то 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 достаточно развитый язык *программирования* (*в дальнейшем просто* ***Maple****-язык*), пакет позволяет программировать в своей *среде* самые разнообразные задачи из различных приложений. Обсуждение данного аспекта нашло отражение в нашей книге [103], а именно о *дилетантском взгляде* на вопрос программирования в ***Maple,*** да, впрочем, и в ряде других математических пакетов.

В среде пакета можно работать в двух основных режимах: *интерактивном* и *автоматическом* (*программном*). ***Интерактивный*** режим аналогичен работе с калькулятором, пусть и весьма высоко интеллектуальным **–** в ***Input****-параграфе* вводится требуемое ***Maple****-*выражение и в следующем за ним ***Output-****параграфе* получаем результат его вычисления. Так шаг за шагом можно решать в среде пакета достаточно сложные математические задачи, в процессе работы формируя *текущий документ* (**ТД**). Для такого режима требуется относительно небольшой объем знаний о самом пакете, а именно:

* знание общей структуры **ТД** и синтаксиса кодирования выражений;
* знание синтаксиса используемых функциональных конструкций;
* общие правила выполнения, редактирования и сохранения текущего документа.

Таким образом, средства ***Maple****-*языка позволяют пользователю работать в среде пакета без сколько-нибудь существенного знания даже основ программирования, а подобно конструктору собирать из готовых функциональных компонент языка на основе его синтаксиса требуемый вам алгоритм, включая его выполнение, отображение результатов на экране (*в обычном и/или графическом виде*), в файле и в твердой копии, и (**2**) использовать всю мощь языка как для создания развитых систем конкретного назначения, так и средств, расширяющих собственно саму программную среду пакета, возможности которых определяются только вашими собственными умением и навыками в сфере алгоритмизации и программирования.

Программирование в среде ***Maple****-*языка в большинстве случаев не требует *особого* программистского навыка (*хотя его наличие и весьма нелишне*), ибо в отличие от языков универсального назначения и многих проблемно-ориентированных ***Maple****-*язык включает большое число математически ориентированных функций, позволяя одним вызовом функции решать достаточно сложные самостоятельные задачи, например: вычислять минимакс выражения, решать системы дифуравнений или алгебраических уравнений, вычислять производные и интегралы, выводить графики сложных функций и др. Тогда как интерактивность языка обеспечивает не только простоту его освоения и удобство редактирования, но также в значительной мере предоставляет возможность *эвристического* программирования, когда методом «*проб и ошибок*» пользователь получает возможность запрограммировать свою задачу в полном соответствии со своими нуждами. Реальная же мощь языка обеспечивается не только его управляющими структурами и структурами данных, но и всем богатством его функциональных (*встроенных*, *библио-*

течных, *модульных*) и прикладных (***Maple****-документов*) средств, созданных к настоящему времени пользователями из различных прикладных областей, прежде всего физико-математических и технических, а в целом, естественно-научных.

Однако для более сложной работы (*выполняющейся*, *как правило*, *в программном режиме*) требуется знание встроенного ***Maple***-языка программирования, который позволяет использовать всю вычислительную мощь пакета и создавать сложные **ПС,** не только решающие задачи пользователя, но и расширяющие средства самого пакета. Примеры такого типа представлены как в нашей книге [103], так и в настоящей книге. Именно поэтому, далее представим основные элементы программирования в среде пакета ***Maple***.

Для *лингвистического* обеспечения решения задач пакет снабжен развитым встроенным проблемно-ориентированным ***Maple****-языком*, поддерживающим *интерактивный* режим. Являясь *проблеммно-ориентированным* языком программирования, ***Maple***-язык характеризуется довольно развитыми средствами для описания задач математического характера, возникающих в различных прикладных областях. В соответствии с языками этого класса структуры управляющей логики и данных ***Maple***-языка в существенной мере отражают специфику средств именно для математических приложений. Особую компоненту языка составляет его функциональная составляющая, поддерживаемая развитой библиотекой функций, покрывающих большую часть математических приложений. Наследуя многие черты ***C***-языка, на котором был написан компилятор интерпретирующего типа, ***Maple***-язык позволяет обеспечивать как численные вычисления с любой точностью, так и символьную обработку выражений и данных, поддерживая все основные операции традиционной математики [8-14].

Средства ***Maple***-языка позволяют пользователю работать в среде пакета в двух режимах: (**1**) на основе функциональных средств языка с использованием правил оформления и работы с ***Maple***-документом предоставляется возможность на интерактивном уровне формировать и выполнять требуемый алгоритм прикладной задачи *без* сколько-нибудь существенного знания даже *основ программирования*, а подобно конструктору собирать из готовых функциональных компонент языка на основе его синтаксиса требуемый вам алгоритм, включая его выполнение, отображение результатов на экране (*в обычном или графическом виде*), в файле и в твердой копии, и (**2**) использовать всю *мощь* языка как для создания развитых систем конкретного назначения, так и средств, расширяющих собственно саму среду пакета, возможности которых определяются только вашими собственными умением и навыками. При этом, первоначальное *освоение* языка не предполагает предварительного серьезного знакомства с основами программирования, хотя знание их и весьма предпочтительно. Тут же уместно отметить, что значительная часть функциональных средств самого пакета написана и скомпилирована именно на ***Maple***-языке, позволившим создать довольно эффективные относительно *основных* ресурсов **ЭВМ** загрузочные модули. Анализ данных средств является *весьма* неплохим подспорьем при серьезном освоении ***Maple***-языка. Наряду с этим, в архиве [109] представлены исходные тексты процедур и программных модулей нашей *Библиотеки* [41], которые также могут служить в качестве неплохого иллюстративного материала для осваивающих программную среду пакета.

В настоящей главе рассматриваются базовые элементы ***Maple***-языка в предположении, что читатель в определенной мере имеет представление о работе в ***Windows***-среде и с самим пакетом в режиме его *графического меню* (**GUI**) в пределах главы **3** [12] либо, например, книг [29-33,45,46,54-62,78-89,103,108,111-130] и подобных им изданий.

1.1. *Базовые элементы Maple-языка пакета*

Определение ***Maple-***языка можно условно разбить на четыре базовые составляющие, а именно: *алфавит*, *лексемы*, *синтаксис* и *семантику*. Именно *две последние* составляющие и определяют суть того или иного языка. *Синтаксис* составляют *правила* образования корректных предложений из *слов* языка, которые должны строго соблюдаться. В случае обнаружения на входе ***Maple-***предложения с синтаксической ошибкой выводится диагностическое сообщение, сопровождаемое "**^**"-*указателем* на место возможной ошибки или установкой в место ошибки **|**-*курсора*, как это иллюстрирует следующий фрагмент:

```
> A:= sqrt(x^2+y^2+z^2); V:=x*56  |Z:= sin(x)/(a+b);
Error, missing operator or `;`
> A:= sqrt(x^2+y^2+z^2):  V:= x**|*56:  Z:= sin(x)/(a+b);
Error, `*` unexpected
> read `C:\\ARM_Book\\Grodno\\Academy.99`:
on line 0, syntax error, `*` unexpected:
VGS:=56***sin(x)/sqrt(Art^2+Kr^2)-98*F(x,y);
          ^
Error, while reading `C:\ARM_Book\Grodno\Academy.99`
```

При этом следует иметь в виду два обстоятельства: (**1**) идентифицируется только *первая* встреченная ошибка и (**2**) при наличии *синтаксических* ошибок (*например*, *несогласования открывающих* и *закрывающих* скобок) в сложных выражениях язык может затрудняться с точной их диагностикой, идентифицируя ошибочную ситуацию сообщением вида "*`;` unexpected*", носящим в целом ряде случаев весьма приблизительный характер. ***Maple-***язык производит *синтаксический контроль* не только на входе конструкций в *интерактивном* режиме работы, но и в момент считывания их из файлов. В последнем случае *синтаксическая* ошибка инициирует вывод соответствующего *диагностического* сообщения указанного выше вида с указанием номера *первой* считанной строки, содержащей ошибку, и идентификацией спецификатора файла, как это иллюстрирует последний пример фрагмента. Ниже вопросы *синтаксического* анализа ***Maple-***конструкций будут рассмотрены более детально, прежде всего, при рассмотрении ***parse***-функции пакета.

В отличие от *синтаксиса*, определяющего *правила* составления корректных языковых конструкций, *семантика* языка определяет алгоритмы их обработки, т.е. определяет их *понятийное* назначение с точки зрения самого языка. Например, результатом обработки конструкции вида "**W:= 57*sin(19.99);**" является присвоение **W-**переменной результата произведения целого числа "**57**" и значения ***sin-***функции в точке "**19.99**" ее вызова. При этом, определяется как собственно результат, так и его тип. В связи со сказанным наряду с *синтаксическими*, как правило, распознаваемыми языком, могут возникать и *семантические* ошибки, которые язык не распознает, если они при этом, не инициируют, в свою очередь, ошибок *выполнения*. Типичным примером семантических ошибок является конструкция вида "**A/B*C**", трактуемая языком как "**(A*C)/B**", а не как "**A/(B*C)**" на первый взгляд. Как правило, *семантические* ошибки выявляются на стадии выполнения ***Maple-***программы или вычисления отдельных ***Maple-***выражений и данная процедура относится к этапу *отладки* программ и процедур, рассматриваемому несколько ниже.

Синтаксис ***Maple-***языка определяется выбранным набором *базовых* элементов и *грамматикой*, содержащей *правила* композиции корректных конструкций языка из *базовых* элементов. Рассмотрение базовых элементов начнем со *входного алфавита* языка, в качестве элементов которого используются следующие символы:

- *заглавные* и *прописные* буквы *латинского* алфавита (**A .. ÷Z; a .. ÷z; 52);**

- ♦ *десятичные* цифры (**0 .. 9; 10);**
- ♦ *специальные* символы **(` ! @ # $ % ^ & * ( ) _ + { } : " < > ? | - = [ ] ; ' , . / \; 32);**
- ♦ *заглавные* и *прописные* буквы *русского* алфавита (*кириллицы*: **А .. Я; а .. я; 64).**

Синтаксис ***Maple-***языка объединяет символы входного алфавита в *лексемы*, состоящие из *ключевых* (*зарезервированных*) слов, *операторов*, *строк*, натуральных чисел и знаков пунктуации. Рассмотрим несколько детальнее каждую из составляющих. В качестве ***ключевых Maple-***язык использует слова, представленные в следующей табл. **1.**

Таблица 1

Ключевые слова:	***Смысловая нагрузка:***
if, then, else, elif, fi	*условное* ***if-***предложение языка
for, from, in, by, to, while, do, od	предложения *циклических* конструкций
proc, local, global, option, description, end	*процедурные* выражения языка
read, save	функции *чтения* и *записи* выражений
done, quit, stop	функции завершения работы
union, minus, intersect	операторы над *множествами*
and, or, not	*логические* операторы языка
mod	оператор вычисления по *модулю*

Так как ключевые слова несут определенную смысловую нагрузку, то они не могут использоваться в качестве переменных, в противном случае инициируется ошибка**:**

> *local*:= 64;

Error, reserved word `local` unexpected

В ***Maple-***языке существует целый ряд других слов, имеющих *специальный* смысл, например идентификаторы (***имена***) функций и типов, однако пользователь может использовать их в программах в определенных контекстах. При этом, защита их от *модификации* обеспечивается совершенно иным механизмом (*базирующемся на* ***protected****-механизме*), рассматриваемым несколько ниже.

Операторы языка относятся к *трем типам***:** *бинарные*, *унарные* и *нульарные*. Допускаемые языком *унарные* операторы представлены в следующей табл. **2.**

Таблица 2

Унарный оператор:	***Смысловая нагрузка оператора:***
+	*префиксный плюс*
-	*префиксный минус*
{**!**|***factorial***}	*факториал* (*постфиксный* оператор)
$	*префиксный* оператор *последовательности*
.	десятичная точка (*префиксный* или *постфиксный*)
%<*целое*>	оператор *метки*
not	*префиксный* логический оператор *отрицания*
&<*строка*>	*префиксный* пользовательский оператор

Унарные операторы (**+, -, not**) и десятичная точка (**.**) имеют вполне прозрачный смысл и особых пояснений не требуют. Здесь мы кратко остановимся на операторе *метки* (**%**)**;** при этом понятие *метки* в ***Maple-***языке существенно иное, чем для традиционных языков программирования. *Унарный* оператор *метки* кодируется в виде **%<***целое***>** и служит для представления общих *подвыражений* большого выражения в целях более наглядного представления *второго*. Данный оператор используется ***Maple-***языком для вывода выражений на экран и на принтер в удобном виде. После возвращения языком представлен-

ного в терминах %-оператора выражения к переменным %<*целое*> можно обращаться как к обычным определенным переменным. Для возможности использования представления выражений в терминах %-оператора используются две опции ***interface***-переменной оболочки пакета: *labelling* и *labelwidth*, определяющих соответственно допустимость такого представления и длину *меченных* подвыражений. При этом, данная возможность допустима не для всех форматов ***Output***-параграфа текущего документа.

Допустимые языком базовые *бинарные* операторы представлены в следующей табл. **3.**

Таблица 3

Оператор	*Смысловая нагрузка:*	*Оператор*	*Смысловая нагрузка:*
+	сложения	**<**	меньше чем
-	вычитания	**<=**	меньше чем или равно
*	умножения	**>**	больше чем
/	деления	**>=**	больше чем или равно
{**|^}	степени	**=**	равно
:=	присваивания	**<>**	не равно
:-	выбора элемента модуля	**,**	разделитель выражений
$	последовательности	**->**	функциональный
@	композиции функций	**mod**	взятие модуля
@@	кратной композиции	**union**	объединение множеств
::	определения типа	**minus**	разность множеств
&<*строка***>**	нейтральный *инфексный*	**intersect**	пересечение множеств
||	конкатенации строк	**and**	логическое **И**
..	ранжирования	**or**	логическое **ИЛИ**

С большинством из приведенных в табл. **3** *унарных операторов* читатель должен быть хорошо знаком, тогда как такие как (**:=, $, @, @@, ..**) в определенной мере специфичны для математически ориентированных языков и будут рассмотрены ниже. По остальным же пояснения будут даваться ниже по мере такой необходимости.

Наконец, три *нульарных* оператора **%, %%** и **%%%** (*один, два и три знака процентов*) представляют специальные идентификаторы ***Maple***-языка, принимающие в качестве значений результат вычисления соответственно *последнего, предпоследнего* и *предпредпоследнего* предложения. Следующий фрагмент иллюстрирует применение *нульарных* операторов:

> AG:= 59: AV:= 64: AS:= 39: (%%+3*%+%%%-2); ⇒ 238

> P:=proc() ***args; nargs;*** **%, %% end proc: P(59, 64, 39, 17, 10, 44);** ⇒ 6, 59, 64, 39, 17, 10, 44

Как правило, *нульарные* операторы используются в интерактивном режиме работы с пакетом и выступают на уровне *обычных* переменных, позволяя обращаться к результатам предыдущих вычислений на глубину до трех. Однако использование их вполне допустимо и в теле процедур, где они выступают на уровне локальных переменных, как это иллюстрирует второй пример фрагмента. Вопросы организации процедур рассматриваются ниже. Наряду с возможностью использования {%|%%|%%%}-конструкции для доступа к результатам предыдущих вычислений в текущем сеансе работы языком предоставляется ***history-****механизм* обращения к любому полученному в рамках его *истории* вычислению. Детально данный механизм рассмотрен в нашей книге [12]. В частности, там же представлен и *анализ* данного механизма, говорящий о недостаточно продуманной и реализованной системе, обеспечива-ющей ***history-****механизм*; в целом весьма полезного средства, но при наличии существенно более широких возможностей, подобных,

например, пакету ***Mathematica*** [6,7], где подобный механизм представляется нам намного более развитым.

Приоритетность операторов ***Maple***-языка в порядке убывания определяется как:

|| :- :: % & ! {^, @@} { ., *, &*, /, @, intersect} {+, -, union, minus} mod subset .. {<, <=, >, >=, =, <>, in} $ not and or xor implies -> , assuming :=

В качестве *строк* ***Maple***-язык рассматривает любые последовательности символов, кодируемые в *верхних двойных кавычках* ("), например**: "Академия Ноосферы".** При этом, составлять строку могут *любые* допустимые *синтаксисом* языка символы. При необходимости поместить в *строку* верхнюю двойную кавычку ее следует *дублировать* либо вводить комбинацией вида (\"). Это же относится и к ряду других символов, вводимых посредством *обратного* слэша (\), как это иллюстрирует следующий весьма простой фрагмент**:**

```
> `Строка`:= "Международная Академия Ноосферы";
Строка:= "Международная Академия Ноосферы"
> `Строка 1`:= "Международная\nАкадемия\nНоосферы";
Строка 1 := "Международная
    Академия
    Ноосферы"
> `Строка 2`:= "Internatio\nal Academy of Noosphere";
Строка 2 := "Internatio
   nal Academy of Noosphere"
> `Строка 3`:= "Российская Эколо"гическая Академия";
Error, missing operator or `;`
```

Однако, как иллюстрируют примеры фрагмента, если наличие в строке *одинарной двойной* кавычки (*помимо ограничивающих*) вызывает *синтаксическую* ошибку**,** то для случая *обратного* слэша в общем случае могут возникать *семантические* ошибки. Максимальная длина строки зависит от используемой платформы: для 32-битных **ЭВМ -** 524271 символов, а для 64-битных **-** 34359738335 символов. О средствах работы со *строчными* структурами детальнее речь будет идти несколько ниже.

В качестве *символов* ***Maple***-язык рассматривает *любые* последовательности символов, в общем кодируемые в *верхних обратных кавычках* (`), например**: `Академия Ноосферы 64`.** При этом, составлять *символ* могут *любые* допустимые синтаксисом языка символы. При необходимости поместить в *символ* верхнюю обратную кавычку ее следует дублировать либо вводить комбинацией вида (\`). Это же относится и к ряду других символов, вводимых посредством *обратного* слэша (\), как это иллюстрирует фрагмент**:**

```
> `Строка`:= `Международная Академия Ноосферы`;
Строка:= Международная Академия Ноосферы
> `Строка 1`:= `Международная\nАкадемия\nНоосферы`;
Строка 1 := Международная
    Академия
    Ноосферы
> `Строка 2`:= `Internatio\nal Academy of Noosphere`;
Строка 2 := Internatio
   nal Academy of Noosphere
> `Строка 3`:= `Российская Эколо`гическая Академия`;
Error, missing operator or `;`
```

Сразу же следует отметить одно принципиальное отличие *строк* от *символов* ***Maple***-языка. Если символам можно присваивать значения, то строки такой процедуры не допус-

кают, вызывая ошибочные ситуации, например:

```
> "Данные":= 2006;
Error, invalid left hand side of assignment
> Данные:= 2006: `Результат`:=350: Данные, Результат;
                                2006, 350
```

При этом, если символ не содержит *специальных* знаков, то его можно кодировать и без ограничивающих его обратных одинарных кавычек, как иллюстрирует второй пример.

В качестве *натурального целого* язык рассматривает любую последовательность из одной или более десятичных цифр, при этом все ведущие нули игнорируются, т.е. **005742** рассматривается как **5742.** Длина таких чисел зависит от используемой пакетом *платформы* и на большинстве 32**-**битных **ЭВМ** составляет порядка 524280 десятичных цифр**,** тогда как на 64-битных она достигает уже 38654705646 цифры. В качестве *целого* рассматривается натуральное целое со знаком или без. Для работы с арифметикой целых чисел язык располагает достаточно развитыми средствами, рассматриваемыми ниже.

Наконец, знаки *пунктуации* ***Maple-***языка представлены в следующей табл. **4.**

Таблица 4

Знак пунктуации:	***Смысловая нагрузка:***
; и **:**	*точка с запятой* и *двоеточие*
(и **)**	левая и правая *круглые* скобки
< и **>**	левая и правая *угловые* скобки
{ и **}**	левая и правая *фигурные* скобки
[и **]**	левая и правая *квадратные* скобки
"	*двойная верхняя* кавычка
|	*вертикальный* разделитель
` ' , .	*кавычка, апостроф, запятая* и *точка*

Представим использование ***Maple-***языком указанных в табл. **4** знаков пунктуации**:**

: и **;** **-** служат для идентификации конца *предложений* ***Maple-***языка**;** различия между ними обсуждаются ниже**;**

() **-** для группировки термов в выражениях, а также формальных или фактических аргументов при определениях или вызовах функций/процедур**;**

<> **-** для определенной пользователем *группировки* выражений**;**

{} **-** для определения структур данных типа *множество***;**

[] **-** для определения структур данных типа *список*, а также для образования индексированных переменных и оператора *выбора* элемента из индексированных выражений**;**

` ' , **-** соответственно для определения *идентификаторов*, *невычисляемых* выражений и структур типа *последовательность***;** при этом, следует четко различать при кодировании конструкций ***Maple-***языка символы *верхней обратной кавычки* (**96**) и *апострофа* (**39**), в скобках даны их *десятичные* коды по *внутренней* кодовой таблице**;**

" **-** *верхние двойные кавычки* служат для определения *строчных* структур данных**.**

Для разделения *лексемов* синтаксис языка допускает использование как знаков *пунктуации*, так и *пробелов***,** понимаемых в расширенном смысле, т.е. в качестве пробелов допускается ис-ользование символов**:** собственно *пробела* (*space*), *табуляции* (*tab*), *перевода строки* и *возврата каретки*. При этом, сами символы пробела не могут входить в состав *лексемов*, являясь их разделителями. Между тем**,** *space***-**символ может входить в состав *строк* и

символов, образованных *двойными* или *одинарными верхними* кавычками. Использование же его в составе *лексема*, как правило, инициирует ошибочную ситуацию, например**:**

```
> AGN: = sqrt(x^2 + y^2 + z^2 + 19.47);
Error, `=` unexpected
```

Под *программной* строкой в ***Maple-***языке понимается строка символов**,** завершающаяся символами *перевода строки* и *возврата каретки* (16-*ричные коды* **0D0A**); при этом, сами эти символы строке не принадлежат. *Пустые* программные строки образуются простым нажатием ***Enter-***клавиши. Завершение программной строки по ***Enter-***клавише в интерактивном режиме вызывает *немедленное* вычисление всех содержащихся в ней выражений и предложений языка. Если в *программной* строке содержится **#-***символ***,** то язык рассматривает всю последующую за ним информацию в качестве *комментария* и обработки ее не производит**.** Этот прием можно использовать для комментирования текстов ***Maple-***программ**,** тогда как в интерактивном режиме особого смысла он не имеет. Вывод длинных строк производится в несколько строк экрана, идентифицируя символом *обратного слэша* (\) продолжение строки. При этом символ обратного слэша несет более широкую смысловую нагрузку, а именно**:** наряду с функцией *продолжения* он может выступать в качестве *пассивного* оператора и средства ввода *управляющих* символов.

В *первом* качестве он используется**,** как правило**,** для *разбиения* на *группы* строчных структур или *цифровых* последовательностей в целях более удобного их восприятия. Тогда как во *втором* случае он позволяет вводить *управляющие символы*, производящие те или иные действия. В этом случае кодируется конструкция следующего вида**:** "**\<***управляющий символ***>**"**,** где управляющий символ является одним из следующих восьми {**a, b, e, f, n, r, t, v**}**;** например, по конструкции "**\n**" производится *перевод строки*, а по "**\a**" **–** звонок, с другими детально можно ознакомиться по книге [10] либо по ***Help-***системе пакета посредством предложения вида: **> ?*backslash*.** Следующий фрагмент иллюстрирует использование символа *обратного слэша* в указанных выше контекстах**:**

```
> A:= `aaaaaaaaaaa\nbbbbbbbbbbbb\nccccccccccccc`;   # Перевод строки
A := aaaaaaaaaaa
     bbbbbbbbbbb
     ccccccccccccc
> A:= `aaaaaaaaaa\a\nbbbbbbbbbbb\a\nccccccccccccc`; # Звонки с переводом строки
A := aaaaaaaaaa●
     bbbbbbbbbbb●
     ccccccccccc
> 1942.1949\1959\1962\1966\1972\1995\1999\2006;
                    1942.1949195919621966197219951999 2006
```

Употребление *обратного* слэша без следующего за ним одного из указанных управляющих символов в середине лексема приводит к его игнорированию**,** т.е. он выступает в качестве *пустого* оператора. Для действительного ввода в строку *обратного слэша* его нужно кодировать *сдвоенным*. При этом, *следует* иметь в виду, что в случае кодирования *спецификаторов* файлов в функциях доступа в качестве *разделителей* подкаталогов можно использовать *сдвоенные* обратные слэши (\\) или *одинарные* прямые слэши (/), в противном случае возможно возникновение *семантических* ошибок. По конструкции вида ***kernelopts*(*dirsep*)** возвращается стандартный *разделитель подкаталогов*, принимаемый по умолчанию, однако его переопределение невозможно.

После успешной загрузки пакета ***Maple*** производится выход на его *главное* окно, структура и назначение компонент которого детально рассмотрены, например, в [9-14]**.** В области текущего документа в первой строке устанавливается **>-***метка ввода*, идентифи-

цирующая запрос на ввод информации, как правило, завершающийся {**;**|**:**}-*разделителем* с последующим нажатием ***Enter***-клавиши либо активацией **!**-*кнопки* **4**-й *строки* главного окна. Результатом данной процедуры является передача пакету *программной строки*, содержащей отдельное ***Maple***-выражение, вызов функции либо несколько предложений языка. Под ***Maple***-*программой* (*документом*) понимается *последовательность* предложений языка, описывающая алгоритм решаемой задачи, и выполняемая, если не указано противного, в *естественном* порядке следования своих предложений.

В *интерактивном* режиме ввод программной строки производится непосредственно за **>**-меткой ввода (***Input***-*параграф текущего документа*) и завершается по клавише ***Enter***, инициирующей синтаксический контроль введенной информации с последующим вычислением (*если не обнаружено синтаксических ошибок*) *всех* входящих в строку ***Maple***-предложений. В дальнейшем мы ради удобства представления иллюстративных примеров программную строку и результат ее вычисления будем иногда представлять в следующем достаточно естественном формате:

> ***Maple-предложение*** **{:|;}** $\Rightarrow$ ***Результат вычисления***

При этом, в зависимости от завершения ***Maple***-*предложения* {**;**|**:**}-*разделителем* результат его вычисления соответственно {*выводится*|*не выводится*} в *текущий* документ, т. е. {*формируется*|*не формируется*} ***Output***-параграф. Однако следует иметь в виду, что «*не формируется*» вовсе не означает «*не возвращается*» **–** результат выполнения ***Input***-*параграфа* всегда возвращается и его можно получать, например, по нульарному %-оператору, как это иллюстрирует следующий весьма простой фрагмент:

```
> Tallinn:= 2006;  ⇒ Tallinn := 2006
> %;  ⇒ 2006
> Tallinn:= Grodno:  %;  ⇒ Grodno
```

Однако здесь следует иметь в виду, что может возвращаться и *NULL*-значение, т.е *ничего*. В этом случае %-оператор возвращает последний результат, отличный от *NULL*, как это иллюстрирует следующий простой пример:

> x:= 5: assign('y' = 64); assign('z' = 59); assign('s' = 39); %; $\Rightarrow$ 5

Использование (:)-разделителя служит, главным образом, для того, чтобы избавиться в ***Maple***-документе от *ненужного* вывода промежуточной и несущественной информации. *Структурная* организация ***Maple***-документов достаточно детально была рассмотрена в [9-14,78-89]. Здесь мы лишь отметим особые типы программных строк, начинающихся с двух *специальных* управляющих {**?, !**}-символов ***Maple***-языка пакета.

Использование в самом начале *программной* строки в качестве *первого*, отличного от *пробела*, **?**-символа рассматривается ***Maple***-языком как инструкция о том, что следующая за ним информация **-** фактические аргументы (*последовательность которых разделяется запятой* "**,**" *или прямым слэшем* "**/**") для ***help***-процедуры, обеспечивающей вывод справочной информации по указанным аргументами средствам пакета. Например, по конструкции формата "**> ?integer**" выводится справка ***integer***-раздела ***Help***-системы пакета.

Использование в самом начале *программной* строки в качестве *первого*, отличного от *пробела*, **!**-символа рассматривается ***Maple***-языком как инструкция о том, что следующая за ним информация предназначена в качестве *команды* для *ведущей* **ЭВМ**. Однако данная возможность поддерживается не всеми операционными платформами, например, для ***Windows***-платформы в данном случае идентифицируется синтаксическая ошибка. Рассмотрев *базовые* элементы ***Maple***-языка, переходим к более сложным его конструкциям.

1.2. *Идентификаторы, предложения присвоения и выделения Maple-языка*

Основной базовой единицей языка является *предложение*, представляющее собой любое допустимое ***Maple***-выражение, завершающееся {**;**|**:**}-разделителем. Более того, не нарушая синтаксиса ***Maple***-языка, под ***предложением*** будем понимать любую допустимую ***Maple***-конструкцию, завершающуюся {**;**|**:**}-*разделителем*; при этом, предложение может завершаться и по ***Enter***-клавише (*символы перевод строки и возврат каретки*), если оно содержит единственное выражение. В ряде случаев допустимо использование в качестве разделителя ***Maple***-предложений даже запятой, если они находятся внутри программной строки, что позволяет выводить результаты вычислений в строчном (*разделенном запятой*) формате. Однако это требует *весьма* внимательного подхода, как к *нетипичному приему*. Следующий фрагмент иллюстрирует все перечисленные способы кодирования ***Maple***-предложений:

```
> V:= 64:  G:= 59:  S:= 39:  Art:= 17:  Kr:= 10:  V ,G, S, Art, Kr;
                                  64, 59, 39, 17, 10
> ?HelpGuide                          # вывод справки по HelpGuide
> R := evalf(sqrt(Art^2 + Kr^2 + V^2 + G^2 + S^2), 6)      Z := sqrt(Art^2 + Kr^2 + V^2 + G^2 + S^2)
                                  R := 97.4012
                                  Z := sqrt(9487)
> if 17 <= evalf(sqrt(Art^2 + Kr^2)) then evalf(89/Art) else evalf(96/Kr) end if
                                  5.235294118
> V:= 64:  G:= 59:  Art:= 17:  Kr:= 10:  V, G, S, Art, H:= 2006:  Kr;
Error, cannot split rhs for multiple assignment
                                  10
> assign('V', 42), assign('G', 47), assign('Art', 89), assign('Kr', 96), V, G, Art;
                                  42, 47, 89
```

Как следует из двух последних примеров фрагмента, запятую в качестве разделителя предложений можно использовать лишь для разделения выражений. Тогда как третий и четвертый примеры иллюстрируют ввод предложений в стандартной математической нотации.

Согласно сказанному ***Maple***-язык оперирует предложениями различных типов, определяемых типом конструкции, завершающейся *рассмотренным* выше способом. Ниже данный вопрос получит свое дальнейшее развитие. Однако, прежде всего нам необходимо определить такую конструкцию как *выражение*, состоящее из ряда более простых понятий. В *первую* очередь, рассмотрим понятие *идентификатора*, играющего одну из *ключевых* ролей в организации вычислительного процесса в среде ***Maple***-языка аналогично случаю других современных языков программирования.

Идентификаторы. В терминологии пакета под *символами* (*symbol*) понимаются как собственно цепочки символов, так и *идентификаторы*, удовлетворяющие соглашениям пакета, т.е. *имена* для всех конструкций ***Maple***-языка пакета. *Идентификаторы* служат для установления связей между различными компонентами вычислительного процесса как логических, так и информационных, а также для образования выражений и других вычислительных конструкций. В качестве *идентификатора* в ***Maple***-*языке* выступает цепочка из не более, чем 524271 символов для 32-битной платформы и не более 34359738335

символов для 64-битной платформы, начинающаяся с буквы либо символа *подчеркивания* (_). Идентификаторы *регистро-зависимы*, т.е. одинаковые буквы на верхнем и нижнем регистрах клавиатуры полагаются различными. Данное обстоятельство может служить, на первых порах, источником синтаксических и семантических ошибок, ибо в большинстве современных **ПС** идентификаторы, как правило, *регистро-независимы*. В качестве примеров простых идентификаторов можно привести следующие:

AVZ, Agn, Vs_A_K, Ar_10, KrV, Tall_Est, Salcombe_Eesti_Ltd_99, Vasco_97

Для возможности определения русскоязычных идентификаторов либо идентификаторов, содержащих *специальные* символы, включая *пробелы*, их следует кодировать в верхних кавычках, как это иллюстрирует следующий простой пример:

```
> `ТРГ=TRG_9`:= 64: `Значение 1`:= 1942: evalf(`Значение 1`/`ТРГ=TRG_9`, 12);
                                  30.3437500000
> Таллинн:= 56: Гродно:= 84: Вильнюс:= 100: Таллинн + Гродно + Вильнюс;
                                  240
```

В принципе, простые *русскоязычные* идентификаторы могут кодироваться и без *кавычек*, однако во избежание возможных недоразумений рекомендуется для них использовать кавычки. В этом случае в качестве идентификатора может выступать произвольная цепочка символов, что существенно расширяет выразительные возможности ***Maple***-языка, позволяя в ряде случаев переносить функции комментария на идентификаторы конструкций языка.

Наряду с такими простыми конструкциями *идентификаторов* ***Maple***-язык пакета допускает и более сложные, определяемые несколькими путями. Прежде всего, идентификаторы, начинающиеся с комбинации символов (***_Env***) полагаются ядром *сеансовыми*, т. е. их действие распространяется на весь *текущий сеанс* работы с пакетом. Так как они служат, прежде всего, для изменения пакетных *предопределенных* переменных, то их использованию следует уделять особое внимание. По конструкции ***anames*('*environment*')** можно получать в текущем сеансе все активные пакетные переменные:

```
> anames('environment');
```

Testzero, UseHardwareFloats, Rounding, **%, %%%,** *Digits, index/newtable, mod,* **%%,** *Order, printlevel, Normalizer, NumericEventHandlers*

Ниже данный тип переменных будет нами рассматриваться несколько детальнее. Для обозначения определенного типа *числовых* значений ***Maple***-язык использует целый ряд *специальных* идентификаторов таких, как: ***_N*, *_NN*, *_NNp*, *_Z~*, *_Zp~*, *_NN~*** и др. Например, ***_Zp~*** и ***_NN~*** используются для обозначения целых и целых неотрицательных чисел, тогда как ***_Cp***-переменные используются для обозначения постоянных интегрирования и т.д.

Пакетные переменные такие как ***Digits*, *Order*, *printlevel*** имеют *приписанные* им по умолчанию значения соответственно **[10, 6, 1]**, которые можно переопределять в любой момент времени. Однако, если после выполнения **restart**-предложения, восстанавливающего исходное состояние ядра пакета, сеансовые переменные становятся неопределенными, то пакетные переменные восстанавливают свои значения по умолчанию. Более сложные виды идентификаторов, включающие целые фразы как на английском, так и на национальных языках, можно определять, кодируя их в верхних *обратных* кавычках; при этом, внутри ограничивающих кавычек могут кодироваться любые символы (*при использовании верхних кавычек они дублируются*). В случае простого **R**-идентификатора конструкции **R** и **`R`** являются эквивалентными, а при использовании ключевых слов

Maple-языка (*указанных в* **1.1**) в качестве идентификаторов они должны кодироваться в верхних обратных кавычках.

Пустой символ, кодируемый в виде ``, также может использоваться в качестве идентификатора, например:

> ``:= 64: ``; ⇒ 64

однако по целому ряду соображений этого делать не следует, то же относится и к символам ` `, `  `, `  ` и т.д. В противном случае могут возникать ошибочные и непредсказуемые ситуации. Вместе с тем, *пустой символ* (*как и строка*) отличен от значения глобальной ***NULL***-переменной пакета, определяющей отсутствие выражение, т.е. ничего.

В последующем мы все более активно будем использовать понятие ***функции,*** поэтому здесь на содержательном уровне определим его. В ***Maple***-языке функция реализует определенный алгоритм обработки либо вычислений и возвращает результат данной работы в точку своего вызова. Каждая функция идентифицируется уникальным *именем* (*идентификатором*) и ее вызов производится по конструкции следующего формата:

Имя*(Последовательность фактических аргументов***)**

где передаваемые ей фактические аргументы определяют исходные данные для выполнения алгоритма функции и возвращаемый ею результат. Детально вопросы организации и механизма функциональных средств ***Maple***-языка рассматриваются ниже.

Вторым способом определения сложных идентификаторов является кодирование объединяющих их составные части ||-оператора *конкатенации* и/или встроенной ***cat***-функции конкатенации строк, имеющей простой формат кодирования ***cat*(x, y, z, t, ...)** и возвращающей объединенную строку (*символ*) **xyzt...** , где: **x, y, z, t … -** строки, символы или ||-*операторы* (*при этом,* ||-*оператор не может быть первым или последним*), как это иллюстрирует следующий весьма простой фрагмент:

> cat(111, aaa||bbb, ccc), cat("111", aaa||bbb, ccc||ddd), `111`||cat(aaa, bbb)||"222";
111aaabbbccc, "111aaabbbcccddd", (111cat(aaa, bbb)) || "222"
> `11111`||22222, "11111"||22222, cat(`11111`, 22222), cat("11111", 22222);
1111122222, "1111122222", *1111122222*, "1111122222"
> 11111||22222;
Error, `||` unexpected
> cat(11111, 22222); ⇒ *1111122222*

При этом, как показывает второй пример фрагмента, тип возвращаемого в результате конкатенации нескольких аргументов значения определяется типом *первого* аргумента, а именно: если первый аргумент является строкой (*символом*), то и результат конкатенации будет строкой (*символом*). Это справедливо как для ***cat***-функции, так и для **||**-оператора. Вообще говоря, имеет место следующее определяющее соотношение для обоих методов конкатенации, а именно:

***cat*(x1, x2, ..., xj, ..., xn) = x1||x2|| ... ||xj|| ... xn**

Результат конкатенации должен использоваться в *кавычках*, если составляющие его компоненты включали *специальные* символы. Полученные в результате конкатенации символы используются в дальнейшем как *единые* идентификаторы или просто символьные (*строчные*) значения. В принципе, допуская конкатенацию произвольных ***Maple***-выражений, функция ***cat*** и **||**-*оператор* имеют ряд ограничений, в частности, при использовании их в качестве аргументов и операндов функциональных конструкций, которые могут инициировать некорректные (*а в ряде случаев и непредсказуемые*) результаты:

> cat(ln(x), sin(x), tan(x)); ⇒ || (ln(x)) || (sin(x)) || (tan(x))

```
> ``||F(x)||G(x), " "||F(x)||G(x);   ⇒ ( F(x)) || G(x), (" F"(x)) || G(x)
> cat(F(x), G(x)), H||evalf(Pi);   ⇒ || (F(x)) || (G(x)), Hevalf(Pi)
> H||(evalf(Pi));   ⇒ H || (3.141592654)
> cat(`sin(x)`, "cos(x)");   ⇒ sin(x)cos(x)
> [A||max(59, 64), A||(max(59, 64))];   ⇒ [Amax(59, 64), A64]
```

При этом, как следует из *последних* примеров предпоследнего фрагмента, ***cat***-функция является более универсальным средством, чем **||**-*оператор*. В любом случае данные средства рекомендуется использовать, как правило, относительно строчных либо символьных структур, апробируя допустимость их в других более общих случаях. Это связано и с тем обстоятельством, что **||**-оператор конкатенации имеет максимальный приоритет, поэтому второй операнд может потребовать круглых скобок, как показано выше. С другой стороны, это может способствовать расширению в качестве операндов типов. Детальнее вопрос использования средств конкатенации рассматривается ниже.

Идентификаторы ключевых слов ***Maple***-языка кодируются в верхних кавычках при использовании их в качестве аргументов функций либо обычных переменных, однако по целому ряду соображений последнего делать не рекомендуется. Не взирая на возможность использования в качестве идентификаторов *произвольных* символов, использовать ее рекомендуется только в случае необходимости контекстного характера, ибо в противном случае выражения с ними становятся мало обозримыми. В качестве идентификаторов допускается использование и русских имен и выражений, однако их следует как при определении, так и при использовании кодировать в верхних обратных кавычкам.

Наряду с рассмотренными типами ***Maple***-язык допускает использование *индексированных* идентификаторов, имеющих следующий формат кодирования:

Идентификатор*[Последовательность индексов***]**

при этом, в отличие от традиционных языков программирования, *индексированный* идентификатор не означает принадлежности его к массиву, обозначая просто индексированную переменную, как это иллюстрирует следующий простой фрагмент:

```
> A:= 64*V[1, 1] + 59*V[1, 2] - 10*V[2, 1] - 17*V[2, 2];
```

$$A := 64\,V_{1,1} + 59\,V_{1,2} - 10\,V_{2,1} - 17\,V_{2,2}$$

```
> AG[47][59]:= 10*Kr[k][96] + 17*Art[j][89];
```

$$AG_{47_{59}} := 10\,Kr_{k_{96}} + 17\,Art_{j_{89}}$$

Однако, при условии **V**-*массива* **V[k, h]**-переменная идентифицирует его (**k, h**)-элемент. И так как индексированность сохраняет свойство быть идентификатором, то его можно последовательно индексировать, как это иллюстрирует последний пример фрагмента.

На основе *индексированной* переменной базируется и предложение *выделения*, кодируемое подобно первой в следующем простом формате:

Идентификатор*[Последовательность индексов***] {;|:}**

возвращающее элемент структуры с указанным идентификатором и заданными значениями индексов. Данное предложение имеет смысл только для структур, в которых можно выделить составляющие их элементы (*списки*, *множества*, *массивы*, *матрицы* и др.). Следующий простой пример иллюстрирует применение предложения выделения:

```
> L:= [59, 64, 39, 10, 17]:  S:={V, G, Sv, Art, Kr}:  L[3], S[5], S[3], L[4];
```

$$39,\ Sv,\ Art,\ 10$$

Наконец, идентификатор функции/процедуры кодируется в следующем формате:

Идентификатор*(Последовательность фактических аргументов***)**

определяя вызов функции с заданным именем с передачей ей заданных фактических аргументов, например: **sin(20.06);** ⇒ 0.9357726776.

Для идентификаторов любых конструкций ***Maple***-языка допускается использование *алиасов* (*дополнительных имен*), позволяющих обращаться к конструкциям как по их *основным* именам, так и по дополнительным. Данный механизм языка обеспечивается встроенной ***alias***-функцией, кодируемой в следующем формате:

***alias*(*alias*_1=*Id*, *alias*_2=*Id*, ... , *alias*_n=*Id*)**

где ***Id* -** основное имя, а ***alias*_j -** присваиваемые ему алиасы (j=1..n), например:

```
> G:= `AVZ`:  alias(year = 'G', Grodno = 'G', `ГрГУ` = 'G');   ⇒ year, Grodno, ГрГУ
> [G, year, Grodno, `ГрГУ`];   ⇒ [AVZ, AVZ, AVZ, AVZ]
> alias(e = exp(1));   ⇒ year, Grodno, ГрГУ, e
> evalf(e);   ⇒ 2.718281828
```

В приведенном фрагменте проиллюстрировано, в частности, определение более привычного **e**-алиаса для основания натурального логарифма, вместо принятого (*на наш взгляд не совсем удачного*) в пакете ***exp*(1)**-обозначения. *Алиасы* не допускаются только для числовых констант; при этом, в качестве ***Id***-параметров ***alias***-функции должен использоваться невычисленный идентификатор (*т.е. кодируемый в апострофах*), а не его значение. Если вызов ***alias***-функции завершается (**;**)-разделителем, то возвращается последовательность всех на текущий момент присвоенных алиасов сеанса работы с ядром пакета. Механизм алиасов имеет немало интересных приложений, детальнее рассматриваемых в наших книгах [10-12,91].

Близкой по назначению к ***alias***-функции является и функция ***macro*** формата:

***macro*(X1 = Y1, ..., Xn = Yn)**

возвращающая ***NULL***-значение, т.е. ничего, и устанавливающая на период сеанса работы с ядром пакета односторонние соотношения **X1 ⇒ Y1, ..., Xn ⇒ Yn**. Точнее, любое вхождение **Xj**-конструкции в ***Input***-параграфе либо при чтении ее из файла замещается на приписанную ей по ***macro***-функции **Yj**-конструкцию. Исключением является вхождение **Xj**-конструкций в качестве *формальных* аргументов и *локальных* переменных процедур. В данном отношении ***macro*-*функция*** отличается от традиционного понятия *макроса* и более соответствует *однонаправленному* алиасу. Функция ***macro*** может быть определена для любой ***Maple***-конструкции, исключая числовые константы. Более того, фактические аргументы ***macro***-функции не вычисляются и не обрабатываются другими ***macro***-функциями, не допуская рекурсивных ***macro***-определений. Для изменения определения ***macro*** вполне достаточно выполнить новый вызов ***macro***-функции, в которой правые части уравнений имеют другое содержимое. Тогда как для отмены ***macro***-определения достаточно произвести соответствующий вызов функции ***macro*(Xp = Yp)**. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> restart;  macro(x = x*sin(x), y = a*b+c, z = 56):
> HS:= proc(x) local y;  y:= 42:  y*x^3  end proc:
> macro(HS=AGN, x = x, y = 47, z = 99):  map(HS, [x, y, z]);
                                  [AGN(x), AGN(47), AGN(99)]
> HS:= proc(x) local y;  y:= 42:  y*10^3  end proc:
> map(HS, [x, y, z]);   ⇒ [42000, 42000, 42000]
> macro(HS=AGN, x = x, y = 47, z = 99):  map(HS, [x, y, z]);
                                  [AGN(x), AGN(47), AGN(99)]
> restart:  HS:= proc(x) local y;  y:= 42:  y*x^3  end proc:  HS(1999);
                                      335496251958
```

```
> macro(HS=AGN):   [AGN(1999), HS(1999)];
                                   [AGN(1999), AGN(1999)]
> alias(AGN=HS):   [AGN(1999), HS(1999)];
Warning, alias or macro HS defined in terms of AGN
                                   [335496251958, AGN(1999)]
```

Из примеров фрагмента, в частности, следует вывод о необходимости достаточно внимательного использования ***macro***-функции для идентификаторов процедур, ибо их переопределение приводит к неопределенности нового идентификатора со всеми отсюда вытекающими последствиями. При этом, в целом, ситуацию не исправляет и последующее использование ***alias***-функции. На это следует обратить особое внимание.

Определение переменной в *текущем документе* (**ТД**) является *глобальным*, т.е. *доступным* любому другому **ТД** в течение *текущего* сеанса работы с ядром пакета. Сказанное не относится к режиму *параллельного сервера*, когда все загруженные документы являются независимыми. Режим параллельного сервера детально рассмотрен в [9-12]. При этом, определение считается сделанным только после его реального вычисления. После *перезагрузки* пакета все определения переменных и пользовательских функций/процедур (*отсутствующих в библиотеках*, *логически сцепленных с главной библиотекой пакета*) теряются, требуя *нового* переопределения. Без перезагрузки пакета этого можно добиться по *предложению* **restart,** приводящему все установки ядра пакета в исходное состояние (*очистка* **РОП,** *отмена всех сделанных ранее определений*, *выгрузка всех загруженных модулей и т.д.*), либо присвоением идентификаторам переменных невычисленных значений вида ***Id*:='*Id*'**. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> x:= 19.42:  y:= 30.175:  Grodno:= sin(x) + cos(y);   ⇒ Grodno := 0.8639257079
> restart;   Grodno:= sin(x) + cos(y);   ⇒ Grodno := sin(x) + cos(y)
> G:= proc() nargs end proc:   G(42, 47, 67, 62, 89, 96);   ⇒ 6
> G:= 'G':   G(42, 47, 67, 62, 89, 96);   ⇒ G(42, 47, 67, 62, 89, 96)
```

Из фрагмента хорошо видно, что ранее сделанное определение ***Grodno***-переменной отменяется после выполнения **restart**-предложения. Выполнение **restart**-предложения следует лишь на *внешнем* уровне **ТД** и не рекомендуется внутри ряда его конструкций (*функции, процедуры и др.*) во избежание возникновения *особых* и *аварийных* ситуаций, включая "*зависание*" пакета, требующее *перезагрузки* **ПК**. При этом, следует иметь в виду, что освобождаемая в этом случае память не возвращается операционной среде, а присоединяется к собственному пулу свободной памяти пакета. Поэтому при необходимости получения *максимально* возможной памяти для решения больших задач пользователю рекомендуется все же производить *перезагрузку* пакета в ***Windows***-среде. Тогда как второй способ отмены определенности для переменных более универсален. В книге [103] и в прилагаемой к ней *Библиотеке* представлены средства, обеспечивающие возможность восстановления из процедур исходного состояния объектов. Например, вызов процедуры ***prestart*()** очищает все переменные, определенные в текущем сеансе, исключая пакетные переменные.

Предложение присвоения. *Идентификатору* может быть присвоено любое допустимое ***Maple***-выражение, делающее его *определенным*; в противном случае идентификатор называется *неопределенным*, результатом вычисления которого является символьное представление самого идентификатора, что весьма прозрачно иллюстрирует следующий простой пример:

```
> macro(Salcombe = Vasco):  Eesti:= 19.95:  Vasco:= Noosphere:  Tallinn:= 20.06:
> TRG:= sqrt(Lasnamae*(Eesti + Tallinn)/(Salcombe + Vasco)) + `Baltic Branch`;
```

$$TRG := 4.472694937\sqrt{\frac{Lasnamae}{Noosphere}} + Baltic\ Branch$$

Присвоение идентификатору определенного или неопределенного значения производится посредством традиционного (**:=**)-оператора *присвоения* вида **A:= B**, присваивающего левой **A**-части **B**-значение. При этом, в качестве левой **A**-части могут выступать простой идентификатор, индексированный идентификатор или идентификатор функции с аргументами. Точнее, присвоение **A**-части **B**-значения корректно, если **A** имеет ***assignable***-тип, т.е. **type(A, 'assignable');** ⇒ *true*. Вычисленное (*или упрощенное*) значение **B**-части присваивается идентификатору **A**-части.

Оператор *присваивания* допускает возможность *множественного* присваивания и определяется конструкциями следующего простого вида:

***Id1*, *Id2*, …, *Idn* :=** <*Выражение*_1>, <*Выражение*_2>, …, <*Выражение*_n>

При этом, при *равном* количестве компонент правой и левой частей присвоения производятся на основе взаимно-однозначного соответствия. В противном случае инициируются ошибочные ситуации "*Error, ambiguous multiple assignment*" либо "*Error, cannot split rhs for multiple assignment*". Следующий фрагмент иллюстрирует случаи множественного присваивания:

```
> A, B, C:= 64, 59:
Error, ambiguous multiple assignment
> V, G, S, Art, Kr, Arn:= 64, 59, 39, 17, 10, 44:  [V, G, S, Art, Kr, Arn];
                                [64, 59, 39, 17, 10, 44]
> x, y, z, t, h:= 2006: [x, y, z, t, h];
Error, cannot split rhs for multiple assignment
                                [x, y, z, t, h]
```

Примеры фрагмента достаточно прозрачны и *особых* пояснений не требуют. Сам принцип реализации множественного присваивания также достаточно прост. Вместе с тем, языком не поддерживаются уже достаточно *простые* конструкции множественного присваивания, что иллюстрируют первый и последний примеры фрагмента.

Между тем, в ряде случаев возникает необходимость назначения того же самого выражения достаточно длинной последовательности имен или запросов функций. Данная проблема решается оператором **`&ma`**, который имеет идентичный с оператором **`:=`** приоритет. Оператор **`&ma`** имеет два формата кодирования, а именно: *процедурный* и *операторный* форматы:

&ma('x', 'y', 'z', ..., <*rhs*>) и **('x', 'y', 'z', ...) &ma (<*rhs*>)**

В общем случае, в обоих случаях в конструкции ***lhs* &ma *rhs*** элементы ***lhs*** должны быть закодированы в невычисленном формате, т.е. в *апострофах* ('). Исключение составляет лишь первый случай присвоения. 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 также кодироваться в скобках. Наконец, если необходимо присвоить ***NULL***-значение (*т. е. ничего*) элементам левой части ***lhs***, то в качестве правой части ***rhs*** кодируется ***_NULL***-значение. Успешный вызов процедуры **`&ma`** или применения оператора **&ma** возвращает ***NULL***-значение, т. е. ничего с выполнением указанных присвоений. В целом ряде приложений оператор/процедура **&ma** представляются достаточно полезными [103]. Ниже приведен ряд примеров на применение оператора **&ma**:

Процедурный формат оператора:

```
> &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', 2006); x, y, z, g, h; ⇒ 2006, 2006, 2006, 2006, 2006

> &ma('x', 'y', 'z', 'g', 'h', 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)

Операторный формат:

> ('x', 'y', 'z', 'g', 'h') &ma _NULL; x, y, z, g, h;

> ('x', 'y', 'z', 'g', 'h') &ma 2006; x, y, z, g, h; ⇒ 2006, 2006, 2006, 2006, 2006

> ('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;

$$\frac{a+b}{c-d}, \frac{a+b}{c-d}, \frac{a+b}{c-d}, \frac{a+b}{c-d}, \frac{a+b}{c-d}$$

Для проверки идентификатора на предмет его *определенности* используется встроенная функция ***assigned*** языка, кодируемая в виде ***assigned*(***Идентификатор***)** и возвращающая значение *true* в случае определенности идентификатора (*простого*, *индексированного или вызова функции/процедуры*), и *false*-значение в *противном* случае. При этом, следует иметь в виду, что *определенным* идентификатор полагается тогда, когда он был использован в качестве левой части (**:=**)-оператора *присвоения*, даже если его правая часть являлась неопределенной. Или он получил присвоение по ***assign***-процедуре. Следующий простой фрагмент иллюстрирует вышесказанное:

> agn:=1947: avz:=grodno: assign(vsv=1967, art=kr): seq(assigned(k), k=[agn, avz, vsv, art]);

true, *true*, *true*, *true*

> map(type,[agn, avz, vsv, art],'assignable'); ⇒ [*false*, *true*, *false*, *true*]

> map(assigned, [agn, avz, vsv, art]);

Error, illegal use of an object as a name

С другой стороны, по конструкции ***type*(*Id*, 'assignable')** мы можем тестировать *допустимость* присвоения ***Id***–переменной (*простой*, *индексированной или вызова функции/процедуры*) выражения: возврат *true*-значения говорит о такой возможности, *false* – нет. Следует обратить внимание на *последний* пример фрагмента, иллюстрирующий *некорректность* использования встроенной функции ***map*** при попытке организации простого цикла.

Вызов функции ***indets*(W** {**,** <*Тип*>**})** возвращает множество *всех* идентификаторов заданного его *первым* фактическим **W**-аргументом ***Maple***-выражения. При этом, **W**-*выражение* рассматривается функцией *рациональным*, т.е. образованным посредством {**+**, **-**, *****, **/**}-операций. Поэтому в качестве результата могут возвращаться не только простые идентификаторы **W**-выражения, но и неопределенные его подвыражения. В случае кодирования *второго* необязательного аргумента, он определяет *тип* возвращаемых идентификаторов, являясь своего рода *фильтром*. В качестве *второго* фактического аргумента могут выступать как отдельный тип, так и их множество в соответствии с типами, распознаваемыми *тестирующей* ***type***-функцией языка, рассматриваемой детально ниже. Следующий фрагмент иллюстрирует применение ***indets***-функции для выделения идентификаторов переменных:

> indets(x^3 + 57*y - 99*x*y + (z + sin(t))/(a + b + c) = G);

{sin(t), x, y, z, t, a, b, c, G}

```
> indets(x^3 + 57*y - 99*x*y + (z + sin(t))/(a + b + c) = G, function);   ⇒ {sin(t)}
> indets(x^3 + z/y - 99*x*y + sin(t), {integer, name});
                                  {-99, -1, 3, x, y, z, t}
> indets(x^3 + z/y - 99*x*y + sin(t), {integer, name, `*`, `+`});
                                  { -99, -1, 3, x, z, y, t, z/y, x^3 + z/y - 99 x y + sin(t), -99 x y }
> indets(x^3 + z/y - 99*x*y + sin(t), {algnum, trig});
                                  {-99, -1, 3, sin(t)}
```

Из приведенного фрагмента, в частности, следует, что по ***indets***-функции можно получать не только неопределенные идентификаторы, но и числовые компоненты тестируемого выражения, а также совокупности комбинаций составляющих его компонент. Таким образом, ***indets***-функция несет существенно более развитую смысловую нагрузку, чем определение простых идентификаторов. По своим возможностям она представляется достаточно эффективным средством при решении задач символьных обработки и анализа выражений, а также в целом ряде других важных задач.

Для возможности использования произвольного **A**-выражения в качестве объекта, которому могут присваиваться значения, ***Maple***-язык располагает ***evaln***-функцией, кодируемой в виде ***evaln*(A)** и возвращающей имя выражения. В качестве **A**-выражения допускаются: простой идентификатор, индексированный идентификатор, вызов функции/процедуры или конкатенация символов. В результате применения к **A**-выражению функции ***evaln*** оно становится доступным для присвоения ему значений, однако если ему не было сделано реального присвоения, то применение к нему ***assigned***-функции возвращает значение *false*, т.е. **A**-выражение остается неопределенным. Таким образом, по конструкции ***Id*:= *evaln*(*Id*)** производится отмена присвоенного ***Id***-идентификатору значения, делая его *неопределенным*, как это иллюстрирует следующий простой фрагмент:

```
> Asv:= 32:  assigned(Asv);  Asv:= evaln(Asv):  Asv, assigned(Asv);
                                  true
                                  Asv, false
> Asv:= 67:  assigned(Asv);  Asv:= 'Asv':  Asv, assigned(Asv);
                                  true
                                  Asv, false
```

Из приведенного фрагмента непосредственно следует, что первоначальное присвоение ***Asv***-идентификатору значения делает его определенным, на что указывает и результат вызова ***assigned***-функции. Тогда как последующее выполнение ***Asv*:=*evaln*(*Asv*)** предложения делает ***Asv***-идентификатор вновь *неопределенным*. *Вторым* способом приведения идентификатора к неопределенному состоянию является использование конструкции вида ***Id*:= *'Id'*,** как это иллюстрирует второй пример предыдущего фрагмента.

Для выполнения *присвоений* можно воспользоваться и ***assign***-процедурой, допускающей в общем случае три формата кодирования, а именно: ***assign*({A, B | A = B | C}),** где **A** – некоторый идентификатор, **B** - любое допустимое выражение и **C** - список, множество либо последовательность уравнений. В первых двух случаях применение ***assign***-процедуры *эквивалентно* применению (**:=**)-оператора присвоения, тогда как третий случай применяется для обеспечения *присвоения* левой части каждого элемента списка, множества или последовательности уравнений. Простой фрагмент иллюстрирует вышесказанное:

```
> assign(AGn, 59);  assign(AVz=64);  assign([xx=1, yy=2, zz=3, h=cos(y)*tan(z)]);
> assign({xx1=4, yy1=5, zz1=6, y=x*sin(x)});  [AGn, AVz, xx, yy, zz, xx1, yy1, zz1, y, h];
                          [59, 64, 1, 2, 3, 4, 5, 6, x sin(x), cos(x sin(x)) tan(z)]
```

```
> assign('x', assign('y', assign('z', 64))); [x, y, z];   ⇒ [64]          # Maple 7 и выше
> assign('x', assign('y', assign('z', 64)));  [x, y, z];                  # Maple 6 и ниже
Error, (in assign) invalid arguments
> `if`(type(AV, 'symbol') = true, assign(AV, 64), false);  AV;   ⇒ 64
> GS:= op([57*sin(19.95), assign('VG', 52)]) + VG*cos(19.99);   ⇒ GS := 72.50422598
```

Успешное выполнение ***assign***-процедуры производит указанные присвоения и возвращает ***NULL***-значение, в противном случае инициируется соответствующая ошибочная ситуация. Данная ситуация, в частности, возникает при попытке рекурсивного вызова ***assign***-процедуры для релизов **6** и ниже, тогда как в более старших релизах подобного ограничения нет. По отношению к процедуре ***assign*** релизы пакета характеризуются весьма существенной несовместимостью, что стимулировало нас к созданию аналога стандартной процедуры, который не только устраняет указанную несовместимость, но и расширяет функциональные возможности [31,39,41-43,45,46,103]. Это и другие наши средства рассмотрены детально в книге [103] и представлены в прилагаемой к ней *Библиотеке* программных средств.

При этом, следует отметить, что в целом ряде случаев ***assign***-процедура является единственно возможным способом *присвоения* значений, например, внутри выражений, как это иллюстрирует последний пример фрагмента, который содержит структуры и функции, рассматриваемые ниже. Механизм ***assign***-процедуры достаточно эффективен в различных вычислительных конструкциях, многочисленные примеры применения которого приводятся ниже при рассмотрении различных аспектов ***Maple***-языка, а также в нашей *Библиотеке* [103].

Обратной к ***assign***-процедуре является ***unassign***-процедура с форматом кодирования:

***unassign*(**<*Идентификатор*_1>**, **<*Идентификатор*_2>**, ...)**

отменяющая определения для указанных *последовательностью* ее фактических аргументов *идентификаторов*. Успешный вызов процедуры ***unassign*** выполняет отмену назначений, возвращая ***NULL***-значение. Однако, процедура не действует на идентифицированные с ***protected***-атрибутом, инициируя ошибочную ситуацию с выводом соответствующей диагностики. Приведем простые примеры на использование ***unassign***-процедуры.

```
> AS:= 39:  AV:= 64:  AG:= 59:  Kr:= 10:  Art:= 17:  AS, AV, AG, Kr, Art;
                                  39, 64, 59, 10, 17
> unassign(AS, AV, AG, Kr, Art);
Error, (in unassign) cannot unassign '39' (argument must be assignable)
> unassign('AS', 'AV', 'AG', 'Kr', 'Art');  AS, AV, AG, Kr, Art;
                                  AS, AV, AG, Kr, Art
> `if`(type(AV, 'symbol') = true, assign(AV, 64), unassign('AV'));  AV;   ⇒ 64
```

В приведенном фрагменте пяти переменным присваиваются целочисленные значения, а затем по ***unassign***-процедуре делается попытка отменить сделанные назначения. Попытка вызывает ошибочную ситуацию, обусловленную тем, что в точке вызова процедуре ***unassign*** передаются не сами идентификаторы, а их значения (*кстати, именно данная ситуация одна из наиболее типичных при ошибочных вызовах* ***assign***-*процедуры*). Для устранения ее идентификаторы следует кодировать в невычисленном формате (*кодируя в апострофах*), что иллюстрирует повторный вызов ***unassign***-процедуры. Последний пример иллюстрирует применение процедур ***assign*** и ***unassign*** в условном **if**-предложении языка, по которому **AV**-переменной присваивается целочисленное значение, если она была неопределенной, и отменяется ее определение в противном случае.

Для отмены значений имен, имеющих ***protected***-атрибут, может оказаться достаточно полезной и процедура ***Unassign*('n1', 'n2', ...),** возвращающая *NULL*-значение и отменяющая назначения для имен {**'n1', 'n2', ...**}, закодированных в невычисленном формате. Ее исходный текст представляется однострочным экстракодом следующего вида:

$$Unassign := (\ ) \rightarrow \text{`if`}(\text{nargs} = 0, NULL, \text{op}([\text{unprotect}(\text{args}), \text{unassign}(\text{args})]))$$

Приведем примеры применения процедур стандартной ***unassign*** и нашей ***Unassign:***

```
> x, y, z, t, h:= 42, 47, 67, 89, 95:  protect('x','y','z', 't', 'h');
> unassign('x','y','z', 't', 'h');
Error, (in assign) attempting to assign to `x` which is protected
> x, y, z, t, h;   ⇒  42, 47, 67, 89, 95
> Unassign('x','y','z', 't', 'h');
> x, y, z, t, h;   ⇒  x, y, z, t, h
> Unassign();
> unassign();
Error, (in unassign) argument expected
```

Как следует из приведенного фрагмента, наша процедура ***Unassign*** в отличие от стандартной корректно обрабатывает особую ситуацию *«отсутствие фактических аргументов»*, возвращая и в этом случае *NULL*-значение.

В целях защиты идентификаторов от возможных модификаций их определений (*назначений*) им присваивается ***protected***-атрибут, делающий невозможной какую-либо модификацию указанного типа. Большинство пакетных идентификаторов имеют ***protected***-атрибут, в чем легко убедиться, применяя к ним ***attributes****-функцию*, кодируемую в следующем формате:

***attributes*(**<*Идентификатор*>**)**

и возвращающую значения атрибутов заданного идентификатора, в частности атрибута ***protected***. Если идентификатору не приписано каких-либо атрибутов, то вызов на нем ***attributes***-функции возвращает ***NULL***-значение, т.е. ничего. Для защиты от модификации либо снятия защиты используются процедуры ***protect*** и ***unprotect Maple-***языка соответственно, кодируемые в следующем простом формате:

{***protect*** | ***unprotect***}**(**<*Идентификатор*_1>**,** <*Идентификатор*_2>**, ...)**

Следующий весьма простой фрагмент ***Maple***-документа иллюстрирует вышесказанное:

```
> protect(AV_42);  attributes(AV_42);   ⇒  protected
> unassign(AV_42);
Error, (in assign) attempting to assign to `AV_42` which is protected
> AV_42:= 64:
Error, attempting to assign to `AV_42` which is protected
> unprotect(AV_42);  attributes(AV_42);  AV_42:= 64;   ⇒  AV_42 := 64
```

Следует при этом отметить, что действие ***protect****-процедуры* не распространяется на *глобальные предопределенные* переменные ***Maple,*** значения которых можно модифицировать согласно условиям пользователя. Такая попытка вызывает ошибочную ситуацию:

```
> map(attributes, [Digits, Order, printlevel]);  protect('Digits');  protect('Order');
  protect('printlevel');   ⇒  []
Error, (in protect) an Environment variable cannot be protected
Error, (in protect) an Environment variable cannot be protected
Error, (in protect) an Environment variable cannot be protected
```

Хотя по ***unprotect***-процедуре отменяется ***protected***-атрибут *любого* идентификатора, однако для пакетных идентификаторов этого (*по целому ряду причин, здесь не рассматриваемых*) делать не рекомендуется.

В целом ряде случаев в качестве весьма полезных средств могут выступать две встроенные функции со следующими форматами кодирования: ***unames*()** и ***anames*({ |<*Тип*>}),** возвращающие последовательности соответственно *неопределенных* и *определенных* идентификаторов (*как пользовательских, так и пакетных*), приписанных текущему ***Maple***-сеансу. При этом, для случая ***anames***-функции можно получать подборку определенных идентификаторов, значения которых имеют указанный *Тип*. Следующий фрагмент иллюстрирует результат вызова указанных выше функций:

```
> restart;  unames();
    identical, anyfunc, equation, positive, Integer, restart, radical, And, gamma, neg_infinity, none,
     default, nonposint, relation, odd, infolevel, indexable, algebraic, SFloat, RootOf, TABLE, float,
    real_to_complex, embedded_real, vector, _syslib, realcons, name, assign, INTERFACE_GET, ...
> restart:  SV:= 39:  GS:= 82:  Art:= sin(17):  Kr:= sqrt(10):  AV:= 64:  anames();
sqrt/primes, type/interfaceargs, GS, sqrt, AV, csgn, interface, type/SymbolicInfinity, Art, sin, SV, Kr
> anames('integer');    # Maple 8
                              sqrt/primes, GS, Digits, printlevel, Order, AV, SV
> anames('environment');
    Testzero, UseHardwareFloats, Rounding, %, %%%, Digits, index/newtable, mod, %%, Order,
                              printlevel, Normalizer, NumericEventHandlers
> SV:= 39:  GS:= 82:  `Art/Kr`:= sin(17):  Kr:= sqrt(10):  _AV:= 64: anames('user');  # Maple 10
                                          Kr, SV, GS
> anames('alluser');   ⇒ Kr, SV, GS, _AV, Art/Kr
```

При этом, ***unames***-функция была вызвана в *самом* начале сеанса работы с пакетом и возвращаемый ею результат представлен только начальным отрезком достаточно длинной последовательности пакетных идентификаторов. Что касается ***anames***-функции, то она в качестве *второго* необязательного аргумента допускает *тип*, идентификаторы с которым будут ею возвращаться. При этом, дополнительно к типу и в зависимости от релиза в качестве второго аргумента допускается использование таких параметров как ***user, environment, alluser,*** назначение которых можно вкратце охарактеризовать следующим образом. Параметр ***environment*** определяет возврат имен *предопределенных* переменных текущего сеанса пакета, параметр ***user*** (*введенный, начиная с* ***Maple 9***) определяет возврат имен, определенных пользователем, тогда как параметр ***alluser*** (*введенный, начиная с релиза* ***10***) аналогичен по назначению предыдущему параметру ***user,*** но без фильтрации имен пользователя, содержащих прямой слэш «**/**» и префикс «_» (*подчеркивание*).

В ряде случаев требуется на основе определенного выражения определить имя (*имена*), которым в текущем сеансе было присвоено данное выражение. Данная задача решается нашей процедурой ***nvalue,*** базирующейся на стандартной процедуре ***anames*** и использующей некоторые особенности пакета. Ниже эта процедура будет представлена более детально, здесь же мы приведем лишь фрагмент ее применения, а именно:

```
> a:= 63: b:= 63: c:= "ransian63": c9:= "ransian63": d:= table([1=63, 2=58, 3=38]): L:= [x, y, z]:
  assign('h'=`svegal`): t47:= 19.42: t42:= 19.42: R:= a+b*I: B:= a+b*I: Z:= (a1+b1)/(c1+d1):
  f:=cos(x): g:=proc() `+`(args)/nargs end proc: r:=x..y: m:=x..y: n:= x..y: Lasnamae:= NULL:
> Tallinn:=NULL: Grodno:= NULL: Vilnius:= NULL: Moscow:= NULL: map(nvalue, [63,
  "ransian63", table([1=63,2=58,3=38]), svegal, 19.42, [x,y,z], a+b*I, (a1+b1)/(c1+d1), cos(x),
  proc () `+`(args)/nargs end proc, x..y]), nvalue();  ⇒ {a, b}, {c9, c}, {d}, {h}, {t47,t42}, {L}, {R,B},
  {Z}, {f}, {g}, {m, r, n}, {Tallinn, Grodno, Vilnius, Moscow, Lasnamae}
```

1.3. *Средства Maple-языка для определения свойств переменных*

Важным средством управления вычислениями и преобразованиями в среде ***Maple***-языка является ***assume***-процедура и ряд сопутствующих ей средств. Процедура имеет следующие три формата кодирования**:**

assume(x1, p1, x2, p2, ...) **assume(x1::p1, x2::p2, ...)** **assume(x1p1, x2p2, ...)**

и позволяет наделять *идентификаторы* (*переменные*) или допустимые ***Maple***-выражения **xj** заданными свойствами **pj,** т.е. устанавливать определенные свойства и соотношения между ними**.** Третий формат процедуры определяет соотношения, налагающие свойство **pj** на выражение **xj**. Например**,** простейшие, но весьма часто используемые вызовы ***assume*(x >= 0)** процедуры, определяют для некоторой **x**-переменной свойство быть неотрицательной действительной константой. Наделяемое по ***assume***-процедуре свойство не является *пассивным* и соответствующим образом обрабатывается ***Maple***-языком пакета при выполнении вычислений либо преобразований выражений, содержащих переменные, наделенные подобными свойствами. Тестировать наличие приписанного **x**-переменной свойства можно по вызову процедуры ***about*(x)**, тогда как *наделять* **x**-переменную *дополнительными свойствами* можно по вызову процедуры ***additionally*(x,** *Свойство***)**. Следующий фрагмент иллюстрирует вышесказанное**:**

```
> assume(x >= 0): map(about, [x, y, z]);   ⇒ [ ]
        Originally x, renamed x~:  is assumed to be: RealRange(0, infinity)
        y:  nothing known about this object
        z:  nothing known about this object
  > assume(a >= 0): A:= sqrt(-a): assume(a <= 0): B:= sqrt(-a): [A, B];   ⇒ [√a~ I, √-a~ ]
> simplify(map(sqrt, [x^2*a, y^2*a, z^2*a]));   ⇒ [x~ √-a~ I, √y^2 a~, √z^2 a~ ]
> additionally(x, 'odd'): about(x);
Originally x, renamed x~:  is assumed to be:
AndProp(RealRange(0, infinity), LinearProp(2, integer, 1))
> map(is, [x, y, z], 'odd', 'posint');   ⇒ [true, false, false]
> assume(y, 'natural'): map(is, [x, y], 'nonnegative');   ⇒ [true, true]
> unassign('x'):  about(x);
x: nothing known about this object
> assume(y, {y >= 0, y < 64, 'odd'}): about(y);
Originally y, renamed y~:  is assumed to be: {LinearProp(2,integer,1), 0 <= y, y < 64}
> hasassumptions(y);   ⇒ true
```

Из примеров данного фрагмента, в частности, следует**,** что переменная с приписанным ей свойством выводится помеченной символом *тильды* (**~**), а по ***about***-процедуре выводится информация о всех приписанных переменной свойствах либо об их отсутствии**.** Один из примеров фрагмента иллюстрирует влияние наличия свойства *положительной* определенности переменной на результат *упрощения* содержащего ее выражения. В *другом* примере фрагмента иллюстрируется возможность определения для переменной *множественности* свойств, определяемых как поддерживаемыми языком стандартными свойствами, так и допустимыми *отношениями* для переменной. При этом, вызов процедуры ***hasassumptions*(x)** возвращает *true***,** если на **x**-выражение было наложено какое-либо соотношение, и *false* в противном случае.

Режим идентификации ***assume*-**переменных определяется *showassumed-параметром* процедуры ***interface*,** принимающим значение {**0**|**1** (*по умолчанию*)|**2**}**: 0 -** отсутствует идентификация, **1 -** переменные сопровождаются знаком *тильды* и **2 -** все такие переменные перечисляются в конце выражений, как это иллюстрирует следующий фрагмент**:**

> assume(V >= 64): assume(G >= 59): S:= V+G; interface(showassumed=0); V + G;

$$S := V\sim + G\sim$$

$$V + G$$

> assume(Art >= 17, Kr >= 10): interface(showassumed=2): S^2 + (Art + Kr)^2;

$$S^2 + (Art + Kr)^2$$

with assumptions on ***Art*** and ***Kr***

> assume(x, 'odd', y::integer, z >= 0), is(x + y, 'integer'); $\Rightarrow$ *true*

> x:= 'x': unassign('y'), map(is, [x, y], 'integer'), is(z, 'nonnegative'); $\Rightarrow$ [*false, false*], *true*

До **7**-го релиза включительно оперативно переопределять режим идентификации переменных ***assume*** можно было переключателями функции ***Assumed Variables*** группы **Options GUI**.

По тестирующей процедуре ***is*(x,** *Свойство***)** возвращается *true*-значение, если **x-**переменная обладает указанным вторым фактическим аргументом *свойством***,** и значение *false* в противном случае. При невозможности идентифицировать для **x-**переменной свойство (*например***,** *если она по* ***assume-****процедуре свойствами не наделялась*) возвращается *FAIL*-значение. Наконец, отменять приписанные **x-**переменной *свойства* можно посредством выполнения простой конструкции **x:= 'x'** либо вызовом процедуры ***unassign*('x');** сказанное иллюстрируют *последние* примеры предыдущего фрагмента. При этом, проверять можно как конкретную отдельную переменную, так и выражение по нескольким ведущим переменным и набору искомых свойств.

***Maple*-**язык поддерживает работу со свойствами шести основных групп, а именно**:**

1) *имя свойства*, например, *continuous*, *unary*;

2) *большинство имен типов*, например, *integer*, *float*, *odd*, *even*;

3) *числовые диапазоны*, например, RealRange(a, b), RealRange(-*infinity*, b), RealRange(a, *infinity*), где a и b могут быть или числовыми значениями или Open(a), где a – числовое значение

4) AndProp(a, b, ...) **– and**-выражение свойств <a **and** b **and** ...>, где a, b, ... – свойства, определенные выше

5) OrProp(a, b, ...) **– or-**выражение свойств, где объект может удовлетворять любому из a, b, ... свойств

6) *диапазон свойств* p1 **..** p2**,** где p1 и p2 - свойства. Данное свойство означает, что объект удовлетворяет по меньшей мере p2, но не более, чем p1; например, integer .. rational удовлетворяется integers/2.

За более детальной информацией по поддерживаемым ***Maple*-**языком *свойствам* остальных групп можно обращаться или к справке по пакету, или к книгам [8-14,78-86,88,105].

Механизм *свойств***,** определяемый процедурой ***assume*** и сопутствующей ей *группой* процедур ***coulditbe*, *additionally*, *is*, *about*, *hasassumptions*** и ***addproperty*,** использует специальную глобальную ***_EnvTry***-переменную для определения режима как *идентификации* у переменных приписанных им свойств, так и их *обработки*. При этом**,** данная переменная допускает только два значения**:** ***normal*** (*по умолчанию*) и ***hard*,** из которых указание второго значения может потребовать при вычислениях существенных временных затрат. В текущих реализациях пакета значение глобальной ***_EnvTry*-**переменной, опреде-

ляющей режим обработки переменных, наделенных ***assume-****свойствами*, не определено, что иллюстрирует следующий достаточно прозрачный фрагмент**:**

```
> _EnvTry, about(_EnvTry);   ⇒ _EnvTry
_EnvTry: nothing known about this object
> assume(V >= 64): about(V);
Originally V, renamed V~: is assumed to be: RealRange(64, infinity)
> `if`(is(V, RealRange(64, infinity)), ln(V) + 42, sqrt(Art + Kr));   ⇒ ln(V~) + 42
> assume(47 <= G, G <= 59): about(G);
Originally G, renamed G~: is assumed to be: RealRange(47, 59)
> `if`(is(G, RealRange(47, 59)), [10, 17, Sv, Art, Kr], Family(x, y, z));   ⇒ [10, 17, Sv, Art, Kr]
> assume(x >= 0), simplify(sqrt(x^2)), simplify(sqrt(y^2));   ⇒ x~, csgn(y) y
> sqrt(a*b), sqrt(a^2), assume(a >= 0, b <= 0), sqrt(a*b), sqrt(a^2);  ⇒ √(a b), √(a^2), √(-a~ b~) I, a~
```

Механизм *приписанных свойств* является достаточно развитым и мощным средством как *числовых* вычислений, так и *символьных* вычислений и преобразований. Использование его оказывается весьма эффективным при программировании целого ряда важных задач во многих приложениях**.** Последние примеры предыдущего фрагмента иллюстрируют некоторые простые элементы его использования в конкретном программировании. Тогда как конкретный ***assume-***механизм базируется на алгоритмах Т. Вейбеля и Г. Гоннета**.** С интересным обсуждением принципов его применения, реализации и ограничений можно довольно детально ознакомиться в интересных работах указанных авторов, цитируемых в конце *справки* по пакету (*см*. **?*assume***), и цитируемых в них многочисленных источниках различного назначения.

В ряде случаев требуется выполнить *обмен значениями* между переменными. Например, переменные **x, y** и **z,** имеющие значения 64, 59 и 39 должны получить значения 39, 59 и 64 соответственно. Следующая процедура ***varsort*** решает данную задачу.

```
varsort := proc()
local a, b, c, d, k;
    `if`(nargs = 0, RETURN( ), assign(c = [ ]));
    d := proc(x, y)
            try if evalf(x) < evalf(y) then true else false end if
            catch : `if`(map(type, {x, y}, 'symbol') = {true}, lexorder(x, y), true)
            end try
        end proc;
    a := [seq(`if`(type(args[k], 'boolproc'), assign('b' = args[k]),
        `if`(type(args[k], 'symbol'), op(args[k], assign('c' = [op(c), k])), NULL)),
        k = 1 .. nargs)];
    a := sort(a, `if`(type(b, 'boolproc'), b, d));
    for k to nops(a) do assign(args[c[k]] = a[k]) end do
end proc
> x:=64: y:=59: z:=39: m:=vsv: n:=avz: p:=agn: varsort('x', 'z', 'y', 'm', 'p', 'n'), x, y, z, m, n, p;
                                    39, 64, 59, agn, vsv, avz
> x:=64: y:=59: z:=39: m:=vsv: n:=avz: p:=agn: varsort('x', 'y', 'z', 'm', 'n', 'n'), x, y, z, m, n, p;
                                    39, 59, 64, avz, vsv, agn
```

Вызов ***varsort*****('x', 'y', 'z', …)** возвращает *NULL*-значение, обеспечивая обмен значениями между переменными **x, y, z, …** в порядке возрастания. Подробнее с возможностями данной процедуры можно ознакомиться в [41-43,103,109].

1.4. *Типы числовых и символьных данных Maple-языка пакета*

Средства ***Maple-***языка поддерживают работу как с простыми, так и сложными типами данных числового или символьного (*алгебраического*) характера. В первую очередь рассмотрим типы простых данных *числового* характера, предварив краткой информацией по очень важным *встроенным* функциям ***nops*** и ***op*,** непосредственно связанных со структурной организацией ***Maple***-выражений. Первая функция возвращает число *операндов выражения*, заданного ее *единственным* фактическим аргументом. Тогда как вторая имеет более сложный формат кодирования следующего вида:

***op*(**{ | **n,** | **n..m,** | <*Список*>**,**} <*Выражение*>**)**

где первый необязательный фактический аргумент определяет возврат соответственно: **n**-го операнда, с **n**-го по **m**-й операнды или операнды согласно *Списка* их позиций в порядке возрастания уровней вложенности *Выражения*. При этом, при **n = 0** возвращается *тип* самого выражения, а в случае отсутствия указанного первым аргументом операнда инициируется ошибочная ситуация. Для случая **n<0** выполняется соотношение ***op*(n,V) ≡ *op*(*nops*(V) + n + 1, V),** где **V** – выражение, а для неопределенного ***Id***-идентификатора имеют место соотношения: **op(0, *Id*) ⇒** *symbol* и ***op*(1, *Id*) ⇒** *Id***.** Отсутствие первого аргумента ***op***-функции аналогично вызову вида ***op*(1 .. *nops*(V), V)**. Для вывода структурной организации произвольного выражения может оказаться полезной конструкция вида:

***op*('k',** <*Выражение*>**) $'k'=0 .. *nops*(**<*Выражение*>**)**

вычисление которой возвращает последовательность типа и всех операндов *первого* уровня вложенности указанного выражения, как это иллюстрирует следующий пример:

> Art:= 3*sin(x) + 10*cos(y)/AV + sqrt(AG^2 + AS^2)*TRG: op('k', Art)$'k'=0 .. nops(Art);

$$+,\ 3\sin(x),\ \frac{10\cos(y)}{AV},\ \sqrt{AG^2+AS^2}\ TRG$$

Ниже будет рассмотрено достаточно средств ***Maple-***языка, ориентированных на задачи символьной обработки выражений, включая и те, которые базируются на их структурном ана-изе. Целый ряд средств для решения подобных задач предоставляет и наша *Библиотека* [103]. Нам же для дальнейшего будет пока вполне достаточно информации по встроенным функциям ***nops*** и ***op*** пакета.

• ***Целые*** (*integer*); представляют собой цепочки из одной или более цифр, максимальная *длина* которых определяется используемой платформой **ЭВМ**: так для 32-битной она не превышает 524280 цифр, а для 64-битной – 38654705646 цифр. Целые могут быть со знаком и без: **1999**, **-57**, **140642**. На числах данного типа функции ***op*** и ***nops*** возвращают соответственно значение числа и значение **1,** тогда как функция ***type*** идентифицирует их тип как *integer*, например:

> op(429957123456789), nops(429957123456789); ⇒ 429957123456789, 1
> type(429957123456789, 'integer'); ⇒ *true*

• ***Действительные*** (*float*) с *плавающей точкой*; представляют собой цепочки из десятичных цифр с десятичной точкой в {*начале*|*середине*|*конце*} цепочки; числа данного типа допускают следующие два основных формата кодирования:

[<*знак*>]{<*целое*>**.**<*целое*> | **.**<*целое*> | <*целое*>**.**}
Float([<*знак*>]<*мантисса*>, [<*знак*>]<*экспонента*>**) ≡**
[<*знак*>]<*мантисса*>**.{E**|**e}**[<*знак*>]<*экспонента*>

В качестве *мантиссы* и *экспоненты* используются *целые* со знаком или без; при этом, *мантисса* может иметь *любую* длину, но *экспонента* ограничивается длиной машинного слова: для 32-битной платформы значение экспоненты не превышает целого 2147483647, а для 64-битной платформы – целого значения 9223372036854775807. Тогда как максимальное допустимое число цифр мантиссы аналогично максимальному допустимому числу цифр целого (*integer*) числа. Число цифр мантиссы, участвующих в операциях арифметики с плавающей точкой, определяется предопределенной ***Digits***-переменной ядра пакета, имеющей по *умолчанию* значение **10**. В книге [103] представлен ряд полезных средств, позволяющих оформлять числовые значения в принятых для документирования и печати форматах.

Второй способ кодирования действительных чисел применяется, как правило, при работе с очень большими или очень малыми значениями. Для конвертации значений в действительный тип используется ***evalf***-функция, возвращающая значение *float*-типа. В вышеприведенных примерах применение данной функции уже иллюстрировалось; функция имеет простой формат кодирования ***evalf*(**<*Выражение*> [**, n**]**)** и возвращает результат *float*-типа вычисления выражения (*действительного или комплексного*) с заданной **n**-точностью (*если она определена вторым фактическим аргументом*). ***Maple*** испытывает затруднения при вычислениях уже целого ряда простых *радикалов* с рациональными степенями от отрицательных значений, если знаменатель экспоненты – нечетное число. В этом случае даже стандартная функция ***evalf*** оказывается бессильной совместно с использованием пакетного модуля **RealDomain,** что очень хорошо иллюстрируют довольно простые примеры, а именно:

> R:=(58 + (-243)^(1/5) + (-8)^(2/3))*(10 + (-1331)^(2/3) + (-8)^(2/3))/((63 +(-32)^(4/5) - (-27)^ (2/3))* (17 + (343)^(2/3) + (-32)^(3/5))); with(RealDomain): evalf(R), Evalf(R);

$$R := \frac{(58 + (-243)^{(1/5)} + (-8)^{(2/3)})\,(10 + (-1331)^{(2/3)} + (-8)^{(2/3)})}{(63 + (-32)^{(4/5)} - (-27)^{(2/3)})\,(17 + 343^{(2/3)} + (-32)^{(3/5)})}$$

$$-0.7722517003 + 1.867646862\,\mathbf{I},\ 1.961822660$$

Тогда как процедура ***Evalf*****,** находящаяся в упомянутой *Библиотеке* [103], вполне успешно решает данную задачу, что и иллюстрируе данный пример в среде ***Maple 10***.

Практически все встроенные функции пакета возвращают результаты *float*-типа, если хоть один из их аргументов получает значение данного типа. Автоматически результат операции возвращается *float*-типа, если один из операндов имеет данный тип. По умолчанию число цифр выводимого действительного числа равно **10,** но в любое время может переопределяться в глобальной ***Digits***-переменной ядра пакета. Примеры: **1.42, -.57, 17., 8.9, 2.9E-3, -5.6e+4, -5.4E-2.** При этом, конструкции типа <*целое*>**.{E|e}**<*целое*> вызывают синтаксическую ошибку с диагностикой "missing operator or `;`".

Третьим способом определения действительных чисел является *функция* **Float(**<*мантисса*>**,** <*экспонента*>**),** возвращающая число *float*-типа с указанными *мантиссой* и *экспонентой*, например: **Float(2006, -10);** ⇒ 0.2006**e**-6. Иногда **Float**-функцию называют *конструктором float-чисел*.

Действительное число имеет *два* операнда: *мантиссу* и *экспоненту*, поэтому вызов функции ***op*(**{**1**|**2**}**,** <*число*>**)** соответственно возвращает {*мантиссу*|*экспоненту*} указанного ее вторым аргументом числа, тогда как вызов функции ***nops*(**<*число*>**)** возвращает значение **2** по числу операндов; при этом, функция ***type*** идентифицирует *тип* таких чисел как *float*, например:

> op(2, 19.95e-10), op(1, 19.95e-10), nops(19.95e-10); ⇒ -12, 1995, 2
> type(19.95e-10, 'float'); ⇒ *true*

Вопросы арифметики с числами *float*-типа детально рассматриваются, например, в книгах [12,13] и в ряде других изданий, поэтому ввиду наших целей здесь они не детализируются. Впрочем, данный тип числовых выражений и так достаточно прозрачен.

• ***Рациональные*** (*rational*); представляют собой числа (дроби), кодируемые в форме вида [<*знак*>] **a/b,** где **a** и **b** - целые числа; в частности, *целые* числа также рассматриваются пакетом в качестве частного случая рациональных, имеющих *единичный* знаменатель. Для перевода *рациональных* чисел в числа *float*-типа используется упомянутая выше ***evalf***-функция, например: **-2, 64, 59/47, -2006/9, *evalf*(59/47)=1.255319149**. На числах данного типа функции ***op*** и ***nops*** возвращают соответственно значение *числителя*, *знаменателя* и значение **2** по числу операндов, тогда как функция ***type*** идентифицирует тип таких чисел как *fraction* или *rational*, например:

> op(350/2006), nops(350/2006); ⇒ 175, 1003, 2
> type(350/2006, 'fraction'), type(350/2006, 'rational'); ⇒ *true*, *true*

Для работы с рациональными числами ***Maple***-язык располагает целым рядом функциональных средств, достаточно детально рассматриваемых ниже.

• ***Комплексные*** (*complex*); представляют собой числа вида **a+b*I** (**b ≠ 0**), где **a** и **b** – числа рассмотренных выше трех типов, а **I** – комплексная единица (**I = √-1**). Части **a** и **b** комплексного числа называются соответственно *действительной* и *мнимой*; отсутствие второй делает число действительным. Примеры: **-19.42 + 64*I, 88.9*I, 57*I/42**. Для комплексных чисел различаются целые, действительные, рациональные и числовые в зависимости от того, какого типа их действительная и мнимая части. Например, число **64 - 42*I** полагается комплексным целочисленным, тогда как *числовое комплексное* предполагает числовыми *действительную* и *мнимую* части. На числах *комплексного* типа функции ***op*** и ***nops*** возвращают соответственно значения действительной и мнимой частей, и число **2** операндов, тогда как функция ***type*** идентифицирует тип таких чисел как *complex* (*может указываться и подтип*), например:

> op(64 - 42*I), nops(64 - 42*I); ⇒ 64, -42, 2
> type(64 - 42*I, 'complex'('integer')); ⇒ *true*

Следует отметить, что ***Maple***-языком некорректно распознается тип вычисления ряда комплексных выражений, ориентируясь только на наличие в вычисляемом выражении комплексной **I**-единицы. Следующий простой пример иллюстрирует вышесказанное:

```
> type(I*I, 'complex'), type(I^2, 'complex'), type(I^4, 'complex'), type(52 + b*I,
  'complex'({'symbol', 'integer'})), type(a + 57*I, 'complex'({'symbol', 'integer'}));
                    true, true, true, true, true
> type(I*I, 'complex1'), type(I^2, 'complex1'), type(I^4, 'complex1'), type(52 + b*I,
  'complex1'({'symbol', 'integer'})), type(a + 57*I, 'complex1'({'symbol', 'integer'}));
                    false, false, false, true, true
```

Согласно соглашениям пакета вызов функции ***type*(x, *complex*)** возвращает ***true***, если **x** – выражение формы **a+b*I,** где **a** (*при наличии*) и **b** (*при наличии*) *конечны*, имея тип *realcons*. В принципе, с формальной точки зрения все нормально. Однако в целом ряде случаев необходимо точно идентифицировать комплексный тип, имеющий форму **a + b*I** при **b ≠ 0**. С этой целью нами был дополнительно определен тип *complex1* [103], решающий данную задачу. В предыдущем фрагменте можно сравнить результаты тестирования на типы *complex* и *complex1*.

• ***Булевские*** (*boolean*); представляют собой логические значения *true* (*истина*), *false* (*ложь*) и *FAIL* (*неопределенная истинность*). Третье значение используется в случае, когда истин-

ность какого-либо выражения неизвестна. Функция ***type*** идентифицирует *тип* таких значений как *boolean*, например:

> map(type, [true, false, FAIL], 'boolean'); ⇒ [*true*, *true*, *true*]

Язык ***Maple*** использует *трехзначную* логику для выполнения операций *булевой* алгебры. *Булевы* выражения образуются на основе базовых логических операторов {**and, or, not**} и операторов отношения {**<, <=, >, >=, =, <>** (*не равно*)}. Наша *Библиотека* [103] определяет ряд достаточно полезных средств, расширяющих стандартные средства пакета для работы с булевой алгеброй.

• ***Константы*** (*constant*); представляют собой постоянные значения любого из вышерассмотренных *пяти* типов. Данные конструкции весьма прозрачны и особых пояснений, так же как и иллюстрирующих их примеров, не требуют.

Таким образом, в процессе организации вычислений в среде пакета пользователь имеет доступ к следующим четырем основным типам числовых данных:

(**1**) ***целые числа со знаком*** (1942; -324; 34567; -43567654326543; 786543278);
(**2**) ***действительные со знаком*** (19.95; -345.84; 7864.87643; -63776.2334643);
(**3**) ***рациональные со знаком*** (18/95; -6/28; -4536786/65932; 765987/123897);
(**4**) ***комплексные числа*** (48+53***I**; 28.3-4.45***I**; ½+5/6***I**; -4543.87604+53/48***I**)

Каждый из перечисленных *типов* числовых данных идентифицируется специальным идентификатором: *integer* – *целые*; *float* – *действительные* с плавающей точкой; *rational, fraction* – рациональные (*дроби вида* **m/n; m, n** – *целые*) и *complex* – *комплексные* числа. Каждый из этих идентификаторов может быть использован для тестирования типа переменных и выражений посредством ***type***-функции, уже упоминаемой выше, но детально рассматриваемой ниже.

Для обеспечения работы с *числовыми* значениями ***Maple***-язык располагает как функциями общего назначения, так и *специальными*, ориентированными на конкретный числовой тип. Например, по функциям ***ifactor, igcd, iperflow*** возвращается соответственно: разложение на целочисленные множители целого числа, наибольший общий делитель целых чисел и результат проверки целого числа на возможность представления его в виде $\mathbf{n^p}$, где **n** и **p** – оба целые числа, например: **[ifactor(64), iperfpow(625, 't'), igcd(42, 7)], t;** ⇒ $[(2)^6, 25, 7], 2$. Детально как с общими, так и специальными функциями работы с числовыми выражениями можно ознакомиться в книгах [12,103], в других изданиях, но наиболее полно в справке по пакету.

Наряду с *десятичными* числовыми значениями пакет поддерживает работу с *бинарными*, **8**- и **16**-*ричными*, а также произвольными **q**-*ричными* числами (**q** - *основание системы счисления*). Для преобразования чисел из одной системы счисления в другую служит функция ***convert*** языка, рассматриваемая ниже. Наряду с *числовыми* данными, пакет поддерживает работу с нечисловыми (*символьными*, *алгебраическими*) выражениями, характеризуемыми тем, что им не приписаны какие-либо числовые значения. С *символьными* данными и их обработкой познакомимся детальнее несколько *позднее*. Здесь лишь отметим базовый тип символьных данных – данные типов *string* и *symbol*.

• ***Строка*** (*string*); любая конечная последовательность символов, взятая в верхние двойные кавычки; данная последовательность может содержать и *специальные* символы, как это иллюстрирует следующий простой пример:

"Dfr@t4#\78578"; "A_V_Z; A+G-N; "" 574%!@#$%"; "_Vasco&Salcombe_2006"

На *строках* функции ***op*** и ***nops*** возвращают соответственно саму строку и количество **1** операндов, тогда как функция ***type*** идентифицирует тип таких выражений как *string*, например:

> op(“123456”), nops(“123456”), type(“123456”, ‘string’); ⇒ “123456”, 1, *true*

• ***Символ*** (*symbol, name*); любая конечная последовательность символов, взятая в *верхние обратные* кавычки; данная последовательность может содержать и *специальные* символы, как это иллюстрирует следующий простой пример:

`Dfr@t4#\78578`; `A_V_Z; A+G-N; “” 574%!@#$%`; `_Vasco&Salcombe_2006`; AVZ

Между тем, в отличие от *строк*, требующих обязательного ограничивания их *двойными* кавычками, для *символов* ограничения верхними обратными кавычками требуется только в том случае, когда они содержат *специальные* символы, например, *пробелы*. На *символах* функции ***op*** и ***nops*** возвращают соответственно сам *символ* и число **1** операндов, тогда как ***type-***функция идентифицирует тип таких выражений как *symbol* либо *name*, например:

> op(`123456`), nops(`12345`), map2(type,`123456`, [‘symbol’, ‘name’]); ⇒ 12345, 1, [*true*, *true*]

В отличие от **4**-го релиза последующие релизы ***Maple*** четко различает понятия *символа* и *строки*. *Символьный* тип играет основополагающую роль в *символьных* (*алгебраических*) вычислениях и обработке информации. *Строчные* данные, наряду с *символьными*, играют основную роль при работе с символьной информацией и ***Maple-***язык располагает для работы с ними довольно развитыми средствами, которые с той или иной степенью полноты рассматриваются нами ниже. Немало дополнительных средств для работы с выражениями типов {*symbol*, *name*, *string*} представлено и нашей *Библиотекой* [103]. Многие из них позволяют весьма существенно упростить программирование целого ряда приложений в среде пакета ***Maple*** релизов **8 – 10.**

Например, процедура **_ON(*expr*)** обеспечивает конвертацию *алгебраического* выражения ***expr*** в стилизованный формат, в котором каждое *float*-число представляется в виде **n.m.**

```
_ON := proc(expr)
local a, b, c, d, p, k;
    assign( a = cat(" ", convert( expr, 'string'), " "),
        b = { seq( convert( k, 'string'), k = 0 .. 9 ) } ), assign( d = Search2( a, {"."} ) );
    d := [ seq( `if`( a[ d[ k ] – 1 ] = " " and a[ d[ k ] + 1 ] = "." or
        a[ d[ k ] – 1 ] = "." and a[ d[ k ] + 1 ] = " " or type( a[ d[ k ] + 1 ], 'letter') or
        type( a[ d[ k ] – 1 ], 'letter') or
        member( a[ d[ k ] + 1 ], b ) and member( a[ d[ k ] – 1 ], b ), NULL, d[ k ] ),
        k = 1 .. nops( d ) ) ];
    p := [ 1 $ ( k = 1 .. nops( d ) ) ];
    for k to nops( d ) do
        if not member( a[ d[ k ] – 1 ], b ) then
            a := cat( a[ 1 .. d[ k ] – 1 ], "0", a[ d[ k ] .. -1 ] ); d := p + d
        elif not member( a[ d[ k ] + 1 ], b ) then
            a := cat( a[ 1 .. d[ k ] – 1 ], a[ d[ k ] + 1 .. -1 ] ); d := d – p
        end if
    end do;
    came( a )
end proc
```

```
> R:= 8.*a + 47*I+(( .5 + 8.*a +c . d) + cos(6.*gamma + x-7.) - 3.*7^(-2.*a) + int(f(x), x=1. .. .5))/
  (( 8.^a.b - .8*Pi )*sqrt( a - 3. ) + exp(8.*a - 87.*Pi + 16.*I) + array(1..2, 1..2, [[1., .2], [3., 4.]]));
```

$$R := 8.\,a + 47\,I + \frac{0.5 + 8.\,a + (c\,.\,d) + \cos(6.\,\gamma + x - 7.) - 3.\,7^{(-2.\,a)} + \int_{1.}^{0.5} \mathrm{f}(x)\,dx}{(((8.^{a})\,.\,b) - 0.8\,\pi)\sqrt{a - 3.} + \mathbf{e}^{(8.\,a - 87.\,\pi + 16.\,I)} + \begin{bmatrix} 1. & 0.2 \\ 3. & 4. \end{bmatrix}}$$

```
> _ON(R);
```

$$8\,a + 47\,I + \frac{0.5 + 8\,a + (c\,.\,d) + \cos(6\,\gamma + x - 7) - 3\,7^{(-2\,a)} + \int_{1}^{0.5} \mathrm{f}(x)\,dx}{(((8^{a})\,.\,b) - 0.8\,\pi)\sqrt{a - 3} + \mathbf{e}^{(8\,a - 87\,\pi + 16\,I)} + \begin{bmatrix} 1 & 0.2 \\ 3 & 4 \end{bmatrix}}$$

При этом, результат, возвращаемый процедурой ***_ON,*** пригоден для непосредственных вычислений. Между тем, данная процедура носит описательный характер, не изменяя самой сути вычисления выражений.

Тогда как процедура ***Evalf*(x)** совместно с процедурой ***ffp*** устраняет затруднения ***Maple*** при вычислениях ряда простых *радикалов* **x** с рациональными степенями от отрицательных значений, если знаменатель их показателя – нечетное число [41-43,103,109].

```
Evalf := proc(E::algebraic)
local a, b, c, d, h, g, k, ψ;
    assign(a = convert(E, 'string'), g = [ ]), assign(b = Search2(a, {")^("}));
    ψ := x → `if`(type(x, 'integer') or x = 0, parse(cat(convert(x, 'string'), ".")), x);
    if b = [ ] then
        nulldel(evalf(E, `if`(nargs = 2 and type(args[2], 'posint'), args[2], NULL)))
            ;
        ψ(%)
    else
        for k in b do
            c := a[nexts(a, k, "(", 1)[-1] .. nexts(a, k, ")")[-1]];
            try d := map(convert, map(parse, SLD(c, "^")), 'fraction')
            catch : next
            end try;
            if type(d[1], 'negative') and type(d[2], 'fraction') and
            type(denom(d[2]), 'odd') then
                if type(numer(d[2]), 'even') then
                    g := [op(g), c = cat("(", c[3 .. -1])]
                else g := [op(g), c = cat("(-1)*(", c[3 .. -1])]
                end if
            end if
        end do;
        nulldel(evalf(parse(SUB_S(g, a))),
            `if`(nargs = 2 and type(args[2], 'posint'), args[2], NULL)));
        ψ(%)
    end if
end proc
```

1.5. *Базовые типы структур данных Maple-языка*

Наряду с простыми данными ***Maple***-язык поддерживает работу с наиболее распространенными структурами данных такими как *последовательности*, *списки*, *множества*, *массивы* и *таблицы*. Вкратце остановимся на каждой из перечисленных структур.

• ***Последовательность*** (*sequence*) – последовательная структура, широко используемая пакетом для организации как ряда других типов структур данных, так и для разнообразных вычислительных конструкций, представляет собой базовую структуру данных и определяется конструкциями следующего весьма простого вида:

***Sequence* := B1, B2, B3, ..., Bn; Bj (j=1..n) –** *любое допустимое выражение языка*

Последовательность не является ни списком, ни множеством, но она лежит в основе определения этих типов структур данных пакета. Пример: **GLS:= 47, Gr, -64*L, `99n+57`, F**. Структура типа *последовательность* выражений или просто последовательность (*exprseq*), как уже отмечалось, образуется (,)-оператором *разделителя* выражений (*запятая*) и представляет интерес не только в качестве самостоятельного объекта в среде ***Maple***-языка, но и в качестве основы таких важных структур как: функциональная, список, множество и индексированная. Важным свойством данного типа структуры данных является то, что если ее элементы также последовательности, то результатом является *раскрытая единая* последовательность, как это иллюстрирует следующий простой пример:

> AV:= a, b, c, d, e: GS:= x, y, z: Sv:=2, 9: AG:= 1, 2, 3, AV, 4, Sv, 5, 6, GS;

AG := 1, 2, 3, a, b, c, d, e, 4, 2, 9, 5, 6, x, y, z

Длина произвольной **SQ**-последовательности (*число ее элементов*) вычисляется по конструкции ***nops*([SQ]),** тогда как ее **k**-й элемент получаем посредством оператора *выделения* **SQ[k]**. *Оператор выделения* имеет весьма простой вид: ***Id*[**<*Выражение*>**],** где ***Id* –** идентификатор структуры типа массив, список, множество или последовательность; значение выражения ***Id*[k]** определяет искомый элемент структуры. Если ***Id*** не определен, то он возвращается индексированным, например:

```
> SQ:= V, G, S, Art, Kr, Arne:  {SQ[4], nops([SQ]), R[S]};   ⇒ {6, R[S], Art}
> SQ:= [SQ]:  SQ[6]:= Aarne:  SQ:= op(SQ):  SQ;   ⇒ V, G, S, Art, Kr, Aarne
> Z:= SQ: whattype(SQ);   ⇒ exprseq
> type(Z, 'exprseq');
Error, wrong number (or type) of parameters in function type
> hastype(Z, 'exprseq');
Error, wrong number (or type) of parameters in function hastype
```

Так как по конструкции вида **SQ[k]:=**<*Выражение*> недопустимо присвоение заданного выражения **k**-му элементу **SQ**-последовательности, то для этих целей можно воспользоваться *цепочкой* ***Maple***-предложений вида: **SQ:=[SQ]:SQ[k]:= <***Выражение***>: SQ:=op(SQ):,** как это иллюстрирует второй пример последнего фрагмента. При этом, следует иметь в виду, что тип последовательности тестируется только ***whattype***-процедурой языка, т. к. при передаче *последовательности* в качестве аргумента другим тестирующим функциям она рассматривается как *последовательность* фактических аргументов. Последние три примера предыдущего фрагмента иллюстрируют сказанное. Для прямого тестирования структур типа `***expressions sequence***` нами была определена процедура ***typeseq*,** описанная в книге [103] и включенная в прилагаемую к ней *Библиотеку* [109]. Ниже дано несколько примеров ее применения, а именно:

> typeseq("Kr",Art,6,14,"RANS",IAN,Tallinn, 'seqn'(integer, string, symbol)); ⇒ *true*

> typeseq(a, b, 10, 17, 'seqn'), typeseq("Kr", Art, 10, 17, "RANS", IAN, Tallinn, Vilnius, 'seqn'), typeseq(G, [a], {b}, 61, 10/17, 'seqn'('integer', 'symbol', 'list', 'set', 'fraction'));

true, true, true

В определении различного назначения последовательностей важную роль играют так называемые *ранжированные* конструкции, образованные (**..**)-оператором *ранжирования* и имеющие следующий простой формат кодирования:

<*Выражение*_1> **..** <*Выражение*_n>

где вычисляемые *выражения* определяют соответственно *нижнюю* и *верхнюю* границы *ран-жирования* с шагом **1**. В зависимости от места использования *ранжированной* конструкции для значений выражений допускаются значения типов {*float*, *integer*}. Особо самостоятельного значения ранжированное выражение не имеет и используется в качестве *ранжирующей* компоненты во многих стандартных ***Maple***-конструкциях (*последовательностные структуры*, *суммирование*, *произведение*, *интегрирование и др.*). В качестве самостоятельного применения оператор *ранжирования* можно использовать, например, с **||**-оператором *конкатенации* для создания последовательностей, например:

> n:= 1: m:= 9: SL:= x||(n .. m); ⇒ *SL := x1, x2, x3, x4, x5, x6, x7, x8, x9*
> A||("g" .. "s"); ⇒ *Ag, Ah, Ai, Aj, Ak, Al, Am, An, Ao, Ap, Aq, Ar, As*
> A||("м" .. "э"); ⇒ *Ам, Ан, Ао, Ап, Ар, Ас, Ат, Ау, Аф, Ах, Ац, Ач, Аш, Ащ, Аъ, Ы, Аь, Аэ*
> GS(x||(1 .. 5)):= (x1*x2*x5 + sin(x4 + x5))*(x1^2 + x2^2 + x3^2 + x4^2);

$$GS(x1, x2, x3, x4, x5) := (x1\ x2\ x5 + \sin(x4 + x5))\ (x1 + x2 + x3 + x4)$$

> A||(1 .. 3)||(4 .. 7); ⇒ *A14, A15, A16, A17, A24, A25, A26, A27, A34, A35, A36, A37*

Из приведенного фрагмента четко прослеживается одно существенное отличие бинарного **||**-оператора конкатенации от других бинарных операторов ***Maple***-языка. Если стандартным в языке является порядок вычисления выражений «*слева направо*», то для **||**-оператора *конкатенации* используется обратный ему порядок вычислений. В последнем примере вычисляется сначала правый операнд, а затем левый. Более того, из второго и третьего примеров явствует, что наряду с числовыми значениями для ранжированной конструкции допускаются и буквы английского и национальных алфавитов, закодированные в строчном формате. На ранжированных выражениях функции ***op*** и ***nops*** возвращают соответственно *последовательность* их *левых* и *правых* частей, и число **2** операндов, тогда как функция ***type*** идентифицирует тип таких выражений как *range*, например:

> op(a .. b), nops(a .. b); ⇒ a, b, 2
> type("a" .. "z", 'range'), type(1 .. 64, 'range'), type(-1 .. -6.4, 'range'); ⇒ *true, true, true*

Детальнее структуры типа *последовательность* и оператор *ранжирования* будут рассматриваться ниже в различных контекстах, включая иллюстративные фрагменты. Здесь же мы лишь упомянем два функциональных средства ***Maple***-языка, предназначенных для определения последовательностных структур данных, а именно: **$**-*оператор* и ***seq***-функция, которые имеют соответственно следующие наиболее общего вида форматы:

V(k) $ k = p .. n; ⇒ **V(p), V(p + 1), ... ,V(n)**
***seq*(V[k], k = p .. n);** ⇒ **V(p), V(p + 1), ... ,V(n)**

где **V –** допустимое ***Maple***-выражение, в общем случае зависящее от **к**-переменной *ранжирования*. Следующий простой фрагмент иллюстрирует примеры структур типа *последовательности*, образованные **$**-оператором и ***seq***-функцией ***Maple***-языка:

> G(h) $ h = 9.42 .. 14.99; ⇒ G(9.42), G(10.42), G(11.42), G(12.42), G(13.42), G(14.42)
> seq(G(x), x = 9.42 .. 14.99); ⇒ G(9.42), G(10.42), G(11.42), G(12.42), G(13.42), G(14.42)

```
> 67 $ 15;  ⇒  67, 67, 67, 67, 67, 67, 67, 67, 67, 67, 67, 67, 67, 67, 67
> Gs $ h = -0.25 .. 7.99;  ⇒  Gs, Gs, Gs, Gs, Gs, Gs, Gs, Gs, Gs
> cat(seq(x, x = "a" .. "z"));  ⇒  "abcdefghijklmnopqrstuvwxyz"
> cat(seq(x, x = "A" .. "Z"));  ⇒  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
> H:= [x, y, z, h, r, t, k, p, w, z, d]:  H[s] $ s=1 .. 11;  ⇒  x, y, z, h, r, t, k, p, w, z, d
> seq(H[s], s = 1 .. 11);  ⇒  x, y, z, h, r, t, k, p, w, z, d
```

С учетом сказанного приведенный фрагмент весьма прозрачен и особых пояснений не требует. Более детально вопросы, связанные с **$**-оператором и ***seq***-функцией, будут рассмотрены в связи с обсуждением функциональных средств языка. Здесь лишь отметим, что при всей близости обоих средств между ними существует различие, выражающееся в более универсальном характере ***seq***-функции в плане ее применимости для *генерации* последовательностей, что будет проиллюстрировано ниже на примерах и детально рассмотрено в [12,41-43,103].

• ***Список*** (*list*) **–** *списочная* структура, широко используемая пакетом для организации вычислений и обработки разнообразной информации, образуется посредством помещения последовательности выражений в квадратные скобки, формируя конструкцию следующего общего вида:

List := [B1, B2, B3, ..., Bn], где **Bj** (**j=1 .. n**) – *любое допустимое выражение* ***Maple****-языка*

Длина списка определяется числом входящих в него элементов, а идентичными полагаются списки, имеющие одинаковую длину и одинаковые значения соответствующих элементов. По конструкции вида **List[n]** можно получать значение **n**-го элемента списка с **List**-идентификатором, а на основе вызова функции ***nops*(List)** – число его элементов. При этом, по вызову функции ***op*(List)** можно конвертировать *списочную* **List**-структуру в *последовательность*. *Замена* **к**-го элемента *списка* **L** выполняется по конструкции **L[k] :=** <*Выражение*>, тогда как для его удаления используется конструкция ***subsop*(k = *NULL*, L).** Функция ***type*** идентифицирует *тип* списочных структур как *list*. Следующий простой пример иллюстрируют вышесказанное:

```
> L:=[sqrt(25),September,2,2006,GS]: [L[4],nops(L)],op(L);  ⇒ [2006,5], 5,September,2, 2006,GS
> L[2]:= October:  L;  L:= subsop(3 = NULL, L):  L, type(L, 'list');  ⇒ [5, October, 2, 2006, GS]
                                                                        [5, October, 2006, GS], true
```

Между тем, для *удаления* **к**-го элемента *списка* **L** с последующим *обновлением* списка *«на месте»* можно использовать конструкцию вида **L[k] := 'NULL',** весьма удобную в целом ряде приложений. Однако данный подход, корректно работая в одиночных вычислениях, не дает искомого результата в *циклических*. Поэтому во втором случае после каждого вычисления вида **L[k]:= 'NULL'** следует использовать операцию присвоения **L:= L**. Следующий весьма простой фрагмент иллюстрирует вышесказанное:

```
> restart; L:= [q, w, e, r, t, y, u, j, o, p, a, d, g, h, k, z, x]:  L[10]:= 'NULL':  L;
                                  [q, w, e, r, t, y, u, j, o, a, d, g, h, k, z, x]
> L:=[q,w,e,r,t,y,u,j,o,p,a,d,g,h,k,z,x]:  while L <> [] do L[1]:= 'NULL': L:= L  end do: L  ⇒ []
```

На основе предложенного приема можно предложить несложную процедуру ***mlist***, обновляющую список на месте. Ее исходный текст и примеры применения даются ниже.

```
> L:= [q,w,e,r,t,y,u,j,o,p,d,a,s,k,x,z,y]:  subsop(5 = VSV, L), subsop(10 = NULL, L),  L;
      [q, w, e, r, VSV, y, u, j, o, p, d, a, s, k, x, z, y], [q, w, e, r, t, y, u, j, o, d, a, s, k, x, z, y],
                                  [q, w, e, r, t, y, u, j, o, p, d, a, s, k, x, z, y]
> mlist:= proc(L::uneval, a::posint, b::anything) if type(eval(L), 'list') then if belong(a, 1 ..
nops(eval(L))) then L[a]:= b; L:=eval(L); NULL else error "in list <%1> does not exist
```

```
element with number %2", L, a end if else error "1st argument should be a list but had
received <%1>", whattype(eval(L)) end if end proc;
 mlist := proc(L::uneval, a::posint, b::anything)
      if type( eval( L ), 'list' ) then
           if belong( a, 1 .. nops( eval( L ) ) ) then L[ a ] := b; L := eval( L ); NULL
           else error "in list <%1> does not exist element with number %2",L, a
           end if
      else error
           "1st argument should be a list but had received <%1>",whattype( eval( L ) )
      end if
 end proc
> mlist(L, 10, 'NULL'), L;   ⇒  [q, w, e, r, t, y, u, j, o, d, a, s, k, x, z, y]
> mlist(L, 16, 'NULL'), L;   ⇒ [q, w, e, r, t, y, u, j, o, d, a, s, k, x, z]
> mlist(L, 3, AVZ), L;       ⇒ [q, w, AVZ, r, t, y, u, j, o, d, a, s, k, x, z]
> L:= [q,w,e,r,t,y,u,j,o,p,d,a,s,k,x,z,y]: while L <> [] do mlist(L,1,'NULL') end do: L;   ⇒ []
> mlist(L, 64, AGN),  L;
Error, (in mlist) in list <L> does not exist element with number 59
> mlist(B, 64, AGN),  L;
Error, (in mlist) 1st argument should be a list but had received <symbol>
```

Вызов процедуры ***mlist*(L, a, b)** возвращает ***NULL***-значение, заменяя **a**-й элемент списка **L** на ***Maple***-выражение **b**; при этом, обновление списка производится *«на месте»*. Тогда как вызов процедуры ***mlist*(L, a, 'NULL')** также возвращает ***NULL***-значение, удаляя **a**-й элемент из списка **L** и обновляя список *«на месте»*. В ряде приложений процедура ***mlist*** представляется достаточно полезной.

Списки могут иметь различный *уровень вложенности*, определяя различного рода структуры данных и конструкции, как это иллюстрирует следующий простой пример:

> Avz:= [[[1, 2, 3], [5, 6]], [6, G], V, S]: op(Avz), nops(Avz); ⇒ [[1, 2, 3], [5, 6]], [6, *G*], *V*, S, 4

Для *вложенных* списков, чьи элементы имеют одинаковую длину, ***Maple*** определяет тип *listlist*, играющий весьма важную роль при организации *индексированных* структур (*матрица, массив и др.*). Нами дополнительно определен тип *nestlist* [103], характеризующий более общий тип вложенности списков, например:

> Avz:= [[[1, 2, 3], [5, 6]], [6, G], V, S]: type(Avz, 'listlist'), type(Avz, 'nestlist'); ⇒ *false, true*
> Agn:= [[1, 2, 3], [a, b, c], [x, y, z]]: type(Agn, 'listlist'), type(Agn, 'nestlist'); ⇒ *true, true*

Пустой список обозначается как **L0:=[]** и ***nops*(L0); ⇒** 0, ***op*(L0); ⇒** *NULL*. ***Maple***-язык располагает весьма обширным набором функциональных средств для работы со *списочными* структурами, которые будут рассматриваться нами довольно детально ниже. Пакетный модуль **ListTools** содержит набор средств для работы со списками. В свою очередь, наша *Библиотека* [103] также содержит ряд полезных средств для такого типа структур.

Списочные структуры являются весьма широко используемыми пакетом объектами (*исходные и выходные данные, управление порядком вычислений, представление массивов, матриц и тензоров и др.*). При этом, в достаточно широких пределах допускается сочетание их с другими типами данных и структур данных.

• ***Множество*** (*set*) – структура данных, весьма широко используемая пакетом, в первую очередь, для организации вычислений в традиционном *теоретико-множественном* смысле, образуется посредством помещения последовательности в фигурные скобки, формируя конструкцию следующего простого вида:

Set := {B1, B2, B3, ..., Bn}, где **Bj** (**j=1 .. n**) – *любая допустимая конструкция **Maple**-языка*

Мощность множества определяется числом входящих в него элементов, а идентичными полагаются множества, имеющие одинаковый набор вычисленных значений элементов без учета их кратности. По конструкции вида **Set[n]** можно получать значение **n**-го элемента множества с ***Set***-идентификатором, а на основе вызова функции ***nops*(Set)** - число его элементов. Более того, по вызову функции ***op*(Set)** можно *конвертировать* ***Set***-множество в последовательность. При этом, функция ***type*** идентифицирует тип *множественных* структур как *set*. Следующий пример иллюстрируют вышесказанное:

> Set1:= {q,6/3,e,r,5,t,w}: Set2:= {q,w,e,r,2,t,w,sqrt(25),r,q,q}: [op(Set1)], [op(Set2)], [nops(Set1)], [nops(Set2)], Set2[5]; ⇒ [2, 5, *r, q, e, t, w*], [2, 5, *r, q, e, t, w*], [7], [7], *e*

Из приведенного примера можно заметить ряд особенностей применения функций ***op*** и ***nops*** к структуре данных *set*-типа. Прежде всего, производится вычисление элементов множества и их *упорядочивание* согласно соглашениям пакета, поэтому порядки элементов исходного множества и результата его вычисления могут не совпадать. Это же относится и к мощности множества - в результате вычисления его элементов одинаковые результаты сводятся к одному, например: **S := {56/2, 7*4, 28, 7*2^2}: nops(S);** ⇒ 1. *Пустое* множество обозначается как **S0:= {}** и ***nops*(S0);** ⇒ 0, ***op*(S0);** ⇒ *NULL*.

Maple-язык поддерживает ряд операций над множествами, аналогичных классическим теоретико-множественным операциям, включая базовые операции *объединения* (**union**), *разности* (**minus**) и *пересечения* (**intersect**) множеств, например:

> S1:= {q,r,64,t,w,x,h,z}: S2:= {h,n,v,x,59,z,k,s}: S1 union S2, S1 intersect S2, S2 minus S1;

{59, 64, *x, h, s, z, n, r, v, q, t, w, k*}, {*x, h, z*}, {59, *s, n, v, k*}

Множества могут иметь различный уровень вложенности, определяя различного рода структуры данных и конструкции. Нами дополнительно определен тип *setset* [103], аналогичный стандартному типу *listlist* для случая списков, например:

> map(type, [{{}}, {{},{}}, {{{7}}}, {{a, b}, {c, d}}, {{a, b}, {c}}, {{10, 17}, {64, 59}, {39, 44}}], 'setset');

[*true, true, true, true, false, true*]

Ввиду принципиально различных принципов упорядочения элементов *списка* и *множества* для *замены* элементов множества можно использовать два способа, а именно: (**1**) по номеру элемента и (**2**) по его значению, как это иллюстрирует следующий фрагмент:

(**1**) **S := S minus {S[k]} union {**<*Выражение*>**}**
(**2**) **S := subs(Sk =** <*Выражение*>**, S)**

где **S** – произвольное множество и **S[k], Sk** – его **к**-й элемент и значение **к**-го элемента соответственно. Конкретные примеры иллюстрируют применение обоих способов:

> S:= {q, r, 64, t, w, x, h, z}; ⇒ *S* := {64, *x, h, z, r, q, t, w*}
> S:= S minus {S[4]} union {Avz}; ⇒ *S* := {64, *x, h, r, q, t, w, Avz*}
> S:= subs(Avz = Agn, S); ⇒ S := {64, *x, h, r, q, t, w, Agn*}
> S:= subs(Agn = NULL, S); ⇒ S := {64, x, r, h, q, t, w}

Одной из наиболее полезных и часто используемых над объектами типов *set* и *list* является ***map***-функция ***Maple***-языка, имеющая в общем виде формат кодирования:

***map*(F, f(x, y, z, ...), a, b, c, …)** ⇒ **f(F(x, a, b, c, …), F(y, a, b, c, …), F(z, a, b, c, …), ...)**

где **F** – некоторая функция или процедура, **f** – выражение от переменных **x, y, z, ...** и **a, b, c, …** – некоторые выражения, и возвращающая результат применения функции или процеду-ры **F** к каждому аргументу **f(x, y, z, ...)**-конструкции (*а в более общей постановке, к каждому операнду выражения*), как это иллюстрирует следующий простой фрагмент:

> map(F, [x, y, z], a, b, c, d); ⇒ [F(x, a, b, c, d), F(y, a, b, c, d), F(z, a, b, c, d)]
> map(F, {x, y, z}, a, b, c, d); ⇒ {F(x, a, b, c, d), F(y, a, b, c, d), F(z, a, b, c, d)}
> map(F, f(x, y, z), a, b, c, d); ⇒ f(F(x, a, b, c, d), F(y, a, b, c, d), F(z, a, b, c, d))
> Seq:= x, y, z: map(F, Seq, a, b, c, d); ⇒ F(x, y, z, a, b, c, d)

Между тем, если по **map(F, V, a1, ..., an)**-функции применяется определенная ее первым фактическим **F**-аргументом функция к *каждому* операнду **V**-выражения с передачей ей дополнительных фактических **aj**-аргументов (**j=1..n**), то по ее модификации – функции ***map2*** вида **map2(F, a1, V, a2, ..., an)** производится *применение* к **V**-выражению **F**-функции со *специфическим* ***a1***-аргументом и возможностью передачи ей дополнительных **aj**-аргументов (**j=2 .. n**), что является существенным расширением ***map***-функции. Следующий пример иллюстрирует принцип формального выполнения ***map2***-функции:

> map2(F, a1, [x,y,z], a, b, c, d, e); ⇒ [F(*a1*, *x, a, b, c, d, e*),F(*a1*, *y, a, b, c, d, e*),F(*a1*, *z, a, b, c, d, e*)]
> map2(S, x, G(a, b, c), y, z, h, t); ⇒ G(S(*x, a, y, z, h, t*), S(*x, b, y, z, h, t*), S(*x, c, y, z, h, t*))
> map(F, W, x, y, z, t, h), map2(F, W, x, y, z, t, h); ⇒ F(W, *x, y, z, t, h*), F(W, *x, y, z, t, h*)

Из данного фрагмента несложно усматривается не только *общий* принцип выполнения функции ***map2***, но и ее эквивалентность ***map***-функции на определенных наборах аргументов. И выше, и в дальнейшем обе эти функции достаточно часто используются в иллюстративных примерах, лучше раскрывая припцип своего выполнения. Нами был определен ряд новых процедур ***map3, map4, map5, map6, mapN, mapLS, mapTab*** (*расширяющих возможности функций* ***map*** *и* ***map2***), описанных в нашей книге [103] и включенных в состав прилагаемой к ней *Библиотеке* [109]. ***Maple***-язык располагает обширным набором средств для работы с множествами и списками, которые будут рассматриваться нами детальнее ниже в различных контекстах. Более того, данный набор дополнен нашими средствами работы со списочными структурами [103,109].

• ***Массив*** (*array*) **–** структура данных, весьма широко используемая ***Maple,*** в первую очередь, при работе с матричными и векторными объектами. *Массив* представляет собой определенное обобщение понятия списочной структуры, когда ее элементам приписываются *индексы*, идентифицирующие *местоположение* элементов в структуре. При этом, размерность массива может быть более единицы, а сами индексы элементов могут принимать как положительные, так и отрицательные значения, например:

> M:= array(-10 .. 16, [k$k = 1 .. 27]): M[-7], M[-1], M[0], M[16]; ⇒ 4, 10, 11, 27

Характерной чертой массива является возможность переопределения его элементов, не затрагивая всей структуры в целом. В общем случае **M**-массив определяется конструкцией следующего достаточно простого вида:

M := array(<*Индексная функция*>**,** <*Размерность*>**,** <*Начальные значения*>**)**

где *Индексная функция* определяет специальный *индексный* атрибут (*например*, *symmetric атрибут определяет соотношение* **M[n, m]=M[m, n]**). *Размерность* задается в виде интервалов "**а .. в**" по каждому из *индексов* массива и *Начальные значения* задают исходные значения для элементов массива. При этом, следует иметь в виду, что ***Maple***-язык автоматически не определяет элементы массива, поэтому сразу же после создания массива его элементы являются неопределенными, например: **a:=** ***array*****(1..3):** ***print*****(a);** ⇒ [a_1, a_2, a_3].

Обращение к массиву производится по его идентификатору, а к его элементам в виде индексированного идентификатора, например: ***print*****(M); M[5, 6, 2]**. Вывод массива на печать производится по ***print***-функции, тогда как по функциям ***eval*** и ***evalm*** не только предоставляется возможность вывода содержимого массива, но и возврата его в качестве результата для возможности последующего использования в вычислениях. Примене-

ние данных функций проиллюстрировано ниже. Ни один из *трех* аргументов функции ***array*** не является обязательным, однако должен присутствовать по меньшей мере один из двух последних для возможности построения массива. Наиболее простой формат ***array***-конструкции имеет следующий вид:

M := *array*(J11 .. J1n, J21 .. J2n, ..., Jm1 . .Jmn {, [] | })

определяя *пустой* (*неопределенный*) **M**-массив **mxn**-размерности. *Переопределение* элементов **M**-массива производится по конструкции вида **M[p, k, h, ...]:=**<*Выражение*>, которая остается действительной и для списочных структур. В целом ряде случаев при определении массива целесообразно сразу же задавать начальные значения для его элементов полностью или частично, что иллюстрирует следующий простой фрагмент:

```
> G:= array(1..3, 1..3, []): G[1,1]:= 42: G[1,2]:= 47: G[1,3]:= 67: G[2,1]:= 89:  G[2,2]:= 96:
  G[2,3]:= 99: G[3,1]:= 95: G[3,2]:= 59: G[3,3]:= 62:  print(G);
```

$$\begin{bmatrix} 42 & 47 & 67 \\ 89 & 96 & 99 \\ 95 & 59 & 62 \end{bmatrix}$$

```
> assign(MMM = [eval(G), evalm(G)]), MMM, sum(MMM[k], k = 1..2);
```

$$\left[\begin{bmatrix} 42 & 47 & 67 \\ 89 & 96 & 99 \\ 95 & 59 & 62 \end{bmatrix}, \begin{bmatrix} 42 & 47 & 67 \\ 89 & 96 & 99 \\ 95 & 59 & 62 \end{bmatrix}\right], 2\ _addmultmp$$

```
> eval(_addmultmp), map(sqrt, _addmultmp);
```

$$\begin{bmatrix} 42 & 47 & 67 \\ 89 & 96 & 99 \\ 95 & 59 & 62 \end{bmatrix}, \begin{bmatrix} \sqrt{42} & \sqrt{47} & \sqrt{67} \\ \sqrt{89} & 4\sqrt{6} & 3\sqrt{11} \\ \sqrt{95} & \sqrt{59} & \sqrt{62} \end{bmatrix}$$

```
> S:= array(1..3, 1..3, [[a, b, c], [d, e, f], [g, h, k]]):  map([eval, evala, evalm], S);
```

$$\begin{bmatrix} [a, a, a] & [b, b, b] & [c, c, c] \\ [d, d, d] & [e, e, e] & [f, f, f] \\ [g, g, g] & [h, h, h] & [k, k, k] \end{bmatrix}$$

```
> Art:=array(symmetric, 1..4, 1..4):  Art[1, 1]:= 42:  Art[1, 2]:= 47:  Art[1, 3]:= 67:  Art[1, 4]:= 62:
  Art[2, 2]:=  57:  Art[2, 3]:= 89:  Art[2, 4]:= 96:  Art[3, 3]:= 52:  Art[3, 4]:= 99:  Art[4, 4]:= 9:
> whattype(eval(Art)),  type(Art, 'array');   ⇒  array, true
> [op(0, eval(Art))], [op(1, eval(Art))], [op(2, eval(Art))], [op(3, eval(Art))];
    [array], [symmetric], [1 .. 4, 1 .. 4], [[(1, 1) = 42, (2, 2) = 57, (4, 4) = 9, (1, 2) = 47, (3, 3) = 52,
                    (2, 4) = 96, (2, 3) = 89, (1, 4) = 62, (1, 3) = 67, (3, 4) = 99]]
```

Как следует из приведенного фрагмента в первом примере определяется *пустой* **G**-массив с последующим *прямым* определением его элементов путем *присваивания*; в качестве разделителей предложений выбрано *двоеточие*, чтобы не загромождать документ выводом *промежуточных* результатов. Затем над полученным **G**-массивом производятся простые операции, о которых говорилось выше. Во втором примере определяется **S**-массив той же (3x3)-размерности с одновременным заданием начальных значений для всех его элементов. Результаты определения массива используются для выполнения нестандартной процедуры, использующей рассмотренные выше функции и которая может оказаться полезной в практической работе с объектами типа *array*. Там же иллюстрируется применение ***map***-функции для *поэлементной* обработки массива. При определения начальных значений элементов непосредственно в ***array***-конструкции они представляются в виде вложенного списка *listlist*-типа, структура которого должна строго соответствовать структуре определяемого массива, как это иллюстрируется фрагментом. В качестве встроенных *индексных* атрибутов пакет использует: *symmetric*, *antisymmetric*, *sparse*,

diagonal, *identity* и ряд определяемых пакетными модулями, которые детально обсуждаются при рассмотрении матричных объектов, например, в нашей книге [12].

При этом, массив одновременно выводится и возвращается по функциям ***evalm*** и ***eval***, тогда как по ***print***-функции производится *только* вывод массива на экран. Предыдущий фрагмент иллюстрирует сказанное. Наряду с этим, примеры *фрагмента* иллюстрируют использование функции ***op*** для получения *типа* ***Art***-объекта, *индексной функции* (*первый аргумент* ***array***-*функции*), *размерности* массива (*второй аргумент*) и содержимого всех начальных значений для входов массива (*третий аргумент*). Иллюстрируется и применение ***map***-функции для поэлементной обработки массива.

Ниже обсуждение структур данных *array-типа* будет детализироваться и рассматриваться на протяжении последующих глав книги. При этом, следует иметь в виду то обстоятельство, что частным случаем понятия *массива* являются *векторы* (*одномерный массив*) и *матрицы* (*двухмерный прямоугольный массив*), для обеспечения работы с которыми ***Maple***-язык располагает достаточно развитым набором средств, прежде всего, определяемых пакетным модулем **linalg,** ориентированным на решение задач линейной алгебры [11-14,80-89]. Ниже этот вопрос будет рассмотрен несколько детальнее.

• ***Массив*** *hfarray-типа*. Для обеспечения численных вычислений на основе машинной *арифметики с плавающей точкой* (**МАПТ**) ***Maple***-язык поддерживает новый тип структуры данных – массивы *hfarray*-типа, определяемые ***hfarray***-функцией формата вида:

Mhf := *hfarray*({<*Размерность*>}{**,** <*Начальные значения*>}**)**

где назначение формальных аргументов функции полностью аналогично случаю функции ***array*****;** при этом, оба аргумента необязательны. Размерность определяется одним либо несколькими *ранжированными* выражениями, при ее отсутствии используется установка по *умолчанию*. Пределы изменения индексов по каждому измерению массива определяются используемой платформой и для 32-битной платформы находятся в диапазоне от -2147483648 до 2147483647 и для 64-битной платформы от -9223372036854775808 до 9223372036854775807.

Данные массивы составляют специальный *hfarray*-тип, распознаваемый тестирующими функцией ***type*** и процедурой ***whattype***. Массивы данного типа в качестве *значений* своих элементов содержат числа с *плавающей точкой двойной* точности стандарта **IEEE-754**. При этом, поддерживаются три специальные значения данного стандарта: **+Infinity**, **-Infinity** и **Quiet NaN,** последнее из которых определяется как **`*undefined*`**. Для ***hfarray***-функции в качестве начальных значений могут выступать любые допустимые ***Maple***-выражения, результатом вычисления которых являются числа *float*-типа, или значения *undefined*, *infinity*, *-infinity*. Полностью массивы *hfarray*-типа возвращаются и выводятся на экран соответственно по функциям ***eval*** и ***print***. Следующий фрагмент иллюстрирует вышесказанное:

```
> МАПТ:= hfarray([evalf([sqrt(10), sqrt(17)]), evalf([sqrt(64), sqrt(59)])]);
```

$$МАПТ := \begin{bmatrix} 3.16227766000000000 & 4.12310562600000008 \\ 8. & 7.68114574799999960 \end{bmatrix}$$

```
> МАПТ:= hfarray([[sqrt(10), sqrt(17)], [sqrt(64), sqrt(59)]]);
Error, unable to store '10^(1/2)' when datatype=float[8]
> type(МАПТ,'hfarray'), type(МАПТ,'array'), whattype(eval(МАПТ)); ⇒ true, false, hfarray
> eval(МАПТ), print(МАПТ);
```

$$\begin{bmatrix} 3.16227766000000000 & 4.12310562600000008 \\ 8. & 7.68114574799999960 \end{bmatrix}$$

$$\begin{bmatrix} 3.16227766000000000 & 4.12310562600000008 \\ 8. & 7.68114574799999960 \end{bmatrix}$$

```
> evalf(array([[sqrt(10), sqrt(17)], [sqrt(64), sqrt(59)]]));
```

$$\begin{bmatrix} 3.162277660 & 4.123105626 \\ 8. & 7.681145748 \end{bmatrix}$$

Структуры данных *hfarray*-типа широко используются с ***evalhf***-функцией, поддерживающей **MAПТ,** и достаточно детально рассматриваются, например, в [11,12]. При этом, следует иметь в виду, что работа с такого типа массивами вне рамок **MAПТ** не эффективна, ибо требуется выполнение соглашений между **MAПТ** и ***Maple***-арифметикой с плавающей точкой. Последний пример фрагмента иллюстрирует различия между массивами типов *array* и *hfarray* при одной и той же установке предопределенной ***Digits***-переменной пакета. Тогда как второй пример фрагмента иллюстрирует недопустимость определения элементов *hfarray*-массива типами данных, отличными от *extended_numeric*-типа (*обобщение типов numeric, undefined либо infinity, -infinity*). Для устранения данного недостатка используется ***evalf***-функция.

• ***Таблица*** (*table*) – структура данных, весьма широко используемая ***Maple,*** прежде всего, при работе с различного рода табличными объектами. *Таблица* представляет собой определенное *обобщение* понятия двумерного массива, когда в качестве значений *индексов* массива могут использоваться не только *целочисленные* значения, но и произвольные выражения, в *первую очередь*, символьные и строчные значения в качестве названия *строк* и *столбцов*. Характерной чертой таблицы является возможность работы со структурами данных, включающими естественные нотации (*фамилии*, *имена*, *названия и т.д.*). Итак, в общем случае **T**-таблица определяется конструкцией следующего простого формата:

T := *table*(<*Индексная функция*>**,** <*Список/множество начальных значений*>**)**

где *Индексная функция* определяет специальный *индексный* атрибут (*например*, *symmetric-атрибут*), аналогичный случаю массива, и второй аргумент определяет *исходные* значения для элементов таблицы (ее *входы* и *выходы*). *Пустая* таблица определяется по конструкции вида **T:= *table*();** при этом, первый аргумент ***table***-функции необязателен.

Второй аргумент ***table***-функции определяется, как правило, в виде списка или множества, элементами которого могут быть как отдельные допустимые ***Maple***-выражения, так и уравнения, т.е. конструкции вида "**A=B**". *Левые* **A**-части уравнений выступают в качестве идентификаторов *строк* таблицы и определяют ее *входы*, а *правые – выходы*, т. е. по конструкции формата **T[**<*Вход*>**]** возвращается строка-*выход*, отвечающая указанному в качестве значения аргумента *Входу*. Поэтому, если **T:=table([X1=Y1, X2=Y2, ...]),** то **T[Xk];** ⇒ Yk. Если в списке/множестве начальных значений хоть один из элементов не является уравнением, то ***Maple*** в качестве *входов* в таблицу использует *целые* неотрицательные числа (**1** .. <*число строк*>), располагая при этом строки таблицы в определенном порядке, отличном от естественного. Это одна из причин, почему в качестве элементов второго аргумента функции ***table*** следует кодировать уравнения, т.е. это не тот случай, когда решение вопроса стоит отдавать на откуп пакету. В качестве *правых* **B**-частей уравнений списка/множества функции могут выступать произвольные ***Maple*** объекты, позволяя создавать достаточно сложные *табличные* структуры данных, примером чего может служить следующий весьма простой фрагмент:

```
> T:= table([A=[AV, 42, 64, 350], B=[42, 64, array([[RANS, IAN], [REA, RAC]])],
  C=array([[{G, S}, {Kr, Ar}], [{Vasco}, {Salcombe}]])]):  eval(T);
```

$$\text{table}([C=\begin{bmatrix}\{S,G\} & \{Kr,Ar\}\\ \{Vasco\} & \{Salcombe\}\end{bmatrix}, B=\begin{bmatrix}42, 64, \begin{bmatrix}RANS & IAN\\ REA & RAC\end{bmatrix}\end{bmatrix},$$
$$A=[AV, 42, 64, 350]$$
$$])$$

При этом, более простым способом создания таблицы является *непосредственное* присвоение индексированной переменной значений, как это иллюстрирует следующий весьма простой фрагмент:

```
> Tab[Grodno]:= 1962: Tab[Tartu]:= 1966: Tab[Tallinn]:= 1999: Tab[Gomel]:= 1995:
  Tab[Moscow]:= 1994: eval(Tab), print(Tab);
        table([Grodno = 1962, Tartu = 1966, Tallinn = 1999, Gomel = 1995, Moscow = 2006])
        table([Grodno = 1962, Tartu = 1966, Tallinn = 1999, Gomel = 1995, Moscow = 2006])
> (Tab[Grodno]-Tab[Tallinn]-Tab[Moscow])/(Tab[Tartu]-Tab[Gomel]-350);   ⇒ 2043/379
> Tab[Grodno], Tab[Tartu], Tab[Tallinn], Tab[Gomel], Tab[Moscow];
                              1962, 1966, 1999, 1995, 2006
> whattype(Tab), whattype(eval(Tab)), type(Tab, 'table'), type(eval(Tab), 'table'),
  map2(type, Tab, ['matrix', 'array']);   ⇒ symbol, table, true, true, [false, false]
> op(0, eval(Tab)), [op(1, eval(Tab))], op(2, eval(Tab));
      table, [], [Grodno = 1962, Tartu = 1966, Tallinn = 1999, Gomel = 1995, Moscow = 2006]
```

Обращение к таблице производится по ее идентификатору, а к ее элементам в форме *индексированного* идентификатора, например: **T[C]**. Вывод таблицы на печать производится по функции ***print*** (*применение которой проиллюстрировано выше*); это же относится и к выводу ее отдельных выходов, если они не являются конструкциями базовых типов {*число*, *строка*, *список*, *множество*}. При этом, подобно массиву таблица одновременно выводится и возвращается по функции ***eval*,** тогда как к ней не применимы функции ***evalm*** и ***evala***. Приведенный выше фрагмент иллюстрирует сказанное. Наряду с этим, примеры фрагмента иллюстрируют использование ***op***-функции для получения типа ***Tab***-объекта, индексной функции (*первый аргумент* ***table****-функции*) и содержимого всех входов таблицы. По причине отсутствия индексной функции вызов ***op*(1, *eval*(*Tab*))** возвращает ***NULL***-значение.

Наиболее простой формат ***table***-функции имеет следующий вид:

T := *table*([X1, X2, X3, ..., Xn]) или **T := *table*({X1, X2, X3, ..., Xn})**

определяя **T**-таблицу из **n** строк, идентифицируемых входами-числами. *Переопределение* элементов **T**-таблицы производится по конструкциям вида: **T[**<*вход*>**]:=**<*Выражение*>, которые работают и со списочными структурами. Для удаления из **T**-таблицы выхода (*соответствующего заданному входу*) выполняется следующая конструкция: **T[***вход***]:=*NULL*,** тогда как для удаления *всех* выходов произвольной **T**-таблицы достаточно выполнить следующее простое ***Maple***-предложение цикла:

for k in [<*Список входов*>**] do T[k]:= *NULL* end do:**

либо эквивалентное ему выражение следующего вида:

***eval*(*parse*(*convert*(*subs*(F = *null*, *mapTab*(F, T, 1)), 'string')))**

где ***null*** и ***mapTab*** – процедуры из *Библиотеки*, описание которых находится в [103]:

```
> Tab[Grodno]:= 1962: Tab[Tartu]:= 1966: Tab[Tallinn]:= 1999: Tab[Gomel]:= 1995:
  Tab[Moscow]:= 2006: eval(parse(convert(subs(F = null, mapTab(F, Tab, 1)), 'string')));
                table([Tartu = (), Tallinn = (), Gomel = (), Moscow = (), Grodno = ()])
```

При работе с *массивами* и *таблицами* наиболее часто используемыми являются две функции ***indices*** и ***entries*,** имеющие следующий простой формат кодирования;

{***indices*** | ***entries***}**(T),** где **T –** *массив или таблица*

и возвращающие *индексы/входы* и соответствующие им *элементы/выходы* массива/таблицы **T** соответственно, как это иллюстрирует следующий простой фрагмент:

```
> Tab[Grodno]:= 1962:  Tab[Tartu]:= 1966:  Tab[Tallinn]:= 1999:  Tab[Gomel]:= 1995:
  Tab[Moscow]:= 2006:   indices(Tab),  entries(Tab);
        [Tartu], [Tallinn], [Gomel], [Moscow], [Grodno],  [1966], [1999],  [1995], [2006], [1962]
> A:= array([[a, b, h], [c, d, k], [x, y, z]]):  indices(A),  entries(A);
   [1, 1], [2, 2], [2, 3], [2, 1], [3, 1], [3, 2], [1, 2], [1, 3], [3, 3],    [a], [d], [k], [c], [x], [y], [b], [h], [z]
> with(Tab);  eval(%);
                                  [Gomel, Grodno, Moscow, Tallinn, Tartu]
                                    [1995, 1962, 2006, 1999, 1966]
> Tab1[1942]:= 1962:  Tab1[2006]:= 1966: type(Tab1, 'table');  with(Tab1);   ⇒  true
Error, (in pacman:-pexports) invalid arguments to sort
> map(op, [indices(Tab)]), map(op, [entries(Tab)]);
                  [Tartu, Tallinn, Gomel, Moscow, Grodno],  [1966, 1999, 1995, 2006, 1962]
```

В частности, последний пример фрагмента представляет конструкции, полезные при работе, прежде всего, с таблицами, и позволяющие представлять их *входы* и *выходы* в виде списка. Во фрагменте (*пример* **3**) проиллюстрировано также использование процедуры ***with*** (*применяемой, как правило, лишь с пакетными и программными модулями*) для получения списка *входов* таблицы. При этом, следует учитывать, что если в качестве *входов* **T**-таблицы выступают значения {*symbol* | *name*}-типа, то по выхову ***with*(T)** возвращается их отсортированный *лексикографически* список, в противном случае возникает ошибочная ситуация, как это иллюстрирует *пример* **4** предыдущего фрагмента.

Имея многочисленные приложения, табличные структуры используются и для организации пакетных и/или библиотечных *модулей*. Такой подход особенно широко использовался в ранних релизах пакета и об этом детальнее будет сказано ниже. Представим здесь один простой пример такого подхода. В этом случае *входами* таблицы **T** являются *имена* процедур, а ее *выходами* – соответствующие им *определения*. Тогда вызов таким образом погруженных в табличную структуру процедур принимают следующий простой вид, а именно: **T[***имя***](***Аргументы***).** Следующий фрагмент иллюстрирует сказанное:

```
> T:= table([sr = (() -> `+`(args)/nargs), (ds = (() -> sqrt(sum((args[k] - sr(args))^2,
  k=1..nargs)/nargs)))]);
```

$$T := \mathrm{table}\left([sr = \left(() \to \frac{\texttt{`+`}(\mathrm{args})}{\mathrm{nargs}}\right), ds = \left(() \to \sqrt{\frac{\sum_{k=1}^{\mathrm{nargs}} (\mathrm{args}_k - \mathrm{sr}(\mathrm{args}))^2}{\mathrm{nargs}}}\right)]\right)$$

```
> with(T), 6*T[sr](64,59,10,17,39,44), 6*T[ds](64,59,10,17,39,44);  ⇒ [ds, sr], 233, √14249
> save(T,"C:\\Temp\\lib.m");  restart;  read("C:\\Temp\\lib.m"); with(T);  ⇒ [ds, sr]
> 6*T[sr](64,59,10,17,39,44), 6*T[ds](64,59,10,17,39,44);   ⇒  233, √14249
```

Данный подход эффективен, например, когда требуется создавать средства модуля пакета табличного типа поэтапно **–** *экспорты* (*входы таблицы*). В этом случае одноименные *входы* заменяют соответствующие им *выходы*, тогда как новые входы просто добавляются в таблицу. Данный подход имеет и другие привлекательные особенности.

Относительно структур типа *массив* (*array*) и *таблица* (*table*) следует сделать *одно* существенное пояснение. Все используемые ***Maple*** структуры, *кроме этих двух*, обладают свойством ***ненаследования*,** т.е. для них справедливы следующие соотношения:

X:= a: Y:= X: Y:= b: [X, Y]; ⇒ [a, b]

т.е. присвоение **X**-значения **Y**-значению с последующим переопределением второго не изменяет исходного **X**-значения. В случае же структур типа *массив* и *таблица* это вполне естественное свойство не соблюдается. Поэтому ***Maple***-язык располагает специальной процедурой ***copy*(<*Массив/Таблица*>),** позволяющей создавать копии указанного массива/таблицы, модификация которой не затрагивает самого оригинала. Следующий весьма простой фрагмент иллюстрирует вышесказанное:

```
> X:= a:   Y:= X:    Y:= b:    [X, Y];      ⇒ [a, b]
> A:= array(1..5, [42, 47, 67, 88, 96]);   ⇒ A := [42, 47, 67, 88, 96]
> C:= copy(A);   ⇒ C := [42, 47, 67, 88, 96]
> C[3]:= 39:  C[5]:= 10:  [A[3], A[5]];   ⇒ [67, 96]
> B:= A:  B[3]:= 95:  B[5]:= 99:  [A[3], A[5]];   ⇒ [95, 99]
```

Создание по ***copy***-процедуре копий объектов указанных двух типов позволяет модифицировать только их копии без изменения самих объектов-оригиналов.

В качестве встроенных индексных атрибутов ***Maple*** для ***table***-функции используются: *sparse*, *symmetric*, *antisymmetric*, *diagonal* и *identity* (*а также определяемые пакетными модулями*), детальнее обсуждаемые при рассмотрении матричных объектов. Обсуждение *table*-типа структуры данных будет постоянно детализироваться и многоаспектно рассматриваться на протяжении последующих глав книги. Целый ряд полезных средств для работы с *табличными* объектами представляет и наша *Библиотека*, прилагаемая к книге [103] и находящаяся в свободном доступе по адресам [109].

В завершение рассмотрения структур данных, поддерживаемых пакетом, будет вполне уместно представить и связанные с объектами *array*-типа, уже упоминаемые *матрицы* и *векторы*. *Матрицы* являются *весьма* широко применимым понятием в целом ряде естественно-научных дисциплин, составляя самостоятельный объект исследования современной математики **–** *теорию матриц*, выходящую за рамки традиционного университетского курса высшей алгебры. Основу теории матриц составляет алгебра *квадратных* матриц, для обеспечения которой ***Maple***-язык располагает целым рядом важных и полезных средств, поддерживаемых функциональными средствами **–** модулей **LinearAlgebra** и **linalg.** Данные модули содержат определения **118** и **114** процедур соответственно (*в зависимости от релиза пакета***;** *пример приведен для* ***Maple 10***), обеспечивающих расширенные средства линейной алгебры по работе с матричными и векторными выражениями. Данные средства не входят в задачу настоящей книги, акцентирующей внимание на вопросах собственно программирования в ***Maple*,** поэтому здесь мы *лишь* представим основные структуры, с которыми имеют дело средства *линейной алгебры* пакета, а именно: *матрицы* (*matrix*) и *векторы* (*vector*).

• ***Матрицы*** (*matrix*); в среде ***Maple***-языка *матрица* представляется в виде 2D-*массива*, нумерация строк и столбцов которого производится, начиная с *единицы*. Матрица определяется либо явно через рассмотренную выше ***array***-функцию, например **M:= *array*(1..n, 1..p),** или посредством процедуры ***matrix*,** имеющей следующие два простых формата кодирования, а именно:

***matrix*(L)** и ***matrix*(n, p, {L | Fn | Lv})**

где **L –** вложенный список (*listlist*) векторов элементов матрицы, **n** и **p –** число ее строк и столбцов, **Fn –** функция генерации элементов матрицы и **Lv –** *список* или *вектор* элементов матрицы. Функция **Fn** генерации элементов матрицы определяется в форме пользовательских функции или процедуры, например **Fn:=(k,j) -> Φ(k,j),** такой, что **M[k,j]:= Fn(k,j).** Процедура ***randmatrix*** модуля **linalg** позволяет стохастически генерировать матрицы (**mxn**)-размерности, что удобно, например, для различного рода отладок и тестов.

По тестирующей функции ***type*(M, {matrix|'matrix'(K{, *square*})})** возвращается ***true***-значение, если **M**-выражение является матрицей; при этом допускается проверка на принадлежность значений ее элементов заданной **K**-области и/или *квадратности* (***square***) матрицы. В противном случае возвращается ***false***-значение. Так как матрица является одним из типов более общего понятия *массив*, то функция ***type*(M, 'array')** также возвращает ***true***-значение, если **M** - матрица. Обратное же в общем случае неверно, например, в случае одномерных массивов-векторов. По функции ***convert*(L, 'matrix')** возвращается матрица, если **L -** соответствующая ей вложенная списочная структура. С другой стороны, по функции ***convert*(M, 'multiset')** производится конвертация матричной структуры в структуру *вложенных* списков, каждый элемент-список которой содержит три элемента, где первые два определяют координаты элемента матрицы, а третий **–** его значение. Следующий фрагмент иллюстрирует вышесказанное.

> M:= matrix(3, 3, [x, y, z, a, b, c, V, G, S]);

$$M := \begin{bmatrix} x & y & z \\ a & b & c \\ V & G & S \end{bmatrix}$$

> type(M, 'array'), type(M, 'matrix'), type(M, 'matrix'(symbol, square)); ⇒ *true*, *true*, *true*

> convert([[x, y, z], [a, b, c], [V, G, S]], 'matrix');

$$\begin{bmatrix} x & y & z \\ a & b & c \\ V & G & S \end{bmatrix}$$

> convert(M, 'multiset');

[[3, 2, G], [2, 2, b], [1, 1, x], [1, 3, z], [1, 2, y], [2, 1, a], [3, 1, V], [2, 3, c], [3, 3, S]]

> convert((a+b)*x/(c+d)*y, 'multiset'); ⇒ [[a+b, 1], [x, 1], [c+d, -1], [y, 1]]

Следует отметить, что функция ***convert*(M, 'multiset')** имеет *более* широкое применение, допуская в качестве своего первого фактического **M**-аргумента *любое* ***Maple***-выражение (*при этом, трактовка составляющих возвращаемого ею вложенного списка весьма существенно зависит от типа* **M**-*выражения*), однако наибольший смысл она имеет для **M**-выражений типа массив или содержащих термы-множители. Не отвлекаясь на частности, рекомендуем читателю определить влияние типа **M**-выражения на семантику возвращаемого функцией вложенного списка, тем более, что в ряде случаев это может оказаться весьма полезным приемом в практическом программировании.

Обращение к элементам **M**-матрицы производится по *индексированной* **M[k, j]**-конструкции, идентифицирующей ее (**k, j**)-й элемент. Посредством этой конструкции элементам **M**-матрицы можно как присваивать значения, так и использовать их в вычислениях в качестве обычных переменных. Следовательно, данная *индексированная* конструкция обеспечивает адресное обращение к элементам **M**-матрицы. Совместное использование функций ***map*** и **F**-функции/процедуры позволяет применять последнюю одновременно ко *всем* элементам **M**-матрицы, не идентифицируя их отдельно, т.е. имеет место следующее определяющее соотношение:

***map*(F, M {, <*опции*>}) ≡ ∀(k)∀(j) (M[k, j] := F(M[k, j] {, <*опции*>}))**

Например:

> M:= matrix(3, 3, [x, y, z, a, b, c, V, G, S]): map(F, M, Art, Kr, Arn);

$$\begin{bmatrix} \mathrm{F}(x, Art, Kr, Arn) & \mathrm{F}(y, Art, Kr, Arn) & \mathrm{F}(z, Art, Kr, Arn) \\ \mathrm{F}(a, Art, Kr, Arn) & \mathrm{F}(b, Art, Kr, Arn) & \mathrm{F}(c, Art, Kr, Arn) \\ \mathrm{F}(V, Art, Kr, Arn) & \mathrm{F}(G, Art, Kr, Arn) & \mathrm{F}(S, Art, Kr, Arn) \end{bmatrix}$$

В целом же **M**-матрица рассматривается как единый объект и для его вывода или возвращения в матричной нотации можно использовать соответственно функции ***print*** и {***op***|***evalm***}. Однако на основе *адресного* подхода обработку матриц можно производить и рассмотренными функциональными средствами, ориентированными на сугубо скалярные аргументы. В частности, по конструкции ***numboccur*(*eval*(M),** <*Элемент*>**)** можно тестировать наличие в указанной **M**-матрице заданного элемента. Позволяет это делать и наша процедура ***belong*** [103,109].

С другой стороны, целый ряд процедур позволяют обрабатывать **M**-матрицу как единый объект; такие функции будем называть *матричными*, т.е. в качестве одного из ведущих аргументов их выступает идентификатор матрицы. Язык ***Maple*** располагает достаточно обширным набором матричных процедур, характеристика которых весьма детально рассмотрена в книгах [8-14,55-60,86-88] с той или иной степенью охвата и в полном объеме в справке по пакету. Лишь некоторые из них будут рассмотрены ниже.

По функции ***evalm*(VM)** возвращается результат вычисленим **VM**-выражения, содержащего матрицы, применяя функциональные ***map***-преобразования над матрицами. Однако, применение ***evalm***-функции требует особой осмотрительности, ибо ***Maple***-язык перед передачей фактических аргументов ***evalm***-функции может производить их *упрощения* (*предварительные вычисления*), в ряде случаев приводящие к некорректным с точки зрения матричной алгебры результатам. Например, по ***evalm*(*diff*(M, x))** возвращается нулевое значение, тогда как предполагается результат дифференцирования **M**-матрицы. Это связано с тем обстоятельством, что отличные от матричных функции воспринимают идентификатор матрицы неопределенным. В **VM**-выражении все неопределенные идентификаторы полагаются ***evalm***-функцией в зависимости от их использования *символьными* матрицами либо скалярами. В суммах, содержащих матрицы, все скалярные константы рассматриваются умноженными на единичную матрицу, что позволяет естественным образом определять *матричные полиномы*. Так как операция произведения матриц *некоммутативна*, то для нее следует использовать матричный (**&***)-оператор произведения, приоритет которого идентичен приоритету (*)-оператора скалярного произведения. В качестве операндов бинарного матричного (**&***)-оператора могут выступать матрицы либо их идентификаторы. Следующий фрагмент иллюстрирует способы определения, тестирования и вычисления матричных выражений:

```
> M1:= convert([[42, 47], [67, 89]], 'matrix'):  M2:=matrix(2, 3, [x, x^2, x^3, x^3, x^2, x]):
> M3:= matrix([[T, G], [G, M]]):  M4:=matrix(2, 2, [ln(x), x*sin(x), Catalan*x^2, sqrt(x)]):
> map(type, [M1, M2, M3, M4], 'matrix');   ⇒  [true, true, true, true]
> map(op, [M1, M2, M3, M4]);
```

$$\left[\begin{bmatrix}42 & 47\\ 67 & 89\end{bmatrix}, \begin{bmatrix}x & x^2 & x^3\\ x^3 & x^2 & x\end{bmatrix}, \begin{bmatrix}T & G\\ G & M\end{bmatrix}, \begin{bmatrix}\ln(x) & x\sin(x)\\ Catalan\,x^2 & \sqrt{x}\end{bmatrix}\right]$$

```
> [map(diff, M2, x),  map(int, M4, x)];
```

$$\left[\begin{bmatrix}1 & 2x & 3x^2\\ 3x^2 & 2x & 1\end{bmatrix}, \begin{bmatrix}x\ln(x)-x & \sin(x)-x\cos(x)\\ \dfrac{Catalan\,x^3}{3} & \dfrac{2x^{(3/2)}}{3}\end{bmatrix}\right]$$

```
> evalm(10*M1^2 + 17*M1);
```

$$\begin{bmatrix}49844 & 62369\\ 88909 & 112213\end{bmatrix}$$

```
> evalm(M4&*(M1 + M3));
```

$$\begin{bmatrix}\ln(x)(42+T)+x\sin(x)(67+G) & \ln(x)(47+G)+x\sin(x)(89+M)\\ Catalan\,x^2(42+T)+\sqrt{x}(67+G) & Catalan\,x^2(47+G)+\sqrt{x}(89+M)\end{bmatrix}$$

С учетом сказанного, примеры фрагмента представляются достаточно прозрачными и каких-либо особых дополнительных пояснений не требуют.

• ***Векторы*** (*vector*); в среде ***Maple***-языка *вектор* представляется в виде **1D**-*массива*, нумерация строк и столбцов которого производится, начиная с *единицы*. Вектор определяется либо явно через рассмотренную выше ***array***-функцию, например **V:= *array*(1..n),** или посредством процедуры ***vector*,** имеющей следующие два простых формата:

***vector*([X1, X2, ..., Xn])** и ***vector*(n, { |Fn|[X1, X2, ..., Xn]})**

где **Xk** - элементы вектора, **n** - число его элементов и **Fn** - функция генерации элементов *вектора* аналогично случаю определения матриц. По тестирующей же функции ***type*(V,** {**'vector'(K)** | **vector**}**)** возвращается значение ***true*,** если **V**-выражение является вектором; при этом, допускается проверка на принадлежность значений его элементов заданной **K**-области. В противном случае функцией возвращается ***false***-значение. Так как вектор является одним из типов общего понятия *массив*, то функция ***type*(V, 'array')** также возвращает ***true***-значение, если **V –** вектор. Обратное в общем случае неверно, например, в случае списка. По ***convert*(L, 'vector')**-функции возвращается *вектор*, если **L –** соответствующая ему списочная структура.

Следует отметить, что в силу специфики организации *базовых* структур, на которых основываются векторы, массивы, и матрицы, они не вполне однозначно тестируются пакетными процедурой ***whattype*** и функцией ***type*,** как иллюстрирует фрагмент ниже:

```
> v:=vector(3,[]): m:=matrix(3,3,[]): a:= array(1..2,1..3,[]): map(whattype, map(eval, [v,m,a]));
                              [array, array, array]
> map(type, [v,m,a], 'array'), map(type, [v,m,a], 'vector'), map(type, [v,m,a], 'matrix');
          [true, true, true], [true, false, false], [false, true, true]
```

Так, если процедура ***whattype*** отождествляет все типы {*array*, *matrix*, *vector*} как *обобщающий array*-тип, то функция ***type*** обеспечивает *более* дифференцированное тестирование, хотя и в этом случае типы *array* и *matrix* ею не различаются. Предыдущий фрагмент хорошо иллюстрирует сказанное.

• ***Основные характеристики матриц и векторов.*** Прежде всего, среди класса матричных объектов выделим *квадратные* (*square*) матрицы, которые будут основным объектом нашего рассмотрения и для работы с которыми ***Maple***-язык располагает довольно широким набором функциональных средств. Поэтому, если не оговаривается противного, то под понятием «*матрица*» в дальнейшем понимается именно квадратная матрица. Для работы с такого типа матрицами ***Maple*** располагает большим набором средств, находящихся в пакетных модулях **linalg** и **LinearAlgebra**.

Прежде всего, по функции ***indices*(M)** возвращается последовательность **2**-элементных списков, определяющих индексное простанство **M**-матрицы, а по функции ***entries*(M) –** последовательность **1**-элементных списков, содержащих значения элементов **M**-матрицы. В случае использования в качестве **M**-аргумента таблицы функция {***indices*|*entries***} возвращает соответственно ее *входы* и *выходы*. При этом, следует иметь в виду, что в случае матриц обе функции возвращают последовательности списков, имеющих внутреннее взаимно-однозначное соответствие (*при отсутствии оного в их выходных порядках*), не управляемых пользователем. Вместе с тем, на основе несложной ***Pind***-процедуры [103], в качестве *единственного* фактического аргумента использующей выражение типа {*array*, *matrix*, *table*}, можно получать (**nxp**)-размерность произвольной **M**-матрицы в виде [n, p]-списка, как это иллюстрирует следующий весьма простой фрагмент:

> Fn:= (k, j) -> k^2+3*k*j: M:= matrix(3, 10, Fn): M1:= matrix(3,3,[]): map(evalm, [M, M1]);

$$\left[\begin{bmatrix} 4 & 7 & 10 & 13 & 16 & 19 & 22 & 25 & 28 & 31 \\ 10 & 16 & 22 & 28 & 34 & 40 & 46 & 52 & 58 & 64 \\ 18 & 27 & 36 & 45 & 54 & 63 & 72 & 81 & 90 & 99 \end{bmatrix}, \begin{bmatrix} M1_{1,1} & M1_{1,2} & M1_{1,3} \\ M1_{2,1} & M1_{2,2} & M1_{2,3} \\ M1_{3,1} & M1_{3,2} & M1_{3,3} \end{bmatrix}\right]$$

> indices(M);

[1, 3], [3, 2], [2, 1], [3, 6], [1, 6], [2, 5], [3, 10], [1, 2], [1, 4], [3,3], [1, 7], [1, 9], [3, 7], [2, 9], [1,5], [2,10], [2, 4], [1, 1], [3, 4], [2, 7], [1, 8], [3, 8], [2, 6], [2, 2], [1, 10], [2, 8], [3, 1], [3, 5], [2, 3], [3, 9]

> entries(M); ⇒ [10], [27], [10], [63], [19], [34], [99], [7], [13], [36], [22], [28], [72], [58], [16], [64], [28], [4], [45], [46], [25], [81], [40], [16], [31], [52], [18], [54], [22], [90]

> Pind(M), Pind(M1); ⇒ [3, 10], [3, 3]

> assign(Kr=matrix(3,3,[[a,e*ln(x),c],[d,e*ln(x),f],[e*ln(x),h,k]])), eval(Kr), Mem1(Kr,e*ln(x));

$$\left[\begin{matrix} a & e\ln(x) & c \\ d & e\ln(x) & f \\ e\ln(x) & h & k \end{matrix}\right], 3, [1, 2], [2, 2], [3, 1]$$

Во фрагменте иллюстрируется определение **M**-матрицы на основе ***matrix****-функции* с использованием **Fn**-функции генерации элементов матрицы. Представлено применение ***Pind***-процедуры для определения размерности матриц. Процедура ***Mem1*(M, h)** возвращает последовательность, первый элемент которой определяет число вхождений в **M**-матрицу элементов, заданных вторым **h**-аргументом функции, а последующие определяют списки-координаты искомых элементов. В нашей *Библиотеке* [103] представлен целый ряд других полезных процедур для работы с матрицами. Можно определять целый ряд других полезных процедур как на основе представленных, так и других функций ***Maple***-языка, что оставляется читателю в качестве весьма полезного практического упражнения. В реальной работе с пакетом это вполне возможные ситуации.

Как таблица, так и общая функция ***array*(**{<ИФ>}**,** <*Размерность*>}**,** <НЗ>**)** допускает **3** необязательных ключевых аргумента: ИФ **–** индексная функция, размерность, кодируемая в виде диапазонов (**..**) изменения значений индексов, и НЗ **–** список начальных значений для определения элементов массива. Два последних из них достаточно просты и неоднократно обсуждались при рассмотрении *массивов* и *таблиц*. Несколько детальнее остановимся на *индексной* функции, имеющей для матриц особое значение, определяя общего уровня их классификацию. В общем случае *индексная* функция определяет правило присваивания значений элементам массива или таблицы, или их обработки. При отсутствия ИФ-аргумента используется стандартный метод индексации элементов массива (*матрицы*). *Индексная* функция может определяться пользовательской процедурой, принцип организации которой здесь не рассматривается, но общий подход был указан выше. Для этих целей может использоваться, например, процедура ***indexfunc*** пакетного модуля **linalg.** Язык ***Maple*** располагает пятью основными *встроенными* ИФ с идентификаторами *symmetric*, *antisymmetric*, *sparse*, *diagonal* и *identity* для функций ***table*** и ***array***. Мы рассмотрим вкратце данные индексные функции.

Индексная ***symmetric***-функция применима в качестве первого фактического аргумента для определения симметричности элементов матрицы/таблицы относительно ее главной *диагонали*, т.е. для **M**-матрицы предполагается определяющее соотношение **M[k,j] = M[j, k].** В свою, очередь ***antisymetric***-аргумент определяет **M**-матрицу с соотношением **M[k, j] = -M[j, k],** следовательно **∀(k)∀(j)(k=j)(M[k, j]=0).** Если же при определении такого типа матрицы были заданы *ненулевые* начальные значения для ее диагональных элементов, то они получают нулевые значения с выводом соответствующей диагностики. Аргумент ***diagonal*** определяет диагональную **M**-матрицу, для которой справедливо со-

отношение **∀(k)∀(j)(k≠j)(M[k, j]=0)**. Посредством ***sparse***-аргумента определяется **M**-матрица, чьи неопределенные входы получают нулевые значения. Например, по вызову ***array*(*sparse*, 1..n, 1..p**) возвращается нулевая матрица (**nxp**)-размерности. Наконец, по ***identity***-аргументу возвращается единичная матрица, для которой имеет место соотношение **∀(k)∀(j)[(k=j) → (M[k, j] = 1)] & [(k≠j) → (M[k,j] = 0)]**. Следующий пример иллюстрирует получение рассмотренных выше пяти типов матриц:

> MS:=array(*symmetric*, 1..3, 1..3): MS[1, 1]:=10: MS[1, 2]:=17: MS[1, 3]:=39: MS[2, 2]:=64: MS[2, 3]:=59: MS[3, 3]:=99: MD:=array(*diagonal*, 1..3, 1..3): MD[1, 1]:=2: MD[2, 2]:=10: MD[3, 3]:=32: MAS:= array(*antisymmetric*, 1..3, 1..3): MAS[1, 2]:=10: MAS[1, 3]:=32: MAS[2, 3]:=52: MI:= array(*identity*, 1..3, 1..3): MSp:=array(*sparse*, 1..3, 1..3): map(evalm, [MS, MAS, MD, MI, MSp]);

$$\left[\begin{bmatrix}10 & 17 & 39\\ 17 & 64 & 59\\ 39 & 59 & 99\end{bmatrix}, \begin{bmatrix}0 & 10 & 32\\ -10 & 0 & 52\\ -32 & -52 & 0\end{bmatrix}, \begin{bmatrix}2 & 0 & 0\\ 0 & 10 & 0\\ 0 & 0 & 32\end{bmatrix}, \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\right]$$

Так как *вектор* представляет собой (**1xn**)-*матрицу*, то к нему применим и целый ряд сугубо матричных функций, например ***evalm***-функция, позволяющая возвращать вычисленные векторные выражения в векторной (*списочной*) нотации. Более того, все имеющее силу относительно матриц в полной мере (*с поправкой на размерность*) относится и к векторам. В частности, это относится и к (**&***)-оператору матрично/векторного произведения. *Длину* **V**-вектора, подобно случаю матриц, можно вычислять, в частности, посредством конструкции вида ***nops*([{*indices* | *entries*}(V)]**); при этом, нулевое значение возвращается, если элементам **V**-вектора не присваивалось значений. Следующий простой фрагмент иллюстрирует некоторые способы определения, тестирования и вычисления простых векторных выражений в среде ***Maple***-языка:

> V1:= array(1..9): V2:= array(1..6, [64, 59, 39, 10, 17, 44]): V3:= vector(5, [42, 47, 67, 89, 96]): V4:= vector(5, [64, 59, 39, 17, 10]): Fn:= k -> 3*k^2 + 10*k + 99: V5:= vector(5, Fn): V6:= vector(6, [V, G, S, Art, Kr, Ar]): map(type, [V1, V2, V3, V4, V5, V6], 'vector');

[*true*, *true*, *true*, *true*, *true*, *true*]

> map(evalm, {V5, V6}); ⇒ {[112, 131, 156, 187, 224], [*V, G, S, Art, Kr, Ar*]}

> map(nops, [[indices(V6)], [entries(V6)]]); ⇒ [6, 6]

> V:=vector(6, []): map(nops, [[indices(V)], [entries(V)]]); ⇒ [0, 0]

> [type(V4, 'vector'(integer)), type(V6, 'vector'(symbol))]; ⇒ [*true*, *true*]

> Z:= vector([W, G, S]): M:= array(1..3, 1..3, [[1, 2, 3], [4, 5, 6], [7, 8, 10]]): evalm(M&*Z);

[*W* + 2 *G* + 3 *S*, 4 *W* + 5 *G* + 6 *S*, 7 *W* + 8 *G* + 10 *S*]

Для обеспечения работы с *матричными* и *векторными* выражениями ***Maple***-язык располагает довольно развитым набором функциональных средств, поддерживаемых, в первую очередь, пакетными модулями ***linalg*** и ***LinearAlgebra*.** Данные модули содержат определения **114** и **118** процедур соответственно (*в зависимости от релиза пакета; пример приведен для* ***Maple 10***), обеспечивающих расширенные средства линейной алгебры по работе с *матричными* и *векторными* выражениями. Учитывая обилие указанных средств и направленность данной книги, мы не будем *акцентировать* на них внимания, отсылая к нашей книге [12] либо к ее авторскому оригинал-макету, который можно загрузить с университетского *Web*-сайта **www.grsu.by/cgi-bin/lib/lib.cgi?menu=links&path=sites.** В них рассмотрены средства пакета, составляющие определенную базу для обеспечения работы с матрично/векторными выражениями в рамках классической линейной алгебры. Лишь кратко определим способы доступа к таким модульным средствам пакета.

Ряд средств *линейной алгебры* имеют классический формат вызова вида ***Id*(**<*аргументы*>**),** тогда как *основная* масса таких средств определяется пакетным модулем **linalg.** Поэтому

первый вызов любого из них должен предваряться предложением одного из следующих трех форматов, а именно: **with(linalg), with(linalg,** <*Процедура_1*>**,** <*Процедура_2*>**,** <*Процедура_3*>**, ...)** или **linalg[**<*Процедура*>**](**<*аргументы*>**).** Исходя из обстоятельств, пользователь будет производить *вызов* процедур **linalg**-модуля необходимым ему способом. *Первый* формат обеспечивает доступ сразу ко всем процедурам модуля, но их загрузка требует дополнительной памяти в рабочей области пакета, *второй* формат активирует определения конкретных процедур, используемых в текущем сеансе, и *третий* формат определяет *разовый* вызов *конкретной* процедуры на конкретных фактических аргументах. Последний формат часто используется в пользовательских процедурах. Вызов процедуры ***packages*()** возвращает список пакетных модулей, загруженных в *текущий* сеанс. Вышесказанное иллюстрирует следующий простой пример:

> restart; packages(), map(type, [col, det], 'procedure'); ⇒ [], [*false*, *false*]
> with(linalg, col, det), packages(), map(type, [col, det], 'procedure');
[*col*, *det*], [*linalg*], [*true*, *true*]

Из приведенного примера следует, что загрузка даже *отдельных* процедур модуля идентифицируется процедурой ***packages*** как загрузка модуля в целом.

С учетом сказанного, средства модуля **linalg** не представляют каких-либо затруднений при использовании их знакомым с основами линейной алгебры читателем, а приведенные здесь и в прилож. **1** [12] примеры и дополнительные замечания вполне достаточно иллюстрируют средства матричной алгебры, поддерживаемые ***Maple***-языком. Наряду с наиболее общими **linalg**-модуль располагает целым рядом других, более специальных, матричных средств, включая средства создания *специального* типа матриц (*Вандермонда*, *Теплица*, *Сильвестра и др.*), в полном же объеме со средствами матричной алгебры, обеспечиваемыми ***Maple***-языком, рекомендуется ознакомиться по книгам [10-14,59,80,86,90].

• ***Базовые структуры данных модуля*** **LinearAlgebra.** Одну из самых больших особенностей ***Maple*** составляет его модуль **LinearAlgebra,** определяющий новые стандарты эффективности, надежности, полезных свойств и точности для вычислительной линейной алгебры. Данная задача была решена путем интегрирования в пакет современных программ линейной алгебры фирмы ***NAG*** (*Numerical Algorithm Group*) через *внешний* механизм вызовов [13,14,39]. Это позволяет использовать мощные вычислительные алгоритмы ***NAG*** для решения задач линейной алгебры с высокими точностью и производительностью. Однако, прежде, чем переходить к характеристике средств **LinearAlgebra**-модуля, кратко остановимся на *различиях* между ним и рассмотренным выше **linalg**-модулем линейной алгебры.

Если в основе векторно-матричных объектов, с которыми оперирует модуль **linalg,** лежит структура данных *array*-типа, то основу объектов, обрабатываемых средствами модуля **LinearAlgebra,** составляет так называемая *rtable*-структура данных, генерируемая одноименной встроенной функцией. Данная функция и генерируемые ею *rtable-объекты* детально были нами рассмотрены в [13,14,39]; ряд новых средств по работе с такого типа объектами как отдельно, так и в совокупности с *array*-объектами представлен нами в книгах [39,41,42,45,46,103], а также в нашей *Библиотеке* [103,109].

Визуализация *rtable*-объекта (*Array*, *Matrix*, *Vector*) определяется его размером. Если размер его меньше или равен значению, определенному предопределенной ***rtablesize***-переменной процедуры ***interface*** (*по умолчанию* ***rtablesize*=10**), то объект визуализируется *полностью*. В противном случае он заменяется специальным *шаблоном*. Установка опции ***rtablesize*=0** определяет вывод любого *rtable*-объекта в виде шаблона, тогда как установка ***rtablesize=infinity*** определяет вывод *rtable*-объекта полностью безотносительно его размера, как иллюстрирует следующий фрагмент:

> interface(rtablesize=2); Kr:= rtable(1..3, [89, 11, 99]); restart: Kr:=rtable(1..3, [89, 11, 99]);

$$Kr := \left[\begin{array}{l} \text{1..3 1-D Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{array}\right]$$

$$Kr := [89, 11, 99]$$

> interface(rtablesize = 6); Kr:= rtable(1..3, 1..3, [[42, 47, 67], [64, 59, 39], [44, 10, 17]]);

$$Kr := \begin{bmatrix} 42 & 47 & 67 \\ 64 & 59 & 39 \\ 44 & 10 & 17 \end{bmatrix}$$

Из фрагмента нетрудно заметить, что шаблон *rtable*-объекта представляет собой стандартный описатель объекта (*размерность*, *тип и основные характеристики*). В случае больших размеров массивов данное средство оказывается весьма удобным. При этом, следует иметь в виду следующее важное обстоятельство. Если размер *rtable*-объекта больше определяемого ***rtablesize***-переменной, то выводится его шаблон даже в случае наличия *рекурсивности* в его определении, в противном случае сразу же идентифицируется аварийная ситуация, требующая *перезагрузки* пакета. Причиной этого является переполнение системного стэка. Впрочем, работа пакетного стэка и так имеет много нареканий.

Для работы с *rtable*-объектами пакет располагает рядом полезных функций и процедур, детально рассмотренных в [45,46]. Данные средства применимы к любому *rtable*-объекту (*Array*, *Matrix*, *Vector)*, однако каждый из *трех* типов объектов располагает и собственными *аналогичными* средствами, кратко рассматриваемыми ниже. Для вывода *rtable*-объектов можно использовать и форматирующие функции ***printf***-группы: ***printf***, ***fprintf***, ***sprintf***, ***nprintf***, с опциями которых для этого случая можно ознакомиться по справке пакета. Аналогично этому к любому *rtable*-объекту применимы и функции ***scanf***-группы (***scanf***, ***fscanf***, ***sscanf***) для выполнения синтаксического анализа объектов.

• ***Объекты rtable-типа.*** На основе упомянутой ***rtable***-функции определяются три базовых объекта **LinearAlgebra**-модуля: ***Array***, ***Matrix*** и ***Vector.*** Непосредственно посредством ***rtable***-функции в *среде* пакета можно определять *любой* из указанных трех объектов. После их создания с ними можно работать в рамках алгебры, определяемой операциями, довольно детально рассмотренными в книгах [39,41,42,45,46,103]. Приведем простой фрагмент, иллюстрирующий использование операций *rtable*-алгебры пакета.

> A:=rtable(1..4, 1..4, random(4..11, 0.95), subtype=Array, storage=sparse, datatype=integer):
M:= rtable(1..4, 1..4, random(42..99, 0.58), subtype=Matrix, storage=rectangular):
C:=rtable(1..4, 1..4): V:= rtable(1..4, random(42..99, 0.64), subtype=Vector[column], storage=rectangular): A, M, V, C;

$$\begin{bmatrix} 10 & 6 & 7 & 8 \\ 8 & 10 & 5 & 5 \\ 6 & 5 & 6 & 10 \\ 9 & 5 & 10 & 6 \end{bmatrix}, \begin{bmatrix} 69 & 73 & 54 & 55 \\ 46 & 0 & 68 & 49 \\ 95 & 95 & 85 & 76 \\ 0 & 0 & 53 & 0 \end{bmatrix}, \begin{bmatrix} 45 \\ 84 \\ 47 \\ 72 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

> Vr:= rtable(1..4, random(47..99, 0.99), subtype = Vector[row], storage = sparse, datatype = integer); $\Rightarrow$ $Vr := [65, 56, 57, 90]$

> Vr.M^(-2) + 6*Vr;

$$\left[\frac{208520698}{897237}, \frac{787693955}{1794474}, \frac{13386043721}{34095006}, \frac{164952480446}{301172553}\right]$$

> A^2 + 10*A + 99, M^2 + 17*M + 95, (Vr.M + 17*Vr).(10*M + 17);

$$\begin{bmatrix} 299 & 195 & 218 & 243 \\ 243 & 299 & 174 & 174 \\ 195 & 174 & 195 & 299 \\ 270 & 174 & 299 & 195 \end{bmatrix}, \begin{bmatrix} 14517 & 11408 & 17113 & 12411 \\ 10416 & 9913 & 12017 & 8531 \\ 20615 & 16625 & 24383 & 17632 \\ 5035 & 5035 & 5406 & 4123 \end{bmatrix},$$

$$[31512880, 27251067, 36562329, 26471066]$$

С учетом сказанного, примеры фрагмента особых пояснений не требуют. В этой связи имеет смысл лишь вкратце пояснить *различие* между *табличной* организацией собственно ***Maple***-среды (*базируется на* ***table****-функции*) и ***NAG***-организацией (*базируется на функции* ***rtable***). В *первом* случае *массивы* и *таблицы* базируются на внутренних хэш-таблицах пакета, тогда как во *втором* случае используется формат импортированного ***NAG***-модуля линейной алгебры. В этом случае для каждого измерения *rtable*-объекта используется по одному *вектору индексов* и отдельный вектор отводится под значения элементов массива. На основе такого *представления* формируются такие структуры данных как *Array*, *Matrix*, *Vector*[*row*] и *Vector*[*column*]. Более того, следует иметь в виду, что одноименные рассмотренным структуры *array*, *matrix* и *vector*, начинающиеся со строчных букв (*исключение составляют пассивные функции*), относятся к средствам собственно ***Maple-***языка и их обработка производится иными средствами, детально рассмотренными в цитируемых выше книгах. При этом, следует отметить, что функции на основе ***rtable***-функции принципиально отличаются от одноименных функций *array*, *matrix* и *vector*, как отмечалось выше.

• ***Базовые объекты модуля*** **LinearAlgebra.** В качестве таких объектов выступают *Array*, *Matrix*, *Vector*[*row*] и *Vector*[*column*], кратко рассмотренные выше в связи с ***rtable-***функцией, на основе которой они формируются. Между тем, пользователь имеет возможность непосредственно создавать указанные объекты на основе встроенных функции ***Array*** и процедур ***Matrix*** или ***Vector,*** кратко рассматриваемых ниже.

По функции ***Array*****(ИФ, D, НЗ,** <*Опции*>**)** создается *массив* с заданными характеристиками, определяемыми ее фактическими аргументами. Каждый из аргументов функции не является обязательным; если же вызов функции ***Array*** определен без фактических аргументов, то возвращается *пустой* 0-*мерный* массив. Смысл и форматы кодирования аргументов функции (**ИФ** - *индексирующая функция*, **D** - *размерность*, **НЗ** - *начальные условия и* <*Опции*>) довольно прозрачны и особых пояснений не требуют. Поэтому на данном вопросе детальнее останавливаться не будем, а приведем несколько примеров на прямое определение массивов *Array*-типа.

```
> A:= Array(1..6, [42, 47, 67, 62, 89, 96], datatype=integer, readonly):  Ao:=Array():  A, Ao;
```

[42, 47, 67, 62, 89, 96], Array({}, *datatype = anything*, *storage = rectangular*, *order = Fortran_order*)

```
> A[4]:= 2006;
```

Error, cannot assign to a read-only Array

```
> ArrayOptions(A);
```

datatype = integer, *storage = rectangular*, *order = Fortran_order*, *readonly*

```
> B:=Array(symmetric, 1..3, 1..3, datatype=integer):  B[1, 1]:=42:  B[2, 2]:=47:  B[3, 3]:=67:
  B[1, 2]:= 64:  B[1, 3]:=59: B[2, 3]:=39:   B, ArrayIndFns(B), ArrayOptions(B);
```

$$\begin{bmatrix} 42 & 64 & 59 \\ 64 & 47 & 39 \\ 59 & 39 & 67 \end{bmatrix}, symmetric, datatype = integer, storage = triangular_{upper},$$

$$order = Fortran_order$$

```
> op(1, B), ArrayOptions(B, order=C_order), ArrayOptions(B);
```

$$symmetric, datatype = integer, storage = triangular_{upper}, order = C_order$$

```
> ArrayDims(B), ArrayNumDims(B);   ⇒ 1 .. 3, 1 .. 3, 2
> ArrayElems(B);
                {(1, 1) = 42, (1, 2) = 64, (1, 3) = 47, (2, 2) = 59, (2, 3) = 39, (3, 3) = 67}
> ArrayNumElems(B, NonZeroStored), ArrayNumElems(B, All);   ⇒ 6, 9
```

Относительно сугубо ***Maple***-объектов ***NAG***-объекты характеризуются двумя важными чертами, а именно: (**1**) ссылка на их идентификатор возвращает непосредственно сам объект, не требуя таких функций как ***evalm,*** и (**2**) объекты создаются *всегда* с определенными элементами, по меньшей мере нулевыми, если не было определено противного. Второе обстоятельство весьма упрощает ряд процедур с такими объектами, не требуя их предварительного обнуления. Обусловлено это тем, что если ***Maple***-объекты изначально ориентированы на символьные вычисления, то ***NAG***-объекты на числовые.

Для работы с *Array*-объектами ***Maple***-язык располагает рядом полезных функций, детально рассмотренных в наших книгах [13,14,29,30,33]. Примеры предыдущего фрагмента иллюстрируют создание посредством ***Array***-функции одномерного **A**-массива и двумерного **B**-массива, а также применение к ним ряда функций работы с *массивами*, которые ввиду их простоты не требуют дополнительных пояснений.

Вызов процедуры ***Matrix*****(n, m, H3,** <*Опции*>**)** возвращает *матричную* структуру данных (*матрицу*), являющуюся одной из базовых структур, с которыми работают функциональные средства **LinearAlgebra**-модуля. Первые два формальных аргумента процедуры определяют число строк и столбцов матрицы соответственно; при этом, фактические значения для аргументов могут кодироваться или целыми неотрицательными числами, или диапазонами вида **1 .. h** (**h –** *целое неотрицательное*). *Третий* аргумент определяет начальные значения, тогда как *опции* **-** *дополнительные* характеристики создаваемого матричного объекта. Все аргументы процедуры *необязательны* и при их отсутствии возвращается матрица (**0x0**)-размерности. В общем же случае при создании матричного объекта вызов ***Matrix***-процедуры должен содержать достаточное количество информации о его структуре и элементах. Сказанное по начальным значениям относительно ***rtable***-функции в полной мере переносится и на третий аргумент ***Matrix***-процедуры. Среди допустимых опций (*четвертый аргумент*) многие аналогичны общему случаю ***rtable****-функции*, рассмотренной выше. Однако, среди них имеются и специфические для матричной структуры. Детальнее с описанием опций функции можно ознакомиться по справке пакета. Приведем примеры на создание некоторых простых матричных объектов и их тестирование.

```
> M1:=Matrix(3,[[42,47,67],[64,59,39],[89,96,62]],readonly): M2:=Matrix([[a,b],[c,d]]): M1,M2;
```

$$\begin{bmatrix} 42 & 47 & 67 \\ 64 & 59 & 39 \\ 89 & 96 & 62 \end{bmatrix}, \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```
> M1[3, 3]:= 47;
Error, cannot assign to a read-only Matrix
> M3:= Matrix(3, (j, k) -> 4*j^2+11*k^2-89*j-96*k+99*j*k, datatype=integer[2]):  M4:=
  Matrix(1..3,1..3,rand(42..99,0.58)): M5:= <<42,47,67> | <64,59,39> | <10,17,44>>: M3, M4, M5;
```

$$\begin{bmatrix} -71 & -35 & 23 \\ -49 & 86 & 243 \\ -19 & 215 & 471 \end{bmatrix}, \begin{bmatrix} 61 & 58 & 53 \\ 67 & 44 & 86 \\ 71 & 98 & 99 \end{bmatrix}, \begin{bmatrix} 42 & 64 & 10 \\ 47 & 59 & 17 \\ 67 & 39 & 44 \end{bmatrix}$$

```
> map(type, [M1, M2, M2, M5], 'Matrix');   ⇒ [true, true, true, true]
> M6:= Matrix(2, {(1, 1)=64, (1, 2)=59, (2, 1)=10, (2, 2)=17}): type(M5,'matrix'), whattype(M5),
  type(M6, 'Array'), type(M6, 'Matrix');   ⇒ false, Matrix, false, true
```

Фрагмент представляет различные способы определения *Matrix*-объектов, принцип которых легко усматривается из самих примеров. Из фрагмента также следует, что в компактном виде *Matrix*-объект можно определять конструкцией следующего вида:

M := <M11, M21, M31>|<M12, M22, M32>|<M13, M23, M33>

недостатком которой является невозможность указания для матрицы в точке определения других ее характеристик. Более того, последний пример фрагмента иллюстрирует тот факт, что ***type***-функция не распознает *Matrix*-объект в качестве ***Maple***-матрицы, тогда как тестирующая ***whattype***-процедура определяет его как *Matrix*-объект. Детальнее с описанием и применением функции ***Matrix*** можно ознакомиться в книгах [13,14,42] и в справке по пакету. С учетом сказанного *Matrix*-объекты (***NAG****-матрицы*) довольно прозрачны, но за более подробной информацией по их определению можно обращаться к справке по пакету (*например*, *оперативно по конструкции* **?Matrix**).

Наконец, по функции ***Vector*[T](n,H3,** <*Опции*>**)** возвращается *векторная* структура данных (*вектор*), являющаяся одной из основных структур, с которыми работают функциональные средства **LinearAlgebra**-модуля пакета. Индекс **T** определяет сам тип вектора (***column*** **–** *столбец*, ***row*** **–** *строка*, *по умолчанию полагается* ***column***). Первый формальный аргумент функции определяет число элементов вектора; при этом, фактические значения для аргумента могут кодироваться или целым неотрицательным числом, либо диапазоном вида **1..h**. Второй аргумент определяет начальные значения, тогда как *опции* **–** *дополнительные* характеристики создаваемого векторного объекта. Все аргументы функции необязательны и при их отсутствии возвращается вектор **0**-размерности, т.е. элемент **0**-мерного векторного пространства. В общем же случае при создании векторного объекта вызов ***Vector****-функции* должен содержать достаточное количество информации о его структуре и элементах. Сказанное выше по начальным значениям относительно ***rtable***-функции в полной мере переносится и на *второй* аргумент ***Vector***-функции. Среди допустимых опций (*третий аргумент*) многие *аналогичны общему* случаю ***rtable***-функции, рассмотренной выше. Однако допускаемые ими значения имеют векторную специфику. Детальнее с описанием опций ***Vector***-функции можно познакомиться в справке по пакету. Приведем простые примеры на создание векторных объектов.

> V1:= Vector(1..4, [42, 47, 67, 89]): V2:= Vector[row](4, {1=64, 2=59, 3=39, 4=17}):
V3:= Vector(1..4): V1, V2, V3;

$$\begin{bmatrix} 42 \\ 47 \\ 67 \\ 89 \end{bmatrix}, [64, 59, 39, 17], \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

> VectorOptions(V2);

shape = [], *datatype* = *anything*, *orientation* = *row*, *storage* = *rectangular*, *order* = *Fortran_order*

> V4:= <Sv, Ar, Art, Kr>: V5:= <Sv|Ar|Art|Kr>: V4, V5, VectorOptions(V4, readonly);

$$\begin{bmatrix} Sv \\ Ar \\ Art \\ Kr \end{bmatrix}, [Sv, Ar, Art, Kr], false$$

> V6:= Vector[row](6, rand(42..99)): V7:=Vector(): V6, type(V6, vector), whattype(V6), V7;

[57, 89, 80, 52, 48, 71], *false*, *Vector*[*row*], []

Фрагмент представляет различные способы определения *Vector*-объектов, принцип которых усматривается из самих примеров. Из фрагмента также видно, что в компактной форме объект *Vector*-типа можно определять конструкциями следующего вида:

V := <V1, V2, V3, …, Vn> или **V := <V1|V2|V3| … |Vn>**

недостатком которых является невозможность задания для вектора в точке его определения других характеристик. При этом, *первый* формат определяет вектор-*столбец*, тогда как второй – вектор-*строку*. Более того, последний пример фрагмента иллюстрирует тот факт, что ***type***-функция не распознает *Vector*-объект в качестве ***Maple***-вектора, тогда как тестирующая ***whattype***-*процедура* определяет его как *Vector*-объект. С учетом сказанного *Vector*-типа объекты (***NAG***-*векторы*) довольно прозрачны и за более детальной информацией по их определению можно обращаться к справке (*например*, *оперативно по конструкции* **?Vector**).

Как уже отмечалось выше, между ***Maple***-объектами (*array*, *vector* и *matrix*) и ***NAG***-объектами (*Array*, *Vector* и *Matrix*) имеются принципиальные различия. Более того, классификация вторых относительно тестирующих функции ***type*** и процедуры ***whattype*** выгодно отличается однозначностью, тогда как первые распознаются ***whattype***-процедурой как объекты *array*-типа. В приведенном ниже фрагменте этот момент иллюстрируется весьма наглядно:

> a:=array(1..2, 1..2, [[42,47], [64,59]]): A:=Array(1..3, 1..3): v:=vector([1,2,3]):
V:=Vector([1,2,3]): m:=matrix([[G, S, Vic], [47, 67, 42]]): M:=Matrix([[G, S, Vic], [47, 67, 42]]):
> type(a, 'array'), type(a, 'matrix'), type(v, 'vector'), type(v, 'array'), type(m, 'matrix'),
type(m,'array'); ⇒ *true*, *true*, *true*, *true*, *true*, *true*
> type(A, 'Array'), type(A, 'Matrix'), type(V, 'Vector'), type(V, 'Array'), type(M, 'Matrix'),
type(M, 'Array'); ⇒ *true*, *false*, *true*, *false*, *true*, *false*
> map(whattype, map(eval, [a, v, m])), map(whattype, [A, V, M]);
[*array*, *array*, *array*], [*Array*, *Vector*[*column*], *Matrix*]
> convert(a, 'listlist'), convert(A, 'listlist'); ⇒ [[42, 47], [64, 59]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
> a[1]:= [x, y, z];
Error, array defined with 2 indices, used with 1 indices
> A[1]:= [x, y, z]; ⇒ A[1] := [x, y, z]
> convert(A, 'listlist'); ⇒ [[[x, y, z], [x, y, z], [x, y, z]], [0, 0, 0], [0, 0, 0]]
> Ar:=Array(1..2, 1..2, 1..4, [[[64, 59, 39, 10], [47, 67, 42, 6]], [[c2, b2, a2, 17], [c3, b3, a3, 6]]]):
> convert(Ar, 'listlist'); ⇒ [[[64, 59, 39, 10], [47, 67, 42, 6]], [[c2, b2, a2, 17], [c3, b3, a3, 6]]]
> Ar[1, 2]:= AVZ; ⇒ Ar[1, 2] := *AVZ*
> convert(Ar, 'listlist'); ⇒ [[[64,59,39,10], [*AVZ*,*AVZ*,*AVZ*,*AVZ*]], [[*c2*,*b2*,*a2*,17], [*c3*,*b3*,*a3*,6]]]
> A1:= Array(1..3, 1..3, [[a, b, c], [x, y, z], [42, 47, 6]]); ⇒ $A1 := \begin{bmatrix} a & b & \tilde{n} \\ x & y & z \\ 42 & 47 & 6 \end{bmatrix}$
> A1[2]:= [Grodno, Tallinn]: A1;

$$\begin{bmatrix} a & b & \tilde{n} \\ [Grodno, Tallinn] & [Grodno, Tallinn] & [Grodno, Tallinn] \\ 42 & 47 & 6 \end{bmatrix}$$

В целом ряде случаев точная идентификация *rtable*-объектов тестирующими средствами пакета играет весьма существенную роль. Еще на одном моменте данного фрагмента имеет смысл обратить *внимание*, а именно. Если **2**-мерный ***Maple***-массив (*array*) не допускает возможности замены своих строк путем присвоения, инициируя ошибочную ситуацию, то массив ***NAG*** (*Array*) такую операцию допускает, *однако* присваиваемое значение дублируется по числу элементов строки. Соответствующим образом это обобщается и на **n**-мерные массивы, как показано выше. В ряде случаев это может представить практический интерес при программировании ряда приложений в среде пакета.

```
rtabobj := proc()
local a, b, c, k, j, t, A, M, V, x, y, z;
    assign( a = map( convert, { anames('rtable') }, 'string'), b = { }, c = { }, x = [ ],
        y = [ ], z = [ ]);
    seq(`if`( search(k, "RTABLE_SAVE/", 't') and t = 1, assign('c' = {k, op(c)} ),
        assign('b' = {k, op(b)} )), k = a);
    for k in b do
        if type( ``||k, 'Array') then
            A := [k];
            for j in c do
                if convert( ``||k, 'listlist') = convert( ``||j, 'listlist') then
                    A := [op(A), j]
                end if
            end do;
            x := [op(x), A]
        elif type( ``||k, 'Matrix') then
            M := [k];
            for j in c do
                if convert( ``||k, 'listlist') = convert( ``||j, 'listlist') then
                    M := [op(M), j]
                end if
            end do;
            y := [op(y), M]
        else
            V := [k];
            for j in c do
                if convert( ``||k, 'listlist') = convert( ``||j, 'listlist') then
                    V := [op(V), j]
                end if
            end do;
            z := [op(z), V]
        end if
    end do;
    x, y, z
end proc
> rtabobj();
        [["A"], ["A1", "RTABLE_SAVE/12988492", "RTABLE_SAVE/12468956"]],
         [["M", "RTABLE_SAVE/15041652", "RTABLE_SAVE/15049896"], ["M1",
                "RTABLE_SAVE/15041652", "RTABLE_SAVE/15049896"]],
          [["V", "RTABLE_SAVE/3384632"], ["V1", "RTABLE_SAVE/14862784",
                "RTABLE_SAVE/15017212", "RTABLE_SAVE/15026360"]]
```

Вместе с тем, ***Maple***-объекты *существенно проще* ***NAG***-объектов и несложно конвертируются во *вторые*. В наших книгах [29,33,42,43,103] и приложенной к ним *Библиотеке* пред-

ставлены дополнительные средства конвертации ***Maple***-объектов в ***NAG***-объекты, и наоборот. Наряду с ними, представлен ряд других средств по работе с *rtable*-объектами, которые существенно расширяют стандартные средства пакета. Эти средства оказываются достаточно полезными при продвинутом программировании разнообразных задач, имеющих дело с *rtable*-объектами, обеспечивая ***Maple***-программиста целым рядом дополнительных возможностей. Так, например, вызов представленной выше процедуры ***rtabobj*()** возвращает трехэлементную последовательность вложенных (*в общем случае*) списков, где элементы данной последовательности представляют информацию о *rtable*-объектах в разрезах типов *Array*, *Matrix* и *Vector* соответственно, активных в текущем сеансе. При этом, каждый подсписок *первым* элементом в строчном формате определяет имя *rtable*-объекта соответствующего типа, тогда как остальные его элементы определяют в строчном формате соответствующие ему идентификационные номера вида «RTABLE_SAVE/nnnnnnnn». Более детальный анализ данной информации позволяет, например, прослеживать историю работы с *rtable*-объектами текущего сеанса [42,103].

Процедура ***MwsRtb*(F)** анализирует ***mws***-файл **F** на наличие в нем ***rtable***-объектов [103].

```
MwsRtb := proc(F::{string, symbol, name})
local a, b, c, d, k, p, h, t;
    if not type(F,'file') or Ftype(F) ≠ ".mws" then
        ERROR("%1 is not a mws-datafile or does not exist",F)
    else
        assign(a = RS([ `\n` = ``], readbytes(F, 'TEXT', ∞))), close(F);
        if not search(a, "{RTABLE_HANDLES", 't') then RETURN(
            WARNING("rtable-objects do not exist in datafile <%1>",F))
        end if;
        assign('a' = a[t + 1 .. -1], c = "{RTABLE "), search(a, "}{", 't'),
            assign(b = map(SD, SLD(a[15 .. t − 2], " ")), 'a' = a[t + 1 .. -1]),
            `if`(1 < nargs and type(args[2], 'symbol'), assign(args[2] = b), NULL)
    end if;
    assign(p = SLD(a, c), d = map2(cat, "RTABLE_SAVE/", b),
        h = MkDir(cat([libname][1][1 .. 2], "\rtbobj")));
    for k to nops(p) do
        assign('a' = cat(h, "\_", b[k], ".m")), writeline(a, cat(p[k][1 .. 4], "
            ", p[k][5 .. -2])), close(a);
        read a
    end do;
    d, WARNING("rtable-objects of datafile <%1> are active in the current sessio

        and have been saved in directory <%2>; their names are %3",F, h,
        map2(cat, "_", b, ".m"))
end proc
> MwsRtb("C:/Academy\\Examples\\MPL9MWS\\IdR.mws", 'h'); h;
Warning, rtable-objects of datafile <C:/Academy\Examples\MPL9MWS\IdR.mws> are
active in a current session and have been saved in directory <c:\program files\
maple 9\rtbobj>; their names are [_5290620.m, _5603484.m, _5188712.m]
    ["RTABLE_SAVE/5290620", "RTABLE_SAVE/5603484", "RTABLE_SAVE/5188712"]
                                [5290620, 5603484, 5188712]
```

При наличии таких объектов возвращаются их вызовы в *строчном* формате, иначе возвращается *NULL*-значение. Через необязательный второй аргумент возвращаются идентификационные номера (*rtable-индексы*) ***rtable***-объектов, находящихся в файле **F.**

Весьма детальный обзор функциональных средств модуля **LinearAlgebra,** его базовые структуры данных, средства их создания и алгебра над ними детально рассмотрены в вышеупомянутых книгах [13,14,29,30,33]. Пакетный модуль ***LinearAlgebra*** предназначен как для интерактиного режима использования, так и для эффективного программирования в ***Maple.*** Для этого пакет прикладных программ линейной алгебры фирмы ***NAG*** был имплантирован в среду пакета ***Maple*** как программный модуль. Данная организация позволяет обращаться к его средствам следующими двумя способами, а именно:

(**1**) как к стандартному модулю пакета по конструкции формата:

LinearAlgebra[*Функция***](***Аргументы***)**

(**2**) как к программному модулю по конструкции формата:

LinearAlgebra:- *Функция*(*Аргументы***)**

В зависимости от удобства и ситуации может быть выбран любой из двух способов. Как правило, это определяется как опытом пользователя, так и самим алгоритмом программируемой в среде пакета задачи.

Начиная с **6**-го релиза, пакет ***Maple*** предоставляет два альтернативных выбора при решении задач *линейной алгебры*, базирующихся соответственно на модулях **LinearAlgebra** и **linalg.** Средства первого модуля и его концепция были довольно детально рассмотрены в наших книгах [8-14], тогда как второго в книгах [29-33]. Отметим здесь только наиболее характерные отличительные черты модуля **LinearAlgebra.** Прежде всего, модуль **LinearAlgebra** представляет собой большой набор процедур линейной алгебры, покрывающих почти все функциональные возможности пакета **linalg.** Вместе с тем, он имеет значительно более четкие структуры данных, располагает дополнительными средствами для создания специальных типов матриц, и улучшенными возможностями для решения матричных задач. Его преимущества *особенно* наглядны при вычислениях с большими числовыми матрицами, элементами которых являются значения *float*-типа (*как данные пакетной float-арифметики*, *так и данные машинной float-арифметики*). В качестве иллюстрации представим ряд примеров использования средств модуля **LinearAlgebra** для решения некоторых массовых задач линейной алгебры:

```
> alias(LA = LinearAlgebra):  A:= <<64, 59, 39>|<42, 47, 67>|<38, 62, 95>>:  B:= <10, 17, 99>:
> LA[Transpose](LA[LinearSolve](A, B)), LA[Transpose](LA[LeastSquares](A, B));
```

$$\left[\frac{-3308}{1855}, \frac{6921}{1855}, \frac{-6}{7}\right], \left[\frac{-3308}{1855}, \frac{6921}{1855}, \frac{-6}{7}\right]$$

```
> LA:- Determinant(A);
Error, `LA` does not evaluate to a module
> LA[Determinant](A);   ⇒  -33390
> M:= rtable([[64, 59, 39], [42, 47, 67], [43, 10, 17]], subtype = Matrix, readonly = true);
```

$$M := \begin{bmatrix} 64 & 59 & 39 \\ 42 & 47 & 67 \\ 43 & 10 & 17 \end{bmatrix}$$

```
> LA[Determinant](M),  LA[Rank](M),  LA[Trace](M);   ⇒  73670, 3, 128
> evalf(LA[Eigenvalues](M, output = 'Vector[row]'));
        [131.7149459, -1.85747297 + 23.57676174 I,  -1.85747297 - 23.57676174 I]
```

```
> evalf(LA[MatrixInverse](M, method='LU'), 6);   ⇒
```

$$\begin{bmatrix} 0.00175105 & -0.00832089 & 0.0287770 \\ 0.0294150 & -0.00799511 & -0.0359712 \\ -0.0217320 & 0.0257500 & 0.00719424 \end{bmatrix}$$

```
> evalf(LA[QRDecomposition](evalf(M), output = 'NAG'), 6);
```

$$\begin{bmatrix} -87.8008999999999986 & -70.3864999999999981 & -68.8033999999999964 \\ 0.276677999999999980 & -28.9090999999999988 & -26.8831999999999988 \\ 0.283266000000000018 & -0.664429999999999965 & 29.0239000000000011 \end{bmatrix},$$

$$\begin{bmatrix} 1.72892000000000001 \\ 1.38748000000000004 \\ 0. \end{bmatrix}$$

```
> LA[FrobeniusForm](M);   LA[CharacteristicPolynomial](%, h);
```

$$\begin{bmatrix} 0 & 0 & 73670 \\ 1 & 0 & -70 \\ 0 & 1 & 128 \end{bmatrix}$$

$$h^3 - 128\,h^2 + 70\,h - 73670$$

```
> evalf(LA[SingularValues](evalf(M), output = 'list'), 6);   ⇒ [136.209, 30.5125, 17.7259]
> evalf(LA[Eigenvectors](evalf(M), output = 'vectors'), 8);
```

$$[0.720627899999999988 + 0.\,I,$$
$$0.278720599999999984 + 0.381360960000000027\,I,$$
$$0.278720599999999984 - 0.381360960000000027\,I]$$
$$[0.613184240000000047 + 0.\,I, -0.671878590000000053 + 0.\,I,$$
$$-0.671878590000000053 + 0.\,I]$$
$$[0.323574589999999995 + 0.\,I,$$
$$0.315224249999999984 - 0.475490770000000006\,I,$$
$$0.315224249999999984 + 0.475490770000000006\,I]$$

```
> LinSolve:=(P::package, A::{Matrix, matrix}, B::{Vector, vector}) -> op([assign('K'=with(P)),
  `if`(`if`(P = linalg, det, Det)(A)<>0, `if`(P = LinearAlgebra, LinearSolve, linsolve)(A, B),
  ERROR("Matrix is singular"))]):  LinSolve(LinearAlgebra, A, B);                    # (1)
```

$$\text{LinearSolve}\left(\begin{bmatrix} 64 & 42 & 38 \\ 59 & 47 & 62 \\ 39 & 67 & 95 \end{bmatrix}, \begin{bmatrix} 10 \\ 17 \\ 99 \end{bmatrix}\right)$$

```
> a:= matrix([[64, 59, 39], [42, 47, 67], [38, 62, 95]]):  b:= vector([10, 17, 99]):          # (2)
> alias(la = linalg): evalf(LinSolve(linalg, a, b));  ⇒ [-4.75624439, 5.94860737, -0.937646002]
```

Прежде всего, следует обратить внимание на одно важное обстоятельство. С целью устранения неудобства, связанного с многократным использованием довольно длинного имени модуля **LinearAlgebra**, ему был присвоен алиас "**LA**", однако этот довольно удобный подход выявил ряд его *особенностей*. Прежде всего, алиас *нельзя* использовать в конструкциях вида **LA:-** ***Функция*****,** как это хорошо иллюстрирует *третий* пример фрагмента. В этом случае следует использовать конструкцию **LA[***Функция***].** Общей рекомендацией является определение алиаса для имени модуля вне тела процедуры/функции, в таком случае он может использоваться наравне с *основным именем*. *Второй* пример фрагмента иллюстрирует решение системы линейных уравнений посредством 2-х процедур ***LinearSolve*** и ***LeastSquares*** модуля. Посредством ***Transpose***-процедуры результат решения системы линейных уравнений **A.X = B** по-лучаем в виде вектора-*строки*. Последующие примеры фрагмента иллюстрируют применение различных процедур пакетного модуля **LinearAlgebra**. Исключение составляют последние примеры **1** и **2,** иллюстрирующие принципиальное отличие модуля **linalg** от **LinearAlgebra** относительно их использования внутри определений функций/процедур. В примере (**2**) показано, что использование вызова **with(linalg)** внутри тела функции делает доступными процедуры модуля **linalg** как в области определения самой функции, так и вне ее. Тогда как согласно примера (**1**) аналогичный подход, но на основе модуля **LinearAlgebra** не работает, что в

определенной мере сужает выразительные возможности программирования с использованием его функциональных средств. Имеется ряд других различий пакетных модулей **linalg** и **LinearAlgebra,** обусловленных проблемами полной интеграции второго в среду пакета, однако мы не будем здесь на них останавливаться. Заинтересованный же читатель отсылается к нашим книгам [13-14,29-33,39,41-43,45,46,103].

С учетом сказанного рассмотренные средства *линейной* алгебры и их базовые структуры данных не представляют каких-либо затруднений при использовании их *знакомым* с основами *линейной* алгебры читателем. В свете сказанного следует иметь в виду, что нами были представлены наиболее употребительные форматы матрично-векторных средств ***Maple***-языка с определенными акцентами скорее на особенностях их реализации и выполнения, чем на их математической сущности, хорошо известной имеющим опыт в данном разделе математики. Поэтому за деталями их описания необходимо обращаться к справочной системе пакета либо к цитированной выше литературе. Однако, многие вопросы снимаются при практической апробации рассмотренных средств. В настоящее время имеется целый ряд внешних модулей пакета (*часть из них поставляется по выбору*), весьма существенно расширяющих рассмотренные базовые средства матрично-векторных операций ***Maple***-языка. Эти средства постоянно расширяются. На базе рассмотренных средств ***Maple***-языка пользователь имеет возможность программировать и собственные, недостающие для решения его матрично-векторных задач, средства. Пакет постоянно, расширяя свои приложения, между тем, не в состоянии в должной мере учесть все потребности, поэтому его *программная* среда и предоставляет пользователям возможность расширять его новыми средствами под ваши специфические нужды. Типичным примером такого подхода и является наша *Библиотека* программных средств [41,42,103,109]. Ниже вопросы решения задач *линейной* алгебры в среде ***Maple*** (*учитывая специфику настоящей книги*) не рассматриваются. Заинтересованный читатель отсылается к книгам [8-14,78,84,86,88,55,59-62], а также к [91] с адресом сайта, с которого можно бесплатно загрузить некоторые книги, и к [109] с нашей *Библиотекой*.

• ***Псевдослучайные числа.*** Еще с одним видом значений - *псевдослучайных* пользователь имеет возможность работать на основе встроенного генератора псевдослучайных чисел (**ГПСЧ**), активизируемого по ***rand***-процедуре, имеющей три формата кодирования. По вызову ***rand*()** возвращается псевдослучайное неотрицательное *integer*-число (**ПСЧ**) длиной в 12 цифр, тогда как по вызову ***rand***({**n**|**n..p**}) (**n, p –** *целочисленные значения и* **n ≤ p**) возвращается равномерно распределенное на интервале соответственно [**0 .. n**] и [**n .. p**] целое **ПСЧ**. Для установки на-чального значения для **ГПСЧ** используется *предопределенная* ***_seed***-переменная пакета, имеющая значение **427419669081**, которое в любой момент может быть переопределено пользователем, например: ***_seed*:= 2006**.

Для этих же целей, но с *более* широкими возможностями, используется и ***randomize***-процедура, кодируемая в одном из допустимых форматов вида: ***randomize***({|n}**),** где **n > 0 –** целочисленное выражение, значение которого и присваивается ***_seed***-переменной. Если используется формат ***randomize*()** вызова процедуры, то для ***_seed***-переменной устанавливается *значение*, базирующееся на текущем значении системного таймера. Так как вызов ***randomize*()** возвращает базовое значение для **ГПСЧ** на основе текущего значения таймера, то таким способом можно достаточно эффективно генерировать различные последовательности **ПСЧ**. Сохраняя необходимые значения ***_seed***-переменной, можно повторять процесс вычислений с *псевдослучайными* числами, что особенно важно в случае необходимости проведения повторных вычислений.

Вызов ***rand*** может инициировать вывод самого тела соответствующей ей процедуры, рекомендуемый подавлять по завершающему обращение к процедуре двоеточию. В слу-

чае же намерений пользователя написать собственную процедуру для подобных целей текст пакетной ***rand***-процедуры может оказаться определенным прообразом. В общем же случае для обеспечения доступа к **ГПСЧ** рекомендуется использовать конструкции вида ***Id*:= rand({|n|n .. p}),** выполнение которых открывает доступ к последовательностям **ПСЧ** по вызовам ***Id*(),** каждое использование которого инициирует получение очередного равномерно распределенного на заданном интервале **ПСЧ**. Следующий пример иллюстрирует применение рассмотренных средств ***Maple***-языка по работе с *псевдослучайными* числами:

```
> rand();  AVZ:= rand(1942 .. 2006):  _seed:= 2006:   ⇒ 403856185913
> seq(AVZ(), k=1 .. 10);   ⇒ 2003, 1960, 1944, 1990, 1986, 1972, 1986, 1992, 2006, 1964
> randomize(2006): _seed;   ⇒ 2006
> ПСЧ:= array[1 .. 10]: for k while k <= 10 do ПСЧ[k]:= AVZ() end do:
> seq(ПСЧ[k], k=1 .. 10);      ⇒ 2005, 1946, 1958, 1951, 1983, 1992, 1946, 1955, 1995, 1951
> restart; [_seed, rand(), _seed];   ⇒ [_seed, 427419669081, 427419669081]
```

Наряду с приведенными ***Maple*** располагает и другими подобными средствами, в частности, для решения статистических задач (*модуль* **stats**), работы со *стохастическими* объектами (*модуль* **RandomTools**), стохастической *генерации* матриц (**linalg[*randmatrix*]**), полиномов (*процедура* ***randpoly***) и др. Средства поддержки работы с *псевдослучайными* числами могут быть использованы во многих вычислительных задачах стохастического и квазистохастического характера, а также в задачах, требующих наборов данных для отладки программ и тестирования вычислительных алгоритмов. Рассмотренные средства широко используются нами в различного рода иллюстративных примерах для генерации разнообразных числовых данных.

Выше мы уже не раз употребляли такое понятие как математическое выражение (*либо просто* ***выражение***), являющееся одним из важнейших понятий ***Maple,*** да и математики в целом. Работа с математическими выражениями в символьном виде — основа основ символьной математики. Не меньшую роль они играют и в численных вычислениях. ***Выражение*** — *центральное* понятие всех математических систем. Оно определяет то, что должно быть вычислено в численном или символьном виде. Не прибегая к излишнему формализму, несколько поясним данное понятие. Математические выражения строятся на основе чисел, констант, переменных, операторов, вызовов функций/процедур и различных специальных знаков, например, скобок, изменяющих порядок вычислений. *Выражение* может быть представлено в общепринятом виде (*как математическая формула или ее часть*) с помощью операторов, например, **c*(x + y^2+sqrt(z))** либо **(a+b)/(c+d),** оно может определять просто вызов некоторой функции или процедуры **F(x,y,z),** либо их комбинацию. Используемые в дальнейшем многочисленные иллюстративные фрагменты представят достаточное число примеров на определение допустимых ***Maple***-выражений, что уже практически позволит уяснить данное ключевое понятие пакета.

Наряду с рассмотренными ***Maple***-язык поддерживает работу с рядом других *структур* данных (*стэк*, *очередь*, *функциональные ряды*, *связные графы*, *графические объекты и т.д.*). В этом направлении нами был создан ряд ролезных средств, расширяющих и дополняющих стандартные. В частности, для работы со структурами типа стэк (*stack*) и очередь (*queue*), а также введен новый тип структур прямого доступа (*dirax*) [41,42,103]. Пока же нам будет вполне достаточно приведенных сведений по типам данных и структур данных для *понимания* сути излагаемого материала, который ссылается на данные понятия. Переходим теперь к средствам ***Maple***-языка, тестирующим вышерассмотренные типы *данных*, *структур данных* и *выражений*.

1.6. *Средства тестирования типов данных, структур данных и выражений*

Согласно аксиоматике ***Maple*** под *типом* понимается **T**-*выражение*, распознаваемое ***type***-функцией и инициирующее возврат логического {*true*|**false**}-значения на некотором множестве допустимых ***Maple***-выражений. В общем случае **T**-*выражения* языка относятся к одной из четырех групп: (**1**) *системные*, определяемые идентификаторами языка {*float*, *integer*, *list*, *set и др.*}; (**2**) *процедурные*, когда тип входит в качестве аргумента в саму тестирующую функцию {***type(***<*Выражение*>**,** <*Тип*>**)**}; (**3**) *приписанные* и (**4**) *структурные*, представляющие собой ***Maple***-выражения, отличные от строковых и интерпретируемые как *тип* {***set(***<*Id*> **=** *float***)**}. К *пятой* группе можно отнести *типы*, определяемые модульными средствами пакета. Несколько детальнее о данной классификации типов будет идти речь ниже по мере рассмотрения все более сложных ***Maple***-объектов.

Уже неоднократно упоминавшееся понятие *выражения*, хорошо знакомое обладающему определенной компьютерной грамотностью читателю, является одним из фундаментальных понятий ***Maple***-языка. Понятие *выражения* ***Maple***-языка аккумулирует такие рассмотренные конструкции языка как константы, идентификаторы, переменные, данные и их структуры, а также рассматриваемые детально ниже *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 данном вопросе не будем акцентироваться, а отсылаем заинтересованного читателя, например, к таким книгам как [9-14, 29-33, 59-62, 78-89, 103].

Для определения рассмотренных типов данных и структур данных язык пакета располагает развитыми средствами, базирующимися на специальных *тестирующих* процедуре ***whattype*** и функциях ***typematch, type,*** имеющих следующие форматы кодирования:

{***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 логическое *true*-значение, если значение выражения ***Maple*** имеет *тип*, определяемый ее *вторым* аргументом, и *false*-значение в противном случае. При определения в качестве второго аргумента множества типов функция {***type***|***typematch***} возвращает логическое *true*-значение в том случае, если тип значения ***Maple***-выражения принадлежит данному множеству, и *false*-значение в противном случае. При этом, следует помнить, что в качестве *второго* аргумента 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*). Для *второго* аргумента {***type***|***typematch***}-функции допускается *более* **202** определяемых пакетом типов (***Maple 10***), из которых здесь рассмотрим только те, которые *наиболее* употребляемы на первых этапах программирования в ***Maple*** и 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 она рассматривается несколько ниже.

Наконец, по тестирующей процедуре ***whattype*(**<*Выражение*>**)** возвращается собственно *тип* ***Maple***-выражения, определяемого ее фактическим аргументом. При этом, следует отметить, что данная процедура в ряде случаев решает задачу тестирования более эффективно, например для *последовательностных* структур и в случае неизвестного типа, что позволяет избегать перебора подвергающихся проверке типов. Тут же уместно отметить, что средства тестирования *типов*, обеспечиваемые, в частности, { ***typematch*|*type***}-функцией существенно более развиты, чем подобные им средства ***Mathematica*** [28-30,32,42,43]. На основе данных средств предоставляется возможность создания достаточно эффективных средств программного анализа типов данных и их структур. В табл. **5** представлены некоторые допустимые ***Maple***-языком типы, тестируемые функцией {***type*|*typematch***} и используемые в качестве значений ее второго фактического аргумента, а также их назначение.

Таблица 5

Id типа	***тестируемое Maple-выражение; пояснения и примечания:***
algnum	*алгебраическое* число
array	*массив*; дополнительно позволяет проверять вид *массива*, тип его элементов и другие характеристики
Array	*массив rtable-типа*; дополнительно позволяет проверять вид *массива*, тип его элементов и другие характеристики
hfarray	*массив* **МАПТ-**типа; используется средствами **МАПТ**
anything	любое допустимое ***Maple-****выражение*, кроме *последовательности*
boolean	*логическая* константа {***true*, *false*, *FAIL***}
complex	*комплексная* константа; не содержит нечисловых констант {***true*, *false*, *FAIL*, *infinity***}, тестирует тип *действительной* и *комплексной* частей
complexcons	*комплексная* константа; **a+b*I,** где *evalf*(a) и *evalf*(b) **-** *float***-**числа
constant	*числовая* константа
{***odd*|*even***}	{*нечетное*|*четное*} *целое* выражение
float	*действительное* выражение
fraction	число вида **a/b;** где **a, b -** *целые* числа
indexed	*индексированное* выражение
infinity	значение *бесконечности*; ***+infinity, -infinity, complex infinity***
integer	*целочисленное* выражение
exprseq	*последовательность;* распознается только ***whattype-***процедурой
{***list*|*set***}	{*список*|*множество*}; позволяет проверять *тип* элементов
listlist	*вложенный список* (**BC**); элементы **BC** имеют то же число членов
literal	*литерал;* значение одного из типов *integer*, *fraction*, *float*, *string*
matrix	*матричный* объект, *массив*; дополнительно позволяет проверять вид *массива*, *тип* его элементов и др. характеристики
Matrix	*матричный* объект *rtable*-типа, *массив*; дополнительно позволяет проверять вид *массива*, *тип* его элементов и другие характеристики
{***positive*|*negative*|*nonneg***}	выражение {**> 0**|**< 0**|**≥ 0**}
{***posint*|*negint*|*nonnegint***}	*целое* выражение {**>0**|**<0**|**≥0**}
numeric	*числовое* выражение; числовое значение {*integer*|*float*| *fraction*}-типа
protected	*protected*-свойство; select(type, {unames(), anames(anything)}, 'protected ')

rational	*рациональное* выражение (*дробь*, *целое*)
range	*ранжированное* выражение**;** выражение вида **a .. b**
realcons	*действительная* константа**;** включает *float*-тип и ***±infinity***
string	*строчное* выражение
symbol	*символ***;** значение, являющееся не индексированным *именем*
table	*табличный* объект**;** корректно тестирует *таблицы*, *массивы*, *матрицы*
type	тестирует значение на допустимость в качестве *типа*
vector	*вектор*, **1-**мерный массив**;** позволяет проверять и *тип* элементов
Vector	*rtable-вектор*; позволяет проверять и *тип* элементов
procedure	*процедурный* объект
`module`	*модульный* объект

Смысл большинства типов достаточно прозрачен и особого пояснения не требует. Таблица **5** отражает далеко не полный перечень типов, распознаваемых пакетом. Данный перечень значительно шире и с каждым *новым* релизом пакета пополняется новыми типами. Например, для ***Maple 8*** этот перечень содержит **176** типов, ***Maple 9* – 182** и ***Maple 10* – 202**. При этом, пользователь также имеет возможность расширять пакет новыми типами и нами был определен целый ряд новых и важных типов, отсутствующих в ***Maple***. Все они описаны в нашей последней книге [103] и включены в прилагаемую к ней *Библиотеку*. Ниже мы представим механизм определения пользовательских типов. Следующий достаточно простой фрагмент иллюстрирует применения {***type, typematch***}-функций и ***whattype***-процедуры:

```
> [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(47.59), whattype(A and B), whattype(10/17), 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, 64, 42], [G, 47, 59]]:  K:= "Академия":  W:= array(1 .. 100):  W[57]:= 99:
> [type(A, 'algnum'), 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, 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, 10.17, `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, практически, все типы, представленные выше и тестируемые рассмотренными {***type, typematch***}-функциями и ***whattype***-процедурой, достаточно прозрачен и особых пояснений не требует. Ранее отмечалось, что ***whattype***-процедура позволяет тестировать *последовательностные* структуры, тогда как

{***type*|*typematch***}-функция этого сделать не в состоянии. Более того, в отличие от вторых, ***whattype***-процедура ориентирована, в первую очередь, на тестирование выражений, структурно более сложных, чем данные и структуры данных. При этом, следует иметь в виду, что и 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 уже упомянутая процедура ***whattype***, имеющая простой формат кодирования: ***whattype*(**<*Выражение*>**)** и возвращающая 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 SDMPolynom Vector[column] Vector[row]

Приведенный *перечень* идентифицируемых типов выражений, включая некоторые данные и их структуры, представлен для ***Maple 10***, тогда как для более низких релизов этот перечень несколько короче. Следует еще раз подчеркнуть, что хотя тип последовательности (*exprseq*) не является определяемым функциями ***type, typematch*** типом, однако он идентифицируется ***whattype***-процедурой. При этом, процедура возвращает только тип *высшего* уровня вложенности выражения в соответствии с приоритетным порядком составляющих его операторов. Следующий пример иллюстрирует применение тестирующей ***whattype***-процедуры:

```
> [whattype(64), whattype(x*y), whattype(x+y), whattype(x<=y), whattype(a<>b),
  whattype([]), whattype(a = b), whattype(a^b), whattype(Z), whattype(h(x)),
  whattype(a[17]), whattype({}), whattype([]), whattype(x,y), whattype(table()),
  whattype(x..y), whattype(5.9), whattype(A and B), whattype(proc()  end proc),
  whattype(module()  end module), whattype(10/17), whattype("SV"), whattype(a.b),
  whattype(hfarray(1..10))];
```

[*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 следует учитывать при использовании указанной процедуры тестирования. Обусловлено это тем обстоятельством, что предварительно вызов процедуры ***whattype*(A)** вычисляет и/или упрощает выражение **A.** 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 выражений:

```
> [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, 51, 42, 67, 62], 'indexed'), type(5*y^4
  + x^3 - 47*x, 'quartic(y)'), type(arctanh, 'mathfunc'), type(`Salcombe Eesti`, 'symbol'),
```

```
type(ln, 'mathfunc'), type(10/17, 'numeric'), type(A -> B, 'operator'), type({G = 51, V = 56},
'point')];
    [true, true, true, true, true, true, true, true, true, true, true, true, true, true, true, true]
```

Следует упомянуть еще об одной тестирующей функции ***hastype*(expr,t),** чей вызов возвращает *true*-значение, если выражение ***expr*** содержит *подвыражение* заданного **t**-типа, 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]
> map2(hastype, (10^Kr + 96)/(Art^17 + 89), [even^symbol, symbol^odd]);  ⇒ [true, true]
```

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

Дополнительно к рассмотренным средствам тестирования рассмотрим еще **6** весьма полезных процедур из класса так называемых ***is*-**процедур языка. 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, <*Свойство*>)** тестирует наличие у **V-**выражения указанного свойства, тогда как процедура ***ispoly*(P, {1 | 2 | 3 | 4}, x)** тестирует будет ли **P**-выражение полиномом {**1 | 2 | 3 | 4**}-степени по ведущей **x-**переменной. Процедура ***isdifferentiable*(V,x,n)** тестирует **V-**выражение, содержащее кусочно-определенные функции (***abs, signum, max*** *и др.*), на предмет принадлежности его к $\mathbf{C^n}$-классу по ведущей **x-**переменной. Наконец, процедура ***iscont*(W, x=a .. b{, 'closed'})** тестирует факт наличия *непрерывности* **W-**выражения на [**a, b**]-интервале по **x-**переменной**,** включая его граничные точки, если закодирован необязательный ***closed*-**аргумент. Простой фрагмент иллюстрирует использование указанных ***is*-**процедур**:**

```
> 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
```

Под *структурированным* типом в ***Maple*-**языке понимается выражение**,** отличное от имени (*не идентифицируемое отдельным словом*), но которое может быть интерпретировано как *тип*. В общем случае структурированный тип представляет собой алгебраическое выражение от известных языку типов, которое в основных чертах наследует структуру тестируемого выражения. На основе механизма *структурированных* типов предоставляется возможность тестировать как структуру выражения в терминах его подвыражений, так и их типовую принадлежность в терминах базовых типов языка. Особый смысл данная возможность приобретает в задачах символьных обработки и вычислений. Следующий фрагмент иллюстрирует вышесказанное:

```
> type([64, V, sqrt(Art)], [integer, name, `^`]), type(59^sin(x), integer^trig);  ⇒ true, true
> type(G(x)^cos(y), function^symmfunc), type(sqrt(F(x)), sqrt(function));  ⇒ true, true
> type(57+sin(x)*cos(y), `&+`(integer, `&*`(trig, symmfunc)));  ⇒ true
> type(sqrt(Art^10 + Kr^3 + 99),sqrt(`&+`(name^even, symbol^odd, integer)));  ⇒ true
> type((G(x) + cos(y))^(47*I + sin(x)),(`&+`(function, trig)^`&+`(complex, trig(x))));  ⇒ true
```

Из примеров представленного фрагмента *вполне* прозрачно прослеживается тесная взаимосвязь между структурным деревом термов тестируемого выражения и выбранным для него структурированным типом. Детальнее с вопросами структурированных типов языка можно ознакомиться по книгам [33,42,45,83] либо оперативно по конструкции вида **?*type*, {*anyfunc*| *identical*|*specfunc*|*structure*}** в среде текущего сеанса пакета.

Следует иметь в виду, что по конструкции следующего простого вида**:**

***type*('<*Id*>', *name*(<*Tun*>))**

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

```
> GS:= cos(x)*sin(y) + F(z) + 99:   AV:= 42 .. 64:   type('AV', name(range));   ⇒ true
> SV:= sqrt(Art + Kr):  type('SV', name(sqrt(`&+`(name, symbol))));   ⇒ true
> type('GS', name(`&+`(`&*`(symmfunc, trig), function, odd)));   ⇒ true
```

Как следует из приведенного фрагмента, описанный механизм позволяет производить тестирование типов значений определенных переменных в терминах как *базовых* типов языка, так и *структурированных* типов, существенно расширяя возможности языка. Для автоматического тестирования типов выражений, прежде всего, в процедурных конструкциях, используется (**::**)-*оператор типирования*, детально рассматриваемый несколько ниже в связи с процедурными объектами.

Особого внимания заслуживают еще две тестирующие функции, позволяющие проводить *структурное* тестирование типов как выражений, так и составляющих их подвыражений. В частности, ***hastype*-**функция имеет следующий формат кодирования**:**

hastype*(**<*Выражение*>**, <Структурированный тип***>)**

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

```
> Kr:= 10*sin(x) + 17*cos(y)/AV + sqrt(AG^2 + AS^2)*TRG + H^3 - 59*sin(z) - Catalan:
> map2(hastype,Kr, [name^integer,constant,sqrt,fraction,`*`]);  ⇒ [true, true, true, true, true]
> map2(has,Kr,[10*sin(x), 1/AV, AG^2+AS^2, H^3,-59*sin(z)]);  ⇒ [true, true, true, true, true]
> [hasfun(Kr, sin, z), map2(hasfun, Kr, [sqrt, sin, cos])];   ⇒ [true, [false, true, true]]
```

Из приведенного фрагмента нетрудно заметить, что в качестве второго аргумента функции ***hastype*** могут выступать не просто рассмотренные выше допустимые языком пакета типы, но и их *структурированные* конструкции, отвечающие *структурам* входящих в тестируемое выражение подвыражений. На основе данной функции можно не только тестировать выражения на *типы* составляющих их *подвыражений*, но также на их *структурную типизацию* в рамках общей структуры исходного выражения.

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

Наконец, по тестирующей функции ***hasfun*(V,<*Id-функции*> {,x})** возвращается значение ***true*,** если определенное *первым* фактическим аргументом **V-**выражение содержит вхождение *функции*, заданной своим идентификатором, и, возможно, от указанной третьим необязательным аргументом ведущей **x-**переменной. Последний пример предыдущего фрагмента не только иллюстрирует вышесказанное, но и указывает на то, что, в частности, ***sqrt*-**функция не тестируется ***hasfun*-**функцией. Ряд дополнительных вопросов, относящихся к тестированию выражений, рассматривается несколько ниже**.**

Начиная с ***Maple 8*,** пакет *пополнен* модулем **TypeTools,** пять процедур которого служат для расширения списка типов, стандартно прддерживаемых ***type*-**функцией. Со своей стороны, еще раньше мы определили процедуру ***usertype*,** обеспечивающую ряд массовых операций для работы с пользовательскими типами, поддерживаемых процедурами с именами формата **'*type/name*',** where ***name* –** имя пользовательского типа.

Модуль **TypeTools** может быть использован для *расширения* стандартной ***type*-**функции пакета типами, определенными пользователем, однако его применение приводит к созданию двух непересекающихся систем типизации в среде ***Maple*,** а именно: (**1**) системе типов пользователя, базирующейся на конструкциях вида **`*type/name*`:=proc ... end proc,** сохраненных в ***Maple*-**библиотеках, и (**2**) системе, базирующейся на модуле **TypeTools.**

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.***

```
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 7/LIB/UserLib");
 "C:/Program Files/Maple 7/LIB/UserLib", [Lower, Table, Upper, arity, assignable1, binary, rlb,
 boolproc, byte, complex1, digit, dir, dirax, file, file1, fpath, heap, letter, libobj, lower, mla, mlib, mod1,
 nestlist, nonsingular, package, path, plot3dopt, plotopt, realnum, sequent, setset, ssign, upper]
> usertype('active');   ⇒ [dir, mla, mlib]
```

Требуемая функция процедуры ***usertype*** может быть получена использованием ее вызовом 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. Данная функция позволяет получать только имена типов, зарегистрированных, используя вышеуказанный метод. В частности, она не может использоваться для получения *встроенных* типов и созданных на основе **TypeTools**-модуля. Каждый возвращаемый вышеуказанный 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*`**, расположенных в ***Maple***-библиотеке, определенной полным путем **L** к ней. Список предваряется полным путем к ***Maple***-библиотеке, 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* туть к которой определен аргументом **L.** Никаких сообщений не выводится.

В процессе использования для ***Maple*** релизов **6-7** процедура ***usertype*** проявила себя достаточно эффективным средством.

Еще на одном весьма существенном моменте следует акцентировать ваше внимание. В среде пакета вызов ***type*(X, 'type')** тестирует выражение **X** на предмет быть допустимым

выражением типа, т.е. опознаваемым встроенной ***type***-функцией в качестве типа. При этом, по определению под выражением типа **X** понимается такое выражение, для которого успешно выполняется вызов ***type*(<*Выражение*>, X),** где в качестве *первого* аргумента выступает произвольное ***Maple-выражение***. В качестве *типов* допускаются системные типы {*integer*, *float*, *symbol*, *string* и др.}, типы, определяемые процедурами с именами формата **`*type/name*`**, а также типы, определяемые присвоением либо комбинацией типов. Однако, данная проверка не столь корректна, как декларируется. Следующий простой фрагмент подтверждает вышесказанное.

```
> restart; with(TypeTools);   ⇒ [AddType, GetType, GetTypes, RemoveType, Type]
> AddType(listodd, L -> type(L, 'list'(odd)));
> type([64, 59, 10, 17, 39], 'listodd'), type([59, 17, 39], 'listodd');   ⇒ false, true
> type(listodd, 'type');   ⇒ true
> `type/listeven`:= L -> type(L, 'list'('even')): type(listodd, 'type');   ⇒ true
> type([64, 10, 44], 'listeven'), type([59, 17, 39], 'listeven');   ⇒ true, false
> `type/neg`:= proc(x::{integer, float, fraction}) if x < 0 then true else false end if end proc:
> map(type, [64, -10, 59, -17, 39, -44], '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, [64, -10, 59, -17, 39, -44], '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 **TypeTools**-модуля и процедурами с именами вида **`*type/name*`.** Оказывается, что во втором случае вызов **type(neg, 'type')** некорректно тестирует *neg*-тип. Затем определяется **`*type/Type*`-**процедура**,** устраняющая данный недостаток. В заключительной части фрагмента данная 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 настоящем разделе средств тестирования типов данных, структур данных и выражений в их общем понимании можно уже сделать вполне определенные выводы о возможностях ***Maple-***языка в данном весьма важном аспекте его вычислительных как численных, так и алгебраических средств. Именно поэтому данному разделу возможностей языка было уделено более развернутое внимание в свете существенной ориентации пакета не столько на сугубо численные вычисления**,** сколько на сугубо алгебраические вычисления и преобразования. Теперь мы переходим к весьма важному разделу средств ***Maple-***языка, обеспечивающих конвертацию выражений одного типа в выражения другого типа.

1.7. *Конвертация Maple-выражений из одного типа в другой*

Так как многие типы числовых значений *взаимно* обращаемы, то ***Maple-***язык пакета для *конвертации* одного типа в другой располагает достаточно развитыми средствами, базирующимися на многоаспектной ***convert-***функции, которая служит не только для *конвертации* значений из одного типа в другой, но и конвертации выражений в целом из *одного* типа в другой, при этом понятие *типа* трактуется существенно *более* широко, чем мы говорили до сих пор, например, можно конвертировать *выражения*, содержащие волновые *Айри*-функции, в выражения с функциями Бесселя и т.д. В общем случае функция ***convert*** имеет следующий формат кодирования:

***convert*(**<*Выражение*>**,** <*Формат*> {**,** <*Список опций*>}**)**

где *выражение* представляет собственно сам *объект* конвертации, *формат* определяет тип конвертации (*его в случае столь широкого толкования понятия "**тип**" вполне допустимо называть более емким понятием "**формат**"*), а необязательный *третий* аргумент функции составляет *Список опций*, определяемых спецификой используемого *второго Формат-*аргумента функции. Язык пакета для ***convert-***функции в качестве значения ее второго аргумента допускает следующие *форматы* (*типы*) конвертации:

0F1 1F1 2F1 Abel Abel_RNF abs Airy algebraic and arabic arctrig arctrigh I array base Bessel Bessel_related binary binomial boolean_function boolean_operator bytes Chebyshev compose confrac conversion_table Cylinder D decimal egrees DESol diff dimensions disjcyc Ei_related elementary Elliptic_related equality erf erfc erf_related exp expln expsincos factorial FirstKind float fullparfrac GAMMA function_rules GAMMA_related global Hankel eaviside eun hex hexadecimal Int hypergeom int iupac Kelvin Kummer Legendre linearODE list listlist ln local mathorner Matrix matrix MeijerG metric MobiusR MobiusX MobiusY mod2 ModifiedMeijerG multiset name NormalForm numericproc octal or package parfrac permlist piecewise polynom PLOT3Doptions PLOToptions polar POLYGONS power pwlist radians radical set rational ratpoly RealRange record relation Riccati roman RootOf SecondKind std signum sincos sqrfree StandardFunctions stdle string Sum sum surd symbol tan system table temperature to_special_function trig trigh truefalse units unit_free Vector vector Whittaker windchill xor y_x `*` `+`

Уже из простого перечисления допустимых видов конвертации (*в количестве* **137** *для пакета* ***Maple 10***) следуют весьма широкие возможности ***convert-***функции. Многие из них будут достаточно детально рассмотрены в настоящей книге в различных контекстах, с другими можно детально ознакомиться по справочной системе пакета либо по книгам [34,42,56,63,78,96]. В представительном отношении набор *допустимых* форматов конвертации относительно предыдущих релизов пакета в ***Maple 10*** увеличился, да и в качественном отношении произведено существенное расширение механизма конвертации типов. Данное обстоятельство существенно расширило возможности ***Maple-***языка по преобразованию типов и форматов представления выражений. Следующие довольно распространенные *форматы конвертации* типов выражений представлены в табл. **6.**

Таблица 6

Формат-значение	***Конвертация первого аргумента функции в формат:***
RootOf	в терминах ***RootOf-***нотации**;** конвертирует все **I** и радикалы
radical	в терминах **I** и радикалов**;** *обратный* к предыдущему формату

{**and**	**or**}	{**and**	**or**}-представления, если аргумент **-** *список/множество*
array	структуры типа *массив*; конвертирует *списки*, *таблицы*, *массивы*; результат возвращается в уплотненном формате
base	из одной системы счисления в другую; имеет две формы
binary	*бинарного* числового представления
binomial	***GAMMA*-**функции и факториалы в ***binomial*-**функцию
bytes	байтов; конвертация *списка* **16-**ричных цифр или строк в байты
confrac	бесконечной дроби; перевод чисел, рядов, алгебраических выражений в бесконечно-дробную аппроксимацию
{`*`	`+`}	{*произведения*	*суммы*} всех операндов исходного выражения
decimal	**2-, 8-** и **16-**ричные (*в виде строк*) числа в **10-**ричные
degrees	*градусного* представления; обратным к нему является ***radians***
double	***float*-**числа *двойной* точности в другие форматы; поддерживается конвертация между платформами ***IBM, VAX, MIPS***
{***equality***	***lessthan***	***lessequal***}	равенства или отношения; {*отношение*} →{**=**	**<**	**≤**}
exp	тригонометрические функции в экспоненциальные
expln	элементарные функции в терминах функций {***exp***, ***ln***}
expsincos	тригонометрические функции в терминах {***exp***, ***sin***, ***cos***}
Ei	тригонометрические, гиперболические и логарифмические интегралы в экспоненциальные интегралы ***Ei*(x)**
factorial	конвертация ***GAMMA*-**функции и биномиалов в факториалы
float	***float*-**типа; в значительной мере подобна ***evalf*-**функции
fullparfrac	*рациональное* выражение в полностью линейное рациональное
hex	десятичное неотрицательное число в **16-**ричное *строчное*
horner	полинома в форме Горнера
list	таблицы, векторы или выражения в *списочную* структуру (*в случае выражения элементами списка будут его операнды*)
listlist	*вложенного* списка; конвертируются *списки/массивы*
ln	обратные тригонометрические функции в логарифмические
mathorner	полином в *матричную* форму Горнера
matrix	массив или вложенный список в *матричную* структуру
metric	английскую систему мер в метрическую
mod2	приведения по (**mod 2**); допустимо вхождение {**and, or, not**}
multiset	в специальную ***multiset*-**форму
{***symbol***	***name***}	конвертация в {***symbol***	***name***}-тип; ***symbol* -** синоним ***name***
numericproc	*символьную* **F(x,y)-**функцию в *числовую* **F`(x,y)-**функцию
octal	**8-**ричного представления; допустимо определение точности
parfrac	перевод в частично-дробный формат; расширенные опции
piecewise	в формат кусочно-определенной функции
polar	комплексные числа в *полярную* форму представления
pwlist	конвертация кусочно-определенной функции в список
radians	перевод из *радианной* меры в *градусную*; обратная к ***degrees***
{***rational***	***fraction***}	перевод из ***float*-**формата в приближенный *рациональный* вид
set	перевод табличной структуры, *массива*, *выражения* в множество
{***signum***	***abs***}	замена всех {***abs***	***signum***}-функций выражения на {***signum***	***abs***}

sincos	тригонометрические функции в {***sin, cos, sinh, cosh***}
sqrfree	представление полиномов без квадратов
string	конвертация выражения в *строчный* формат
table	перевод *списочной* структуры или *массива* в *табличную* форму
tan	перевод тригонометрических функций в ***tan-***представление
trig	экспоненциальные и тригонометрические функции в форме выражений из функций {***exp, sinh, cosh, tanh, sech, csch, coth***}
vector	*список* или *массив* в *вектор*, а в общем случае и в *матрицу*

В качестве *примера* использования представленных в табл. **6** значений *формат*-аргумента функции приведем фрагмент, иллюстрирующий возможности функции по конвертации различных форматов представления выражений из одного в другой**:**

```
> convert(table([Kr, G, S, Art]), 'array');   ⇒ [Kr, G, S, Art]
> convert([2, 0, 0, 6], 'base', 10, 2);   ⇒ [0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1]
> convert(2006, 'binary');      ⇒ 11111010110
> convert(Tallinn, 'bytes');   ⇒ [84, 97, 108, 108, 105, 110, 110]
> convert([73, 110, 116, 101, 114, 110, 97, 116, 105, 111, 110, 97, 108, 32, 65, 99, 97, 100, 101, 109,
  121, 32, 111, 102, 32, 78, 111, 111, 115, 112, 104, 101, 114, 101], 'bytes');
                          "International Academy of Noosphere"
> convert(sin(x)/x, 'confrac', x, 8);   ⇒ 1 + x^2 / (-6 + x^2 / (-10/3 + 11 x^2/126))
> convert(x*y*z, `+`), convert(x*y*z – h + 64, `*`);   ⇒ x + y + z,  -64 x y z h
> [convert(101100111, 'decimal', 'binary'), convert(64, 'decimal', 'octal'), convert(`ABCDEF`,
  'decimal', 'hex')];   ⇒ [359, 52, 11259375]
> convert(y*sin(x) + x*cos(y), 'exp');   ⇒ -(1/2) I y (e^(x I) - 1/e^(x I)) + x ((1/2) e^(y I) + (1/2) 1/e^(y I))
> convert([[V, 42, 64], [G, 47, 59], [Art, 89, 96]], 'matrix');   ⇒ [V 42 64; G 47 59; Art 89 96]
> convert([inch, ft, yard, miles, bushel], 'metric');
        [127 m/5000, 381 m/1250, 1143 m/1250, 25146 km/15625, 0.035238775147289395200 m^3]
> convert([Art, Kr, Sv, Arn, V, G], `and`);   ⇒ Art and Kr and Sv and Arn and V and G
> G:= (x, y) -> x*sin(y):  convert(G(x, y), 'numericproc');
 proc(_X, _Y)
 local err;
      err := traperror( evalhf( (x×sin(y))(_X, _Y) ));
      if type( [ err ], [ numeric ] ) then err
      else
           err := traperror( evalf( (x×sin(y))(_X, _Y) ));
           if type( [ err ], [ numeric ] ) then err else undefined end if
      end if
 end proc
> convert(a*x + b*y - 3*x^2 + 5*y^2 - 7*x^3 + 9*y^3, 'horner', [x, y]);
                              (b + (5 + 9 y) y) y + (a + (-3 - 7 x) x) x
```

С учетом сказанного приведенный фрагмент *использования* вышерассмотренных пакетных средств конвертации *типов* (*форматов*) выражений в широком понимании данного термина представляется достаточно прозрачным и особых пояснений не требует**.** Более детальное ознакомление со средствами данного *класса* ***Maple*-**языка рекомендуется проводить при непосредственной практической работе в *его* среде. Нами также был определен ряд довольно полезных средств конвертации объектов ***Maple*-**типа в объекты типа *rtable*, и наоборот [103]. Например, вызов нижеприведенной процедуры ***avm_VM*(G)** возвращает результат преобразования объекта, определенного фактическим аргументом **G** типа {*array*, *vector*, *matrix*}, в объект типа {*Vector*, *Matrix*}, и наоборот **–** объект типа {*Vector*, *Matrix*} в векторы или матрицы типа ***Maple***.

```
avm_VM := proc(G::{Matrix, Vector, matrix, vector, array})
local k, h, S, H, M;
    S := H → rtable(op(2, eval(G)), {op(op(3, eval(G)))}, 'subtype' = H);
    `if`(type(G, 'array') = true and nops([op(2, eval(G))]) = 1, S(Vector['row']),
        `if`(type(G, 'array') = true and nops([op(2, eval(G))]) = 2 and
        nops([seq(k, k = op(2, eval(G))[1])]) = 1, rtable(
        op([assign(h = op(3, eval(G)))], [seq(rhs(h[k]), k = 1 .. nops(h))]]),
        'subtype' = 'Vector'['row']), `if`(
        type(G, 'array') = true and nops([op(2, eval(G))]) = 2 and
        nops([seq(k, k = op(2, eval(G))[2])]) = 1, rtable(
        op([assign(h = op(3, eval(G)))], [seq(rhs(h[k]), k = 1 .. nops(h))]]),
        'subtype' = 'Vector'['column']), `if`(whattype(G) = 'Matrix', op([assign(M =
        Matrix(op(1, G), op(2, G), subs('readonly' = NULL, MatrixOptions(G))))),
        matrix(op(1, G), [op(op(2, G))])]), `if`(whattype(G) = 'Vector'['row'],
        vector(`if`(whattype(op(1, G)) = `..`, rhs(op(1, G)), op(1, G)),
        op([assign(h = op(2, eval(G)))], [seq(rhs(h[k]), k = 1 .. nops(h))]])), `if`(
        whattype(G) = 'Vector'['column'], matrix(
        `if`(whattype(op(1, G)) = `..`, rhs(op(1, G)), op(1, G)), 1,
        op([assign(h = op(2, eval(G)))], [seq(rhs(h[k]), k = 1 .. nops(h))]])),
        S(Matrix)))))))
end proc
```

При этом, следует отметить, что в нашей книге [12] (*прилож.* **1**) представлен целый ряд особенностей выполнения конвертации выражений одного типа в другой посредством ***convert*-**функции, имеющих важное значение при практическом использовании данного средства. Там же приведены и другие полезные *замечания* относительно ***convert*-**функции, сохраняющие свою актуальность и для последующих релизов ***Maple***.

Следует отметить, что в таблицах главы и в последующих не приводится исчерпывающей характеристики функций либо других средств ***Maple*-**языка, а только основные их назначение и аргументы. Некоторые их особенности приводятся в прилож. **3** [12], полную же информацию по любой функции, поддерживаемой пакетом, можно оперативно получать по конструкции **?<***функция***>** или в документации по ***Maple*,** например, [81-83,89]. Следует еще раз напомнить, что кодирование идентификаторов в пакете ***Maple*** *регистро-зависимо*, поэтому необходимо правильно кодировать все используемые идентификаторы. Несоблюдение данного условия лежит в основе многих ошибок.

1.8. *Функции математической логики и средства тестирования пакета*

Для решения задач математической логики, пропозиционального исчисления, а также организации логических конструкций, управляющих вычислительным процессом в документе ***Maple*** либо программе (*условные переходы, ветвления, циклические и итеративные вычисления и др.*), ***Maple-***язык располагает рядом *встроенных, библиотечных* и *модульных* процедур и функций, операторов и иных конструкций, значения аргументов или возвращаемых результатов которых получают *логические значения* ***true*** (*истина*), ***false*** (*ложь*) и ***FAIL*** (*неопределенность*). К группе данных средств относятся и так называемые *тестирующие* функции и процедуры. Такие средства в зависимости от результата тестирования своего аргумента возвращают значение ***true*** или ***false*.** Ряд свойств таких функций нами рассматривался выше в связи с другими вопросами ***Maple-***языка, остальные будут рассматриваться здесь и в последующих разделах по мере необходимости и в контексте с различными вопросами практического программирования.

В основе математической логики, поддерживаемой ***Maple-***языком, лежит понятие ***булевого*** (***boolean;*** логического) выражения. Как уже отмечалось**,** ***Maple-***язык поддерживает трехзначную логику {***true, false, FAIL***}** для всех булевых операций. Булевы выражения формируются на основе *базовых логических* операторов {**and, or, not**}**,** образующих функционально *полную* систему (*в смысле возможности представления на их основе произвольной логической функции*) и операторов ***отношения*** {**<** (*меньше*) | **<=** (*не больше*) | **>** (*больше*) | **>=** (*не меньше*) | **=** (*равно*) | **<>** (*не равно*)}. *Булевский* (***boolean***) тип идентифицируется тестирующими функциями ***type*** и ***typematch,*** и процедурой ***whattype;*** и при этом, ***Maple-***язык дифференцирует ***boolean-***тип на два базовых подтипа**:** ***relation*** и ***logical.*** К ***relation-****подтипу* относятся выражения вида {**<** | **<=** | **=** | **<>**}, а к ***logical-****подтипу* **-** выражения вида {**or, and, not**}**;** тогда как ***boolean-****тип* определяют как собственно выражения двух указанных подтипов**,** так и их сочетания, а также *логические* константы {***true, false***}. Все логические типы тестируются {***type*** | ***typematch***}-функцией следующего формата кодирования**:**

{***type*** | ***typematch***}**(**<*Выражение*>**,** {***boolean*** | ***relation*** | ***logical***}**)**

как это иллюстрирует следующий простой пример**:**

```
> [type(AV <> 64, 'relation'), type(AV and AG, 'logical'), type(true <> false, 'boolean')],
  [whattype(AV and AG), typematch(AV <> 59, 'relation')];   ⇒ [true, true, true], [and, true]
```

Для вычисления ***Maple-***выражений в булевой трактовке используется ***evalb-***функция, имеющая простой формат кодирования**:** ***evalb*****(**<*Выражение*>**)** и возвращающая логическое значение {***true*** | ***false*** | ***FAIL***}**;** если это невозможно, то *выражение* возвращается *невычисленным*. Основной задачей ***evalb-***функции является вычисление ***Maple-***выражений, содержащих операторы отношения**,** в терминах логических значений. Это необходимо в целом ряде случаев**,** связанных с различного рода задачами анализа вычислительных конструкций**,** ибо ***Maple-***язык трактует выражения, содержащие операторы *отношения*, как алгебраические уравнения или неравенства, если выражения дополнительно не содержат *логических* {**and, or, not**}-операторов. И только в качестве аргументов ***evalb-***функции либо в {***if*** | ***while***}-предложениях ***Maple-***языка они получают логическую трактовку. При этом, следует иметь в виду, что ***Maple-***язык конвертирует выражение, содержащее операторы *отношения* {**>, >=**}, в эквивалентное ему выражение в терминах {**<, <=**}-операторов. Более того, т.к. ***evalb-***функция не производит упрощения *выражения*-аргумента, то в ряде случаев ее применение может приводить к некорректным результатам. Поэтому, перед вызовом ***evalb-***функции рекомендуется предварительно упрощать выраже-

ние, передаваемое ей в качестве *фактического* аргумента; это можно, в частности, делать и по ***simplify-***функции. Простой фрагмент иллюстрирует вышесказанное:

```
> whattype(AVZ = AGN), evalb(AVZ = AGN), evalb(AVZ = AVZ);   ⇒ =, false, true
> AV:= 64:  whattype(AV <= 64), evalb(AV = 64), evalb(AV <= 64);   ⇒ <=, true, true
> whattype(sqrt(64) <> ln(17)), evalb(sqrt(64) <> ln(17));   ⇒ <>, true
```

Следует иметь в виду, что вычисление *логических* выражений подчиняется следующему правилу: *первым* вычисляется левый операнд {**and** | **or**}-оператора и вычисление правого его операнда производится *только* тогда, когда логическое значение *левого* операнда может способствовать получению ***true-***значения выражения в целом. Так, правый операнд логического **and-**выражения следующего вида:

```
> G:= 0:  V:= 17:  if (G = 64) and (V/G >= 0.25)  then  printf(`%3s\%4f`, `G=`, V/G)  end if;
> G:= 0:  V:= 20:  if (G = 52) or (V/G >= 9.47)  then  printf(`%3s\%4f`, `G=`, V/G)  end if;
Error, numeric exception: division by zero
```

не вычисляется и не вызывает ошибочной *ситуации "деления на нуль"*, т.к. *левый* его операнд (**G = 64**) для **G = 0** при вычислении возвращает ***false-***значение, не способствующее получению ***true-***значения **and-**оператора в целом, *независимо* от результата вычисления его *правого* операнда, вызывающего на данном **G-**значении указанную ошибочную ситуацию. Иная ситуация, как показано, имеет место для случая **or-**оператора.

Правила выполнения *логических* {**and,or,not**}-операторов на {***true, false, FAIL***}-значениях в качестве операндов определяются следующими таблицами *истинности:*

and	*true*	*false*	*FAIL*
true	*true*	*false*	*FAIL*
false	*false*	*false*	*false*
FAIL	*FAIL*	*false*	*FAIL*

or	*true*	*false*	*FAIL*
true	*true*	*true*	*true*
false	*true*	*false*	*FAIL*
FAIL	*true*	*FAIL*	*FAIL*

	not
true	*false*
false	*true*
FAIL	*FAIL*

Смысл приведенных правил достаточно прозрачен и пояснений не требует, например:

```
> [FAIL and true,  false or FAIL,  true or FAIL,  not FAIL];   ⇒ [FAIL, FAIL, true, FAIL]
```

Следует отметить, что до **6**-го релиза ***Maple*** для расширения круга решаемых задач математической логики и ее приложений *дополнительно* предоставлял **10** модульных функций, поддерживаемых средствами **logic**-модуля. Однако, после нашей *принципиальной* критики ряда его функций, точнее результатов вызовов ***bequal-***функции на ***FAIL***-значениях [10-12], данный модуль был исключен из ***Maple***. И аналога этого (*в целом весьма полезного*) модуля не было до ***Maple 10***. И только в последнем релизе появился модуль **Logic,** чьи средства предна-начены для работы с выражениями, используя двузначную булеву логику, т.е. без ***FAIL***-значения. По конструкции **?Logic** читатель может детально ознакомиться со средствами данного модуля.

```
Bit := proc(
B::{string, symbol, list(integer)}, bits::{procedure, list({equation, integer})})
local a, b, c, k;
    c := x → [seq(parse(x[k]), k = 1 .. length(x))];
    if type(B, {'symbol', 'string'}) and length(B) = 1 then
        a := c(cat("", convert(op(convert(B, 'bytes')), 'binary')));
        b := [0 $ ('k' = 1 .. 8 − nops(a)), op(a)]
    elif type(B, 'list'('integer')) and nops(B) = 1 and belong(B[1], 0 .. 255) then
        a := c(cat("", convert(op(B), 'binary')));
        b := [0 $ ('k' = 1 .. 8 − nops(a)), op(a)]
```

```
    else error "1st argument must be symbol/string of length 1 or integer list, but\
        as received <%1>", B
    end if;
    if type( bits, 'list'('integer') ) and belong( { op( bits ) }, 1 .. 8 ) then
        [ seq( b[ k ], k = { op( bits ) } ) ]
    elif type( bits, 'list'('equation') ) and belong( { seq( lhs( k ), k = bits ) }, 1 .. 8 ) and
    belong( { seq( rhs( k ), k = bits ) }, 0 .. 1 ) then
        seq( assign( 'b'[ lhs( bits[ k ] ) ] = rhs( bits[ k ] ) ), k = 1 .. nops( bits ) );
        convert( parse( cat( op( map( convert, b, 'string' ) ) ) ), 'decimal', 'binary' )
    elif type( bits, 'procedure' ) then
        convert( parse( cat( op( map( convert, bits( b ), 'string' ) ) ) ), 'decimal', 'binary' )
    else error "2nd argument <%1> is invalid", bits
    end if
end proc
> Bit(`S`,[k$k=1..8]),Bit([59],[1,3,8]), Bit([59],[2=1,4=0,8=0]),Bit("G",[6]),Bit("0",[6=1,7=1,8=1]);
                                [0, 1, 0, 1, 0, 0, 1, 1], [0, 1, 1], 106, [1], 7
> R:=(L::list) -> [L[nops(L)-k]$k=0..nops(L)-1]:  Bit("G", [k$k=1..8]), Bit("G", R), Bit("S", R),
  Bit([59], R);     ⇒ [0, 1, 0, 0, 0, 1, 1, 1], 226, 202, 220
```

Для обеспечения *логических операций* и *по-битной* (*поразрядной*) обработки информации нами также был определен ряд процедур и модулей [42,103,109]. Так вышеприведенная процедура ***Bit*(B, *bits*)** обеспечивает выполнение следующих базовых по-битных операций с символом, указанным первым аргументом **B** (*в качестве* **B** *могут выступать* 1-*элементные строка либо символ, а также* 1-*элементный список, чей элемент определяет десятичный код символа из диапазона* 0..255):

(**1**) если второй аргумент ***bits*** имеет вид **[n1, n2, …],** вызов процедуры ***Bit*(B, *bits*)** возвращает значения битов символа **B,** расположенных в его позициях **nj** (**nj** лежит в **1 .. 8**);
(**2**) если второй аргумент ***bits*** имеет вид **[n1 = b1, n2 = b2, …],** возвращается десятичный код символа, полученного заменой значений битов символа **B** (*в его позициях* **nk**) на бинарные значения **bk (nk = 1..8; bk =** {**0**|**1**}).
(**3**) если второй аргумент ***bits* –** имя процедуры, определенной над списками, возвращается результат применения процедуры к списку значений всех битов символа **B.**

Наконец, в двух последних случаях возвращается десятичный код символа – результата обработки исходного символа **B.**

Базовые средства поддержки булевой алгебры обеспечиваются модулем **boolop,** а также нижеследующей процедурой ***BitWise*** [103,108,109]:

```
BitWise := proc(BO::symbol, n::nbinary)
local a, b, c, d, k, h;
    assign( a = length( n ), c = "", h = 0 );
    if not member( BO, { AND, ADD, OR, NOR, XOR, NOT, IMPL } ) then error
        "operation <%1> is different from {ADD, AND, OR, XOR, NOR, NOT\
        IMPL}", BO
    elif BO = NOT then
        parse( cat( seq( `if`( k = "0", "1", "0" ), k = convert( n, 'string' ) ) ) )
    else
        if 2 < nargs and type( args[ 3 ], 'nbinary' ) then assign( b = length( args[ 3 ] ) )
```

```
        else error "3rd argument is absent or has a type different from nbinary'
        end if;
        if a = b then assign('a' = "" || n, 'b' = "" || args[3])
        elif a < b then assign('b' = "" || args[3], 'a' = cat(seq("0", k = 1 .. b - a), n))
        else assign('a' = "" || n, 'b' = cat(seq("0", k = 1 .. a - b), args[3]))
        end if;
        if BO = ADD then
            for k from length(a) by -1 to 1 do
                d := map(parse, {b[k], a[k]});
                if d = {0, 1} then
                    if h = 1 then c := cat("0", c)
                    else assign('c' = cat("1", c), 'h' = 0)
                    end if
                elif d = {1} then
                    if h = 1 then c := cat("1", c)
                    else assign('c' = cat("0", c), 'h' = 1)
                    end if
                else assign('c' = cat("", parse(a[k]) + h, c), 'h' = 0)
                end if
            end do;
            parse(c)
        elif BO = OR then parse(
            cat(seq(`if`([a[k], b[k]] = ["0", "0"], "0", "1"), k = 1 .. length(a))))
        elif BO = NOR then parse(
            cat(seq(`if`([a[k], b[k]] = ["0", "0"], "1", "0"), k = 1 .. length(a))))
        elif BO = XOR then parse(cat(seq(
            `if`([a[k], b[k]] = ["0", "0"] or [a[k], b[k]] = ["1", "1"], "0", "1"),
            k = 1 .. length(a))))
        elif BO = AND then parse(
            cat(seq(`if`([a[k], b[k]] = ["1", "1"], "1", "0"), k = 1 .. length(a))))
        else parse(
            cat(seq(`if`([a[k], b[k]] = ["1", "0"], "0", "1"), k = 1 .. length(a))))
        end if
    end if
end proc
> BitWise(ADD, 1001, 100101101), BitWise(NOT, 1001), BitWise(AND, 1001, 100101101),
  BitWise(XOR, 100101101, 1001), BitWise(OR, 100101101, 1001), BitWise(IMPL, 1001, 1001);
                    100110110, 110, 1001, 100100100, 100101101, 1111
```

Вызов процедуры ***BitWise*(BO, n {, m})** возвращает результат поразрядной операции **BO** над бинарными числами **n** {*и* **m**}. При этом в качестве **BO** выступают традиционные булевы операции {**AND, OR, XOR, NOR, NOT, IMPL**}, где **IMPL** обозначает ***implies***-операцию; тогда как **ADD –** операция поразрядного сложения по **mod 2.**

Так модуль **boolop** [109] экспортирует ряд полезных функций/операторов, обеспечивающих базовые операции булевой алгебры, а именно:

&andB, &orB, &xorB, &notB, &impB

где, в частности, **&andB – n**-арная логическая функция/оператор **and:**

```
boolop:-&andB := proc(x::{0, 1}, y::{0, 1})
local a, k;
    if nargs = 0 or nargs = 1 then
        error "actual arguments in single number or are absent'
    elif nargs = 2 then if [x, y] = [1, 1] then 1 else 0 end if
    else
        if type({args[3 .. -1]}, set({0, 1})) then
            a := procname(x, y);
            for k from 3 to nargs do a := procname(a, args[k]) end do;
            a
        else error "arguments starting with 3rd should have binary type but had re\
            ceived %1", [args[3 .. -1]]
        end if
    end if
end proc
> with(boolop): 0 &andB 0, 1 &andB 1, 0 &andB 1, 1 &andB 0, `&andB`(1, 0, 1, 0, 1, 0);
                                0, 1, 0, 0, 0
```

Следующая процедура ***logbytes*** обеспечивает поддержку базовых логических операций {***xor, and, or, not, implies***} над последовательностями байтов типа {*string*, *symbol*}.

```
logbytes := proc(O::symbol)
local a, k, j;
    if nargs = 1 then error "number of arguments must be > 1"
    elif not member(O, {`and`, `not`, `or`, `xor`, `implies`}) then error "1st argu\
        ment must be {`and`,`or`,`xor`,`not`,`implies`} but has received <%1>", O
    else
        a := {};
        for k from 2 to nargs do
            if (proc(x)
                try type(x, {'symbol', 'string'}) and type(x, 'byte')
                catch : return false
                end try
            end proc)(args[k]) then next
            else a := {k, op(a)}
            end if
        end do
    end if;
    if a ≠ {} then
        error "arguments with numbers %1 must be bytes of type {string, symbol}", a
    else a := map(Bit, [args[2 .. -1]], ['k' $ ('k' = 1 .. 8)])
```

```
    end if;
    if O = `and` then xbyte1(
        [seq(boolop[boolop:-`&andB`](seq(a[k][j], k = 1 .. nops(a))), j = 1 .. 8)])
    elif O = `or` then xbyte1(
        [seq(boolop[boolop:-`&orB`](seq(a[k][j], k = 1 .. nops(a))), j = 1 .. 8)])
    elif O = `xor` then xbyte1(
        [seq(boolop[boolop:-`&xorB`](seq(a[k][j], k = 1 .. nops(a))), j = 1 .. 8)])
    elif O = `not` then xbyte1([seq(boolop[boolop:-`&notB`](a[1][j]), j = 1 .. 8)])
    elif O = `implies` and 2 < nargs then xbyte1(
        [seq(boolop[boolop:-`&impB`](seq(a[k][j], k = 1 .. 2)), j = 1 .. 8)])
    else 'procname(args)'
    end if
end proc
> logbytes(`and`, A,v,z, A,G,N, V,S,V), logbytes(`or`, A,v,z, A,G,N, V,S,V), logbytes(`xor`,
  A,v,z, A,G,N, V,S,V);   ⇒   "@", "□", "V"
```

Логические операции производятся над соответствующими битами байтов, определенных последовательностью аргументов, начиная со второго. Тогда как в качестве *первого* **O**-аргумента выступают указанные логические операции. Результат возвращается в виде байта в *string*-формате. При этом, предполагаются **n**-*арные* операции {**`xor`**,**`and`**,**`or`**}, *бинарная* операция **`implies`** и *унарная* **`not`**.

Рассмотренные в настоящем разделе средства ***Maple-***языка по обеспечению решения задач математической логики и ее прикладных аспектов будут в различных контекстах использоваться при решении разнообразных иллюстративных и прикладных математических задач**,** в первую очередь при программировании *логических* компонент конструкций**,** управляющих вычислительным процессом в ***Maple-***документах и программах (*условные переходы*, *циклы и другие управляющие структуры*).

• ***Тестирующие*** функции, значительная часть которых рассмотрена выше, возвращают значение {***true***|***false***} в зависимости от {*истинности*|*ложности*} того или иного проверяемого *логического условия* (***места***). И в этом смысле они могут входить в состав булевых выражений. Выше был рассмотрен целый ряд тестирующих функций, обеспечиваемых языком пакета. В дальнейшем оставшиеся функции данного типа будут представлены при обсуждении соответствующих приложений ***Maple-***языка.

Здесь мы несколько расширим наше представление о концепции типов, поддерживаемой пакетом. Как уже отмечалось выше, базовыми тестирующими средствами являются функции ***type*** и ***typematch,*** а также процедура ***whattype.*** *Третья* из них возвращает поддерживаемый ***Maple*** тип указанного своим фактическим аргументом ***Maple-****выражения*, тогда как две первые возвращают {***true***|***false***}-значение в зависимости от {*истинности*|*ложности*} факта эквивалентности типа**,** указанного их первым фактическим аргументом (*выражение*) и их вторым фактическим аргументом **–** идентификатором *типа***,** распознаваемого ***Maple-***языком.

Все распознаваемые ***Maple-***языком типы можно классифицировать на *две* группы**:** *внешние* (*поверхностные*) и *вложенные***.** К *первой* группе относятся *типы*, для тестирования которых вполне достаточно информации о самом верхнем уровне структурного дерева тестируемого выражения. В качестве *внешних* ***Maple-***язык расматривает следующие **66** *типов* выражений**:**

algebraic anything applied array boolean equation even float fraction function list indexed integer laurent linear listlist logical mathfunc matrix moduledefinition odd monomial name negative nonnegative numeric point positive procedure radical range rational relation RootOf rtable scalar SDMPolynom SERIES series set sqrt string table taylor trig type uneval vector zppoly `!` `*` `+` `..` `.` `<=` `<>` `<` `=` `and` `intersect` `minus` `module` `not` `or` `union` `^`

Большинство тестируемых вышеупомянутыми средствами ***Maple-***языка типов относятся именно к *первой* группе **–** *внешним типам*. *Типы*, требующие для своего тестирования анализа (*возможно рекурсивного*) *всего* структурного дерева выражений**,** относятся ко *второй* группе **–** *вложенным* типам. В качестве *вложенных* ***Maple-***язык рассматривает следующие **13** типов выражений, а именно**:**

algfun algnum appliable constant cubic expanded linear polynom quadratic quartic radfun radnum ratpoly

Все вышеперечисленные типы относятся к пакету ***Maple 10*****.** Следует отметить, что *константный* тип относится ко второй группе, т.к. для его тестирования требуется анализ всего структурного дерева выражений на предмет отсутствия *вхождения* в него переменных компонент, т.е. **x-**компонент, для которых имеет место определяющее *соотношение* ***type*****(x,*name*) ⇒ *true*.** В приводимых ниже примерах будут детализированы многие практические аспекты работы с типами при организации различных вычислительных конструкций, использующих функциональные средства ***Maple-***языка.

По ***testeq-***процедуре, имеющей формат кодирования ***testeq*****(A** { **{,|=}** **B}),** производится *стохастическое* тестирование эквивалентности двух ***Maple-***выражений либо эквивалентность заданного единственным фактическим аргументом выражения *нулю***.** В случае установления факта эквивалентности возвращается ***true-***значение**,** в противном случае **–** ***false-***значение**.** При этом, ***false-***значение является достоверным**,** тогда как значение ***true*** возвращается корректно с очень высокой степенью достоверности**.** В процессе тестирования процедура не только производит вычисления *выражений*-аргументов, но и их алгебраические преобразования и упрощения. При невозможности произвести тестирование возвращается ***FAIL-***значение. Простой пример иллюстрирует вышесказанное**:**

> [testeq(sin(x)/cos(x), tan(x)), testeq(sin(x)^2 + cos(x)^2 - 1)]; ⇒ [*true, true*]
> [testeq(sqrt(-1) - I), testeq(x^2 + 4*x + 4, (x + 2)^2)]; ⇒ [*true, true*]

Весьма важной для задач *формального* анализа ***Maple-***выражений представляется тестирующая ***match-***процедура, имеющая следующий формат кодирования**:**

match**(<***Выражение***> = <***Шаблон***>, <*Id*>, '<*Id*_1>')**

и возвращающая ***true-***значение**,** если устанавливается соответствие структуры тестируемого**,** заданного первым аргументом**,** *выражения* указанному вторым аргументом функции *шаблону* по указанной *ведущей* ***Id-***переменной. *Шаблон* представляет собой выражение по ведущей ***Id-***переменной с формальными параметрами, на соответствие структуре которого и производится проверка *выражения*. Например, выражение **z+a*x^2+b*x+c** определяет *шаблон* квадратного трехчлена по ведущей **x-**переменной. В случае успешного тестирования в *невычисленную* ***Id*_1-**переменную помещается множество значений параметров *шаблона*, на которых он *структурно эквивалентен* тестируемому выражению. Следующий простой фрагмент иллюстрирует вышесказанное**:**

> [match(ln(Pi)/ln(x) + x^y = a/ln(x) + x^b, x, 'h'), h]; ⇒ [*true*, {b = y, a = ln(π)}]
> [match(1942*sqrt(x) + 64^x - 10*y = a*x^d + b^x + c, x, 'h'), h];
[*true*, {a = 1942, d = 1/2, c = -10 y, b = 64}]
> [match(ln(z)*sqrt(x) + z^x - 10*y = a*x^b + c^x + d, x, 'h'), h];

[*true*, {a = ln(z), c = z, d = -10 y, b = 1/2}]

> [match(ln(3*z/(1 + G(x)))*z + z^x - exp(y) = ln(a*z)*z + z^b + c, z, 'h'), h];

$$\left[true, \{ c = -\mathbf{e}^{y}, a = \frac{3}{1 + \mathrm{G}(x)}, b = x \} \right]$$

> [match(sqrt(3 + 32/G(x))*z + z^ln(x) - z/exp(y) + 10 = a*z + z^b + c, z, 'p'), p];

$$\left[true, \left\{ c = 10, a = -\frac{-\sqrt{\frac{3\,\mathrm{G}(x) + 32}{\mathrm{G}(x)}}\,\mathbf{e}^{y} + 1}{\mathbf{e}^{y}}, b = \ln(x) \right\} \right]$$

Механизм *шаблонов*, поддерживаемый ***match***-процедурой, позволяет устанавливать точную структуру ***Maple***-выражений на *основе* заданного *структурного шаблона* с формальными параметрами, вычисляя значения последних (*если установлено соответствие тестируемого выражения шаблону*) по принципу функционального уравнения. Между тем, успешное применение ***match***-процедуры для структурного тестирования выражений предполагает представление их в алгебраическом виде. В противном случае ***match***-процедура может необоснованно возвращать ***false***-значение, например:

> [match(19.89*sin(x) + x^57 - 10*y = a*sin(x) + x^b + c, x, 'h'), h]; ⇒ [*false*, h]

> [match(convert(19.89*sin(x) + x^57 - 10*y, 'rational') = a*sin(x) + x^b + c, x, 'h'), h];

$$\left[true, \{ a = \frac{1989}{100}, c = -10\,y, b = 57 \} \right]$$

Во избежание подобной ситуации выражения, содержащие *числовые значения*, рекомендуется предварительно конвертировать в эквивалентные выражения *rational*-типа.

К тестовым средствам ***Maple***-языка можно отнести и *шаблоны* проверки типов, базирующиеся на *структурных типах*. В качестве *базовых* ***Maple***-язык располагает *типами*, идентификаторы которых используются рассмотренными выше тестирующими функциями и в исчерпывающем виде представляются следующим списком (*для* ***Maple 10***):

! * + . < <= <> = @ @@ abstract_rootof algebraic algext algfun algnum algnumext And and anyfunc anyindex anything appliable applied arctrig Array array assignable atomic attributed boolean BooleanOpt builtin cubic character ClosedIdeal CommAlgebra complex complexcons composition const constant cx_infinity dependent dictionary dimension disjcyc cx_zero filedesc embedded_axis embedded_imaginary embedded_real equation even evenfunc xpanded extended_numeric extended_rational facint filename finite float float[] form fraction freeof function global hfarray identical imaginary implies in indexable indexed indexedfun infinity integer intersect last_name_eval laurent linear list listlist literal local logical mathfunc Matrix matrix minus module moduledefinition monomial MonomialOrder MVIndex name negative negint negzero neg_infinity NONNEGATIVE nonnegative nonnegint nonposint nonpositive nonreal Not not nothing numeric odd oddfunc operator Or or OreAlgebra package patfunc patindex patlist Point point polynom posint positive poszero pos_infinity prime procedure property protected quadratic quartic Queue radalgfun radalgnum radext radfun radfunext radical radnum radnumext Range range rational ratpoly ratseq realcons real_infinity relation RootOf rtable satisfies scalar SDMPolynom sequential SERIES series set sfloat ShortMonomialOrder SkewAlgebra SkewParamAlgebra SkewPolynomial specfunc specified_rootof Stack specindex sqrt stack std stdlib string subset suffixed symbol SymbolicInfinity symmfunc table tabular taylor TEXT trig trigh truefalse type typefunc

typeindex undefined uneval union unit unit_name Vector vector verification verify xor with_unit zppoly ^ ||

Данные идентификаторы используются в качестве фактического значения второго аргумента {***type*|*typematch***}-функции либо возвращаются ***whattype-***процедурой в результате тестирования ***Maple-***выражений. Часть данных типов рассматривалась выше, многие из оставшихся типов будут рассмотрены ниже при обсуждении соответствующих вопросов ***Maple-***языка. В представленном *списке* количество типов больше, чем в предыдущих релизах пакета и, как правило, с ростом номера релиза количество поддерживаемых им типов также растет.

В качестве типов, отличных от *базовых* (*представляемых единым идентификатором, принадлежащим приведенному выше списку, например*: ***integer, trig, list, set, float*** *и др.*), ***Maple-***язык поддерживает *структурные* типы, формальный *синтаксис* которых может описывать основную структуру тестируемых выражений. ***Maple-***язык допускает структурные типы двух видов: *простые* и *функциональные*. *Простые* структурные типы имеют следующий синтаксис: **<*Tun*_1>#<*Tun*_2>**, где два базовых типа соединены знаком оператора (**#**); в качестве основных язык для них допускает следующие: **`=` `<>` `<` `<=` `>` `>=` `..` and or not &+ &* `^` `.`**. А также: **'<*Tun*>', [<*Tun*>], *name*[<*Tun*>]** и ***fcntype***, определяющие соответственно *невычисленное выражение*, *список*, *индексированную ссылку* указанного типа и *функцию специального* типа. В качестве *функционального структурного* типа ***Maple-***язык допускает следующие основные синтаксически корректные конструкции:

***set*(<*Tun*>) –** *множество* элементов заданного *типа*;
***list*(<*Tun*>) –** *список* элементов заданного *типа*;
`&+`(<*Tun*>) – *сумма* термов заданного *типа*;
`&*`(<*Tun*>) – *произведение* сомножителей заданного *типа*;
***identical*(<*Выражение*>) –** *выражение*, идентичное заданному;
anyfunc*(<*Tun*>) –** произвольная *функция* заданного *типа*, кроме ***exprseq;
***function*(<*Tun*>) –** произвольная *функция* с аргументами заданного *типа*;
***specfunc*(<*Tun*>, F) –** *функция* **F** с аргументами заданного *типа*.

Наряду с рассмотренным выше механизмом *шаблонов* ***Maple-***язык поддерживает и механизм *типированных* шаблонов, базирующийся на структурных типах, ***typematch-***функции и (**::**)-*операторе типизации*. По (**::**)-*оператору*, имеющему максимальный приоритет, конструкция вида ***Id*::<*Tun*>** приписывает ***Id-***идентификатору заданный *тип*, который может быть как *базовым*, так и *структурным*. Рассмотренная в общих чертах ***typematch-***функция имеет формат кодирования следующего вида:

typematch*(**<*Выражение*>, <*Типированный шаблон*> {, '<Id***>'}**)**

В рамках первых *двух* аргументов ***typematch-***функция совпадает с ***type-***функцией и возвращает ***true-***значение в случае соответствия *типированной* структуры тестируемого *выражения* заданному *вторым* аргументом *типированному шаблону*, определяемому *базовым* либо *структурным* типом, иначе возвращается ***false-***значение.

Третий же ее необязательный **'*Id*'-**аргумент определяет невычисленную переменную и в сочетании с (**::**)-оператором *типирования* позволяет существенно расширять механизм типированных шаблонов. Суть расширения данного механизма сводится в общих чертах к следующему: в *типированный* шаблон, определяемый в качестве второго фактического аргумента ***typematch-***функции, каждому *типу* посредством (**::**)-оператора приписывается некоторая переменная (*типированная*) и в случае возврата функцией ***true-***значения в *переменную*, определяемую третьим фактическим аргументом, помещается список конкретных значений типированных переменных шаблона, на которых тестируемое *выражение*, определяемое первым фактическим аргументом функции, обеспечивает

возврат ***true*-**значения. Следующий несложный фрагмент иллюстрирует ряд примеров применения тестирующих функций {***type*, *typematch***}:**

> [typematch((99-42)..(99), a::integer..b::integer, 'h'), h]; ⇒ [*true*, [a = 57, b = 99]]
> [typematch(sin(x)^exp(x), a::trig^b::function, 'h'), h]; ⇒ [*true*, [a = sin(x), b = exp(x)]]
> type([sin(x), exp(y), ln(z)], list(function)); ⇒ *true*
> [typematch([sin(x),exp(y),ln(z)],list(L::function), 'h'), h];
[*true*, [L=sin(x), L=exp(y), L=ln(z)]]
> [typematch(sin(x)^19.99, b::trig^f::float, 'h'), h]; ⇒ [*true*, [b = sin(x), f = 19.99]]
> [typematch(x + y, `&+`(n::name, b::name), 'h'), h]; ⇒ [*true*, [n = x, b = y]]
> type(Art + Kr^(1/3) + 99,`&+`(name, radical, integer)); ⇒ *true*
> type({sin(x), cos(y), tan(z)}, set(trig)); ⇒ *true*
> [typematch({sin(x), cos(y), tan(z)}, set(L::trig), 'h'), h];
[*true*, [L=sin(x), L=cos(y), L=tan(z)]]
> type(F(57/180,47/99,10/89,32/67),function(rational)); ⇒ *true*
> [typematch(F(57/180, 47/99, 10/89, 3/96), f::function(G::rational), 'h'), h];
$$\left[true, \left[G = \frac{19}{60}, G = \frac{47}{99}, G = \frac{10}{89}, G = \frac{1}{32}, f = \mathrm{F}\left(\frac{19}{60}, \frac{47}{99}, \frac{10}{89}, \frac{1}{32} \right) \right] \right]$$
> [typematch(A||V||Z*abs(I^2), k::identical(AVZ), 'h'), h]; ⇒ [*true*, [k = *AVZ*]]
> [typematch(F(x)*G(x) = 57, a::`*`=b::integer, 'h'), h];
[*true*, [a = F(x) G(x), b = 57]]
> [typematch(F(x) <> 19.99, a::function <> b::float, 'h'), h]; ⇒ [*true*, [a = F(x), b = 19.99]]
> typematch((Art + 17)*(Kr + 10),`&+`(p::name, h::odd)&*`&+`(t::name, s::even)); ⇒ *true*
> [typematch((tan(x)*sin(x))^(64/59), (a::trig &* b::trig)^c::rational, 'h'), h];
$$\left[true, \left[a = \tan(x), b = \sin(x), c = \frac{64}{59} \right] \right]$$
> [typematch(sin(x)*cos(x)^(V*G), a::trig &* b::trig^(c::name &* d::name), 'h'), h];
[*true*, [a = sin(x), b = cos(x), c = V, d = G]]
> typematch(Raadiku(64, 59, 39, 17, 10, 44, 95, 99), specfunc(integer, Raadiku)); ⇒ *true*

С учетом сказанного приведенный фрагмент достаточно прозрачен и каких-либо особых пояснений не требует. Вместе с тем, механизмы *шаблонов*, поддерживаемые ***Maple*-**языком требуют для хорошего *усвоения* определенной наработки по их использованию, т. к. содержат целый ряд особенностей, здесь не рассматриваемых (*см. прилож.* **1** [12]).

Структурные типы обоих видов (*простого* и *функционального*) в сочетании с ***typematch*-**функцией позволяют достаточно эффективно тестировать структурную организацию ***Maple*-**выражений, однако в отличие от рассмотренного выше *механизма шаблонов*, поддерживаемого ***match*-**процедурой, в первом случае производится тестирование только структурно-типированного соответствия искомого выражения, определяемого первым фактическим аргументом ***typematch*-**функции, с заданным ее вторым аргументом-*шаблоном***.** Тогда как по ***match*-**процедуре производится тестирование соответствия искомого выражения, определяемого *левой* частью равенства, *правой* части, определяющей *шаблон***,** в математическом контексте, производя, при необходимости, вычисления и упрощения как исходного выражения, так и самого шаблона.

Вместе с тем, для обеспечения структурного типированного анализа выражений весьма полезной представляется ***patmatch*-**процедура, имеющая формат кодирования**:**

***patmatch*(**<*Выражение*>**,** <*Шаблон*> {**, 'h'**})**

где первый фактический аргумент определяет тестируемое *выражение***,** а второй **–** тестирующий *шаблон***,** т.е. *шаблон***,** на соответствие которому проверяется исходное *выражение*.

Шаблон представляет собой типированное (**::**)-оператором алгебраическое выражение от ведущих переменных исходного выражения, на соответствие которому оно и проверяется. Процедура ***patmatch*** возвращает ***true-***значение, если устанавливается структурно-типированное соответствие *тестируемого выражения* заданному *шаблону*, и ***false-***значение в противном случае. В первом случае необязательной **h-**переменной присваивается список *уравнений* таких**,** что имеет место соотношение ***subs*(h, <*Шаблон*>) = <*Выражение*>,** во *втором* **– h-**переменная возвращается невычисленной (*неопределенной*). При этом, ***patmatch-***процедура допускает использование и специального ключевого слова ***conditional,*** обеспечивающего возможность определения для шаблона дополнительных условий. Такие условия кодируются в одном из следующих двух форматах, а именно**:**

conditional*(**<*Шаблон*>**, <*Условие*>)** и ***conditional*(**<*Шаблон*> **= <B>, <Условие***>)**

В качестве *условия* выступают корректные *булевские выражения*, включающие типированные параметры шаблона, операторы отношения и логические операторы {**and, or, not**}. Вместе с тем, *условие* может включать и произвольные *булевские* функции и процедуры, возвращающие логические значения, например**: *isprime, issqr, type, typematch, match*** и даже ***patmatch,*** что позволяет производить *рекурсивный* структурно-типированный анализ ***Maple-***выражений. Следующий фрагмент иллюстрирует применение процедуры ***patmatch*** для структурно-типированного анализа алгебраических выражений:

```
> patmatch(sqrt(17*Art + 10*Kr), sqrt(a::odd*Art + b::even*Kr), 'h'), h;
                          true, [a = 17, b = 10]
> patmatch(10*x^3 + 3*y^2, a::even*x^b::prime + c::odd*y^d::even, 'h'), h;
                          true, [a = 10, b = 3, c = 3, d = 2]
> patmatch(sin(3)*x + F(p*y + t*z), a::trig*x + F(b::symbol*y + c::name*z), 'h'), h;
                          true, [a = sin(3), b = p, c = t]
> patmatch(sqrt(v)*x - ln(10)*Pi - exp(3), a::sqrt*x + b::atomic + c::realcons, 'h'), h;
                          true, [a = sqrt(v), c = -ln(10) Pi - exp(3), b = 0]
> patmatch(sqrt(17*Art + 10*Kr), sqrt(a::even*Art + b::prime*Kr), 't'), t;  => false, t
> patmatch(sqrt(17*Art + 10*Kr), conditional(sqrt(a::odd*Art + b::even*Kr), a< b^2), 'g'), g;
                          true, [a = 17, b = 10]
> patmatch(10*Art + 3*Kr, conditional(a::even*Art + b::odd*Kr, a > b^2 and a + b <= 15 and
  evalf(a/b) > 3 and a*b < 32), 'h'), h;  => true, [a = 10, b = 3]
> patmatch((10*A + 3*K)/(32*S - 52*G), conditional((10*A + b::odd*K)/(c::even*S - 52*G), c/b
> 10 and c + b >= 35 and b^3 < c), 'h'), h;  => true, [b = 3, c = 32]
> patmatch(57*sin(x)*exp(y) + 52.47*ln(z), conditional(a::anything*exp(y) + b::float*ln(z), b
  < 57 and _match(a = v*sin(x), x, 'p')), 'h'), h;  => true, [a = 57 sin(x), b = 52.47]
```

Из представленных примеров фрагмента достаточно прозрачно прослеживается *общий* принцип организации анализа выражений на основе ***patmatch-***процедуры как в ее основном формате, так и с использованием *уточняющих условий* на основе ключевого слова ***conditional***. При этом, следует отметить, что *второй* формат данного слова используется для определения табличных правил, операторов и функций, и несколько *детальнее* рассматривается ниже в связи с обсуждением функций пользователя.

Еще на одном функциональном средстве ***Maple-***языка, использующем механизм *шаблонов* и табличную структуру данных, следует остановиться особо. По вызову процедуры

compiletable*([Шаблон*_1 **=** *Выход*_1**, …,** *Шаблон*_n **=** *Выход*_n**])**

возвращается специальная **CT-**таблица, входами которой являются *шаблоны*, определяющие некоторые ***Maple-***выражения, а *выходами* таблицы **-** результаты соответствующей обработки данных выражений, например интегрирования, дифференцирования и др.

Тогда по вызову процедуры ***tablelook*(<***Выражение***>, CT)** возвращается выход **CT-**таблицы, входу которого по *шаблону* соответствует указанное первым фактическим аргументом *выражение*. В случае отсутствия в **CT-**таблице входа, по шаблону соответствующего выражению, процедурой возвращается ***FAIL-***значение. Модифицировать **CT-**таблицу путем добавления в нее новых *входов* можно посредством ***insertpattern-***процедуры, по которой новые входы помещаются в конец таблицы. При необходимости помещения нового входа в другое место требуется новая компиляция **CT-**таблицы. Следующий простой фрагмент иллюстрирует использование указанных средств ***Maple-***языка для создания таблицы интегралов от простых выражений, содержащей только четыре *входа* (*шаблоны подинтегральных выражений*) и в качестве соответствующих им *выходов* **–** *результаты* интегрирования исходных *шаблонов*-выражений.

```
> T:=([a::algebraic*x::name^(n::integer) = a*x^(n+1)/(n+1), sin(x::name)*cos(x::name)^
  (p::integer) = -cos(x)^(p+1)/(p+1), 1/(a::positive+b::positive*x::name^2) = arctan(b*x/
  (sqrt(a*b)))/(sqrt(a*b)), sin(n::integer*x::name)^m::integer = int(sin(n*x)^m, x)]);
```

$$T := \left[(a::algebraic)\,(x::name)^{(n::integer)} = \frac{a\,x^{(n+1)}}{n+1},\right.$$

$$\sin(x::name)\cos(x::name)^{(p::integer)} = -\frac{\cos(x)^{(p+1)}}{p+1},$$

$$\frac{1}{(a::positive)+(b::positive)\,(x::name)^2} = \frac{\arctan\left(\frac{b\,x}{\sqrt{a\,b}}\right)}{\sqrt{a\,b}},$$

$$\left.\sin((n::integer)\,(x::name))^{(m::integer)} = \int \sin(n\,x)^m\,dx\right]$$

```
> compiletable(T):  map(tablelook, [10*y^3, sin(z)*cos(z)^3, 1/(1 + 2*h^2), sin(10*x)^3], %);
```

$$\left[\frac{5\,y^4}{2}, -\frac{1}{4}\cos(z)^4, \frac{1}{2}\arctan(\sqrt{2}\,h)\,\sqrt{2}, -\frac{1}{30}\sin(10\,x)^2\cos(10\,x) - \frac{1}{15}\cos(10\,x)\right]$$

```
> map(whattype, [T, %%]);  ⇒ [list, function]
```

Из представленного фрагмента достаточно прозрачен сам принцип создания функциональной **CT-**таблицы и последующего ее использования. Рассмотренные средства пакета ***Maple*** обеспечивают простую возможность создания различного рода функциональных таблиц с параметрами и достаточно быстрого их просмотра. При этом, следует иметь в виду, что созданная таким образом *функциональная* таблица не является в строгом понимании ***Maple-***языка структурой данных *table-*типа, как это иллюстрирует последний пример предыдущего фрагмента. Более детально читатель может ознакомиться с принципами организации функциональных таблиц в книгах [80,84,86-88].

При этом, следует отметить**,** что в определенной мере типировать идентификаторы можно и по ***assume-***процедуре, как это иллюстрирует следующий простой пример**:**

```
> assume(A, integer);  assume(B > 0);  frac(A),  sin(A*Pi),  sqrt(-B);  ⇒
```

$$0,\ 0,\ \sqrt{B\sim}\ I$$

Аппарат *шаблонов* ***Maple-***языка представляет собой довольно развитое уникальное средство, позволяющее проводить *структурно-типированный* анализ ***Maple-***выражений, однако он не позволяет *наделять* конструкции языка требуемыми свойствами, подобно тому, как это делает подобный ему механизм математического пакета ***Mathematica*.** В деталях данный вопрос здесь не обсуждается и заинтересованный читатель отсылается к книгам [10-14,29,30,84].

Вместе с тем, (::)-оператор *типирования* поддерживает *механизм* автоматической проверки типов, передаваемых процедуре *значений фактических* аргументов, что при конкретном программировании используется весьма широко. В данном случае определяемые в процедуре *формальные* аргументы по (::)-оператору наделяются соответствующими типами (*типируются*), позволяя при *вызове* процедуры на фактических аргументах проверять их допустимость на заданные типы. В дальнейшем указанные средства тестирования типов будут *широко* использоваться в многочисленных иллюстративных примерах, детализируя их смысловую нагрузку. По конструкции **?{*type*|*typematch*|*whattype*}** можно оперативно получать справочную информацию по {***type, typematch, whattype***} и список всех поддерживаемых ***Maple***-языком типов. Ряд важных особенностей выполнения рассмотренных тестирующих функций можно найти в [12,91], тогда как с введенными нами новыми важными типами можно ознакомиться в [42,103,108,109].

В частности, нами был определен новый *file*-тип, поддерживаемый процедурой:

```
type/file := proc(F::anything)
local a, b, c, k, f, SveGal;
global _datafilestate;
    if not type( F, {'string', 'symbol'} ) then return false
    else c := interface( warnlevel ); null( interface( warnlevel = 0 ) )
    end if;
    SveGal := proc(f)
            try open(f, 'READ'); close(f)
            catch "file or directory does not exist":
                RETURN(false, unassign('_datafilestate'))
            catch "file or directory, %1, does not exist":
                RETURN(false, unassign('_datafilestate'))
            catch "file I/O error": RETURN(false, unassign('_datafilestate'))
            catch "permission denied": RETURN(false, unassign('_datafilestate'))
            end try;
            true, assign67('_datafilestate' = 'close', f)
        end proc;
    if Empty( Red_n(F, " ", 1) ) then
        null( interface( warnlevel = c ) ); ERROR( "argument <%1> is invalid", F)
    else assign67( a = iostatus( ), b = NULL, f = CF(F) ),
        null( interface( warnlevel = c ) )
    end if;
    if nops( a ) = 3 then SveGal(f), null( interface( warnlevel = c ) )
    else
        for k in a[ 4 .. -1 ] do if CF( k[ 2 ] ) = CF(f) then b := b, [ k[ 1 ], k[ 2 ] ] end if
        end do;
        if b = NULL then SveGal(f), null( interface( warnlevel = c ) )
        else true, assign67('_datafilestate' = 'open', b),
            null( interface( warnlevel = c ) )
        end if
    end if
end proc
```

Вызов ***type*(K, *file*)** возвращает *true*, если **K –** реальный файл, иначе *false* возвращается.

Глава 2. *Базовые управляющие структуры Maple-языка пакета*

Для описания *произвольного* вычислительного алгоритма рассмотренных выше конструкций ***Maple***-языка явно недостаточно по причине отсутствия средств по *управлению* вычислительным процессом. Глава и служит целям рассмотрения данных средств ***Maple.***

2.1. *Предварительные сведения общего характера*

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

В задачу структурного кодирования входит получение *корректной программы* (***модуля***) на основе простых *управляющих* структур. В качестве таких *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***.** Более того**,** *первые* из трех указанных управляющих структур (*следования*, *ветвления* и *организации циклов*) составляют тот ***минимальный*** базис, на основе которого можно создавать любой сложности корректную программу с одним *входом*, одним *выходом*, без зацикливаний и недостижимых команд. Детальное обсуждение базисов управляющих структур программирования можно найти, в частности, в книгах [1-3] и в другой доступной литературе по основам программирования.

Следование отражает сам принцип *последовательного* выполнения *предложений* программы, пока не встретится изменяющее эту последовательность предложение. Например: **Avz:=19.4; Ag:=47.52; Sv:=39*Av-6*Ag; Tal:=Art+Kr;** - типичная управляющая структура *следования*, состоящая из последовательности четырех весьма простых ***Maple***-предложений присваивания.

Ветвление определяет выбор одного из возможных путей дальнейших вычислений; типичными предложениями, обеспечивающими данную управляющую структуру, являются предложения типа «**IF A THEN B ELSE C**». Структура «*цикл*» реализует повторное *выполнение* группы предложений, пока выполняется некоторое логическое условие; типичными предложениями, обеспечивающими данную управляющую структуру, являются предложения: **DO, DO_WHILE** и **DO_UNTIL**. Таким образом, базисные структуры определяют соответственно последовательную (*следование*), условную (*ветвление*) и итеративную (*цикл*) передачи управления в программах. При этом, корректная структурированная программа теоретически любой сложности может быть написана с использованием только управляющих структур следования, **IF**-операторов ветвления и **WHILE**-циклов. Однако расширение набора указанных средств, в первую очередь, за счет обеспечения вызовов функций и механизма процедур существенно облегчает программирование, не нарушая при этом структурированности программ и повышая уровень их модульности. При этом, сочетания (*итерации*, *вложения*) корректных структурированных программ, полученные на основе указанных управляющих структур, не нарушают их структурированности и корректности. *Любых* сложности и размера программы можно получать на основе соответствующего сочетания расширенного базиса (*следование*, *ветвление*, *цикл*, *вызовы функций и механизм процедур*) управляющих структур. Такой под-

ход позволяет 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. Именно в свете сказанного ***Maple***-язык представляет собой достаточно хороший пример лингвистического обеспечения при разработке *эффективных структурированных* программ, сочетающего лучшие традиции структурно-модульной технологии с ориентацией на математическую область приложений и массу программистски непрофессиональных пользователей из различных прикладных областей, включая и *не совсем* математической направленности.

Для дальнейшего изложения напомним, что под «*предложением*» в ***Maple***-языке понимается конструкция следующего простого вида:

<*Maple-выражение*> {**;**|**:**}

где в качестве *выражения* допускается любая корректная с точки зрения языка конструкция, например: **A:= sqrt(60 + x): evalb(42 <> 64); sin(x) + x; `Tallinn-2006`:= 6; # Вызов** и др. В рассмотренных ниже иллюстративных фрагментах приведено достаточно много различных примеров относительно несложных предложений в рамках управляющей структуры следования, которая достаточно прозрачна и особых пояснений не требует. *Предложения* кодируются друг за другом, каждое в отдельной строке или в одной строке несколько; завершаются в общем случае {**;**|**:**}-разделителями и выполняются строго последовательно, если управляющие структуры ветвления и цикла не определяют другого порядка. В дальнейшем предложения языка будем называть в соответствии с их определяющим назначением, например предложение *присваивания*, *вызова функции*, *комментария*, **while**-предложение, **restart**-предложение, **if**-предложение и т.д. Сделаем лишь *одно* замечание к предложению присваивания.

Наиболее употребительно определение предложения *присваивания* посредством 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* фактического аргумента допускает и *список/множество* уравнений вида ***Id*** **=** <*Выражение*>, определяющих соответствующие *множественные* присваивания. Например, списочная структура следующего фрагмента допустима лишь с использованием вышеупомянутой ***assign***-процедуры пакета:

```
> [x:= 64, y:= 59*x,  z:=evalf(sqrt(x^2 + y^2)), x + y + z];
Error, `:=` unexpected
> [assign(x= 64), assign(y, 59*x), assign(z, evalf(sqrt(x^2 + y^2))), x+y+z];   ⇒ [7616.542334]
> [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];   ⇒ [6250.639936]
```

Первый пример фрагмента иллюстрирует недопустимость использования (**:=**)-оператора, а три последующих - реализацию этой же списочной структуры на основе процедуры ***assign*.** При этом последний пример фрагмента демонстрирует использование *невычисленных* идентификаторов, обеспечивающих корректность вычислений. Тогда как 3-й пример иллюстрирует ошибочность повторного применения ***assign*** для переопределения *вычисленных* переменных. В дальнейшем ***assign***-процедура широко используется в иллюстративных примерах, а в книге [103] представлен ряд полезных ее расширений.

2.2. *Управляющие структуры ветвления Maple-языка* (**if-*предложение***)

Достаточно сложные алгоритмы вычислений, обработки информации и/или управляющие (*в первую очередь*) не могут обойтись сугубо последовательной схемой, а включают различные конструкции, изменяющие *последовательный* порядок выполнения алгоритма в зависимости от наступления тех или иных условий: циклы, ветвления, условные и безусловные переходы (*такие конструкции иногда называются* ***управляющими***). Для организации управляющих конструкций *ветвящегося* типа ***Maple-***язык располагает довольно эффективным средством, обеспечиваемым **if-***предложением* и имеющим следующие четыре формата кодирования:

(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 булевское выражение, образованное на основе операторов отношения {**<|<=|>| >=|=|<>**}, *логических операторов* {**and, or,not**} и *логических констант* {***true, false, FAIL***}, и возвращающее логическое {***true***|***false***}-значение. *Последовательность предложений* (**ПП**) представляет собой управляющую структуру типа следования, предложения которой завершаются {**;|:**}-разделителем; для последнего предложения **ПП** *кодирование* разделителя необязательно. Во всех форматах, кроме последнего, ключевая фраза **`end if`** определяет закрывающую скобку (*конец*) **if**-предложения и его отсутствие идентифицирует синтаксическую ошибку, вид которой определяется контекстом **if-***предложения*. Каждому **if**-слову должна строго соответствовать своя закрывающая скобка **`end if`**.

Первый формат **if**-предложения несет следующую смысловую нагрузку: если результат вычисления **ЛУ** возвращает ***true***-значение, то 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, то 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.

Второй формат **if**-предложения несет следующую смысловую нагрузку: если результат вычисления **ЛУ** возвращает ***true***-значение, то выполняется указанная за ключевым ***then***-словом **ПП1,** в противном случае выполняется *следующая* за ключевым ***else***-словом **ПП2.** Замечание относительно вывода промежуточных результатов для случая первого формата **if**-предложения сохраняет силу и для второго формата кодирования.

$$\mathbf{R}=\begin{cases}\mathbf{ПП1},\ \textit{если}\ \ \textit{evalb}(\mathbf{ЛУ1})\Rightarrow \textit{true}\\ \mathbf{ПП2},\ \textit{если}\ \ (\textit{evalb}(\mathbf{ЛУ1})\Rightarrow \textit{false})\,\mathbf{and}\,(\textit{evalb}(\mathbf{ЛУ2})\Rightarrow \textit{true})\\ \mathbf{ПП3},\ \textit{если}\ \ (\textit{evalb}(\mathbf{ЛУ1})\Rightarrow \textit{false})\,\mathbf{and}\,(\textit{evalb}(\mathbf{ЛУ2})\Rightarrow \textit{false})\,\mathbf{and}\,(\textit{evalb}(\mathbf{ЛУ3})\Rightarrow \textit{true})\\ \hline\\ \mathbf{ПП n-1},\ \textit{если}\ \ (\forall \mathbf{k}\,|\,\mathbf{k}\le \mathbf{n-2})(\textit{evalb}(\mathbf{ЛУk})\Rightarrow \textit{false})\,\mathbf{and}\,(\textit{evalb}(\mathbf{ЛУ n-1})\Rightarrow \textit{true})\\ \mathbf{ПП n},\ \textit{in other cases}\end{cases}$$

Третий формат **if-**предложения несет смысловую нагрузку**,** легко усматриваемую из *общей* **R-**функции**,** поясняющей принцип выбора выполняемой **ППк** в зависимости от цепочки *истинности* предшествующих ей **ЛУj** (**j = 1 .. k**) в предложении (*см. выше*).

А именно: **if-**предложение *третьего* формата возвращает **R-**результат выполнения **ППк** тогда и только тогда, когда справедливо следующее определяющее соотношение**:**

(∀j | j ≤ k-1) (*evalb*(ЛУj) ⇒ *false*) and (*evalb*(ЛУk) ⇒ *true*)

Данный формат **if-**предложения является наиболее общим и допускает любой уровень вложенности. Ключевое **elif-**слово является сокращением от **"else if"**, что позволяет не увеличивать число закрывающих скобок **`end if`** в случае *вложенности* **if-**предложения.

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

```
> if (64 <> 59)  then  V:= 42: G:= 47: S:= evalf(sqrt(G - V), 6)  end if:  S;   ⇒ 2.23607
> x:= 57: if type(x,'prime') then y:=x^2-99:  z:=evalf(sqrt(y)) else NULL end if;  y+z;  ⇒ y + z
> if (64 = 59) then V:= 42 else G:= 47: S:=evalf(sqrt(57 - G), 6) end if: [V, S];   ⇒ [42, 3.16228]
> if (64 = 59) then V:= 42 elif (64 <= 59) then G:= 47 elif (39 >= 59) then S:= 67 elif (9 = 2)
  then Art:= 17 else  `ArtVGS`  end if;   ⇒ ArtVGS
> AGN:= `if`(64 <> 59, 59, 64):  Ar:= sqrt(621 + `if`(10 < 17, 2, z)^2): [AGN, Ar];   ⇒ [59, 25]
> if (64 = 59) 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  `ArtVGS`  end if;   ⇒ ArtVGS
> V:= 64: G:= 59:  if (V <> G) then S:= 39;  if (V = 64) then Art:= 17;  if (G = 59) then R:= G +
  V end if end if end if;   ⇒ S := 39
> [%, S, Art, R];          ⇒ [123, 39, 17, 123]
> 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)];   ⇒ [39, 59, 59, 64, Z(95), Z(99)]
```

Последний пример фрагмента иллюстрирует использование четвертого формата **if-**предложения для определения *кусочно*-определенной функции. Описание допустимых конструкций в качестве **ЛУ** и **ПП** для **if-**предложения обсуждалось выше, там же приведен целый ряд примеров по его применению. Ниже будет дана его дальнейшая *детализация* в контексте различного назначения иллюстративных фрагментов ***Maple-я***зыка.

Синтаксис ***Maple-я***зыка допускает также **if-**предложения следующего формата**:**

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

ЛУk - *логические условия* и **ПП -** последовательность предложений. Данного формата **if-**конструкция выполняется корректно, но результат возвращается без вывода. Его можно использовать по {%|%%|%%%}-предложению или по *переменной***,** которой он был присвоен заранее, как это иллюстрирует следующий простой фрагмент**:**

```
> V:=64: G:=59: if (V<>G) then if (V=64) then if (G=59) then R:= G+V  end if end if  end if;
> [%, R];   ⇒ [123, 123]
> if (V<>G) then S:=39; if (V=64) then Art:= 17;  if (G=59) then R:=G+V end if end if end if;
                                        S := 39
> [%, S, Art, R];   ⇒ [123, 39, 17, 123]
```

Последний пример фрагмента иллюстрирует существенно более широкую трактовку допустимости **if-**конструкций указанного формата и особенности их выполнения**.** Сле-

дует иметь в виду, что **if-**предложение возвращает {***true***|***false***}-значение даже в случае неопределенного по сути своей результата вычисления логического условия**,** например:

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

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 лежит не в самом **if-**предложении, а в трактовке языком *неопределенности*, которая обсуждалась нами детально в [12].

Предложение **if** представляет собой наиболее типичное средство обеспечения ветвящихся алгоритмов. В этом контексте следует отметить, что *аналогичное* предложение ***Math-***языка пакета ***Mathematica*** [6,7] представляется нам существенно более выразительным, позволяя проще описывать ветвящиеся алгоритмы [33,39,41,42].

Наконец**,** ***Maple-***язык допускает использование и *безусловных переходов* на основе встроенной функции ***goto*****,** кодируемой в виде ***goto*****(**<*Метка*>**).** Данная функция по вполне понятным причинам, обусловленным структурным подходом к программированию, недокументирована. Однако, в целом ряде случаев использование данного средства весьма эффективно, например, при необходимости погрузить в ***Maple***-среду программу, использующую безусловные переходы на основе **goto**-предложения. Типичным примером являются ***Fortran***-программы, широко распространенные в научных приложениях. Из нашего опыта следует отметить, что использование функции ***goto*** существенно упростило погружение в ***Maple*** целого комплекса физико-технических ***Fortran***-программ, использующих ***goto***-конструкции. Между тем, ***goto***-функция имеет смысл только в теле процедуры, обеспечивая в точке *вызова* ***goto*****(**<*Метка*>**)** переход к ***Maple***-предложению, *перед* которым установлена указанная *Метка*. При этом, в качестве *метки* выступает некоторый идентификатор, как это иллюстрирует следующий простой фрагмент:

```
> A:= proc(x)  `if`(x > 64, goto(L1), `if`(x < 59, goto(L2), goto(Fin)));
        L1:  return x^2;
        L2:  return 10*x + 17;
        Fin: NULL
end proc:
> A(350), A(39), A(64), A(10), A(17), A(64), A(59);   ⇒ 122500, 407, 117, 187
```

При использовании вызовов ***goto*****(**<*Метка*>**)** следует иметь в виду, что *Метка* является *глобальной* переменной, что предполагает ее выбор таким образом, чтобы она не пересекалась с переменными текущего сеанса, находящимися *вне* тела процедуры, использующей ***goto***-переходы. Нами была определена процедура ***isplabel*** [41,103], использующая простой подход защиты ***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')**. Это 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*** [103] позволяет в ***Maple***-процедуре работать с *глобальными* переменными, не заботясь о возможности нежелательных последствий такого решения на *глобальном* уровне текущего сеанса пакета. Детальнее вопросы использования ***goto***-функции ***Maple-***языка и средства, сопутствующие ее использованию, рассматриваются в рамках нашей *Библиотеки* [103]. Рассмотрев средства организации ветвления вычислительного алгоритма, переходим к средствам организации *циклических* конструкций в среде языка пакета ***Maple.***

2.3. *Циклические управляющие структуры Maple-языка* (while_do-*предложение*)

Рассмотрев средства организации *ветвления* вычислительного алгоритма, переходим к обсуждению средств организации *циклических* конструкций в среде ***Maple-***языка. Циклическое предложение **while_do** служит для *многократного* вычисления заданного предложения или их последовательности по указанным его *переменным* (*переменным цикла*), принимающим определенное множество значений. При этом, предложение **while_do** имеет *два* основных формата кодирования, наиболее общие из которых имеют следующий принципиальный вид**:**

(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 <B> 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 **while_do-**предложения 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**) возвращает ***true-***значение (*см. прилож.* **3** [12]). В качестве *выражения*, как правило, выступают список, множество и диапазон, тогда как относительно **ЛУ** имеет место силу все сказанное для случая **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 его {**;|:**}-разделителя.

После завершения **while_do-**предложения управление получает *следующее* за ним предложение. Смысл вариантов (**1.b..1.d**) первого формата **while_do-**предложения с учетом сказанного достаточно прозрачен и особых пояснений не требует. Следующий 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 {10, 17, 39, 44, 59, 64}  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=10, k^2=100, k^3=1000|k=17, k^2=289, k^3=4913|k=39, k^2=1521, k^3=59319|k=44,
k^2=1936, k^3=85184|
> for k in (h$h=1 .. 13)  do  printf(`%a|`, k^2)  end do:
1|4|9|16|25|36|49|64|81|100|121|144|169|
> k:= 0: in (h$h=1 .. 99)  while  (not type(k, 'prime'))  do  k:=k+1: printf(`%a|`, k^2)  end do:
1|4|
```

```
> do V:= 64: G:= 59: S:= 39: Ar:= 17: Kr:= 10: Arn:= 44: R:= V + G + S + Ar + Kr + Arn:
  printf(`%s%a`, `R=`, R); break end do:
R=233
> M:= {}: N:= {}: for k in [3, 10, 32, 37, 52, 57] do if type(k/4, 'odd') then break end if;
  M:= `union`(M, {k}); end do: M;      # (1)   ⇒ {3, 10, 32, 37}
> for k in [3, 10, 32, 37, 52, 57] do if type(k/2, 'odd') then next end if; N:= `union`(N, {k});
  end do: N;      # (2)   ⇒ {3, 32, 37, 52, 57}
> 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;    # (4)   ⇒ {R(F(x)), R(G(x)), R(H(x)), R(S(x)), R(Art(x)), R(Kr(x))}
> 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 <= 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:= []:  p:= 0:  while (p <= 10) do p:= p + 1: S:= [op(S), p^3] end do:  S;
                        [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331]
> 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 20.06 do h:= h + k: end do:  h;   ⇒ 12653.340
> 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 из него 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, в результате вычисления которого производится *немедленный* выход из содержащей его конструкции с передачей управления следующему за

ней предложению. 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. Лучше всего различие управляющих значений ***break*** и ***next*** иллюстрируют простые примеры нижней части предыдущего фрагмента. Из них легко заметить, что если ***break-***значение обеспечивает *немедленный* выход из цикла, в котором оно было вычислено, с передачей управления следующему за ним предложению илио *внешнему* относительно его циклу**,** то ***next-****значение* 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**), содержащего ***next-***значение, цикл выполняется полностью, но **N-**множество, в отличие от **M-**множества предыдущего цикла, формируется только из **к-**чисел *списка*, для которых значения **к/2** являются *четными*. Наконец, пример (**3**) фрагмента иллюстрирует совместное использование для управления циклической конструкцией варианта (**1.e**) первого формата **while_do-**предложения управляющих значений ***next*** и ***break*** с целью оценки *производительности* **ПК,** работающего под управлением системы ***DOS+Windows + Maple 8***, в единицах простых операций в с. Для обеспечения возможности последующего использования результатов выполнения циклических конструкций в их телах можно предусматривать аккумуляцию и сохранение результатов**,** получая возможность доступа к ним после завершения (*даже в целом ряде случаев ошибочного или аварийного*) выполнения циклической конструкции.

Наряду с *численными* первый формат **while_do-**предложения позволяет весьма успешно производить и *символьные* вычисления и обработку, так в частности, пример (**4**) предыдущего фрагмента иллюстрирует эквивалентную замену ***map-***функции соответствующей циклической конструкцией и некоторые другие полезные преобразования.

Наиболее общего вида вариант (**2.a**) второго формата **while_do-**предложения несет следующую *смысловую* нагрузку**:** для заданной *переменной цикла* (**ПЦ**), принимающей в качестве значений последовательность значений **A, A+B, A+2*B, ..., A+n*B ≤ C** (*где* **A,B,C -** *результаты вычисления соответствующих выражений*), *циклически* повторяется *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 относительно **ЛУ** имеет место силу *все* сказанное для случая **if-**предложения. Это же относится и к типу завершающего разделителя, используемого для предложения *второго* формата.

В случае отсутствия во *втором* формате **while_do-**предложения *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 этом случае во избежание зацикливания (*бесконечного цикла*) необходимо определять условия *завершения* (*или выхода из*) цикла в **ЛУ** и/или в **ПП.** Минимально допустимым форматом **while_do-**предложения в ***Maple-***языке является конструкция **do ... end do** {**;|:**}**,** определяющая собой *пустой* бесконечный цикл, на основе которого можно программировать не только 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.**

Между тем, стандартные **for_while_do**-конструкции допускают только фиксированный *уровень* вложенности (*вложенные циклы*), тогда как в целом ряде приложений *уровень* вложенности определяется *динамически* в процессе выполнения алгоритма. В состав нашей *Библиотеки* [103] включена процедура ***FOR_DO*,** обеспечивающая работу с *динамически* генерируемыми конструкциями **for_while_do** любого конечного уровня вложенности.

Процедура ***FOR_DO*** обеспечивает динамическую *генерацию* «***for_do***»-конструкций следующих достаточно широко используемых типов, а именно:

(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* [41,103,109]):

***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( ), "/$Art16_Kr9$" ) );
    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
```

```
> ArtKr:= 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],
  "ArtKr[k,j,h,p,v]:=k*j*h*p*v;"); interface(rtablesize = infinity); eval(ArtKr);
```

```
[ 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 ]
```

2.4. *Специальные типы циклических управляющих структур Maple-языка пакета*

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

> add(S(k), k=1..10); ⇒ S(1) + S(2) + S(3) + S(4) + S(5) + S(6) + S(7) + S(8) + S(9) + S(10)
> assign('A'=0); for k from 1 to 5 do A:=A+F(k) end do: A; ⇒ F(1) + F(2) + F(3) + F(4) + F(5)
> mul(F(k), k=1..13); ⇒ 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)
> assign('M'=1); for k from 1 to 7 do M:=M*F(k) end do: M; ⇒ F(1) F(2) F(3) F(4) F(5) F(6) F(7)
> 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)
> assign('S'=[]); for k from 1 to 5 do S:=[op(S),F(k)] end do:op(S); ⇒ F(1), F(2), F(3), F(4), F(5)
> 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)]
> assign('m'=[]); for k in [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10] 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)]
> L:= [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]: member(x3, L); ⇒ *true*
> for k to nops(L) do if L[k]=x10 then R:=true: break else R:=false end if end do: R; ⇒ *true*

В данном фрагменте каждый пример *вызова* функций из рассмотренной группы сопровождается следующим за ним примером**,** представляющим эквивалентную ***Maple-***конструкцию в терминах базовых *управляющих* структур. Читателю в качестве полезного упражнения рекомендуется разобраться в представленных примерах фрагмента.

С другой стороны, указанные функции не только моделируются базовыми *управляющими* структурами, но и сами могут моделировать определенные типы *вторых*, а также допускают использование в точке вызова соответствующих базовых управляющих структур, как это иллюстрирует следующий простой фрагмент**:**

> G:= [59, 64, 39, 44, 10, 17]: add(G[k]*`if`(type(G[k], 'odd'), 1, 0), k=1..nops(G)); ⇒ 115
> mul(G[k]*`if`(type(G[k], 'even'), 1, `if`(G[k]=0, 1, 1/G[k])), k=1..nops(G)); ⇒ 28160
> F:=[10,GS,17]: (seq(F[k]*`if`(type(F[k],'symbol'),1,0),k=1..nops(F)))(x,y,z); ⇒ 0, GS(x,y,z), 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 **if-**структуры для обеспечения *дифференцировки* выбора слагаемых и сомножителей внутри функций ***add*** и ***mul.*** Тогда как третий и четвертый примеры формируют последовательности вызовов функций на основе результатов их тестирования. Данные приемы могут оказаться достаточно полезным средством в практическом программировании в среде языка ***Maple*** различного рода циклических вычислительных конструкций.

В циклических конструкциях типа **`for k in n$n=a..b...`** не допускается отождествления идентификаторов **k** и **n,** не распознаваемого синтаксически, но приводящего к ошиб-

кам выполнения конструкции. Более того, в общем случае *нельзя* отождествлять в *единой* конструкции переменные *цикла* и *суммирования/произведения*, например:

```
> h:= 0:  for k to 180  do  h:= h + sum(1, k = 1 .. 64)  end do;  h;   ⇒ 0
Error, (in sum) summation variable previously assigned, second argument evaluates to 1 = 1 .. 64
> h:= 0:  for k to 180  do  h:= h + sum(1, 'k' = 1 .. 64)  end do:  h;   ⇒ 11520
> h:= 0:  for k to 180  do  h:= h + product(1, k = 1 .. 64)  end do: h;   ⇒ 0
Error, (in product) product variable previously assigned, second argument evaluates to 1 = 1 .. 64
> h:= 0:  for k to 180  do  h:= h + product(2, 'k' = 1 .. 64)  end do: h;
                                  332041393326771929088O
> h:= 0:  for k in  ['k' $ 'k'=1..64]  do  h:= h + k  end do:  h;   ⇒ 2080
> h:= 0:  for k in  [k $ 'k'=1 .. 64]  do  h:= h + k  end do:  h;   ⇒ 4096
> h:= 0:  for k in  [k $ k=1 .. 64]  do  h:= h + k  end do:  h;   ⇒ 0
Error, wrong number (or type) of parameters in function $
```

Вместе с тем, как иллюстрирует фрагмент, корректность выполняется 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. Данное обстоятельство определяется соглашениями ***Maple-***языка по использованию *глобальных* и *локальных* переменных, детально рассматриваемых в следующей главе книги, посвященной процедурным объектам языка.

Дополнительно к сказанному, следует иметь в виду весьма существенное отличие в выполнении ***seq-***функции и логически эквивалентного ей **$-**оператора. Если функция ***seq*** носит достаточно универсальный характер, то **$-**оператор более ограничен, в целом ряде случаев определяя некорректную операцию, что весьма наглядно иллюстрирует следующий простой фрагмент применения обоих средств ***Maple-***языка пакета:

```
> S:="aqwertyuopsdfghjkzxc": 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)));   ⇒ "aqwertyuopsdfghjkzxc"
> X,Y:=99,95: seq(H(k),k=`if`(X<90,42,95)..`if`(Y>89,99,99));  ⇒ H(95),H(96),H(97),H(98),H(99)
```

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

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

При использовании ***`if`-***функции для организации выхода из *циклических* конструкций рекомендуется проявлять внимательность, ибо возвращаемое функцией ***break-***значение не воспринимается в качестве *управляющего* слова ***Maple-***языка пакета, например:

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

Первый пример фрагмента иллюстрирует *успешный* выход из **do-**цикла по достижении **R-**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 **R=0** ***break-***значение, не восприни-

маемое в качестве *управляющего* слова. При этом, если в ***Maple 7–10*** инициируется ошибочная ситуация, то еще в ***Maple 6*** второй пример, не вызывая ошибочной ситуации, выполняет бесконечный цикл, требуя прекращения вычислений по **stop-**кнопке **GUI**. В случае использования вместо ***break*** функций **done, quit** и **stop** выполняется *бесконечный* цикл, требуя прекращения вычислений по **stop-**кнопке **GUI**.

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

{***select*** | ***remove***}**(**<ЛФ>**,** <*Выражение*> {**,** <*Параметры*>}**)**

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

```
> select(issqr, [seq(k, k= 1 .. 350)]);
                [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324]
> ЛФ:= x ->`if`(x >= 42 and x <= 99, true, false): select(ЛФ, [seq(k, k = 1 .. 64)]);
          [42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]
> remove(ЛФ, [seq(k, k = 1 .. 64)]);   ⇒ [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]
```

Приведенный фрагмент достаточно прозрачен и особых пояснений не требует.

В целом ряде случаев работы с данными *индексированного* типа {*array*, *Array и др.*} возникает необходимость динамической генерации вложенных циклических конструкций, когда уровень циклической вложенности заранее неизвестен и вычисляется программно. В этом случае может оказаться весьма полезной процедура ***SEQ*** [41,103]. Процедура обеспечивает динамическую генерацию ***seq***-конструкций следующего типа:

seq**(*****expr*****[k], k = a .. b)** и **seq(*****expr*****[k], k = V)**

Следующий пример иллюстрирует применение ***SEQ***-процедуры:

```
> SG:= Array(1..2, 1..3, 1..4): SEQ([k, 1..2], [j, {1, 2, 3}], [h, 2..3], "assign('SG'[k,j,h]=k+j+h)");
  ArrayElems(SG);
    {(1, 1, 2) = 4, (1, 1, 3) = 5, (1, 2, 2) = 5, (1, 2, 3) = 6, (1, 3, 2) = 6, (1, 3, 3) = 7, (2, 1, 2) = 5,
                    (2, 1, 3) = 6, (2, 2, 2) = 6, (2, 2, 3) = 7, (2, 3, 2) = 7, (2, 3, 3) = 8}
```

Используемый процедурой прием может быть использован для *расширения* других циклических конструкций [41,103]. Детальнее *группа* функциональных средств, объединенных управляющей структурой *циклического* типа, на семантическом уровне рассматривалась выше. Здесь же на нее было акцентировано внимание именно в связи с вопросом механизма организации управляющих структур ***Maple-***языка.

На основе рассмотренных *управляющих* структур *следования*, *ветвления* и *цикла*, поддерживаемых ***Maple-***языком**,** пользователь в сочетании с его функциональными средствами получает достаточно развитый инструмент для создания *собственных* программных средств**,** ориентированных на его конкретные приложения**.** В этом отношении поддерживаемый языком механизм *процедур* позволяет создавать его приложения не только *структурированными* в указанном выше смысле, но и более *модульными*. В следующей главе рассматриваются процедурные и модульные объекты ***Maple***-языка, что позволит с большим пониманием освоить и использовать программную среду пакета для создания приложений в различных областях.

Глава 3. *Организация механизма процедур в Maple-языке пакета*

Сложность в общем случае является достаточно интуитивно-субъективным понятием и его исследование представляет весьма трудную фундаментальную проблему современного естествознания да и познания вообще. Поэтому его использование ниже носит интуитивный характер и будет основываться на субъективных представлениях читателя. За свою историю человечество создала немало весьма сложных проектов и систем в различных областях, к числу которых с *полным* основанием можно отнести и современные *вычислительные системы* (**ВС**) с разрабатываемым для них программным обеспечением (**ПО**)**.** Поэтому обеспечение высокого качества разрабатываемых сложных *программных* проектов представляется не только чрезвычайно важной, но и весьма трудной задачей, носящей многоаспектный характер. В последнее время данному аспекту программной индустрии уделяется особое внимание.

Решение данной задачи можно обеспечивать *двумя основными* путями**:** (**1**) исчерпывающее тестирование готового программного *средства* (**ПС**), устранение всех ошибок и оптимизация его по заданным критериям**;** и (**2**) обеспечение высокого качества на *всех* этапах разработки **ПС.** Так как для большинства достаточно сложных **ПС** первый подход неприемлим, то наиболее реальным является второй**,** при котором вся задача разбивается на *отдельные* объекты (*модули*), имеющие хорошо обозримые структуру и функции, относительно небольшие размеры и сложность и структурно-функциональное объединение (*композиция*) которых позволяет решать исходную задачу. При таком модульном подходе сложность **ПС** редуцируется к существенно *меньшей* сложности составляющих его компонент, каждая из которых выполняет четкие функции, обеспечивающие в совокупности с другими компонентами требуемое функционирование **ПС** в целом. Метод программирования, когда вся программа разбивается на группы *модулей***,** каждый со своей контролируемой структурой, четкими функциями и хорошо определенным интерфейсом с внешней средой, называется *модульным программированием***.**

Поскольку *модульный* является единственной альтернативой *монолитного* (*в виде единой программы*) подхода, то вопрос состоит не в целесообразности разбивать или нет большую программу на ***модули,*** а в том **-** каков должен быть *критерий* такого разбиения. На сегодня 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. Однако наиболее развитым представляется *критерий***,** в *основе* которого лежит *хорошо* известный принцип «*черного ящика*». Данный подход предполагает на стадии проектирования **ПС** представлять его в виде совокупности функционально связанных модулей**,** каждый из которых реализует *одну* из допустимых функций. При этом, способ взаимодействия модулей должен в максимально возможной степени *скрывать* принципы его функционирования и организации. Подобная *модульная* организация приводит к выделению модулей**,** которые характеризуются легко воспринимаемой структурой и могут проектироваться и разрабатываться *различными* проектировщиками и программистами. Более важный аспект состоит в том**,** что многие требуемые модификации сводятся к изменению алгоритмов функционирования отдельных модулей без изменения общей структурно-функциональной организации **ПС** в целом. Вопросы современной концепции ***модульного*** программирования базируются на ряде основных предпосылок, рассматриваемых, например, в книгах [1-3] и в цитируемой в них *весьма* обширной литературе.

3.1. *Определения процедур в Maple-языке и их типы*

Технология *модульного программирования* охватывает *макроуровень* разработки **ПО** и позволяет решать важные задачи программной индустрии. Одним из основных подходов, обеспечивающих модульность программ, является механизм ***процедур***, относительно ***Maple-***языка рассматриваемый нами в настоящей главе книги.

Выделение в большой задаче *локальных подзадач* с достаточно большой частотой использования и применимости позволяет не только существенно продвинуть *вопрос* повышения ее модульности**,** но и 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. Достаточно развитый механизм процедур ***Maple-***языка во многих отношениях отвечает данному решению.

Процедура в среде ***Maple-***языка имеет следующую принципиальную структуру**:**

proc(<*Последовательность формальных аргументов*>**)**{**::***Тип***;**}

local <*Последовательность идентификаторов*>**;**	***Описательная***
global <*Последовательность идентификаторов*>**;**	***часть***
options <*Последовательность параметров*>**;**	***определения***
uses <*Последовательность имен пакетных модулей*>**;**	
description <*Описание*>**;**	***процедуры***

<Т Е Л О процедуры>

end proc {**;**|**:**}

Заголовок процедуры содержит ключевое ***proc-***слово со скобками, в которых кодируется последовательность *формальных аргументов* процедуры**;** данная последовательность может быть и *пустой*, т.е. минимально допустимый *заголовок* процедуры имеет вид **proc( ).** Позади *заголовка* может кодироваться *тип*, относящийся к типу возвращаемого процедурой результата. Подробнее об этом говорится ниже**.** Ключевые слова **local, global** и **options** определяют *необязательные* описательные секции процедуры, представляющие соответственно последовательности *идентификаторов локальных*, *глобальных* переменных и *параметров* (*опций*) процедуры. Необязательная **description-**секция содержит последовательность строк, описывающих процедуру. Если данная секция представлена в определении процедуры, то она будет присутствовать и при выводе последней на печать. Практически все библиотечные процедуры пакета содержат **description-**секцию, хотя она и не выводится на печать. При этом, если данная секция содержит несколько предложений, то она должна кодироваться последней, иначе инициируется ошибочная ситуация, как это иллюстрирует нижеследующий простой фрагмент:

```
> P:= proc() description "Сумма аргументов"; `+`(args); `+`(args) end proc: print(P);
            proc() description "Сумма аргументов"; `+`(args); `+`(args) end proc
> P(42, 47, 67), op(5, eval(P));    ⇒ 156, "Сумма аргументов"
> P:= proc() description "Сумма аргументов"; `+`(args); `+`(args); option trace;  end proc;
Error, reserved word `option` or `options` unexpected
```

В ***Maple 10*** дополнительно к указанным определение процедуры допускает использование необязательной **uses**-секции, определяющей имена пакетных модулей, используе-

мых процедурой. Например, кодирование секции «**uses LinearAlgebra**» эквивалентно использованию в процедуре **use**-предложения следующего формата:

use LinearAlgebra in < ... > end use;

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

> P:= proc() local a,b,c; global d; uses LinearAlgebra; option remember; description "grsu"; BODY end proc;

P := **proc**() **local** a, b, c; **global** d; **option** remember; **description** "grsu"; *BODY* **end proc**

И не возвращается ее содержимое по вызовам ***op*(k, *eval*(P))** (**k=1..8**) подобно случая других секций описания произвольной процедуры **P.**

На наш взгляд, секция **uses** носит непринципиальный характер, в определенной мере лишь упрощая (*сокращая*) оформление процедуры, тогда как с точки зрения читабельности исходного текста использование в теле **use**-предложений даже более предпочтительно. Не говоря уже о том, что данное непринципиальное новшество ведет к несовместимости процедур *«сверху-вниз»* относительно релизов пакета.

Секции **local, global, options, uses** и **description** составляют описательную часть определения ***Maple*-**процедуры, которая в общем случае может и отсутствовать. Управлять выводом тела процедуры можно посредством установки переменной ***verboseproc interface*-**процедуры ***Maple*-**языка. Минимальной конструкцией, распознаваемой ядром в качестве процедуры**,** является структура следующего весьма простого вида, а именно**:**

Proc* := proc() end proc:** ***whattype*(*eval*(*Proc*));** ⇒ ***procedure

Распознаваемый пакетом ***Proc***-объект в качестве процедуры, между тем, особого смысла не имеет, ибо вызов процедуры ***Proc*(*args*)** на любом кортеже фактических аргументов всегда возвращает ***NULL***-значение, т.е. ничего. Для возможности вызова процедуры ее определение присваивается некоторому ***Id*-**идентификатору ***Id* := proc({ |*Args*}) ... end proc {:|;}**, позволяя после его вычисления производить вызов процедуры по ***Id*({|*Args*})-**конструкции с заменой ее *формальных* ***Args*-**аргументов на *фактические* ***Args*-**аргументы. Наряду с классическим определением именованной процедуры ***Maple*-**язык допускает использование и непоименованных процедур, определения которых не присваиваются какому-либо идентификатору. Для вызова *непоименованных* процедур служит конструкция следующего вида**:**

proc({ |*Args*}) ... end proc(*Args*)

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

> proc() [nargs, args] end proc(42, 47, 67, 89, 96, -2, 95, 99); ⇒ [8, 42, 47, 67, 89, 96, -2, 95, 99]
> proc(n) sum(args[k], k=2..nargs)^n end proc(3, x, y, z, a, b, c, h); ⇒ $(x+y+z+a+b+c+h)^3$
> D(proc(y) (y^9 + 3*y^5 - 99)/6*(3*y^2 - 256) end proc)(9); ⇒ 2648753541
> a*proc(x) x^2 end proc(42) + b*proc(y) y^2 end proc(47); ⇒ 1764 a + 2209 b
> D(proc(y) (y^9 + 2*y^5 - 99)/13*(3*y^2 - 256) end proc);
proc(y) 1/13*(9*y^8+10*y^4)*(3*y^2-256)+6/13*(y^9+2*y^5-99)*y **end proc**
> restart: (D@@9)(proc(x) G(x) end proc); ⇒ **proc**(x) (D@@9)(G)(x) **end proc**

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***}, а также с другими функциями и операторами, допускающими *неалгебраические* выражения в качестве своих фактических аргументов и операндов. В то

же время, как правило, рекомендуется использовать *именно именованные* ***Maple-***процедуры, что позволяет исключать различного рода ошибки.

Тело процедуры содержит *текст* описания алгоритма решаемой ею задачи с использованием рассмотренных выше средств ***Maple-***языка, а именно**:** данных и их структур *допустимых* типов, переменных, функций пакетных, модульных и пользовательских, других процедур и т. д., логически связанных управляющими структурами *следования*, *ветвления* и *цикла*, рассмотренными выше. Данное описание должно удовлетворять правилам допускаемого языком синтаксиса. *Завершается* процедура кодированием закрывающей процедурной скобки **`end proc`**. Представленная структура является ***определением*** процедуры, которое активизируется только после его вычисления. При этом, во время вычисления процедуры не производится выполнения предложений ***тела*** процедуры, но интерпретатор ***Maple-***языка производит все возможные упрощения тела процедуры, являющиеся простым типом ее оптимизации. Реальное же выполнение тела процедуры производится только в момент ее вызова с передачей ей *фактических* аргументов, значения которых замещают все вхождения в тело формальных аргументов. Завершается *определение* процедуры обязательным предложением **`end proc {:|;}`** (*закрывающей скобкой*).

Вторым способом определения процедур является использование *функционального* оператора (**->**), позволяющего представлять процедуры в нотации классической операции отображния, а именно в следующем простом виде**:**

(*Args*) -> <*Выражение от* *Args-переменных*>

При этом, в случае *единственного* аргумента кодирование его в скобках не обязательно. Присвоение данной конструкции идентификатору позволяет определить процедуру, например**: > G:= (x, y) -> *evalf*(*sqrt*(x^2 + y^2)): G(42, 47);** ⇒ 63.03173804. Последовательность *формальных* аргументов в таким образом определяемой процедуре может быть пустой**,** а ее тело должно быть единым выражением либо **if-**предложением языка:

```
> W:= () -> `if`(member(float, map(whattype, {args})), print([nargs, [args]]), NULL):
> W(42, 47, 19.98, 67, 96, 89, 10,3, `TRG`);   ⇒ [9, [42, 47, 19.98, 67, 96, 89, 10, 3, TRG]]
> H:= x -> if (x < 42) then 10 elif (x >= 42)and(x <= 99) then 17 else infinity end if:
> map(H, [39, 49, 62, 67, 95, 98, 99, 39, 17, 10, 39]);   ⇒ [10, 17, 17, 17, 17, 17, 17, 10, 10, 10, 10]
> [D(x -> sin(x))(x), (x -> sqrt(x^3 + 600))(10)];        ⇒ [cos(x), 40]
> map((n) -> `if`(type(n/4, 'integer'), n, NULL), [seq(k, k= 42..64)])  ⇒ [44, 48, 52, 56, 60, 64]
> Z:= () -> if sum(args[k], k= 1 .. nargs) > 9 then args else false end if:
> [Z(3, 6, 8, 34, 12, 99), Z(3, -6, 8, -21, 6, 10)];   ⇒ [3, 6, 8, 34, 12, 99, false]
```

При этом, следует иметь в виду, что *второй* способ определения предназначен, прежде всего, для простых однострочных процедур и функций, ибо не поддерживает механизма *локальных* и *глобальных* переменных, а также *опций*. Между тем, в целом ряде случаев он оказывается весьма полезным приемом программирования. Подобно *первому* способу определения процедур, *второй* также допускает использование *непоименованных процедур*, используемых в тех же случаях, что и первый способ. Как это иллюстрируют последние примеры предыдущего фрагмента (*см. прилож.* **3** [12]).

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

***define*(F, F(x_1::<*Тип*_1>, ..., x_n::<*Тип*_n>) = <*Выражение*>(x_1, ..., x_n))**

По этому формату ***define-****процедура* в качестве **F-**процедуры/функции от типированных **(x_1, ..., x_n)-**аргументов/переменных определяет *выражение* от этих же ведущих переменных. При этом, допускается использование не только *типов*, идентифицируемых ***type-***функцией, но и *структурных* типов. Фактические аргументы, определяемые их *типиро-*

ванными формальными аргументами, могут принимать *любые* совместимые с указанными для них *типами* значения. Успешный вызов процедуры ***define*** возвращает *NULL*-значение. Вызов определенной таким образом процедуры/функции производится по конструкции $\mathbf{F(x_1,..., x_n)}$. Следующий фрагмент иллюстрирует применение ***define*-**процедуры для определения пользовательских процедур**:**

```
> 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, `/x1`::integer), `/BIND`(2, 1, `/x2`::anything),
         `/BIND`(3, 1, `/x3`::fraction),
         `/PATTERN`((`/x1`^3 + `/x2`^3 + `/x3`^3)^(1/2))]');
     if look ≠ FAIL then eval(look, `/FUNCNAME`= procname)
     else 'procname'(theArgs)
     end if
 end proc
> [G(42, 64., 19/99), G(42, 64., 1999), G(42., 64, 350/65), G(`REA`, `13.09.06`)];
                  [579.8551604, G(42, 64., 1999), G(42., 64, 70/13), G(REA, 13.09.06)]
> define(S, S(x, y::string, z::integer) = cat(x, "", y, "", z)); S(RANS, " - 13 сентября ", 2006);
                                S(RANS, " - 13 сентября ", 2006)
> proc(x, y::string, z::integer) cat(x, "", y, "", z) end (RANS, " - 13 сентября ", 2006);
                                RANS - 13 сентября 2006
> S1:= (x, y, z) -> cat(x, "", y, "", z): S1(RANS, " - 13 сентября ", 2006);
                                RANS - 13 сентября 2006
```

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

Подобным *третьему* способом определения *пользовательской* функции является применение специальной ***unapply*-**процедуры, имеющей следующий формат кодирования**:**

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

где *выражение* определяет собственно тело самой функции**,** а *ведущие переменные* **–** последовательность *ее* формальных аргументов. В результате своего вызова ***unapply*-**процедура возвращает рассмотренную выше *функциональную* конструкцию в терминах (**->**)-оператора, как это иллюстрирует следующий весьма простой фрагмент**:**

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

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

```
> evalf(F(6.4, 5.9, 3.9)),  F(m, n, p), evalf(F(Pi/2, Pi/4, Pi));
```

$$0.2946229694\ ,\ \frac{\gamma\, m^2 + \sqrt{m\, n\, p} + m\, \sin(n)}{\pi\, n^2 + \ln(m + p) + \tan(n\, p)}\ ,\ 1.674847463$$

```
> W:=[unapply(sin(x),x), unapply(x,x), unapply(x^3,x)]; W(6.4);
```

$$W := [\sin,\ x \to x,\ x \to x^3]$$

$$[0.1165492049,\ 6.4,\ 262.144]$$

Реализация ***unapply*-**процедуры базируется на использовании **λ-***исчисления*, а сама процедура применяется, как правило, при использовании вычисляемых выражений для функциональных конструкций. Эта же процедура позволяет достаточно эффективно производить *функциональные* определения и в рамках структур (*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 и имеются существенные различия.

При необходимости определения функции с *неопределенным* числом формальных аргументов в общем случае второй аргумент ***unapply*-**процедуры (*последовательность ведущих переменных*) может не кодироваться, как это иллюстрирует следующий фрагмент**:**

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

$$SV := (\ ) \to \left[\ \text{nargs},\ \frac{\sqrt{\sum_{k=1}^{\text{nargs}} \text{args}_k^{\,2}}}{\text{nargs}}\ \right]$$

```
> [6*SV(64,39,59,44,10,17), evalf(SV(64,39,59,44,10,17), 6)];  ⇒
```

$$[\,[\,36,\ \sqrt{11423}\,],\ [\,6.,\ 17.8130\,]\,]$$

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

$$VS := (\ ) \to \text{evalf}\left(\left[\ \text{nargs},\ \frac{\sqrt{\sum_{'k'=1}^{\text{nargs}} \text{args}_{'k'}^{\,2}}}{\text{nargs}}\ \right],\ 6\right)$$

```
> k:= 'k': [VS(64,39,59,44,10,17), evalf(VS(64,39,59,44,10,17), 6)];  ⇒ [6., 17.8130], [6., 17.8130]]
```

Первый пример фрагмента представляет ***SV*-**функцию от неопределенного числа формальных аргументов, определенную на основе ***unapply*-**процедуры, а второй **- *VS*-**функцию, определенную на основе функционального (**->**)-оператора и *эквивалентную* (*по реализованному алгоритму*) ***SV*-**функции. Вместе с тем, как иллюстрируют примеры фрагмента, между обоими определениями имеется существенное различие, а именно**:** игнорируется ряд встроенных функций (***evalf***, ***convert***) при определении ***SV*-**функции**.** Поэтому, в общем случае способ определения функции на основе *функционального* (**->**)-оператора является более универсальным.

Следует отметить, что представленный способ определения *пользовательской* функции на основе ***define-***процедуры по целому ряду характеристик предпочтительнее способа, базирующегося на функциональном (**->**)-операторе или ***unapply-***процедуре, как это иллюстрирует следующий весьма поучительный фрагмент**:**

```
> R:=[42,47,67,62,89,96]: S:=[64,59,39,44,17,10]: define(V,V(x::integer)=interp(R,S,x)); V(95);
                                        11
> map(GS, [42, 47, 67, 62, 89, 96]);   ⇒ [64, 59, 39, 44, 17, 10]
> define(H, H(x::anything, y::integer) = sqrt(x^2 + y^2));
> H(sqrt(75), 5), evala(H(sqrt(75), 5)), H(sqrt(75), 59.47);   ⇒ sqrt(100), 10, H(5 sqrt(3), 59.47)
> 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, 1999, 71, 13, 17, 7)), evala(DD(64, 59, 39, 44, 17, 10));
                                sqrt(4001558), DD(64, 59, 39, 44, 17, 10)
> restart;  define(R, R(x::even, y::odd) = sqrt(x^2 + y^2)),  R(10, 17)^2;   ⇒ 389
> define(R, R(x::even, y::odd) = sqrt(x^3 + y^3)), R(10, 17);
Error, (in DefineTools:-define) R is assigned
> R:= 'R':  define(R, R(x::even, y::odd) = sqrt(x^3 + y^3)), R(10, 17)^2;   ⇒ 5913
> 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, [10, 17, 15, 18, 20]);   ⇒ [1596, 336608, 72962, 723000, 3335539]
> define(G3, G3(seq(x||k::integer, k=1 .. 6)) = sum(args[k], k=1 .. nargs)/nargs);
> 3*G3(10, 17, 39, 44, 95, 99);   ⇒ 152
> define(G4, G4()=sum(args[k], 'k'=1..nargs)/nargs):  3*G4(10, 17, 39, 44, 95, 99);   ⇒ 152
> define(G6, G6() = [nargs, args]):
Error, (in insertpattern) Wrong kind of arguments
> G7:= () -> [nargs, args]:  G7(10, 17, 39, 44, 95, 99);   ⇒ [6, 10, 17, 39, 44, 95, 99]
> define(G8, G8() = args), define(G9, G9() = nargs), G8(10, 17), G9(10, 17, 39, 44);   ⇒ 10, 4
> define(S1, define(S2, S2()= sum(args[k], 'k'=1..nargs)), S1() = S2(V, G, S)*nargs);
> S1(10, 17, 39, 44, 95, 99), S2(10, 17, 39, 44, 95, 99);   ⇒ 6 V + 6 G + 6 S, 304
```

Фрагмент иллюстрирует реакцию пакета на переопределения функции**,** определенной на основе ***define-***подхода, и вызовы пользовательской ***H-***функции от двух *типированных* аргументов. На примере ***H-***функции проиллюстрированы следующие два весьма принципиальных момент, а именно**:**

(**1**) результат вызова ***define-***определенной функции в общем случае требует *последующей* обработки ***evala-***функцией для получения окончательного результата**;**

(**2**) вызов ***define-***определенной функции на фактических аргументах, *не отвечающих* определенным для них типам, возвращается *невычисленным***.**

Использование последовательности свойств позволило определить *целочисленную рекуррентную* ***AG-***функцию. Пример определения ***G3-***функции иллюстрирует возможность использования специальных переменных ***args***, ***nargs*** в ***define-***определении функции, тогда как пример ***G4-***функции дополнительно иллюстрирует определение функции от *неопределенного* числа аргументов. Однако, при этом следует иметь в виду, что использование указанных переменных в ***define-***определении функций носит более *ограниченный* характер, чем при *других* способах определения функций пользователя. Более того, как показывают примеры определения двух *эквивалентных* функций ***G6*** и ***G7***, в плане представимости типов выражений, используемых при определении функции, функциональный (**->**)-оператор более предпочтителен. Примеры определения функций ***G8*** и ***G9*** также иллюстрируют ограничения ***define-***способа задания функций. В этом отношении следует отметить, что реализация ***define-***функции *более* младших релизов ***Maple*** во многих отношениях была *более* эффективной [8-10]. Наконец, последний пример фрагмен-

та иллюстрирует (*в ряде случаев весьма полезную*) возможность рекурсивного использования функции ***define*** для определения функций пользователя.

Использование ***assign*-**процедуры для присвоения определения функции некоторому идентификатору (ее *имени*) позволяет включать определения функций *непосредственно* в вычислительные конструкции, соблюдая *только* одно правило**:** вычисление определения функции должно предшествовать ее первому вызову в вычисляемом выражении. Сказанное относится к любому способу определения функции, например**:**

```
> 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),
  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(10, 17);   ⇒ [2, 27]
```

В частности, последний пример фрагмента иллюстрирует вычисление *списочной* структуры, содержащей ***unapply*-**определение ***Kr*-**функции, с последующим ее вызовом. Следует отметить при этом, что механизм *пользовательских* функций, поддерживаемый пакетом ***Mathemaica*** [6, 7], представляется нам существенно более гибким при *реализации* алгоритмов обработки.

При этом следует иметь ввиду, что в случае определения процедуры в ***assign*-**конструкции ее определение становится глобальным, если ее имя не определено в **local**-секции содержащей конструкцию процедуры, как это иллюстрирует следующий фрагмент:

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

Еще об одном ухищрении стоит сказать особо. В релизах **6** и ниже по конструкции вида

m(assign('a' = <*Выражение***>),** где **m –** произвольное целое либо *NULL*

можно было *широко* использовать ***assign*-**конструкции в выражениях, ибо возвращалось значение **m** с вычислением выражений в скобках, например:

```
> m:=10: (m(assign('a'=2006)) + a - 16)/(6(assign('b'=1995)) + b - 1);   ⇒ 1          # Maple 6
```

Тогда как, начиная с релиза **7,** данная возможность была исключена, например:

```
> m:=10: (m(assign('a'=2006)) + a - 16)/(6(assign('b'=1995)) + b - 1);          # Maple 7 – 10
```

$$\frac{-6+a}{5+b}$$

Что породило еще один тип несовместимости релизов пакета «*снизу-вверх*» для тех, кто пытался использовать особенности ***Maple*-**языка при программировании своих задач.

Рассмотрев способы определения процедур в ***Maple*-**языке и их структурную организацию**,** обсудим более детально отдельные компоненты структуры процедур, определяемых *первым* способом, как наиболее универсальным и часто используемым. Это тем более актуально, что позволит вам не только понимать реализацию пакетных процедур и процедур нашей *Библиотеки* [41] (*некоторые из них представлены и в настоящей книге*), но и самому приступать к созданию собственных конкретных приложений в среде ***Maple*,** реализованных в форме процедур и их библиотек.

3.2. *Формальные и фактические аргументы Maple-процедуры*

Формальный аргумент процедуры в общем случае имеет вид **<*Id*>::<*Tun*>,** т. е. ***Id*-**идентификатор с приписанным ему *типом*, который не является обязательным. В случае определения типированного формального аргумента при передаче процедуре в момент ее вызова *фактического* аргумента, последний проверяется на соответствие *типу формального* аргумента. При *несовпадении* типов идентифицируется ошибочная ситуация с возвратом соответствующей диагностики. Совершенно иная ситуация имеет место при несовпадении числа передаваемых процедуре *фактических* аргументов числу ее *формальных* аргументов**:** (**1**) в случае числа фактических аргументов, меньшего определенного для процедуры числа формальных аргументов, как правило, идентифицируется ошибочная ситуация типа **"***Error*, (*in* ***Proc***) *Proc uses* a ***n****th argument* **<*Id*>**, *which is missing*", указывающая на то, что ***Proc*-**процедуре было передано меньшее число фактических аргументов, чем имеется формальных аргументов в ее определении; где ***Id -*** *идентификатор первого* недостающего **n-**го фактического аргумента**;** (**2**) в случае числа фактических аргументов, *большего* определенного для процедуры числа формальных аргументов, ошибочной ситуации не идентифицируется и лишние аргументы игнорируются. Между тем, и в первом случае возможен корректный вызов. Это будет в том случае, когда в теле процедуры не используются *формальные* аргументы *явно*. Следующий простой фрагмент хорошо иллюстрирует вышесказанное**:**

```
> Proc:=proc(a, b, c) nargs, [args]  end proc: Proc(5, 6, 7, 8, 9, 10), Proc(5), Proc(), Proc(5, 6, 7);
                                  6, [5, 6, 7, 8, 9, 10], 1, [5], 0, [], 3, [5, 6, 7]
> Proc:= proc(a, b, c)  a*b*c  end proc:  Proc(5, 6, 7), Proc(5, 6, 7, 8, 9, 10);   ⇒ 210, 210
> Proc(645);
Error, (in Proc) Proc uses a 2nd argument, b, which is missing
> AVZ:= proc(x::integer, y, z::float)  evalf(sqrt(x^3 + y^3)/(x^2 + y^2)*z)  end proc:
> AVZ(20.06, 59, 64);
Error, AVZ expects its 1st argument, x, to be of type integer, but received 20.06
> AVZ(2006, 456);
Error, (in AVZ) AVZ uses a 3rd argument, z (of type float), which is missing
> [AVZ(64, 42, 19.42), AVZ(59, 42, 19.42, 78, 52)];   ⇒ [1.921636024, 1.957352295]
```

В момент вызова процедуры с передачей ей фактических выражений для ее соответствующих *формальных* аргументов *первые* предварительно вычисляются и их значения передаются в *тело* процедуры для замещения соответствующих им формальных аргументов, после чего производится *вычисление* составляющих тело ***Maple*-**предложений с возвратом значения последнего вычисленного предложения, если не было указано 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 несовпадения инициируется ошибочная ситуация, в противном случае выполнение процедуры продолжается. В качестве *типов* формальных аргументов процедуры используются любые из допустимых языком и тестируемых функцией ***type*** и процедурой ***whattype.*** При использования *нетипированного* формального аргумента рекомендуется все же указывать для него *anything*-тип, информируя других пользователей процедуры о том, что для данного формального аргумента допускаются значения любого типа, например, кодированием заголовка процедуры в виде ***proc*(X::*integer*, Y::*anything*).**

В качестве *формальных аргументов* могут выступать последовательности допустимых выражений ***Maple***, типированных переменных, либо пустая последовательность. Типированная переменная кодируется в следующем формате: <*Переменная*>**::**<*Тип*>. Тип может быть как простым, так и сложным. При обнаружении в точке вызова процедуры *фактического* аргумента, типом отличающегося от определенного для соответствующего ему *формальн*ого аргумента, возникает ошибочная ситуация с возвратом через ***lasterror***-переменную соответствующей диагностики и с выводом ее в текущий сеанс, например:

```
> A:=proc(a::integer, b::float) a*b end proc:  A(64, 42);
Error, invalid input: A expects its 2nd argument, b, to be of type float, but received 42
> lasterror;
"invalid input: %1 expects its %-2 argument, %3, to be of type %4, but received %5"
```

Использование типированных формальных аргументов дает возможность контролировать на *допустимость* передаваемые процедуре фактические аргументы, однако *данный* подход не совсем удобен при решении вопроса устойчивости процедур. С этой целью рекомендуеся обеспечивать *проверку* получаемых процедурой *фактических аргументов* в теле самой процедуры и производить соответствующую программную обработку недопустимых фактических аргументов. В качестве простого примера модифицируем предыдущий фрагмент следующим очевидным образом:

```
> A1:=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:=float(b) end if;  a*b  end proc:
  A1(64, 42), 17*A1(10/17, 59);   ⇒  2688, 590
```

Большинство процедур нашей *Библиотеки* [103] использует именно подобный программный подход к обработке получаемых фактических аргументов на их допустимость и, по возможности, производятся допустимые корректировки. Это существенно повышает устойчивость процедур относительно некорректных фактических аргументов.

При организации процедур роль типированных формальных аргументов не ограничивается только задачами проверки входной информации, но несет и ряд других важных нагрузок. В частности, использование *uneval*-типа для формального аргумента позволяет вне процедуры проводить его модификацию (*т.е. обновлять на «месте» **Maple**-объект, определенный вне тела процедуры под этим идентификатором*) как это иллюстрирует следующий фрагмент:

```
> A:= proc(L::list, a::anything)  assign('L' = subs(a = NULL, L)) end proc:
> L:= [64, 59, 39, 44, 10, 17];  A(L, 64);
                                    L := [64, 59, 39, 44, 10, 17]
Error, (in assign) invalid arguments
 A1 := proc(L::uneval, a::anything)
      if not type(L, 'symbol') then
           error "1st argument should be symbol but had received %1",whattype(L)
      elif type(eval(L), {'list', 'set'}) then assign('L' = subs(
           [`if`( not type(a, {'list', 'set'}), a = NULL, seq(k = NULL, k = a))], eval(L)))
      else error "1st argument should has type {list, set} but had received %1-type"
           whattype(eval(L))
      end if
 end proc
> A1(L, 64), L, A1(L, 59), L, A1(L, {59, 39, 44, 10, 17}), L;
                                [59, 39, 44, 10, 17], [39, 44, 10, 17], []
```

```
> A1(AVZ, 64), AVZ;
Error, (in A1) 1st argument must has type {list, set} but had received symbol-type
> A1([1, 2, 3, 4, 5, 6], 64);
Error, (in A1) 1st argument must be symbol but had received list
```

Попытка определить такую операцию для стандартно типированного **L**-аргумента в ***A***-процедуре вызывает ошибку выполнения, тогда как, определив этот же **L**-аргумент как аргумент *uneval*-типа и использовав в дальнейшем обращение к нему через ***eval***-функцию, получаем вполне корректную ***A1***-процедуру, обеспечивающую *обновление «на месте»* 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 нашей *Библиотеки* [41,103,108,109].

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:
> P(a, b), P(10, 17, 5), P(10, 17, 5, 2);
```

$$\frac{a+b}{\sin(x)+\cos(x)}, \frac{27}{5+\cos(x)}, \frac{27}{7}$$

```
> P1:= proc(a, b, c, d) (a+b)/`if`(nargs = 2, (sin(x) + cos(x)), `if`(nargs = 3, (c+cos(x)), (c+d)))
  end proc:
> P1(a, b), P(10, 17, 5), P(10, 17, 5, 2);
```

$$\frac{a+b}{\sin(x)+\cos(x)}, \frac{27}{5+\cos(x)}, \frac{27}{7}$$

При этом, определенные таким способом позиционные аргументы должны быть *последними* в списке формальных аргументов, иначе инициируется ошибочная ситуация как это иллюстрирует первый пример фрагмента. Этого можно избежать, кодируя их в фигурных скобках, как иллюстрирует следующий простой пример:

```
> P:=proc(f::symbol, x::symbol, {b:= evalf}, c::range(posint)) b(int(f(x), x=c)) end proc;
      P := proc(f::symbol, x::symbol, c::range(posint), {b := evalf}) b(int(f(x), x = c)) end proc
> P(sin, y, 1..5, b=G), P(sin, y, 1..5, G), P(sin, y, 1..5, evalf), P(sin, y, 1..5, b);
            G(cos(1) - cos(5)), 0.2566401204, 0.2566401204, true(cos(1) - cos(5))
```

Из него, в частности, следует, что использование в вызове процедуры такого аргумента в виде уравнения либо значения по *умолчанию* приводит к требуемым результатам, тогда как вызов только на левой части вместо фактического аргумента производит подстановку вместо фактического аргумента *true*-значения. В определенной мере данный механизм и его расширения в ряде случаев оказываются довольно полезными, однако вызывают несовместимость с более ранними релизами. Между тем, этот механизм в целом ряде случаев несложно реализуется программно и прежними средствами ***Maple***-языка. Например, второй пример предпоследнего фрагмента может быть реализован *способом*,

представленным последним примером фрагмента. Естественно, при большем количестве позиционных параметров программирование усложняется и представленный механизм более эффективен, одно ограничение – его действие ограничивается последними формальными аргументами.

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

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

```
P := proc(x, y, z)
    if type( eval(y), 'symbol' ) then return procname(x, 2006, z) end if; x + y + z
end proc
> P(a, b, c), P(a, x-y, c), P(sin(x), h, cos(y));  ⇒ a + 2006 + c, a + x - y + c, sin(x) + 2006 + cos(y)
```

Читателю рекомендуется разобраться с используемым практически полезным *приемом*.

Для организации процедуры наряду с *предложениями*, описывающими непосредственный алгоритм решаемой задачи (*а в ряде случаев и для обеспечения самого алгоритма*), язык ***Maple*** предоставляет ряд важных средств, обеспечивающих функции, *управляющие* выполнением процедуры. Прежде всего, к ним можно отнести переменные ***args*** и ***nargs***, возвращающие соответственно *последовательность* переданных процедуре *фактических аргументов* и их *количество*. Оба эти средства имеют смысл только в рамках процедуры, а по конструкциям вида ***args***{||**[n]**||**[n..m]**} можно получать {*последовательность фактических аргументов*|**n**-*й аргумент*| *аргументы с* **n**-*го по* **m**-*й включительно*} соответственно. Тогда как ***nargs***-переменная возвращает количество полученных процедурой *фактических* аргументов. Назначение данных средств достаточно прозрачно и обусловливает целый ряд их важных приложений при разработке пользовательских процедур. В первую очередь, это относится к обработке получаемых процедурой *фактических* аргументов. В частности, ***nargs***-переменная необходима с целью обеспечения определенности выполнения вычислений в случае передачи 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:
> 137*SV(42, 47, 62, 67, 89, 96, 350, 39, 44, 59, 64);
                                  22698342960272179200
> GN:= proc() [nargs, [args]] end proc:  GN(V, G, S, A, Art, Kr);
                                  [6, [V, G, S, A, Art, Kr]]
> map(whattype, [59, 17/10, ln(x), 9.9, "RANS"]);  ⇒ [integer, fraction, function, float, string]
> Arg_Type:= proc() map(whattype, [seq(args[k], k= 1 .. nargs)]) end proc:
> Arg_Type(59, 17/10, ln(x), 9.9, "RANS");  ⇒ [integer, fraction, function, float, string]
> Arg_Type:= proc() map(whattype, [args[k]$k= 1 .. nargs]) end proc:
> Arg_Type(59, 17/10, ln(x), 9.9, "RANS");  ⇒ [integer, fraction, function, float, string]
```

Приведенный фрагмент достаточно прозрачен и особых пояснений не требует. Более того, как иллюстрирует уже первый пример, *число* передаваемых процедуре фактических аргументов не обязательно должно соответствовать числу ее формальных аргументов. Данный пример иллюстрирует, что в общем случае ***Maple***-процедуру можно опре-

делять, не привязываясь к конкретному списку ее формальных аргументов, но определять формальной функциональной конструкцией следующего общего вида:

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

что оказывается весьма удобным механизмом для организации процедур, ориентированных, в первую очередь, на задачи символьных вычислений и обработки [9-14,39].

Дополнительно к переменным ***args*** и ***nargs*** можно отметить еще одну важную переменную ***procname,*** возвращающую имя процедуры, ее содержащей. В целом ряде случаев данная переменная оказывается весьма полезной, в частности, при возвращении вызова процедуры *невычисленным*. С этой целью используется конструкция простого формата '**procname(*args*)**'. Многие пакетные процедуры возвращают результат именно в таком виде, если не могут решить задачу. Ряд процедур и нашей *Библиотеки* [103] поступают аналогичным образом. Между тем, переменная ***procname*** может использоваться и в других полезных приложениях. Следующий фрагмент иллюстрирует применение указанной переменной как для организации возврата вызова процедуры невычисленным, так и для вывода соответствующего сообщения:

```
AVZ := proc()
    if nargs ≤ 6 then
        WARNING("%1(%2) = %3", procname, seqstr(args), `+`(args)/nargs)
    else 'procname(args)'
    end if
end proc
> AVZ(64, 59, 39, 44, 10, 17);
Warning, AVZ(64, 59, 39, 44, 10, 17)=233/6
> AVZ(64, 59, 39, 44, 10, 17, 6);   ⇒ AVZ(64, 59, 39, 44, 10, 17, 6)
```

Процедура ***AVZ*** при получении не более **6** фактических аргументов выводит соответствующее сообщение, описывающее вызов процедуры и его результат, тогда как в противном случае вызов процедуры возвращается *невычисленным*.

Наконец, для *проверки* допустимости передаваемых процедуре *необязательных* аргументов служит процедура ***DefOpt*(*args*),** где ***args*** – передаваемые аргументы [41-43,103].

```
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 *Библиотеки*, прилагаемой к книгам [41,103] и к настоящей, предоставляют хороший иллюстративный материал по использованию аргументов, переменных процедуры ***args, nargs*** и ***procname*** для разработки различных приложений.

3.3. *Локальные и глобальные переменные Maple-процедуры*

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, так и для модификации 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*. В этой связи во избежание возможной *рассинхронизации* вычислений и возникновения ошибочных ситуаций рекомендуется в качестве *глобальных* использовать в процедурах только те переменные**,** значения которых ими не модифицируются, а только считываются. Иначе не исключено возникновение отмеченных ситуаций, включая и непредсказуемые. Следующий пример иллюстрирует некорректность определения в процедуре *глобальной* **x-**переменной:

```
> x:= 64:  proc(y) global x; x:= 0;  y^(x+y) end proc(10);  evalf(2006/x);   ⇒ 10000000000
Error, numeric exception: division by zero
> x:= 64:  proc(y) global x; x:= 0;  y^(x+y) end proc:   evalf(2006/x);   ⇒ 31.34375000
```

вызывающей в дальнейшем ошибочную ситуацию. При этом, следует иметь в виду**,** что вычисление определения процедуры не изменяет *значений* содержащихся в ней *глобальных* переменных, а вычисляются они лишь в момент реального вызова процедуры, как это иллюстрируют оба примера фрагмента.

Локальные переменные также можно типировать подобно аргументам, но действие этого типирования имеет смысл лишь при установке **kernelopts(*assertlevel*=2),** например:

```
> P:= proc(a, b) local c::integer; c:=a+b end proc: P(42.47,6);   ⇒ 48.47
> kernelopts(assertlevel=2): P(42.47,6);
Error, (in P) assertion failed in assignment, expected integer, got 48.47
```

Если для переменных, используемых в определении процедуры, не определена область их действия (***local, global***), то ***Maple-***язык классифицирует их следующим образом. *Каждая* переменная**,** получающая в *теле* процедуры определение по (**:=**)-оператору либо переменная цикла, определяемая функциями {***seq, add, mul***} полагается *локальной* (***local***), остальные полагаются *глобальными* (***global***) переменными. При этом**,** если переменные **for**-цикла не определены локальными *явно*, то выводится предупреждающее сообщение вида "*Warning*, **`k`** *is implicitly declared local to procedure* **`P`",** где **k** и **P** – переменная цикла и имя процедуры соответственно. Тогда как уже для функций ***sum*** и ***product*** *переменные* цикла рассматриваются глобальными**,** не выводя каких-либо сообщений, что предполагает их явное определение в **local**-секции. Однако вне зависимости от наличия предупреждающих сообщений рекомендуется явно указывать *локальные* и *глобальные* переменные, что позволит не только избегать ошибок выполнения, но и более четко воспринимать исходный текст процедуры. Следующий фрагмент иллюстрирует вышесказанное:

```
> G:=2: A:=proc(n) V:=64: [args, assign('G', 5), assign('V', 9), assign(cat(H, n), `h`)] end proc:
Warning, `V` is implicitly declared local to procedure `A`
> [A(99), G, V, A(10), whattype(H9), H9];   ⇒ [[99], 5, 9, [10], symbol, H9]
> k:= 64:  H:= proc() product(args[k], k=1 .. nargs)/sum(args[k], k=1 .. nargs) end proc:
> [k, H(42, 47, 62, 67, 96, 89, 10, 17, 4), k];
Error, (in H) invalid subscript selector
```

```
> k:=64:  P:= () -> [seq(args[k], k=1..nargs)]:  P(1, 2, 3), k;     ⇒ [1, 2, 3], 64
> k:=64:  P:= () -> [sum(args[k], k=1..nargs)]:  P(1, 2, 3), k;
Error, (in P) invalid subscript selector
> k:=64:  P:= () -> [product(args[k], k=1..nargs)]:  P(1, 2, 3), k;
Error, (in P) invalid subscript selector
> k:=64: P:=proc() for k to nargs do  end do end proc:  P(1, 2, 3), k;   ⇒ 64
Warning, `k` is implicitly declared local to procedure `P`
```

Таким образом, в указанных случаях соответствующие переменные процедуры при ее вычислении неявно декларируются 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 **1**-й пример предыдущего фрагмента. Делать это позволяет процедура ***assign.*** Однако работа с такими глобальными переменными чревата непредсказуемыми последствиями. Таким образом, практика программирования в среде ***Maple-***языка рекомендует следовать следующим двум правилам определения области действия переменных**:**

(**1**) ***глобальными*** определять переменные, лишь используемые в режиме *"чтения"***;**

(**2**) ***локальные*** переменные определять явно в **local-**секции процедуры**.**

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

```
> restart;  P:= proc(a, b) local  x,  y;  x, y:= 42, 64;  x, y end proc:
> x,  y:= 42, 64:  P(a, b),  x,  y;    ⇒ 42, 64, 42, 64
> restart;  P:= proc(a, b) local  x,  y;  x, y:= a, b;  x, y end proc:
> x,  y:= 42, 64:  P(a, b),  x,  y;    ⇒ a, b, 42, 64
```

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

По ***assign-***процедуре в теле процедур можно назначать выражения как *локальным* (*заданным явно*), так и *глобальным* (*заданным явно либо неявно*) переменным. Однако здесь имеется одно весьма существенное отличие. Как известно, пакет не допускает динамического генерирования имен в левой части (**:=**)-оператора присваивания, тогда как на основе ***assign-***процедуры это возможно делать. Это действительно существенная возможность, весьма актуальная в целом ряде задач практического программирования [103]. Между тем, если мы по процедуре ***assign*** в теле процедуры будем присваивать выражения *локальным* переменным и сгенерированным одноименным с ними переменным, то во вто-

ром случае присвоения производятся именно *глобальным* переменным, не затрагивая *локальных*. Нижеследующий пример весьма наглядно иллюстрирует вышесказанное.

> restart; V42, G47:= 10, 17: proc() local V42, G47; assign(V42=64, G47=59); assign(cat(V, 42)=100, cat(G, 47)=200); [V42, G47] end proc(), [V42, G47]; ⇒ [64, 59], [100, 200]

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

Следует еще раз отметить, что для предложений *присвоения* в процедурах в целом ряде случаев использование ***assign***-процедуры является единственно возможным подходом. Однако, при таком подходе в общем случае требуется, чтобы *левая* часть уравнения **x=a** в **assign(x=a)** была *неопределенным* именем, т.е. для нее *должно* выполняться соотношение ***type*(x, 'symbol') =** *true*. И здесь вполне допустимо использование конструкций следующего общего формата кодирования:

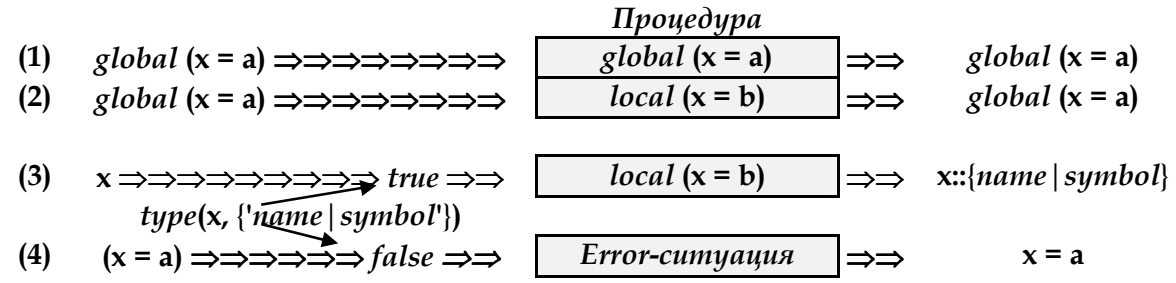
***assign*(*op*([*unassign*('<*Имя*>'), <*Имя*>]) = <*Выражение*>)**

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

```
> x:= 64: assign(op([unassign(x), x]) = 59);  x;    ⇒ 64
Error, (in unassign) cannot unassign '64' (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])=59),k=[P,M,T,A]); [P,M,T,A], map(type, [P,M,T,A],'odd');
                                  [59, 59, 59, 59], [true, true, true, true]
```

В частности, данный прием оказывается весьма удобным при необходимости *присваиваний* выражений *глобальным* переменным либо фактическим аргументам процедуры, передаваемым через формальный *uneval*-аргумент.

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



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

явно**.** Прежде всего, как отмечалось выше, не выводится предупреждающих сообщений о том, что они предполагаются *локальными*. Следовательно, они согласно трактовке языка ***Maple*** должны рассматриваться *глобальными*. Между тем, 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 нее она вновь становится *неопределенной***,** т.е. имеет место *глобальное* поведение переменной. Если на момент вызова процедуры **x-**переменная имела значение, то выполнение процедуры инициирует *ошибочную* ситуацию, а значение **x-**переменной остается неизменным (*случай* **4**). Рассмотренные ситуации еще раз говорят в пользу 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:= 64:  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))];
```

$$\Rightarrow \left[64,\ \frac{y^{(n+1)}\,n^2-2\,y^n\,n^2+y^{(n-1)}\,n^2-y^{(n+1)}\,n+y^{(n-1)}\,n+2\,y^n-2}{(y-1)^3}\right]$$

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

Следующий простой фрагмент иллюстрирует взаимосвязь *локальных* и *глобальных* переменных, когда вторые определяются как *явно* в **global**-секции, так и *неявно* через ***assign***-процедуру. Второй же пример фрагмента иллюстрирует механизм действия *uneval*-типированного формального аргумента, когда *(в отличие от стандарта*) ему могут присваиваться выражения на уровне *глобальных* переменных. Применение подобного весьма полезного приема уже иллюстрировалось в предыдущем разделе.

```
> P:= proc() local a; global b; x:= 56; assign('m' = 67); a:= proc(x) m:= 47; assign('n' = 89); x
  end proc; x*a(3) end proc:
Warning, `x` is implicitly declared local to procedure `P`
Warning, `m` is implicitly declared local to procedure `a`
> n, m:= 100, 100:  P(), n, m;   ⇒ 168, 89, 67
> P1:= proc(m::uneval) local a, x; global b; a, x:= 59, 39; b:= 64; assign('m'=42) end proc:
> m:= 64:  P1(m), a, x, b, m;   ⇒ a, x, 64, 42
```

Еще на *одном* существенном моменте механизма *глобальных* и *локальных* переменных необходимо акцентировать внимание, предварительно пояснив понятие *по-уровневого* вычисления. Правила вычислений в ***Maple-***языке предполагают нормальными *полные* вычисления для *глобальных* переменных и ***1-****уровневые* для *локальных*. Поясним сказанное первым примером следующего 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 простая рекуррентная цепочка выражений, вычисление которой реализует полностью рекуррентная подстановку и обеспечивает возврат конечного числового значения, т.е. производится полное вычисление для **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 иллюстрирует *второй* пример фрагмента.

Для *полного* вычисления произвольного **B-**выражения используется вызов ***eval*(B)-**функции. Однако в первом примере фрагмента **W-**переменная является *глобальной*, что и определяет ее полное вычисление, если (*как это иллюстрирует второй пример*) не определено противного. Наконец, третий пример фрагмента иллюстрирует результат вычисления той же рекуррентной цепочки выражений, но уже 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 и для его полного вычисления требуется использование функции ***eval***, как это иллюстрирует последний пример фрагмента. Данное обстоятельство следует всегда иметь в виду, ибо оно не имеет *аналогов* в традиционных языках программирования и наряду с требованием особого внимания обеспечивает целый ряд весьма интересных возможностей программирования в различных приложениях.

Еще на одном моменте следует остановиться отдельно. По конструкциям ***op*(2, *eval*(P))** и ***op*(6, *eval*(P))** возвращается соответственно последовательность *локальных* и *глобальных* переменных процедуры **P.** И если в первом случае это действительно так, то во втором ситуация совершенно иная, а именно. Если в процедуре производятся *присвоения* переменным по ***assign*-**процедуре и эти переменные *явно* не определены *глобальными*, то второй вызов их не распознает, например:

> restart; P:= proc() local a, b, c; assign(x= 64, y = 59) end proc: P(): seq([op(k, eval(P))], k = [2, 6]), x, y; ⇒ [a, b, c], [], 64, 59

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( ), "\$Art18_Kr10$.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, fremove(f)
end proc
```

```
> P:= proc() local a; global x, y;  a:= `+`(args); 5*assign(v=64), assign(g=59), assign(s=39);
  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' = 64, 'c' = 2006), assign67('y' = 59, 39); a:=proc()
  local z, h; assign67('z' = 10, 17), assign(h = 59); [z], h end proc; a() end proc:  [Proc()],
  Globals(Proc);   ⇒ [[10, 17], 59], {c}, {x, y, z, h}
> restart; Proc:= proc() local a, b; global c; assign('x' = 64, 'c' = 2006), assign67('y' = 59, 39);
  a:=proc() local z, h; assign67('z' = 10, 17), assign(h = 59); [z], h end proc; a() end proc:
  [Proc()]:  c, x, [y], [z], h;   ⇒ 2006, 64, [59, 39], [z], h
> restart; P1:= proc() local a; assign('a' = 64); a end proc: P1(), a;    ⇒ 64, a
> restart; P1:= proc() local a; assign67('a' = 64); a end proc: P1(), a;   ⇒ 64, a
```

С этой целью нами была определена процедура ***Globals***, обеспечивающая более точное тестирование *глобальных* переменных процедур. Вызов процедуры ***Globals*(P)** возвращает 2-элементную последовательность множеств *глобальных переменных* процедуры, определяемой фактическим аргументом **P.** *Первое* множество возвращаемой последовательности содержит имена *глобальных переменных*, определяемых **global**-секцией процедуры **P,** тогда как *второе* множество содержит имена *глобальных* переменных, которые определяются вызовами процедур пакетной ***assign*** и наших ***assign6, assign7, assign67*** и ***Assign*** [41,103]. Процедура ***Globals*** функционирует в среде ***Maple 9*** и выше. При этом, следует иметь в виду следующее важное обстоятельство, а именно.

Для *простой* процедуры (*не содержащей внутри себя других процедур*) вызов ***Globals*(P)** возвращает как определенные в **global**-секции **P** переменные, так и переменные, значения которым присваивались процедурами ***assign, assign6, assign7, assign67*** и ***Assign.*** Совсем иначе ситуация обстоит в случае *вложенных* процедур, когда вложенная процедура (*подпроцедура*) также содержит вызовы указанных процедур, но некоторые из вычисляемых ими переменных определены в **local**-секции подпроцедуры. В этом случае такие переменные, идентифицируясь ***Globals*** глобальными, на самом деле носят *локальный* характер. Сказанное хорошо иллюстрирует пример ***Proc***-процедуры предыдущего фрагмента (*переменные* **z** *и* **h**). Данное обстоятельство следует учитывать при использовании процедуры ***Globals*** для вложенных процедур, содержащих ***assign***-вызовы.

В целях повышения надежности при работе с *глобальными* переменными может быть использована процедура ***uglobal,*** 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 произвольно их использовать.

```
uglobal := proc()
local a, b;
    assign( a = cat( CDM(  ), "/$Art_Kr169$" ), 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:=64: P:=proc() global V; uglobal('V'); V:= `*`(args); V,uglobal() end proc: P(10, 17, 39),V;
                                        6630, 64
```

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

В ***Maple,*** начиная с **9**-го релиза, для встроенной функции ***anames*** был определен аргумент **'user',** по которому вызов ***anames*('user')** возвращает последовательность имен, значения для которых в текущем сеансе были определены пользователем. Данная возможность важна для приложений. В целях получения некоторого аналога данного средства для ***Maple*** релизов **8** и ниже нами была определена процедура ***pusers,*** *исходный* текст которой и пример применения представлены нижеследующим 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
        { Testzero, pusers };
    for k in p do
        unassign( 'h' );
        try ParProc1( k, h ); if type( h, 'symbol' ) then a := { op( a ), k } 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();   ⇒ {Proc, Proc1, Proc2, Proc3, Proc4}
```

Вызов процедуры ***pusers*()** возвращает множество *имен* процедур, определенных пользователем в текущем сеансе. При этом, в него не включаются те процедуры, определения которых находятся в ***Maple***-библиотеках пользователя, логически сцепленных с *главной* библиотекой пакета. Данная процедура функционирует в среде ***Maple*** релизов **6 – 10.**

```
elib := proc(e::symbol, L::{ mlib, mla } )
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 10/lib/userlib", 'h'), h;   ⇒ true, 2007/1/21
```

Вызов представленной выше процедуры ***elib*(e, L)** возвращает *true*-значение, если *объект* **e** находится в ***Maple***-библиотеке **L,** и *false*-значение в противном случае. Если указан и третий аргумент, то через него возвращается дата сохранения **e**-объекта в библиотеке.

3.4. *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 данных секций **description-**секция располагается непосредственно за заголовком процедуры и кодируется в следующем формате**:**

description <*Строчная конструкция*> {**:**|**;**}

Определенная в данной секции *строчная конструкция* не влияет на выполнение процедуры и используется в качестве комментирующей компоненты, т.е. она содержит текстовую информацию, предназначенную, как правило, для документирования процедуры. При этом, в отличие от обычного комментария языка, которое игнорируется при чтении процедуры, описание ассоциируется с процедурой при ее выводе даже тогда, 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 простой фрагмент**:**

```
> REA:= proc() description `Average`; sum(args[k], k= 1 .. nargs)/nargs end proc:
> REA(19.42, 19.47, 19.62, 19, 67, 19, 89, 20.06), eval(REA);
      34.07125000, proc () description Average; sum(args[k],k = 1 .. nargs)/nargs end proc
> REA:= proc() option Copyright; description "Average of real arguments";  sum(args[k],
   k= 1 .. nargs)/nargs end proc:
> eval(REA);    ⇒  proc () description "Average of real arguments"  … end proc
```

Данный фрагмент иллюстрирует результат использования **description-***секции* процедуры в случаях как отсутствия, так и наличия в ней дополнительно и ***Copyright*****-**опции. В примерах фрагмента использовались в **description-**секции комментарии *string*-типа. В связи со сказанным, механизм **description-**секций достаточно прозрачен и особых пояснений не требует. При этом, подавляющее большинство пакетных процедур не содержат **description-**секций.

Рассмотрев секции **local, global** и **description,** несколько детальнее остановимся на секции {**options**|**option**}, которая должна кодироваться непосредственно за двумя первыми (*или быть первой при их отсутствии*) в описательной части определения процедуры. В качестве параметров (*опций*) данной секции допускаются следующие**:** ***builtin, arrow, Copyright, trace, operator, remember*** и ***call_external.*** При этом, перечень опций может зависеть от используемого релиза пакета.

Пакет располагает *тремя* 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. Данный параметр визуализируется при *полном* вычислении процедуры посредством ***eval-***функции либо по ***print-***функции, например**:**

```
> print(eval), eval(readlib);
                    proc () option builtin; 169 end proc
                proc () options builtin, remember; 237 end proc
```

Каждая встроенная функция идентифицируется *уникальным* номером (*зависящим от номера релиза пакета*) и пользователь не имеет прямой возможности определять собственные встроенные функции. В приведенном примере *первым* выводится результат вызова

print-*функции*, а *вторым* ***- eval-***функции, из чего следует, что встроенные функции ***eval*** и ***readlib*** имеют соответственно номера **98** и **152** (***Maple 8,*** *тогда как уже для* ***Maple 10*** *эти номера соответственно будут* **117** *и* **274**), а вторая процедура имеет дополнительно и опцию ***remember***.

Для проверки процедур могут быть полезны и три наши процедуры ***ParProc, ParProc1*** и **Sproc** [103], обеспечивающие возврат как основных параметров процедур, модулей и пакетов, так и их местоположение в библиотеках ***Maple,*** как это иллюстрирует следующий достаточно простой фрагмент:

```
> ParProc(MkDir), ParProc(came);  map(ParProc, ['add', march, goto, iostatus, seq]);
```

$$\left[\begin{array}{c} Arguments = (F::\{string, symbol\}) \\ locals = (cd, r, k, h, z, K, L, \Lambda, t, d, \omega, u, f, s, g, v) \end{array}\right], \left[\begin{array}{c} Arguments = (E::anything) \\ locals = (f, h) \\ globals = (__Art_Kr_) \end{array}\right]$$

[*builtin function*, 91, *iolib function*, 31, *builtin function*, 193, *iolib function*, 13, *builtin function*, 101]

```
> ParProc(DIRAX);
```

DIRAX is module with exports [*new,replace,extract,empty,size,reverse,insert,delete,sortd,printd,conv*]

```
> ParProc(process);
```

⇒ *inert_function*

process is module with exports [*popen, pclose, pipe, fork, exec, wait, block, kill, launch*]

```
> ParProc(Int);
```

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 10/lib/userlib"}]

$$\left[\begin{array}{c} Arguments = (M::\{procedure, module\}) \\ locals = (a, b, c, d, p, h, t, z, cs, L, R, N, \omega, \nu) \\ globals = (_62, ParProc, Sproc) \end{array}\right],$$

[*Proc*, *User*, {"c:/program files/maple 10/lib/userlib"}]

```
> ParProc1(Sockets, 't'),  t;
```

Warning, module Sockets is in library [package, Maple, {"C:\Program Files\Maple 10/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 10/lib"}]

```
Sproc := proc(P::symbol)
local k, L, R, h, s, ω;
    `if`(type(P, procedure) or type(P, `module`),
        assign(L = [libname], R = {  }, s = CF(cat(CDM(  ), "\lib"))),
        RETURN(false)), assign(ω = (proc(R, s)
        local a, b;
            assign(a = map(CF, R), b = `if`(type(P, 'package'), 'package',
                `if`(type(P, procedure), 'Proc', 'Mod'))));
            if {s} = a then b, 'Maple'
            elif member(s, a) then b, `Maple&User`
```

```
            else b, 'User'
            end if
        end proc ));
    `if`(type(P, 'builtin'), RETURN(true,
        `if`(1 < nargs, assign(args[2] = ['Proc', 'builtin', Builtin(P), s]), 7)), 7);
    try assign(h = IO_proc(P)), `if`(type(h, 'integer'), RETURN(true,
        `if`(1 < nargs, assign(args[2] = ['Proc', 'iolib', h, s]), NULL)), NULL)
    catch : seq(`if`(search(convert(march('list', L[k]), 'string'), cat("[", P, ".m","")),
        assign('R' = {L[k], op(R)}), NULL), k = 1 .. nops(L))
    end try;
    `if`(R = {}, RETURN(false),
        RETURN(true, `if`(nargs = 2, assign([args][2] = [ω(R, s), R]), NULL)))
end proc
> Sproc(MkDir, 'h'), h;   ⇒ true, [Proc, User, {"c:/program files/maple 9/lib/userlib"}]
> Sproc(`type/package`, 'h'), h;
                    true, [Proc, Maple&User, {"C:\Program Files\Maple 9/lib",
                                "c:/program files/maple 9/lib/userlib"}]
```

С описанием данных процедур можно ознакомиться в [103], тогда как сами они включены в прилагаемую *Библиотеку* [109]. Там же можно получить и их исходные тексты.

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

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 совмещает процедуры и более ранних релизов. Текст любой пакетной процедуры (*включая их* ***remember-****таблицы*), кроме *встроенных*, можно получать по конструкции следующего простого вида**:**

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

как это иллюстрирует следующий достаточно простой пример**:**

```
> interface(verboseproc = 3): eval(savemp);
(M::{`module`, set(`module`), list(`module`)}, F) → (proc()
local a;
    a := x → `if`(type(x, `module`), [x], x);
    map(mod21, a(M));
    modproc(op(a(M)));
    (proc() save args, F end proc )(op(a(M)))
end proc )( )
```

Данная возможность представляется нам весьма полезной не только *начинающему* программисту в среде ***Maple-***языка, но и даже достаточно искушенному пользователю.

Параметр ***package*** определяет принадлежность процедуры *внутреннему* модулю пакета. Тогда как по ***trace-***параметру определяется режим *трассировки* вызова процедуры неза-

висимо от значения *глобальной* ***printlevel*-**переменной пакета, устанавливающей режим вывода служебной информации на экран монитора.

Совместное использование *двух* 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:= proc() option operator, arrow; evalf(sqrt(sum(args[k]^2, k= 1 .. nargs))) end proc;
```

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

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

S := **proc** () **local** k, p; **options** *operator*, *arrow*; **for** k **to** nargs **do** p := args[k]; **if** type(p, '*numeric*') **then next else** print(p, whattype(p)) **end if end do end proc**

Использование указанных параметров предполагает также *упрощение* тела процедуры. С другой стороны, как иллюстрируют примеры процедур **G** и **S,** далеко не каждая процедура допускает представление в нотации функционального (**->**)-оператора. Данную возможность допускают**,** как правило**,** простые процедуры, реализованные однострочным экстракодом, что иллюстрирует простая **G-**процедура *первого* примера фрагмента.

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

type(Proc, 'procedure') и **whattype(*eval*(Proc))**

возвращающие соответственно значения *true* и *procedure***,** если ***Proc*-**объект является процедурой ***Maple***, определенной любым из *трех* рассмотренных выше способов, и, кроме того, вычисленной. Следующий простой фрагмент иллюстрирует вышесказанное**:**

```
> 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]
```

В частности, *последний* пример фрагмента представляет простую тестирующую процедуру ***Type_Proc*,** возвращающую ***true*-**значение лишь тогда, когда объект, приписанный ***Id*-**идентификатору, является ***Maple*-**процедурой. При этом, еще раз следует обратить внимание на то 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**,** то 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 может выступать любой допустимый идентификатор. В связи со сказанным по ***whattype*(*Proc*)-**вызову возвращается *symbol*-значение, а по вызову ***whattype*(*eval*(*Proc*)) –** *procedure*-значение. Поэтому по ***op*(*n*, *eval*(*Proc*))-**вызову возвращается значение восьми компонент *определения* процеду-

ры, приписаного ***Proc*-**переменной. В табл. **7** представлены **n-**значения для указанной конструкции и их смысловая нагрузка**:**

Таблица 7

n	***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
8	тип возвращаемого результата (*если был определен*)

В случае отсутствия в определении процедуры какой-либо из рассмотренных *секций* на соответствующем ей **n-**значении конструкция ***op*(*n*, *eval*(*Proc*))** возвращает *NULL*-значение. 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 данных компонент не входит само *тело* процедуры и для 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 * 29.03.99`, 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:= 59: V:= 64: S:= 39:  [IAN(19.95, 59, 19.99), IAN(0.1, 17, 1.1)];
                                  [96.82051282, 27.89743590]
> for k to 8 do print(op(k, eval(IAN))) end do;
                                  x::float, y::integer, z::numeric
                                  k, h
Copyright Tallinn Research Group * 29.03.99, remember
                  table([(19.95, 59, 19.99) = 96.82051282, (0.1, 17, 1.1) = 27.89743590])
                                  "G-average of arguments"
                                  G, V, S
> proc() local k; option `Copyright * 29.03.99`;  sum(args[k], k=1..nargs)  end proc;
  % (64, 59, 39, 44, 17, 10);    ⇒  proc() ... end proc          233
```

В приведенном фрагменте определяется простая ***IAN*-**процедура, содержащая все допустимые *компоненты* определения ***Maple*-**процедуры, и производится ее вычисление, выводящее на экран только частичный текст определения процедуры, ибо при ее определении был задействован ***Copyright*-**параметр **option-**секции. Однако, возвращаемое в результате вычисления определение является *полным* и его последующий вызов возвращает корректные результаты, как это иллюстрирует последний пример фрагмента с непоименованной процедурой. После вычисления определения производится двухкратный вызов процедуры**;** последующее использование ***op*(k, *eval*(*IAN*))-**конструкции в (**for_do**)-предложении выводит содержимое всех восьми компонент ***IAN*-**процедуры.

Как следует из вышесказанного, тело процедуры рассмотренными средствами не идентифицируется в качестве ее компоненты и доступ к нему возможен иными средствами, рассматриваемыми несколько ниже. Здесь же уместно отметить лишь ***dismantle*(P)-**процедуру, обеспечивающую вывод структуры данных, определяемой **P-***выражением*. Про-

цедура выводит структуру **P-**выражения (*которое для случая процедуры должно быть полностью вычисленным*) в разрезе составляющих его подвыражений (*компонент*), их длины и *относительные* адреса в {*десятичном*|**16-***ричном*|**8-***ричном*} представлении. Следующий фрагмент иллюстрирует вывод структуры данных, отвечающей ***IAN-***процедуре предыдущего примера**:**

```
> dismantle(eval(IAN));
PROC(9) #[`Copyright Tallinn Research Group * 29.03.99`, remember]
  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(14): `Copyright Tallinn Research Group * 29.03.99`
    NAME(6): remember
  HASHTAB(129)
    HASH(7)
      EXPSEQ(4)
        FLOAT(3): 19.95
          INTPOS(2): 1995
          INTNEG(2): -2
        INTPOS(2): 59
        FLOAT(3): 19.99
          INTPOS(2): 1999
          INTNEG(2): -2
      FLOAT(3): 96.82051282
        INTPOS(4): 9682051282
        INTNEG(2): -8
    HASH(7)
      EXPSEQ(4)
        FLOAT(3): .1
          INTPOS(2): 1
          INTNEG(2): -1
        INTPOS(2): 17
        FLOAT(3): 1.1
          INTPOS(2): 11
          INTNEG(2): -1
      FLOAT(3): 27.89743590
        INTPOS(4): 2789743590
        INTNEG(2): -8
  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)
```

Данный фрагмент представляет внутреннюю структуру ***Maple-***процедуры и на ее основе искушенный пользователь может решать целый ряд весьма интересных задач. Однако, в нашу задачу рассмотрение *данной* проблематики не входит. Здесь уже вполне можно вновь возвратиться к рассмотрению ***remember-***параметра **option-**секции, предварительно дав дополнительную полезную информацию.

Набор из процедуры ***dismantle*** и 4-х функций ***assemble***, ***addressof***, ***disassemble*** и ***pointto*** известен как "*хакерский*" пакет в ***Maple***. Последние четыре функции обеспечивают доступ к внутренним представлениям объектов ***Maple*** и к адресам, указывающим на них. Между тем, пользователь должен быть знаком с внутренним представлением объектов пакета перед использованием данного набора средств. Для этого рекомендуем обратиться, например, к руководствам [83,84]. Некоторые средства подобного типа представлены и в [41,42,103].

Для целого ряда типов процедур (*и в первую очередь для рекурсивных*), характеризуемых многократными вызовами на одних и тех же наборах фактических аргументов, важную задачу приобретает вопрос повышения эффективности их выполнения. Данная задача

решается путем сохранения истории вызовов процедуры в текущем сеансе в специальной ***remember-***таблице. Использование данного *механизма* требует определенных 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 процедур.

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

```
> op(4, eval(Proc));
        table([
                (<Фактические аргументы_1>) = <Возвращаемое значение_1>
                ==========================================
                (<Фактические аргументы_h>) = <Возвращаемое значение_h>])
```

Как следует из представленного, ***remember-***таблица в качестве входов использует последовательности фактических аргументов *каждого* ее вызова, а выходов **-** значения, возвращаемые на соответствующих фактических аргументах. Данная организация обеспечивает эффективную работу с часто используемыми или *рекурсивными* процедурами, ибо при вызове процедуры в случае установления *наличия* в ее ***remember-***таблице аналогичного вызова, производится возврат соответствующего результата без повторного выполнения *тела* процедуры, т.е. входы в таблицу не дублируются. Целый ряд средств пакета определены с ***remember-***параметром, например, ***readlib-***функция и др.

Между тем, использование ***remember-***механизма может *наряду* с повышением эффективности выполнения процедуры 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 использование с ***remember-***параметром ***system-***параметра обеспечивает автоматическую очистку (*без ее удаления*) ***remember-***таблицы в процессе выполнения ядром периодических операций по очистке от "*муссора*" занимаемой текущим сеансом памяти **ПК.** Именно поэтому *не следует* использовать ***system-***параметр для тех процедур, текущий результат выполнения которых зависит от истории вычислений, как это имеет место для рекурсивных процедур. Другие средства ***Maple-***языка позволяют производить *модификацию* ***remember-***таблицы более дифференцированно.

Для *обнуления* ***remember-***таблицы ***Proc-***процедуры применяется процедура ***forget*****(*****Proc*****)**. Вызов ***forget*****(*****Proc*****{, A}{, 0})** позволяет удалять из ***remember-***таблицы *входы* для определяемой *первым* фактическим аргументом ***Proc-***процедуры. При этом, допускается передавать ***Proc-***процедуре фактические **A-**аргументы и использовать необязательные две **O-**опции (***reinitialize***, ***subfunction***), управляющие непосредственным выполнением ***forget-***процедуры. Детальнее с ними можно ознакомиться по справке пакета. Посредством передачи 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, однако оставаясь ассоциированной с данной процедурой. Наконец, посредством выполне-

ния подстановки ***subsop*(4=*NULL*, *eval*(*Proc*))** производится *удаление* таблицы ***remember*** для ***Proc*-**процедуры. Однако, ***forget*-**процедуру следует использовать с определенной осмотрительностью.

Прежде всего, ***forget*-**процедуру не рекомендуется использовать для *расширенного* управления ***remember*-**таблицей процедур, т.к. она, в частности, не работает с целым рядом модульных процедур пакета, а также со многими процедурами, определенными в модулях ***Share*.** Ряд достаточно простых примеров нижеприведенного фрагмента иллюстрирует вышесказанное. Для модификации процедурной таблицы можно использовать и нашу процедуру ***Remember_T*** от неопределенного числа формальных аргументов, по которой производится редактирование *входов* и *выходов* таблицы. Следующий простой фрагмент иллюстрирует текст процедуры [12,103] и примеры ее применения:

```
Remember_T := proc(NP::{symbol, name})
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(10,17,39,44,59,64), P3(96,89,67,62,47,42), P3(30,3,99,350,520,5)]:  op(4, eval(P3));
        table([(96, 89, 67, 62, 47, 42) = 67.16666667, (10, 17, 39, 44, 59, 64) = 38.83333333,
                                    (30, 3, 99, 350, 520, 5) = 167.8333333])
> Remember_T(P3, [10, 17, 39, 44, 59, 64]=2006, [96, 89, 67, 62, 47, 42], [31, 3, 99, 42]=[10, 17]):
> op(4, eval(P3));
 table([(10, 17, 39, 44, 59, 64)=2006, (30, 3, 99, 350, 520, 5)=167.8333333, (31, 3, 99, 42)=[10, 17]])
```

В приведенном фрагменте предварительно определяется ***Remember_T*-**процедура от неопределенного числа формальных аргументов, из которых первым фактическим аргу-

ментом должен выступать идентификатор процедуры, ***remember-***таблица которой модифицируется. Остальные фактические аргументы кодируются в следующем формате**:**

[x_1, x_2, …, x_n] = <*Значение*> либо **[x_1, x_2, …, x_n]**

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

В ряде случаев она оказывается весьма *полезным* средством, в первую очередь, при необходимости *более* эффективного использования памяти при работе с *рекурсивными* и часто используемыми процедурами в циклических конструкциях. Пример в нижеследующем фрагменте иллюстрирует строго линейную зависимость числа входов в таблицу ***remember*** при вызове процедуры ***Proc_30*** в циклической конструкции, однако такая зависимость может носить и нелинейный характер, существенно увеличивая размер таблицы и требуемое для нее место в оперативной памяти. В связи с этим при работе с рекурсивными процедурами или при наличии вызовов процедур в теле *циклических* конструкций рекомендуется оценивать целесообразность использования как ***remember-***таблицы, так и режима ее последующей модификации.

```
> Proc:= proc() options package;  evalf(sum(args[k], k= 1 .. nargs)/nargs) end proc:
> Proc(10, 17, 39, 44, 59, 64);   ⇒ 38.83333333
> Proc1:= proc() options trace;  evalf(sum(args[k], k= 1 .. nargs)/nargs) end proc:
> Proc1(10, 17, 39, 44, 59, 64);
        {--> enter Proc1, args = 10, 17, 39, 44, 59, 64
                                        38.83333333
        <-- exit Proc1 (now at top level) = 38.83333333}
                                        38.83333333
> Proc2:= proc() options operator, arrow;  evalf(sum(args[k], k= 1 .. nargs)/nargs) end proc:
> [eval(Proc2), Proc2(10, 17, 39, 44, 59, 64)];
```

$$\left[( ) \rightarrow \mathrm{evalf}\left(\frac{\sum_{k=1}^{\mathrm{nargs}} \mathrm{args}_k}{\mathrm{nargs}} \right), 38.83333333 \right]$$

```
> Proc3:= proc() options operator, arrow, remember; evalf(sum(args[k], k= 1 .. nargs)/nargs)
  end proc:  [Proc3(10, 17, 39, 44, 59, 64), Proc3(96, 89, 67, 62, 47, 42), Proc3(30, 6, 99, 350,
  520)]:  op(4, eval(Proc3));
 table([(96,89,67,62,47,42)=67.1666667, (30,6,99,350,520)=201., (10,17,39,44,59,64)=38.8333333])
> forget(Proc3, 10, 17, 39, 44, 59, 64):  op(4, eval(Proc3));
              table([(96, 89, 67, 62, 47, 42) = 67.16666667, (30, 6, 99, 350, 520) = 201.])
> Proc_30:= proc() options remember; sum(args[n], n= 1 .. nargs) end proc:
> h:= 0:  for k to 300 do h:= h + Proc_30(a$a = 1 .. k) end do:  h;   ⇒ 4545100
> nops([indices(op(4, eval(Proc_30)))]);   ⇒ 300
> 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(32), time() - T];   ⇒ [2178309, 12.157]
> T:= time():  [Fib1(32), time() - T], nops([indices(op(4, eval(Fib1)))]);   ⇒ [2178309, 0.], 33
```

Между тем, наиболее важной особенностью ***remember-***параметра является возможность на основе ***remember-***таблицы определять эффективные во временном отношении *рекурсивные* процедуры. Обеспечивая возможность определения *рекурсивных* процедур, механизм *ядра*, вместе с тем, определяет эффективность их выполнения в зависимости от использования таблицы ***remember***. В конце предыдущего 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 **32** входа, что не соизмеримо с полученным временным выигрышем. Однако возможно и наоборот, например, при использовании процедур в *циклических* конструкциях (*пример* ***Proc_30-****процедуры фрагмента*).

Для модификации ***remember-***таблицы произвольной ***Proc-***процедуры, определение которой содержало ***remember-***параметр, наряду с приведенной выше ***Remember_T-***процедурой можно использовать конструкции следующего простого вида**:**

Proc**(<***Фактические аргументы* (***ФА***)**>) :=** <*Требуемый результат*> {**:|;**}
T := ***op*****(4,** ***eval*****(*****Proc*****)): T[(<*****ФА*****>)] :=** ***evaln*****(T[(<*****ФА*****>)])** {**:|;**}

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

```
> P:= proc() local k; options remember; sum(args[k], k= 1 .. nargs) end proc:          #  (1)
> [P(10, 17, 39, 44, 59, 64), P(96, 89, 67, 62, 47, 42), P(11, 6, 98, 445, 2006)]:  op(4, eval(P));
   table([(11, 6, 98, 445, 2006) = 2566, (96, 89, 67, 62, 47, 42) = 403, (10, 17, 39, 44, 59, 64) = 233])
> P(42, 47, 67, 89, 96):= 2006:  P(56, 51, 31, 2, 9):= 2500:  op(4, eval(P));          #  (2)
     table([(42, 47, 67, 89, 96) = 2006, (11, 6, 98, 445, 2006) = 2566, (96, 89, 67, 62, 47, 42) = 403,
                                (56, 51, 31, 2, 9) = 2500, (10, 17, 39, 44, 59, 64) = 233])
> T:= op(4, eval(P)):  T[(42, 47, 67, 89, 96)]:= evaln(T[(42, 47, 67, 89, 96)]):  op(4, eval(P));
     table([(11, 6, 98, 445, 2006) = 2566, (96, 89, 67, 62, 47, 42) = 403, (56, 51, 31, 2, 9) = 2500,
                                (10, 17, 39, 44, 59, 64) = 233])
```

В *первом* примере фрагмента определяется **P-**процедура с ***remember-***параметром, затем производится ее вычисление и после трех вызовов процедуры выводится содержимое ассоциированной с ней ***remember-***таблицы**.** Во *втором* примере на основе *присвоения* данная таблица расширяется на два новых входа и выводится ее актуальное состояние. Наконец, в *третьем* примере производится удаление из ***remember-***таблицы ее конкретного входа с выводом нового состояния.

Следует отметить, что ***remember-***механизм может быть *успешно* использован и для решения других важных задач, в частности, имеющих дело с функциями с особенностями (*точки разрыва*, *сингулярные точки*). Идея 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 отсутствия. С этой целью выражение с особенностями оформляется процедурой с параметром ***remember*** в ее секции **option**, процедура вычисляется и сразу же описанным выше способом производит-

ся *включение* в ее ***remember-***таблицу *входов*, соответствующих *особым* точкам выражения, как это иллюстрирует следующий простой пример**:**

```
> GS:=proc(x) evalf(1/x,3) end proc: GS(0):=infinity:  map(GS, [10,0,17]); ⇒ [0.100, ∞, 0.0588]
> op(4, eval(GS));   ⇒  table([0 = ∞])
```

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

(1) ***Proc* := proc(...) option remember; ... end proc** {**:**|**;**}

(2) ***Proc* := proc(...) ... end proc: Proc(ФА) :=** <*Значение*> {**:**|**;**}

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

```
> G:= proc(x) x end proc:  [assign('G(42)', 647), assign('G(47)', 59), assign('G(67)', 39)]:
> G1:= x -> x:  [assign('G1(42)', 64), assign('G1(47)', 59), assign('G1(67)', 39)]:
> define(G2, G2(x::numeric) = x):  [assign('G2(42)', 64), assign('G2(47)', 59), assign('G2(67)',
  39)]:  op(4, eval(G)), op(4, eval(G1)), op(4, eval(G2));
 table([67 = 39, 42=64, 47 = 59]), table([67 = 39, 42=64, 47 = 59]), table([67 = 39, 42=64, 47 = 59])
> 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):= 64:  P1(47):= 59:  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 = 64, 47 = 59])
```

Первые три примера фрагмена иллюстрируют нестандартной прием ассоциирования с процедурами всех допустимых типов ***remember-***таблицы, тогда как последние *два* примера проясняют принципиальные различия ***remember-***таблиц процедур, созданных как посредством ***remember-***параметра**,** так и *нестандартно*. Из приведенного фрагмента следует, что появляется простая возможность *наделять* ***remember-***механизмом (*пусть и 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 *ассоциирована*.

3.5. *Механизмы возврата Maple-процедурой результата ее вызова*

В результате вызова стандартным является возврат результата выполнения процедуры через значение *последнего* предложения ее *тела*. Однако, ***Maple-***язык поддерживает еще *три* основных механизма возврата результата вызова процедуры**:** через *фактический* аргумент, функцию ***RETURN*** (**return**-*предложение*) и ***ERROR*-**функцию (**error**-*предложение*). В случае возникновения ошибочной ситуации в период выполнения процедуры производится аварийный выход из нее с выводом соответствующего сообщения. В отсутствие ошибочных ситуаций предварительно вычисленная процедура с ***Proc-***именем (*идентификатор, которому присвоено определение процедуры*) вызывается подобно *функции* по конструкции следующего формата кодирования**:**

***Proc*(**<*Фактические аргументы*>**)**

возвращая на переданных ей *фактических аргументах* соответствующее значение**,** определя-мое одним из вышеуказанных способов. Из указанных механизмов возврата результата вызова процедуры наиболее часто используемым является использование функции ***RETURN*** либо **return**-предложения следующих форматов кодирования**:**

***RETURN*(**<*Последовательность выражений*>**)**
return <*Последовательность выражений*> {**:**|**;**}

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

```
> AG:= proc(x::integer, y::float,R)  `if`(x*y>R, x, y) end proc:   AG(64, 51.6, 350);   ⇒ 64
> SV:= proc(x, L::list) member(x, {op(L)}) end proc:   SV(10, [64, 59, 10, 39, 17]);   ⇒ 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, 10, 95, 37, 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:=56: MM(2006, [57, 52, 3, 1999, 32, 10, 1995, 37, 180, 23, 1.0499, 2006]), k;  ⇒ [12, {2006}], 12
> MM(1942, [57, 52, 3, 98, 32, 10, 95, 37, 96, 34, 23, 97, 45, 42, 47, 67, 89, 96]);   ⇒ false
> 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(64, 59., 10/17, "RANS", Pi, TRG);
                    [Аргументы:, [3, Числовые], [3, Нечисловые]]
> 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", 2006, F(x), n..p, array(1..3, [10, 17, 39]));
```

$$\begin{bmatrix} Argument & Type \\ RANS & symbol \\ \text{"IAN"} & string \\ 2006 & integer \\ F(x) & function \\ n\,..\,p & .. \\ [10, 17, 39] & array \end{bmatrix}$$

Первые три примера фрагмента представляют простые процедуры ***AG***, ***SV*** и ***MV***, назначение первой из которых легко усматривается из ее определения, а две другие возвращают соответственно результат тестирования и список номеров позиций вхождения **x-**элемента в заданный **L-**список. Все три процедуры возвращают результат стандартным способом через *последнее* предложение тела процедуры. Остальные примеры фрагмента иллюстрируют возврат результата процедуры ***RETURN*-**функцией либо **return**-предложением.

Процедура **MM** возвращает список с номером позиции *первого* вхождения **x-**элемента в **L-**список и сам элемент, в противном случае возвращается ***false*-**значение. Процедура для возврата результата 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 количества аргументов обоих типов. Процедура **VS** возвращает таблицу, первый столбец которой содержит передаваемые процедуре вычисленные фактические аргументы, тогда как второй **-** их типы. Одним из достоинств использования функции ***RETURN*** является возможность эффективного возврата значений *локальных* переменных из целого ряда важных вычислительных конструкций, а также обеспечение гибкой возможности избирательности возврата результатов *вызова* пользовательских процедур. Тогда как предложение **return** имеет существенно меньшие выразительные возможности по представлению вычислительных алгоритмов в среде ***Maple*-**языка.

Следует отметить, что, начиная с ***Maple 6***, разработчики объявили о том, что ***RETURN***-функция является устаревшим средством (*obsolete*) и сохранена для обеспечения обратной совместимости процедур, разработанных в предыдущих релизах пакета. При этом, настоятельно рекомендуя использование именно **return**-предложения. Наш опыт работы с ***Maple*** говорит совершенно об *обратном*. Использование ***RETURN***-функции во многих случаях более предпочтительно, позволяя создавать эффективные *выходы* из вычислительных конструкций, прежде всего, в однострочных экстракодах. Ниже на этом моменте акцентируем несколько больше внимания.

В общем случае функция ***RETURN*(V1,V2 ,..., Vn),** где в качестве фактических аргументов могут выступать произвольные ***Maple*-**выражения, предварительно вычисляемые, может не только возвращать конкретные результаты вычисления **Vk-**выражений, связанных со спецификой тела процедуры, но и возвращать в определенных условиях вызов процедуры *невычисленным*. Для этих целей, как правило, используется конструкция вида ***RETURN*('*Proc*(*args*)'),** где ***Proc* –** имя (*идентификатор*) процедуры. Библиотечные процедуры ***Maple*** в случае некорректного либо прерванного вызова возвращают ***FAIL*-**значение, если нецелесообразно возвращать вызов процедуры *невычисленным*. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> 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(64, 59, 39.37, 19.81, 19.83, 10, 17, G, S);
                                  LO(64, 59, 39.37, 19.81, 19.83, 10, 17, G, S)
> [sqrt(10), sin(17), ln(gamma), exp(Pi)];   ⇒  [√10, sin(17), ln(γ), e^π]
```

В данном фрагменте **LO-**процедура в случае обнаружения *среди* переданных ей фактических аргументов выражения типа, отличного от ***float***, сразу же осуществляет *выход* по **return-**предложению с возвращением своего вызова невычисленным. Тогда как второй пример иллюстрирует возврат *невычисленных* вызовов базовых функций ***Maple*-**языка.

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

```
G := proc(L::list({integer, float, fraction}), x::evaln, y::evaln, z::evaln)
local a, b, c, k;
    `&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
> G([2006, 64, 10/17, 19.42, 59/1947, 350, 65.0, 16.10], a, v, z),  a, v, z;
                                                                                  [10   1 ]
                    50.21094042, [2006, 64, 350], [--, --], [19.42, 65.0, 16.10]
                                                                                  [17  33 ]
G1 := proc(L::list, x::evaln, y::evaln, z::evaln)
local a, b, c, d, k;
    `&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' = {op(d), whattype(eval(k))}))))), k = L),
    assign(x = a, y = b, z = c, `if`(4 < nargs, assign(args[5] = d), NULL)),
    evalf(sqrt(`+`(op(L))))
end proc
> G1([2006,64,10/17,19.42,59/1947,350,65.0,16.10,10+17*I,64-42*I], a, v, z, 'h'), a, v, z, h;
                                                                                  [10   1 ]
        50.94309321 - 0.2453718299 I, [2006, 64, 350], [--, --], [19.42, 65.0, 16.10], {complex}
                                                                                  [17  33 ]
```

Первый пример фрагмента представляет ***G-***процедуру**,** допускающую при своем вызове четыре фактических аргумента, из которых *три* последних имеют ***evaln***-тип и через которые передаются *списки* фактических аргументов, имеющих типы *float*, *integer* и *fraction* соответственно. Тогда как в качестве основного возврата процедуры является результат вычисления корня квадратного из суммы фактических аргументов, переданных процедуре при ее вызове. Пример *вызова* процедуры ***G*** иллюстрирует сказанное. *Второй* пример фрагмента представляет ***G1-***процедуру, являющуюся модификацией предыдущей процедуры и допускающей при своем вызове более четырех фактических аргумента, из которых четыре первых аналогичны случаю процедуры ***G*****,** тогда как через пятый *необязательный* аргумент передается множество типов фактических аргументов процедуры, отличных от типов {*float*, *integer*, *fraction*}. Тогда как *основной* возврат процедуры аналогичен случаю ***G***-процедуры. Пример вызова процедуры иллюстрирует сказанное. В обеих процедурах рекомендуется обратить внимание на использование процедуры/орператора **&ma** [103], обеспечивающего присвоение одного и того же выражения последовательности переменных, и ***assign***-процедуры для присвоения значений фактическим аргументам, через которые обеспечиваются вторичные возвраты процедуры.

Между тем, возвращать *вторичные* результаты вычислений процедур возможно и через ее фактические аргументы, не определяемые *evaln*-типом и имеющие *symbol*-тип; в этом

случае их рекомендуется кодировать в невычисленном формате, т.е. в прямых верхних кавычках. Следующий фрагмент на процедуре ***Kris*** иллюстрирует описанный подход.

```
Kris := proc(x, y, z)
local a, k;
    for k in [args] do
        if type(k, 'assignable') then
            a := k;
            assign(a = map(whattype, map(eval, [args])));
            WARNING("2nd result had been returned via <%1>", a);
            return [args]
        end if
    end do;
    [args]
end proc
> Kris(59, 64.42, z, [4, 1, 2007], x);
Warning, 2nd result had been returned via <z>
                                    [59, 64.42, z, [4, 1, 2007], x]
> z;   ⇒ [integer, float, symbol, list, symbol]
> Kris(59, 64.42, 65, [4, 1, 2007]);   ⇒ [59, 64.42, 65, [4, 1, 2007]]
```

Процедура ***Kris*** при обнаружении в точке своего вызова среди фактических аргументы *symbol*-типа через первый такой аргумент возвращает вторичный результат с выводом соответствующего сообщения и возвратом основного результата. Тогда как при отсутствии оного возвращается только основной результат.

Maple вычисляет *формальные* аргументы один раз, поэтому их не следует использовать в теле процедуры подобно локальным переменным. *Формальному* аргументу, через который будет возвращаться результат *вызова* процедуры, в ее теле должно быть выполнено только одно присвоение, определяющее возвращаемый результат, отличный от *основного*. При этом, в ряде случаев процедура может вполне обходиться без *основного* возврата, ограничиваясь только *вторичными*, а то даже и вовсе без них. В обоих случаях вызов такой процедуры возвращает *NULL*-значение, т.е. ничего. Механизм возврата результатов вызова ***Maple***-процедуры через ее фактические аргументы дает возможность наряду со стандартным или на основе ***RETURN***-функции *подходами* организовывать *дополнительные* (*вторичные*) выходы, определяемые наличием или отсутствием при вызове процедуры соответствующих аргументов. Подобная организация возврата результатов вызова достаточно широко практикуется в библиотечных процедурах как собственно самого пакета ***Maple***, так и наших [41,103,109].

Механизм возврата результатов вызова ***Maple***-процедуры через ее *фактические аргументы* позволяет наряду со стандартным либо на основе ***RETURN***-функции (**return**-*предложения*) подходами организовывать и дополнительные выходы, определяемые как наличием либо отсутствием при вызове процедуры соответствующих аргументов, так и выполнением оп-еделенных условий. Такая организация возврата результата вызова широко практикуется в пакетных процедурах. Наряду с представленными выше механизмами возврата результатов вызова процедуры можно предложить еще один механизм, полезный в целом ряде приложений и используемый рядом процедур нашей *Библиотеки* [103,109]. Суть его состоит в следующем.

Результаты вызова процедуры возвращаются через ее *ключевые* аргументы, представляющие собой некоторые идентификаторы. Эти переменные относительно процедуры

выступают на уровне *глобальных переменных* и их определение производится в *теле* процедуры, реализуясь лишь при указании такого аргумента при вызове процедуры. Схематично такую процедуру ***Proc*** можно представить следующим фрагментом.

```
Proc:= proc({Id1} {, Id2} … {, Idp}) local t;
        if member('Id1', [args], 't') then assign(args[t] = expr1) end if;
        if member('Id2', [args], 't') then assign(args[t] = expr2) 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, что здесь требуется использование ***assign***-процедуры, а не (**:=**)-оператора *присваивания*, т.е. ***assign*(args[t] = *expr*)**, а не **args[t] := *expr***. В противном случае ***Maple*** выводит предупреждение о недопустимости использования переменной ***args*** в качестве локальной переменной с последующим инициированием ошибочной ситуации с диагностикой «Error, ***args*** cannot be declared as a local**».** Это еще *один* пример принципиальных различий оператора присваивания (**:=**) и ***assign***-процедуры.

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

Данная организация 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 весьма простой модификации ее исходного текста*.

Как уже отмечалось выше, определение процедуры допускает следующий формат:

Proc := proc(*args*)::type; … end proc

который допустим только в релизах **7** и выше пакета, вызывая в ***Maple 6*** ошибку.

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 с диагностикой **"***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. Более того, при качественной разработке процедуры в ней уже должна быть предусмотрена (*при необходимости*) проверка типов получаемых ею фактических аргументов. Так что и здесь имеется «*фильтр*» на допустимось фактических аргументов. К тому же, если такого формата процедура находится в библиотеке и используется для программирования, то при возникновении ошибки указанного выше типа пользователю будет весьма слож-

но 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 роде подобна *типизации* формальных аргументов, но относится к результатам ее вызова, и может в ряде случаев представить интерес, например, в случае множественности точек возврата процедуры, упрощая обработку типов результатов вызова. Следующий простой фрагмент иллюстрирует результат использования указанного формата кодирования процедуры:

```
> P:= proc()::float; `+`(args)/nargs  end proc:  P(64, 59, 39, 10, 17, 44);   ⇒ 233/6
> kernelopts(assertlevel = 2):  P(64, 59, 39, 10, 17, 44);
Error, (in P) assertion failed: P expects its return value to be of type float, but computed 233/6
> 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 := 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(64, 59, 39, 10, 17, 44);   ⇒ 38.8333333300
                        P := proc():: `+`; op(8, eval('procname'))(args)/nargs end proc
> 6*P(64, 59, 39, 10, 17, 44);   ⇒ 233
                        P2 := proc()::proc() `if`(type(nargs, 'odd'), `+`, `*`) end proc;
                              eval(op(8, eval('procname')))(args)(args)/nargs
                        end proc
> P2(42, 47, 67, 89, 95), P2(42, 47, 67, 89, 95, 62);   ⇒ 68, 11555161030
```

Фрагмент содержит подпроцедуру ***conv***, результат вызова которой обеспечивает *конвертирование* возвращаемого процедурой ***P1*** выражения в тип, заданный после заголовка процедуры. Возможно, в некоторых случаях такой прием может оказаться полезным. В частности, кодировать после *заголовка* процедуры ***P*** можно любое корректное ***Maple***-выражение, доступное внутри процедуры по вызову ***op*(8, *eval*(*P*))**, что предоставляет целый ряд дополнительных возможностей при программировании приложений, как это иллюстрируют *две* последние процедуры ***P*** и ***P2*** фрагмента.

Последний механизм возврата результатов вызова процедуры связан с особыми и ошибочными ситуациями, возникающими в процессе ее выполнения. Любая *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. прилож.* **1** [12]). Однако, наряду с такого типа ситуациями, обрабатываемыми ***Maple*** автоматически, пользователь имеет возможность как производить *обработку* ситуаций, определяемых спецификой вызываемой процедуры, так и в определенной мере перехватывать обработку ошибочных ситуаций, стандартно *обрабатываемых* пакетом. В следующем разделе рассматриваются средства обработки ошибочных ситуаций, имеющих особый смысл именно для процедурных и модульных объектов.

3.6. *Средства обработки ошибочных и особых ситуаций в Maple-языке пакета*

В процессе вычисления ***Maple-***конструкций возможно появление *особых* и *аварийных* ситуаций**,** которые в большинстве случаев требуют специальной обработки во избежание серьезного нарушения всего вычислительного процесса. Идентифицируемые пакетом ситуации такого рода возвращаются в *предопределенную* ***lasterror-***переменную, значение которой определяет *последнюю* обнаруженную в текущем сеансе ошибку и доступно текущему сеансу для необходимой *обработки* возникшей ситуации. Значение ***lasterror-***переменной определяется только в момент возникновения ошибочной ситуации, а сразу же после загрузки пакета она является неопределенной**.** К ***lasterror-***переменной непосредственно примыкает и ***tracelast-***функция, обеспечивающая вызов *последней* ошибочной ситуации из стэка ошибок пакета. При этом, между ними имеется *принципиальное* различие, а именно**:** через ***lasterror-***переменную возвращается ошибочная диагностика**,** обрабатываемая функциональными средствами *языка*, тогда как ***tracelast-***функция *инициирует* вызов *последней* ошибочной ситуации, если она не была удалена из стэка ошибок. Значение ***lasterror-***переменной имеет *string*-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 *совместно* с ***lasterror-***переменной использовалась 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 определение предопределенной переменной ***lasterror***. Это же производится и по ***traperror*()-**вызову. Если же при вызове функции ***traperror*** не обнаружено особых ситуаций, то она возвращает *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 совместно с ***lasterror-***информацией для организации обработки *особых* и *ошибочных* ситуаций, возникающих в процессе вычислений в документе или ***Maple-***процедуре. Следующий простой пример иллюстрирует вышесказанное**:**

```
> VS:=0: AG:=15.04: if (traperror(AG/VS)=lasterror) then T:=AG/10.17 end if: [T, lasterror];
[1.478859390, "numeric exception: division by zero"]
```

В данном примере на основе вызова ***traperror*(AG/VS)** и ***lasterror*** обрабатывается особая ситуация "*деление на нуль*", в результате чего производится *устранение* данной ситуации путем перехода к вычислению другого выражения. При этом, следует обратить внимание на то обстоятельство, что при возникновении ошибочной ситуации в вычисляемом по ***traperror-***функции выражении *сообщение* о ней поглощается функцией, не отражаясь в документе**.** Большинство *серьезных* особых и аварийных ситуаций идентифицируется предопределенной ***lasterror-***переменной, поэтому *совместное* использование указанных средств может оказаться достаточно эффективным. При этом, 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:***

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

Это объясняется невозможностью восстановления на момент возникновения указанных ситуаций. При этом, следует иметь в виду**,** что вызов ***traperror*****(V)** не связывает с **V-**выражением особой ситуации типа "*деление на нуль*", если в качестве **V-**аргумента выступает конструкция следующего вида {**V/0 | V/(a-a) | V/(a*0)**}, т.е. если знаменатель дроби тождественно равен нулю**,** а не принимает нулевого значения в результате вычисления (*присвоения*) или упрощения выражения. Следующий фрагмент иллюстрирует сказанное**:**

```
> x:= 64: y:= 0: T:= traperror(x/(a - a)):  [lasterror, T];
                          ["numeric exception: division by zero", T]
Error, numeric exception: division by zero
> x:= 64:  y:= 0:  T:=traperror(x/(a*0)):  [lasterror, T];
                          ["numeric exception: division by zero", T]
Error, numeric exception: division by zero
> x:= 10:  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:  if 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 тот факт, что *порядок следования* переменной ***lasterror*** и вызова ***traperror-***функции в *логической* паре в общем случае существенен и первой следует кодировать ***traperror-***функцию. Между тем, повторное выполнение **if-**предложения последнего примера фрагмента возвращает корректные результаты, т. к. значение ***lasterror-***переменной сохраняет последнюю ошибочную ситуацию**.** Ниже мы еще раз вернемся к рассмотренным средствам обработки *ошибочных* и *особых* ситуаций, однако следует отметить, что ***traperror-***функция в значительной мере является устаревшим (*obsolete*) средством и его заменяет появившееся для данных целей в ***Maple 6*** более функционально широкое **try-**предложение, рассматриваемое ниже.

Важный тип особых ситуаций определяется временными ограничениями, связанными с возможностью больших временных затрат на вычисление выражения. 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 **ПК,** 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 {*от 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 ее 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 более удобен для 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. Следующий фрагмент иллюстрирует применение ***time-***функции для временных оценок**:**

```
> t:= time( ): G:= sin(10.3^8!): t1:= time( ): printf(`%s%1.3f%s`,`Время вычисления
  выражения "sin(10.3^8!)" равно: `, t1 - t, ` сек.`);
```

Время вычисления выражения "sin(10.3^8!)" равно: 29.781 сек.

$$Time := ( ) \rightarrow \mathrm{evalf}\left(\mathrm{map2}\left(`*`, \mathrm{time}( ), \left[\frac{1}{60}, \frac{1}{3600}\right]\right), 2\right)$$

> Time(); ⇒ [0.014, 0.00023]

Фрагмент включает пример простой ***Time-***процедуры, возвращающей общее время от начала текущего *сеанса* работы с пакетом в *минутах* и *часах*. Функция ***time*** используется для организации управления вычислениями в контексте их *временных* характеристик и может служить в качестве средства управления непредсказуемыми *по времени* вычислениями (*циклы*, *итерации и др.*).

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

```
> S:= proc(n) local k; for k to n do sin(10.3*n) end do end proc:  S(10^5),  evalhf(S(10^5)),
  timelimit(3, evalhf(S(10^7)));
                  -0.5431520991, -0.543152099236042907, 0.764330635010303183
> timelimit(3, S(10^7));
Error, (in sin) time expired
```

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

```
Timetest := proc(t::{float, integer}, x::anything, V::symbol, p::evaln)
local h, z;
    z := time( );
    if traperror( timelimit( t, assign( h = V(x) ))) = "time expired" then
        p := [ evalf( time( ) – z, 6 ), 'undefined' ]; false
    else p := [ evalf( time( ) – z, 6 ), h ]; 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), p;   ⇒ true, [0.234, 1000000]
> Timetest(0.5, 10^7, G, p), p;   ⇒ false, [0.516, undefined]
```

В качестве примера применения ***timelimit*-**функции предыдущий фрагмент представляет простую ***Timetest*(t, x, V, p)-**процедуру, возвращающую ***true*-**значение только в том случае, когда вычисление **V(x)-**выражения укладывается в отведенный ему *интервал* в **t-**секунд, в противном случае ею возвращается ***false*-**значение. При этом, через аргумент **p** процедуры возвращается список, первый элемент которого определяет время вычисления искомого выражения, а второй **-** *результат* вычисления либо ***undefined*-**значение соответственно. В данном фрагменте ***Timetest*-**процедура применяется для временного тестирования *вызова* простой ***G*-**процедуры. При этом, следует иметь в виду, что в *общем* случае совместное использование функций ***time*** и ***timelimit*** в одной процедуре может приводить к *некорректным* результатам [12]. Вместе с тем, при получении определенного навыка использования функциональных средств языка существующих средств обработки особых и аварийных ситуаций оказывается вполне достаточным для создания в среде ***Maple*** довольно эффективных пользовательских систем обработки ситуаций такого рода. Более того, на основе знания *специфики* реализуемого в среде языка алгоритма решаемой задачи функциональные средства ***Maple*-**языка дают возможность проводить предварительный программный анализ на предмет предупреждения целого ряда возможных особых и аварийных ситуаций. Рассмотрим теперь вопросы обработки особых и аварийных ситуаций в контексте программирования процедур.

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

ERROR({**V1, V2, …, Vn**}**)**

производится немедленный выход из процедуры в точке ее вызова с возвратом диагностического сообщения следующего вида**:**

Error, (in {**Proc** | **unknown**}) {<V1>, <V2>**, ... ,**<Vn>}

где ***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:

error {**Msg** {**, p1, p2, …, pn**}} {**:** | **;**}

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

Error, (in {**Proc** | **unknown**}) { **Msg(**{p1, p2**, …,** 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. Она может содержать пронумерованные параметры вида «**%n**», где **n** – *целое* число 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. Например:

```
> error  "invalid arguments <%1> and <%2>", 3, 7;
Error, invalid arguments <3> and <7>
> 64/0;
Error, numeric exception: division by zero
> error;
Error, numeric exception: division by zero
> restart; error;
Error, unspecified error
```

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

```
> error "%1 and %2 arguments are invalid", 3, 7;
Error, 3rd and 7th arguments are invalid
> error "%-1 and %-2 arguments are invalid", 3, 7;
Error, 3rd and 7th 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 с релиза **6** ***Maple.*** Детальнее с оформлением диагностических сообщений для ***ERROR***-функции (**error**-*предложения*) можно ознакомиться в справке по пакету.

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

```
> restart; lasterror;   ⇒ lasterror
> A:= proc() local k; `if`(type(nargs, 'odd'), ERROR("odd number of arguments"), `+`(args))
  end proc:  A(64, 59, 39, 10, 17, 44, 95, 2006);   ⇒ 2334
> A(64, 59, 39, 10, 17, 44, 2006);
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 производится выход из процедуры с возвратом соответствующего сообщения. После этого на *основе* анализа значения ***lasterror-***переменной производится *выбор* пути дальнейших вычислений. Последний пример фрагмента иллюстрирует возврат ошибочного сообщения, связанного с определенным соотношением *между* аргументами, для непоименованной процедуры пользователя. При этом, использованы как ***ERROR***-фунция, так и эквивалентное ей **error**-предложение.

```
> plot(x);
Plotting error, empty plot
> lasterror, lastexception;   ⇒ lasterror, lastexception
> ulibrary(8, `C:/AVZ/AGV\\VSV/Art\\Kr`, MKDIR);
Error, During delete of MKDIR - could not open help database
> 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, связанных, прежде всего, с графическими объектами (*а в общем случае со средствами* **GUI** *пакета*), как это весьма наглядно иллюстрирует предыдущий достаточно простой фрагмент. В свою очередь, это же *не позволяет* обрабатывать такого типа ошибки средствами пакета. При этом, другого типа *недоработки* ***Maple*** иначе как *«чехардой»* назвать и вовсе трудно. В частности, доступ к отсутствующему файлу или каталогу для релизов **6–9** пакета инициирует ошибочную ситуацию *"file or directory does not exist"***,** тогда как в релизе **10** возвращается несколько иная диагностика такой ситуации *"file or directory*, %1, *does not exist"***.** Диагностика ошибочной ситуации *"wrong number* (*or type*) *of parameters in function …*" релизов **6 – 9** пакета ***Maple*** изменена в релизе **10** на диагностику *"invalid input: …*". Серьезными причинами такие замены объяснить трудно. Указанные моменты потребовали очередной корректировки нашей *Библиотеки* [41,103], например, путем корректировки **try**-предложения, а именно:

```
try …..
……
catch  "file or directory does not exist": …
catch  "file or directory, %1, does not exist":  …… (введено дополнительно)
……....
end try
```

В других ситуациях использовался подобный подход либо корректировалась используемая процедурами диагностика с учетом новых реалий. Естественно, подобная практика не отвечает принципам создания качественного **ПО,** не говоря уже о самом престиже разработчиков подобных программных средств.

Так как механизм **ERROR-**функции (**error**-*предложения*) позволяет производить обработку ситуаций, ошибочных с точки зрения внутренней логики реализуемого процедурой алгоритма, т. е. *прогнозируемых* ошибок, то для этих же 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(6.4, 350, 10, 17);
   lasterror;
Error, (in WM) 4, [6.4, 350, 10, 17]
                                    4, [6.4, 350, 10, 17]
> 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-procedure` end proc: WM1(6.4,350,10,17);
                                    4, [6.4, 350, 10, 17]
> if  whattype(%) = exprseq  then  WM1(op(evalf(lasterror[2])))  end if;
                                End of WM1-procedure
> if  whattype(%%) = exprseq  then  WM1(op(evalf(%%[2])))  end if;
                                End of WM1-procedure
```

В приведенном фрагменте иллюстрируются эквивалентные конструкции по обработке передаваемых ***WM-***процедуре фактических аргументов**,** созданные 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 более предпочтительным.

Механизм возврата результата вызовов процедуры на основе **error-**предложения представляется весьма важным средством обработки особых ситуаций**,** связанных именно со *спецификой* самой процедуры, а не средств ***Maple-я***зыка. В этой связи пользователю пре-

доставляется возможность *организации* собственных алгоритмов обработки как ошибочных, так и особых ситуаций, связанных с алгоритмом вычислений или передаваемыми процедуре аргументами. Как это иллюстрирует следующий простой фрагмент**:**

```
VK := proc()
local k;
    if nargs = 0 then error "procedure call has no arguments"
    else for k to nargs do
            if whattype( args[ k ] ) ≠ 'float' then
                error "%-1 argument has a type different from `float`",k
            end if
        end do
    end if;
    "Procedure body"
end proc
> VK(6.4, 5.9, 10.2, 3, 17);
Error, (in VK) 4th argument has a type different from 'float'
> lasterror;   ⇒ "%-1 argument has type different from 'float'", 4
> VK(6.4, 10.2, 5.9, 3.9, 2.7, 17.7);   ⇒ "Procedure body"
```

Процедура ***VK*** программно обрабатывает ситуацию получения как *NULL*-значения, так и аргумента некорректного типа на основе **error**-предложения с возвратом соответствующей диагностики, которая вне тела процедуры может обрабатываться программно.

Особенности и полезные рекомендации по использованию функции ***ERROR*** (**error**-*предложения*) для организации дополнительных выходов из процедур достаточно детально представлены в наших книгах [12-14,39,41]. Здесь же мы лишь сделаем одно замечание относительно реализации функций {***RETURN***, ***ERROR***} и предложений {**return, error**}. В настоящее время данные средства попарно функционально эквивалентны, однако синтаксис их использования при создании процедур и программных модулей различен, а именно. В пакете ***Maple 5*** и *ниже* использовались функции ***RETURN*** и ***ERROR***, что в силу их концепции позволяло использовать их в *любой* точке, подобно обычным функциям. Во многих случаях это представляется весьма удобным. Однако, начиная с ***Maple 6***, пакет, одновременно допуская указанные функции и предложения, между тем, настоятельно рекомендует использовать все же последние, чей синтаксис *существенно* уже. Вероятно, возможен вариант исключения функций из последующих релизов, что создаст еще один аспект несовместимости релизов уже на уровне исходных текстов. Такой подход несколько настораживает и порождает два естественных вопроса, а именно: (**1**) почему при проектировании встроенного языка – *базовой компоненты пакета* – не были решены столь важные концептуальные моменты (*это наводит на мысль об отсутствии концептуальной целостности пакета*), и (**2**) если будут удалены указанные средства, то нарушится одно из *краеугольных* требований, которым должно удовлетворять качественное **ПО** – совместимость «*снизу-вверх*» на уровне встроенного языка пакета. Все это вызывает определенные недоумения и опасения со стороны пользователей.

Наконец, по *вызову* функции ***traperror*****(**<*Выражение*>**)** возвращается результат *вычисления выражения*, указанного ее фактическим аргументом**;** если при этом в процессе вычисления возникает ошибочная ситуация, то возвращается идентифицирующее ее диагностическое сообщение**,** доступное для последующей обработки и не нарушающее естественный порядок вычислений. Таким образом**,** ***traperror*****-**функция в *отличие* от функции

ERROR (**error-***предложения*) и результата стандартной обработки ошибочной ситуации не прекращает выполнения процедуры**,** а позволяет *"блокировать"* ошибочную ситуацию**,** предоставляя возможность более гибкой ее обработки. Простой фрагмент иллюстрирует вышесказанное**:**

```
H := proc()
local k, h, z;
    assign(z = 64 + sqrt(`+`(args))), assign(h = traperror(z/`+`(args)));
    if z = 0 then 'NULL – argument'
    elif h ≠ "numeric exception: division by zero'then return h
    else z/(59 + `+`(args))
    end if
end proc
> evalf([H(64, -59, 10, 17), H(64, 59), H(0, 0, 0)]);  ⇒ [2.176776695, 0.6104921668, 1.084745763]
```

В данном фрагменте ***H-***процедура использует ***traperror-***механизм для обработки *особой* ситуации**,** которая может приводить к аварийному завершению процедуры в период ее выполнения **-** *нулевая сумма* значений ее фактических аргументов инициирует ошибочную 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, что по ***tracelast-***команде языка можно получать полезную отладочную информацию, представляющую собой последнее содержимое *стэка* ошибок ядра пакета, как это иллюстрирует следующий достаточно простой фрагмент**:**

```
> Svegal:= proc(x) local y;  y:= 64;  1/(x - y)  end proc:   Svegal(64);
Error, (in Svegal) numeric exception: division by zero
> tracelast;
        Svegal called with arguments: 64
        #(Svegal, 2): 1/(x-y)
Error, (in Svegal) numeric exception: division by zero
        locals defined as: y = 64
> S:= proc(n::integer) local k, G; G:= 0: for k to 10^n do G:= G+1 end do end proc:  S(10);
Warning, computation interrupted
> tracelast;
        S called with arguments: 10
        #(S, 3): G := G+1
Error, (in S) interrupted
        locals defined as: k = 9089788, G = 9089787
```

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

Механизм try-предложения пакета. Для *программной* обработки ошибочных ситуаций, отражаемых в ***lasterror***-переменной пакета, ***Maple***-язык располагает встроенной функцией ***traperror,*** рассмотренной выше, и процедурами ***stoperror*** и ***unstoperror,*** наряду с **try**-предложением. Три первых средства остались в наследство от предыдущих релизов пакета (*заинтересованный читатель детально с ними может ознакомиться, например, в* [10-

12]), тогда как **try**-предложение явилось дальнейшим и более эффективным средством обработки ошибочных ситуаций. Данное средство впервые было включено в ***Maple 6.***

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

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

Формат предложения **try** указывает, что его блоки **catch** и **finally** необязательны. Более того, **catch**-блок имеет довольно простую структуру, а именно: "*Строка соответствия*"**:** <*последовательность* ***Maple****-предложений*>. Механизм выполнения **try**-предложения сводится вкратце к следующему. После получения управления **try**-предложение инициирует выполнение блока **A** предложений ***Maple***, в которых мы пытаемся локализовать ошибочные (*особые*) ситуации. Если в процессе выполнения **A**-блока не возникло таких ситуаций, то *выполнение* будет продолжено, начиная с ***Maple***-предложений **finally**-блока, если он был определен. Затем выполнение продолжается с *предложения*, следующего за закрывающей скобкой **`end try`** **try**-предложения.

В случае обнаружения исключительной ситуации в процессе выполнения **A**-блока его выполнение немедленно прекращается и производится *сравнение* каждой из **catch**-строк на соответствие возникшей *исключительной* ситуации (*данная ситуация отражается в переменных пакета* ***lasterror и lastexception***). Если такая **catch**-строка существует, то выполняется последовательность ***Maple***-предложений, соответствующая данному **catch**-блоку (*эти предложения расположены между ключевым словом* **catch** *и элементами* **try**-*предложения, а именно* (**1**) *следующий* **catch**-*блок*, (**2**) **finally**-*блок либо* (**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 также передает обработку исключительной ситуации вне **try**-предложения. Если исключительная ситуация передается вне **try**-предложения, то создается новый специальный объект, который повторяет имя текущей процедуры, текст диагностического сообщения, а также параметры *исходной* исключительной ситуации.

```
F := proc(x)
    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 sin(x)
        end if
    catch "Branch_1": Art(x)
    catch "Branch_2": Kr(x)
```

```
    catch "Branch_3": Sv(x)
    catch "Branch_4": Ar(x)
    end try
end proc
> seq(F(x), x = [42, 45, 64, 100, 350]);   ⇒ Art(42), Kr(45), Sv(64), Ar(100), sin(350)
```

Предложение **try** может содержать любое количество **catch**-блоков и в каждом из этих *блоков* выполняется соответствующая последовательность ***Maple***-предложений. Для иллюстрации вышесказанного и механизма активации **catch**-блоков рассмотрим реализацию некоторой *кусочно*-определенной функции посредством **try**-предложения (*предыдущий фрагмент*); в качестве *строк совпадения* **catch**-блоки используют диагностику по запрограммированным ошибкам, сформированным на основе предложений **error** и **return.** На наш взгляд, данный фрагмент достаточно прозрачно иллюстрирует механизм инициирования **catch**-блоков, обеспечивающий достаточно широкий спектр приложений **try**-предложения для обработки различного типа исключительных (*особых*) ситуаций, возникающих как естественным образом при выполнении некоторого вычислительного алгоритма или работы с внешними данными и т.д., так и созданных пользователем с целью обеспечения ветвления выполнения алгоритма в зависимости от тех или иных условий, заранее предусмотренных.

При нормальных условиях предложения **finally**-блока выполняются всегда, как только **try**-предложение потеряет управление. Данное поведение сохраняется даже, если предложения **catch**-блока обеспечивают обработку *исключительной* ситуации вне **try**-предложения, сгенерированной некоторой *новой* исключительной ситуацией, или выполнением любого из предложений **return, break, next** языка пакета. Если исключительная ситуация возникает в **catch**-блоке и она *распознана отладчиком* (***debugger***), то программа пользователя прекращает выполнение и предложения **finally**-блока не выполняются. Предложения **finally**-блока не выполняются и в том случае, если возникает одна из следующих исключительных ситуаций:

(**1**) вычисление превысило отведенный временной интервал (*эта ситуация может быть обработана только в том случае, если* ***timelimit****-функция, ограничивающая время вычисления, инициирует прерывание типа «time expired», которое может быть опознано и обработано*);
(**2**) вычисление *прервано извне* вычислительного процесса (*клавиши* **Ctrl+C**, **Break** *и т.д.*);
(**3**) возникла внутренняя системная ошибка;
(**4**) неудачное выполнение ***ASSERT***-функции или типификации *локальной* переменной;
(**5**) переполнение пакетного стэка.

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

- *строки соответствия* рассматриваются как префиксы текстов сообщений, генерируемых исключительными ситуациями (*переменные* ***lastexception*** *и* ***lasterror***);
- ни специальный объект, созданный ***ERROR***-функцией (**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.

Результатом выполнения **try**-предложения является результат выполнения его *последнего* ***Maple***-предложения. Таким образом, механизм **try**-предложения носит наиболее общий характер, обеспечивая возможность обработки, практически, любой исключительной ситуации, инициированной как ***ERROR***-функцией (**error**-предложением) на основе *конкретного* вычислительного алгоритма, так и другими средствами пакета. В нашей книгах [41,103] представлен целый ряд интересных применений **try**-предложения для обработки различного типа *ошибочных* и *особых* ситуаций. Использованные в них приемы могут оказаться весьма полезными для программирования приложений в ***Maple.***

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

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

где **Msg** и **pk –** полностью соответствуют описанию **error**-предложения, представленного в начале данного раздела с *очевидной заменой* диагностического типа сообщения **Msg** на *информационное*. В случае установки параметра ***warnlevel*=0** (*по умолчанию* ***warnlevel*= 3**) процедуры ***interface*** вывод сообщений не производится и 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(64, 59, 19.95, 10, 17, 19.99, 39, 44, 3.1, 95.99, 350);   ⇒ 722.03
Warning, Среди полученных значений оказалось 7 integer-типа
> interface(warnlevel = 0);  H(64, 59, 19.95, 10, 17, 19.99, 39, 44, 3.1, 95.99, 350);   ⇒ 722.03
```

Возможность вывода сообщений предоставляет удобный механизм информирования о ходе выполнения текущего документа, процедуры или модуля. Для возможности обработки сообщений, выводимых процедурой ***WARNING,*** нами была создана ее полезная модификация [103], обеспечивающая через глобальную ***_warning***-переменную возврат последнего в текущем сеансе сообщения, генерируемого ***WARNING,*** например:

```
> WARNING("Help database for library <%1> has been created",
   "C:/AVZ/AGN\VSV/Art\Kr");
Warning, Help database for library <C:/AVZ/AGNVSV/ArtKr> has been created
> _warning;
            "Help database for library <C:/AVZ/AGNVSV/ArtKr> has been created"
```

Данный подход позволяет 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 целый ряд весьма интересных процедур, использующих все рассмотренные здесь механизмы *возврата* результатов вызова, а также ряд нестандартных приемов в данном направлении.

3.7. *Расширенные средства Maple-языка для работы с процедурами*

Рассмотрим теперь вопросы использования механизма процедур несколько детальнее, учитывая прикладную значимость процедур для разработки различных приложений в ***Maple.*** Прежде всего, рассмотрим вопрос создания вложенных процедур, т.е. процедур, определения которых, в свою очередь, содержат определения *других* процедур**.** В целом ряде случаев данная возможность может оказаться достаточно эффективным средством при программировании прикладных задач. В общем случае поименованная процедура наряду с общепринятым допускает и определение посредством следующей (*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 внутри другой процедуры, но и внутри вычислительных конструкций, например**:**

> assign(A, proc() local k; sum(args[k], k=1..nargs) end proc); A(59, 64, 67); ⇒ 190
> [assign(G, proc(x, y) evalf(sqrt(x^2 + y^2)) end proc), G(42, 47)]; ⇒ [63.03173804]

Более того, как будет показано ниже, при использовании данных подходов к определению *подпроцедур* имеется принципиальное различие. *Первый* пример фрагмента иллюстрирует определение ***A*-**процедуры описанным способом с последующими ее вычислением и вызовом. Тогда как *второй* пример фрагмента иллюстрирует использование определенной вышеуказанным способом ***G*-**процедуры в вычислительной конструкции.

```
WM := proc(x::numeric, y::numeric, z::{10, 17})
    `if`(z = 10, 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
> [WM(64, 59, 10), AG(1995, 2006)];  ⇒ [[64, 59, 61.4492], [1995, 2006, 2000.49]]
WM1 := proc(x::numeric, y::numeric, z::{10, 17})
local A, G, V, AG, AV;
    if z = 10 then AG := proc(x, y) [x, y, evalf(sqrt(x×y), 6)] end proc ; AG(x, y)
    else AV := proc(x, y) [x, y, evalf(sqrt(x + y), 6)] end proc ; AV(x, y)
    end if
end proc
> [WM1(59, 64, 10), AG(64, 42)];  ⇒ [[59, 64, 61.4492], AG(64,42)]
WM2 := proc(x::numeric, y::numeric, z::{10, 17})
local A, G, V;
global AG, AV;
    if z = 10 then AG := proc(x, y) [x, y, evalf(sqrt(x×y), 6)] end proc
    else AV := proc(x, y) [x, y, evalf(sqrt(x + y), 6)] end proc
    end if;
    eval(parse("" || A || (`if`(z = 10, G, V)) || "(" || x || "," || y || ")"))
end proc
> [WM2(59,64,17), AV(59, 65), AG(64, 42)];  ⇒[[59, 64, 11.0905], [59, 65, 11.1355], AG(64,42)]
```

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

Первый пример фрагмента представляет определение вложенной ***WM-***процедуры, возвращающей результат вызова ***AG-***подпроцедуры, тело которой определяется на основе ***assign-***конструкции в зависимости от значения третьего фактического **z-**аргумента, передаваемого *внешней* процедуре. При этом, в текущем сеансе доступной является и ***AG-***подпроцедура, если ее идентификатор не декларируется во внешней процедуре как *локальная* переменная.

Второй пример представляет *вложенную* ***WM1-***процедуру**,** возвращающую результат вызова одной из подпроцедур (***AG***, ***AV***) в зависимости от значения третьего фактического **z-**аргумента, передаваемого *внешней* процедуре. При этом, для текущего сеанса обе подпроцедуры носят *локальный* характер**,** т.е. доступны лишь в рамках *содержащей* их внешней ***WM1-***процедуры**.** Наконец *третий* пример представляет вложенную ***WM2-***процедуру, *эквивалентную* предыдущей**,** но лишь с той разницей, что входящие в нее подпроцедуры ***AG***, ***AV*** определены глобальными. Результат последующих вызовов процедур ***WM2***, ***AG***, ***AV*** иллюстрирует вышесказанное.

Из сказанного следует вынести следующий важный вывод, иллюстрируемый табл. **8.** Если в случае *явного* декларирования идентификаторов подпроцедур область их действия соответствует декларации независимо от *способа* их определения, то в случае отсутствия декларации определенная стандартно и посредством ***assign-***процедуры подпроцедура становится соответственно *локальной* и *глобальной*.

Таблица 8

	Proc*:= *proc*() ... *end proc	***assign*(*Proc*, *proc*() ... *end proc*)**
global	***global***	***global***
local	***local***	***local***
не декларирована	***local***	***global***

Представленный в табл. **8** принцип декларирования идентификаторов подпроцедур в полной мере распространяется и на идентификаторы вообще, что весьма наглядно иллюстрирует следующий достаточно простой фрагмент**:**

```
> restart:  H:= proc(x) local h; h:= x^2 end proc:  [H(64), h];      ⇒ [4096, h]
> restart:  H:= proc(x) global h; h:= x^2 end proc:  [H(64), h];     ⇒ [4096, 4096]
> restart:  H:= proc(x)  h:= x^2 end proc:  [H(64), h];              ⇒ [4096, h]
Warning, `h` is implicitly declared local to procedure `H`
> restart:  H:= proc(x) local h; assign(h = x^2); h end proc:  [H(64), h];      ⇒ [4096, h]
> restart:  H:= proc(x) global h; assign(h = x^2); h end proc:  [H(64), h];     ⇒ [4096, 4096]
> restart:  H:= proc(x)  assign(h = x^2); h end proc:  [H(64), h];              ⇒ [4096, 4096]
```

Данное обстоятельство следует учитывать при практическом программировании, иначе в целом ряде случаев *его* игнорирование может быть причиной весьма *серьезных* ошибок, как отслеживаемых ядром пакета, так и семантических.

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, находящимся в окружающих их процедурах. Этот *аспект* лежит в

основе обеспечения механизма *инкапсуляции*, на котором базируется современное объектно-ориентированное программирование**.** При этом, для *глобальных* переменных поддерживается только режим их *разделения* вложенными процедурами.

Процедуры допускают любой *уровень* вложенности, определяемый только объемом памяти, доступной рабочей области пакета. Однако, доступность *внутренних* процедур в текущем сеансе определяется двумя моментами: (**1**) типом *внутренней* процедуры (*local*, *global*) и (**2**) ее режимом использования. Если внутренняя процедура определена *локальной* (*local*), то прямой доступ к ней 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* содержащей ее процедуры. И только после первого вызова главной процедуры все ее *внутренние* процедуры, определенные тем либо иным способом *глобальными*, становятся доступными в текущем сеансе *вне* содержащей их процедуры. Следующий фрагмент весьма наглядно иллюстрирует вышесказанное.

```
> P:= proc() local P1;  P1:= () -> `+`(args); P1(args) end proc:
> map(type, [P, P1], 'procedure');   ⇒ [true, false]
> P(64, 59, 39, 10, 17):  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(64, 59, 39, 10, 17):  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(64, 59, 39, 10, 17):  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(64, 59, 39, 10, 17): 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(64, 59, 39,10, 17):  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(64, 59, 39,10, 17):  map(type, [P, P1, P2], 'procedure');   ⇒ [true, false, true]
```

Таким образом, *вложенные* (*глобальные относительно содержащей ее главной процедуры*) процедуры становятся доступными в текущем сеансе только 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

P:= proc() global P; P:= proc() end proc; [args] end proc:

в качестве *вложенности* рассматриваться не может и представляет собой *исключительный* случай, как это наглядно иллюстрирует следующий простой 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(64, 59, 39, 10, 17), eval(P), P(64, 59, 39, 10, 17);   ⇒ [64, 59, 39, 10, 17], () -> `+`(args), 189
```

Следовательно, во избежание недоразумений *не* рекомендуется использовать в качестве имен *глобальных* переменных имена процедур, в которых они определяются.

В свете вышесказанного довольно полезной представляется процедура ***intproc*(P),** 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** (*или идентичную ей*) в качестве *внутренних/вложенных* процедур. Первый элемент обоих списков – *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 и примеры ее применения.

```
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, P3, P2]
> intproc(P1);   ⇒  [analogous, P], [innet, P3, P2]
> intproc(P2);   ⇒  [analogous], [innet]
> intproc(P3);   ⇒  [analogous], [innet]
> intproc(AGN);   ⇒  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 возвращает *FAIL*-значение по причине невозможности установить *истинное* определение символа (*возможно, процедура из библиотеки, 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.

Дальнейшее рассмотрение целесообразно начать с ***CompSeq*-**функции, позволяющей в определенной степени автоматизировать процесс создания процедур на основе последовательности ***Maple*-**предложений. Для организации *вычислительных последовательностей* (**ВП**) в виде *невычисляемых* конструкций (*своего рода макетов вычислительных блоков*) ***Maple*-**язык располагает специальной ***CompSeq*-**функцией, имеющей следующий формат кодирования**:**

***CompSeq*(*locals* = L1, *globals* = L2, *params* = L3, <*Список* ВК>)**

где в качестве первых трех необязательных аргументов выступают списки соответственно *локальных* (**L1**), *глобальных* (**L2**) переменных **ВП** и *параметров* (**L3**). Если локальные переменные областью своего определения имеют *только* 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 ***CompSeq*-**функции представляет собой *список вычисляемых конструкций* (**ВК**) вида **<*Id*-***переменной***> = <***Выражение***>;** последняя **ВК** последовательности и возвращает *окончательный* результат ее вычисления. При этом, **ВП** может быть упрощена, оптимизирована, а также конвертирована в форму процедуры, и наоборот. Следующий фрагмент иллюстрирует использование ***CompSeq*-**функции для создания **ВП,** затем конвертацию полученной конструкции в ***Maple*-**процедуру с ее последующим ее выполнением**:**

```
> 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)]);
```

$$GS := \mathrm{CompSeq}\left(locals = [x, y, z], globals = [X, Y, Z], params = [a, b, c], \left[x = a\sqrt{X^2 + Y^2 + Z^2}, y = \frac{(x+b)\,\mathbf{e}^{Z}}{\ln(X) + \sin(Y)}, z = \frac{x\,y}{a\,x + b\,y}, \frac{x\,y}{x+z} \right] \right)$$

```
> SG:= convert(GS, 'procedure');
```

$SG := \mathbf{proc}(a, b, c)$

local $x, y, z;$

global $X, Y, Z;$

$x := a \times (X\text{^}2 + Y\text{^}2 + Z\text{^}2)\text{^}(1/2);$

$y := (x + b) \times \exp(Z)/(\ln(X) + \sin(Y));$

$z := x \times y/(a \times x + b \times y);$

$x \times y/(x + z)$

end proc

```
> X:= 6.4:  Y:= 5.9:  Z:= 3.9:  evalf(SG(42, 47, 67), 12);   ⇒ 14613.2142097
```

Возможности ***CompSeq*-**функции позволяют описывать относительно несложный 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 упрощать решение многих прикладных, но достаточно простых задач.

Для обеспечения возможности вывода из процедур полезной для пользователя информации в *теле* процедуры можно помещать специальную встроенную функцию ***userinfo*,** имеющую следующий простой формат кодирования**:**

***userinfo*(<*Уровень*>, <*Id*>, <*Выражение*_1> {, ... , <*Выражение*_n>})**

и позволяющую в соответствии с указанным *уровнем* выводить *информацию*, представляемую ее аргументами, начиная с *третьего* (*обязательного*), для процедур, чьи идентификаторы задаются *вторым* аргументом функции, допускающим как отдельное имя, так и их множество. Вывод информации, получаемой в результате вычисления *выражений_***к**, производится только в том случае, если определенный первым аргументом функции *уровень* не превышает установленного в ***infolevel-***таблице для ее ***all-***входа. Исходным состоянием таблицы является: ***print*(*infolevel*); ⇒** table([*hints*=1])). Ее единственный ***hints-***вход определяет вывод результата вызова функции/процедуры *невычисленным*, если невозможно получить его точное значение. Вычисленные выражения ***userinfo-***функции выводятся в ***lprint-***формате, разделенные тремя пробелами.

Предложение ***infolevel*[*all*]:=*m*** определяет *общий* уровень вывода для всех последующих процедур, содержащих ***userinfo-***функцию. Соответствующая информация выводится только тогда, когда определяемый их первым аргументом *уровень* не превышает указанного в предложении ***infolevel*** уровня ***m***, например:

```
> infolevel[all]:= 10;    ⇒ infolevel[all] := 10
> P1:=proc() userinfo(5,P1, `Суммирование `||nargs||` аргументов`); `+`(args) end proc:
> P2:=proc() userinfo(12,P2,`Суммирование `||nargs||` аргументов`); `+`(args) end proc:
> P1(10, 17, 39, 44, 59, 64); restart:   ⇒ 233
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(10, 17, 39, 44, 59, 64); restart:   ⇒ 233
P1: Суммирование 6 аргументов
> P2(10, 17, 39, 44, 59, 64);   ⇒ 233
> print(infolevel);   ⇒ table([hints = 1, P1 = 6, P2 = 9])
```

В частности, библиотечные процедуры пакета используют следующие информационные уровни вывода: (**1**) **-** обязательная информация; (**2, 3**) **-** *общая* информация, включая метод решения проблемы, и (**4, 5**) **-** детальная информация по процедуре.

При отсутствии для процедур общего уровня вывода информации он определяется на основе индивидуальных ***infolevel-***предложений, кодируемых либо в *теле* самих процедур, либо вне их в виде ***infolevel*[<***Имя процедуры***>]:=** *Уровень*. По данному предложению соответствующая информация заносится в ***infolevel-****таблицу*, с которой работают функции ***userinfo*** процедур. *Последний* пример предыдущего фрагмента иллюстрирует сказанное. Средство ***userinfo-***функции довольно полезно для обеспечения пользователя *документированными* процедурами.

В случае определения выхода процедуры через ***RETURN-***функцию (**return-***предложение*) следует кодировать предложения ***infolevel*** и ***userinfo*** *перед* ним, ибо в противном случае действие последних подавляется, как это иллюстрирует следующий простой фрагмент:

```
> infolevel[all]:= 10;    ⇒ infolevel[all] := 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(10, 17, 39, 44, 59, 64);   ⇒ 233
P1: Суммирование 6 аргументов
```

```
> P2(10, 17, 39, 44, 59, 64); restart:   ⇒ 233
> 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(10, 17, 39, 44, 59, 64);  print(infolevel);   ⇒ 233        table([hints = 1])
> P2(10, 17, 39, 44, 59, 64);  print(infolevel);   ⇒ 233        table([hints = 1, P2 = 10])
P2:  Суммирование  6  аргументов
```

При этом, последние два примера фрагмента иллюстрируют необходимость *глобального* определения ***infolevel-***переменной, иначе она не редактирует соответственно ***infolevel-***таблицы 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 указанных процедур или функций в разрезе**:** *количество вызовов*, *временные издержки*, *требуемая оперативная память* и др. Информация выводится в табличном виде, смысл которой особых пояснений не требует. Для инициации режима мониторинга процедур/функций используется процедура ***profile*(P1, ..., Pn)**, которая в случае успешного вызова возвращает значение *NULL* и по которой устанавливается режим мониторинга вызовов **Pj-**процедур, определяемых ее фактическими аргументами. При этом, следует иметь в виду, что попытка вызова процедуры ***profile*** для уже находящейся в режиме ***profile-***мониторинга процедуры либо функции вызывает ошибочную ситуацию**,** как это иллюстрирует пример нижеследующего фрагмента. По вызову ***profile*()** производится мониторинг вызовов всех процедур и функций в текущем сеансе. Однако, в таком объеме ***profile-***средство рекомендуется использовать с большой осторожностью во избежание существенных *замедления* вычислений и *увеличения* используемой памяти, вплоть до *критического*.

Результаты мониторинга носят кумулятивный характер и их можно периодически выводить по ***showprofile*({ |P1,...,Pn})-**процедуре в *разрезе* или *всех* профилируемых процедур, или только относительно **Pj-**указанных в качестве фактических аргументов процедуры. По вызову ***unprofile*({|P1 ,..., Pn})-**процедуры 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 или функциям. Повторное применение ***unprofile-***процедуры вызывает ошибочную ситуацию. Следующий фрагмент иллюстрирует применение рассмотренных средств для мониторинга пользовательской ***VSV-***процедуры и некоторых встроенных функций языка.

```
> VSV:= proc() local k; product(args[k], k=1 .. nargs)  end proc:  profile(VSV);
> [VSV(10, 17, 39, 44, 59, 64), VSV(96, 89, 67, 62, 47, 42), VSV(k$k=1..9)];
                                [1101534720, 70060765824, 362880]
> showprofile(VSV);
function        depth   calls    time    time%       bytes   bytes%
-------------------------------------------------------------------------------
VSV                 1         3   0.000    0.00       23896   100.00
-------------------------------------------------------------------------------
total:              1         3   0.000    0.00       23896   100.00
> profile(VSV);
Error, (in profile) VSV  is already being profiled.
> profile(sin, exp);  [sin(6.4), sin(4.2), exp(0.59), sin(17)*exp(10)]:  showprofile();
```

```
function        depth   calls    time    time%       bytes   bytes%
-------------------------------------------------------------------------------
sin                 1       3    .016   100.00       16172    35.00
exp                 1       2   0.000     0.00        6140    13.29
VSV                 1       3   0.000     0.00       23896    51.71
-------------------------------------------------------------------------------
total:              3       8    .016   100.00       46208   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 назна-чению к ***profile-***процедуре является и ***exprofile-***процедура, обеспечивающая мониторинг всех вызовов пакетных процедур и функций. С другими типами *мониторинга* различных *аспектов* выполнения процедур и/или функций в среде ***Maple-***языка можно ознакомиться в [31, 33,43,84,103]. Рассмотренные средства мониторинга предоставляют полезную информацию, в частности, для оптимизации вычислений.

Создав собственную серьезную процедуру с использованием других своих и пакетных процедур и функций, поместив ее в свою библиотеку, естественно возникает задача ее оптимизации, в частности, с целью раскрытия *частоты* использования средств, содержащихся в ней, и основных компьютерных ресурсов, используемых ими. В этом контексте, проблема оптимизации пользовательских процедур весьма актуальна. Для этих целей достаточно полезной представляется процедура ***StatLib*(L)** [103]**,** обеспечивающая сбор основной статистики по заданной **L-**библиотеке и возврату статистики для последующего анализа. В процессе своего выполнения ***StatLib-***процедура требует некоторых *дополнительных* ресурсов памяти и времени. Подробнее о ней будет идти речь при рассмотрении создания библиотек пользователя.

В заключение настоящего раздела отметим, что более искушенный пользователь может при работе с процедурами воспользоваться *специальными* двумя процедурами ***procbody*** и ***procmake.*** По вызову процедуры ***procbody*(*P*)** возвращается специальная *невычисленная* форма процедуры с ***P-***именем, которая может быть протестирована и *модифицирована*, не затрагивая исходной ***P-***процедуры. *Обратной* к ней является процедура ***procmake*(G),** возвращающая на основе специальной **G-**формы выполняемую ***Maple-***процедуру. Следующий простой фрагмент иллюстрирует вышесказанное**:**

```
> AGN:= proc() `+`(args)/nargs end proc:  AGN1:= procbody(AGN);  procmake(AGN1);
> AGN1:= procbody(AGN);  procmake(AGN1);
```

$$AGN1 := \&\mathrm{proc}\left(\&\mathrm{expseq}(\ ), \&\mathrm{expseq}(\ ), \&\mathrm{expseq}(\ ), \&\mathrm{expseq}(\ ), \frac{+\ \&\mathrm{function}\,\&\mathrm{expseq}(\&args_{-1})}{\&args_0}, \&\mathrm{expseq}(\ ), \&\mathrm{expseq}(\ ), \&\mathrm{expseq}(\ ) \right)$$

proc () `+`(args)/nargs **end proc**

Использование данных функций предполагает достаточную искушенность пользователя по работе в среде ***Maple-***языка, поэтому за детальной информацией заинтересованный читатель отсылается к интересным работам, цитированным в книгах [12,13].

Отметим теперь некоторые принципиальные *новации* относительно процедур, появившиеся в последнем релизе **10**. Во-первых, как правило, количество передаваемых процедуре при вызове фактических аргументов не обязательно должно совпадать с количеством ее формальных аргументов. Однако, если при определении процедуры в конце ее формальных аргументов закодирован символ-маркер «**$**», то передача процедуре при вызове дополнительных фактических аргументов вызывает ошибочную ситуацию с диагностикой "invalid input: %**1** arguments passed to %**2** but only %**3** positional parameters specified", как это весьма наглядно иллюстрирует следующий простой фрагмент:

```
> P:= proc(x, y, z, $) `+`(args) end proc:  P(64, 59, 39), P(64, 59);     # Maple 10   ⇒ 162, 123
> P(64, 59, 39, 10, 17);
Error, invalid input: 5 arguments passed to P but only 3 positional parameters specified
> lasterror;
   "invalid input: %1 arguments passed to %2 but only %3 positional parameters specified"
> P:= proc(x, y, z, $) `+`(args) end proc: P(64, 59, 39), P(64, 59);     # Maple 6 – 9
Error, `$` unexpected
> lasterror;   ⇒ lasterror
```

Тогда как в предыдущих релизах **6 – 9** данная новация вызывает ошибочную ситуацию уже на уровне синтаксиса, *не отображаясь* в ***lasterror***-переменной. Принципиально, данная новация не столь уж существенна, добавляя пакету несовместимость «*сверху-вниз*» (*в принципе, каноны программирования это допускают, но все же*). Тем более, что контроль за передаваемыми процедуре фактическими аргументами легко осуществляется программно и на основе сути реализуемого ею алгоритма с использованием ***nargs***-переменной.

Дополнительно к двум процедурным переменным ***args*** и ***nargs,*** рассмотренным выше, в релизе **10** введен ряд дополнительных процедурных переменных, а именно:

_passed – алиас для переменной ***args***
_params – последовательность *продекларированных* позиционных аргументов, переданных процедуре
_options – последовательность опций в процедуре
_rest – последовательность недекларированных аргументов, переданных процедуре
_npassed – алиас для переменной ***nargs***
_nparams – число *продекларированных* позиционных аргументов, переданных процедуре
_noptions – число опций в процедуре
_nrest – число недекларированных аргументов, переданных процедуре
_nresults - число предполагаемых выходов из процедуры

Смысл их достаточно прозрачен уже из приведенного описания, а детальнее с ними можно ознакомиться по конструкции **?args,** тогда как примеры их применения представляет нижеследующий простой фрагмент:

```
> P:=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],_options,_rest,_nparams,_noptions,_nrest,_nresults]
  end proc:  P(64, 59, 39);   ⇒ [[64, 59, 39], 3, 0, 0, undefined]
> P(64,59,39,10,17);   ⇒ [[64, 59, 39], 10, 17, 3, 0, 2, undefined]
> C:=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], _options, _rest,_nparams, _noptions, _nrest, _nresults]  end proc:
> C(64, 59, 39);   ⇒ [[64, 59, 39], 3, 0, 0, undefined]
> C(64, 59, 39, 10, 17);   ⇒ [[64, 59, 39], 10, 17, 3, 0, 2, undefined]
```

При программировании процедур вышеперечисленные *переменные* в ряде случаев могут упрощать программирование, однако порождают несовместимость «*сверху-вниз*».

3.8. *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 преобразования, расширять диапазон математических функций, обрабатываемых функциями ***evalf*** и ***diff***, расширяя тем самым средства главной ***Maple***-библиотеки. Каждое уникальное расширение средства (*типа* ***type*** *либо* ***diff***) ассоциируется с *именем*. Это *имя* определяет имя *расширяемого* средства, например, для ***type*** это – ***type***. Существуют *два* механизма расширения: *классический* механизм расширения, используемый в большинстве случаев, и *современный* механизм расширения, используемый *более* новыми средствами, например, 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 позволяет именовать 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 расширений можно ознакомиться по запросу **?extension,** здесь же мы вкратце рассмотрим классический механизм расширений средств пакета, носящий глобальный характер

Все *встроенные* процедуры и большинство *библиотечных* процедур, допускающие пользовательские расширения, используют *классический* механизм, основанный на конкатенации имен, а именно. Расширяемой процедуре ***name*** дают *новые* функциональные возможности, определяя выражение (*как правило, процедура*) с именем формата **`*name/new*`**, где ***new*** – имя расширения. Например, новый тип для *бинарных* выражений (*распознаваемый* ***type****-функцией*) может быть определен процедурой с *именем* **`*type/binary*`**. При этом, вводимые расширения для стандартного средства ***name*** должны удовлетворять определенным правилам, которым удовлетворяет данное средство ***name***, и которые отражены в справке по данному средству. Например, при *расширении* стандартной ***type***-функции процедурой с именем **`*type/aa*`** требуется, чтобы она возвращала только значения {***true***, ***false***}, в противном случае инициируется ошибка с диагностикой "result from type `%1` must be true or false", например:

```
> `type/aa`:=proc(x::numeric) if x<0 then true elif x>0 then false else FAIL end if end proc:
> type(0, 'aa');
Error, result from type `aa` must be true or false
> lasterror;   ⇒ "result from type `%1` must be true or false"
```

В качестве примера *расширения* стандартной ***type***-функции определим новый тип ***color***, отсутствующий в пакете текущих релизов и достаточно полезный для большого числа задач, имеющих дело с графическими объектами. Вызов процедуры ***type*****(x, 'color')** возвращает значение ***true***, если **x** – имя цвета, процедура либо выражение, которые можно рассматривать в качестве цвета при формировании графического объекта; в *противном* случае возвращается ***false***-значение. При этом, через глобальную ***_ColorType***-переменную возвращается **2**-элементный *бинарный* список, если вызов процедуры ***type*****(x, 'color')** возвращает ***true***-значение. Его первый и второй элементы определяют допустимость **x**-выражения в качестве цвета (**0** – *нет*, **1** – *да*) при оформлении графического объекта с опцией **color = x** в случае размерности **2** и **3** соответственно. Если же вызов процедуры

возвращает ***false***-значение, то переменная ***_ColorType*** возвращается неопределенной. Процедура `***type/color***` имеет целый ряд 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 приведены исходный текст процедуры **`type/color`** и примеры ее применения.

```
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(RGB(0.64, 0.59, 0.39)), 'color'), _ColorType; ⇒ *true*, [1, 1]

> type(RGB(0.64, 0.59, 0.39), 'color'), _ColorType; ⇒ *true*, [0, 1]

> type([sin(x*y), cos(x*y), tan(x*y)], 'color'), _ColorType; ⇒ *true*, [0, 1]

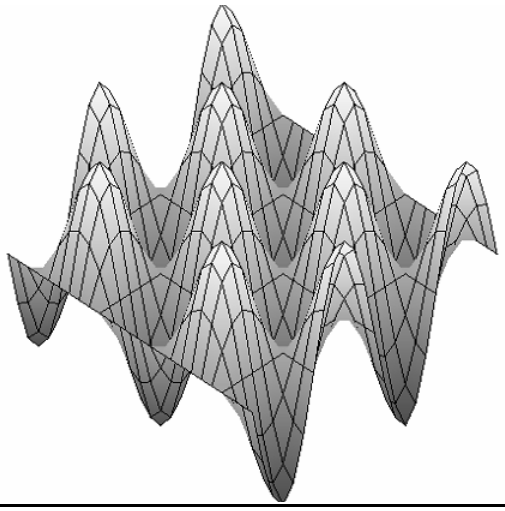
> type(AGN(0.64, 0.59, 0.39), 'color'), _ColorType; ⇒ *true*, [0, 1]

> type([1.64, 0.59, 0.39], '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 = [1.64, 0.59, 0.39]);



Читателю рекомендуется рассмотреть используемый процедурой *прием*, положенный в *основу* алгоритмов *тестирования*, который может быть полезен как для создания средств тестирования других графических опций, так и в целом ряде других задач. Ниже представлен ряд других примеров применения *классическго* механизма расширения средств ***Maple***, которые рекомендуется рассмотреть в качестве довольно *полезного* упражнения.

```
type/float_list := G → `if`(type(G, 'list'),
    `if`(member(false, {op(map(type, G, 'float'))}), false, true), false)
> type([6.4, 5.9, 3.9, 2.6, 17.6, 10.3], 'float_list');  ⇒ true
> type([6.4, 5.9, 3.9, 2.6, 17.6, 10.3, 2006], 'float_list');  ⇒ false
type/complex1 := proc(Z::anything)
local `1`, `2`, a, b, c, d, h, t, k, rm, im, sv, tn;
option `Copyright (C) 2004 by the International Academy of Noosphere. All rights \
 reserved.`;
    assign(`1` = cat("", convert(interface('imaginaryunit'), 'string'))),
        assign(`2` = [cat("*", `1`), cat(`1`, "*"), `1`]);
    if map2(search, convert(Z, 'string'), {op(`2`)}) = {false} then false
    else
        assign(h = interface(warnlevel), a = normal(evalf(Z))),
            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(Z, '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, [-3.65478, 64, 2006, -1995, 10/17], 'complex1');   ⇒ [false, false, false, false, false]
> type(2004, complex1(numeric));   ⇒ false
> type([a, b] - 3*I, complex1({realnum, list}));   ⇒ true
> type(-3.67548, 'complex'), type(-3.67548, 'complex1');   ⇒ true, false
> assume(a, 'integer'); type((a-I)*(a+I), complex(algebraic)), type((a-I)*(a+I),
  complex1(algebraic));   ⇒ true, false
> type((10 - I)*(10 + I), 'complex'), type((10 - I)*(10 + I), 'complex1');   ⇒ true, false
> type(8 + I^2, complex(algebraic)), type(8+I^2, complex1);        ⇒ true, true
> type(8 + I^2, complex1(algebraic)), type(8 + I^2, 'complex1');   ⇒ false, false
> type(64,'complex'),type(64,'complex1'),type((3+4*I)/(5+6*I),'complex1');  ⇒ true, false, true
```

```
convert/uppercase := proc(S::{string, symbol})
local k, h, t, G, R;
    assign(G = "", R = convert(S, 'string'));
    for k to length(R) do G := cat(G, op([assign('t' = op(convert(R[k], 'bytes'))), `if`(
        t ≤ 255 and 224 ≤ t or t ≤ 122 and 97 ≤ t, convert([t − 32], 'bytes'),
        `if`(t = 184, "¨", R[k]))]))
    end do;
    convert(G, whattype(S))
end proc
```

```
> convert("Russian Academy of Natural Sciences - 20.09.2006", 'uppercase');
        "RUSSIAN ACADEMY OF NATURAL SCIENCES - 20.09.2006"
```

```
&ma := proc()
    op(map(assign, [seq(args[k], k = 1 .. nargs − 1)],
    `if`(args[-1] = _NULL, NULL, args[-1])))
end proc
```

```
> x, y, z:= 64;
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', 2006); x, y, z, g, h; ⇒ 2006, 2006, 2006, 2006, 2006
> ('x', 'y', 'z', 'g', 'h') &ma _NULL; x, y, z, g, h;
> ('x', 'y', 'z', 'g', 'h') &ma 2006; x, y, z, g, h; ⇒ 2006, 2006, 2006, 2006, 2006
> ('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;

$$\frac{a+b}{c-d}, \frac{a+b}{c-d}, \frac{a+b}{c-d}, \frac{a+b}{c-d}, \frac{a+b}{c-d}$$

Первый пример фрагмента представляет *расширение* тестирующей ***type*-**функции на новый тип *float_list*, определяющий списочную структуру с элементами *float*-типа. С этой целью идентификатору **`*type/<Id>*`** присваивается *определение* процедуры, вызываемой в тот момент, когда делается попытка протестировать произвольное выражение посредством вызова ***type*(***Выражение*, *float_list*)**.** В остальных случаях применяется вызов встроенной ***type*-**функции пакета. Представленная процедура тестирует, в первую очередь, на соответствие типа *выражения* типу *list* и в случае наличия такого соответствия на втором этапе тестирует на соответствие типа *каждого* элемента списка *float*-типу, возвращая в зависимости от *наличия/отсутствия* такого соответствия *true/false*-значение.

Во *втором* примере определяется модификация стандартного *complex*-типа. Вызов стандартной функции ***type*(Z, *complex*)** возвращает *true*-значение, если **Z** – выражение формы **x + y*I,** где **x** (*если существует*) и **y** (*если существует*), конечны и типа 'realcons'. При этом, вызов процедуры ***type*(X, *complex*(t))** возвращает *true*-значение, если **Re(X)** (*если существует*) и **Im(X)** (*если существует*) – оба типа **t**. К сожалению, стандартная процедура не обеспечивает, на наш взгляд, корректного тестирования ***Maple*-**выражений упомянутого *complex*-типа, включая в него и действительные выражения. Следующий простой пример убедительно подтверждает вышесказанное, а именно:

> map(type, [-9.1942, 64, 350, -2006, 10/17], 'complex'); ⇒ [*true*, *true*, *true*, *true*, *true*]

К сожалению, эта 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, процедура не различает числовые типы {*fraction*, *rational*}, идентифицируя их более общим *numeric*-типом. В отличие от стандартной процедуры, процедура ***'type/complex1'*** обеспечивает более корректную проверку выражений ***Maple*** на *complex*-тип. Приведенные примеры иллюстрирую применение как стандартной, так и ***'type/complex1'*-**процедуры.

В *третьем* примере определяется расширение ***convert*-**функции ***Maple*-**языка на случай конвертации символов строчного значения в их *заглавный* эквивалент**.** В данном случае вызов функции ***convert*(`*Строка*`, *'uppercase'*)** инициирует вызов **`*convert/uppercase*`-**процедуры, обеспечивающей соответствующую конвертация указанной *строки*.

В конструкции **lhs: = rhs** оператор *назначения* **`:=`** присваивает **lhs** выражение **rhs**. Кроме того, оператор назначения допускает многократные назначения. Однако, в этом случае количество элементов последовательности **lhs** должно строго соответствовать количеству элементов последовательности **rhs,** иначе возникает ошибка с диагностикой "Error, ambiguous multiple assignment". Тогда как в ряде случаев возникает необходимость назначения *того же самого* выражения достаточной длинной последовательности имен или вызовов функций.

Данная проблема решается оператором **&ma,** который имеет *идентичный* с оператором **`:=`** приоритет. Оператор **&ma** имеет два формата кодирования, а именно: *процедурный*

и *операторный*. Вообще говоря, в обоих случаях элементы **lhs** должны быть закодированы 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,** то в качестве **rhs** кодируется ***_NULL***-значение. Успешный вызов процедуры **&ma** либо применения оператора **&ma** возвращает *NULL-значение* с выполнением указанных назначений. В целом ряде приложений оператор **&ma** оказывается достаточно полезным.

Наконец, последний пример фрагмента иллюстрирует определение пользовательского оператора **&ma** [103], обеспечивающего *многократные* присвоения одного и того же выражения переменным либо вызовам функций. На данном вопросе имеет смысл остановиться отдельно, учитывая его важность для практического программирования в среде пакета.

Наряду со стандартно определяемыми, ***Maple-***язык 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 следующих символов**:**

& | ( ) [ ] { } ; : ‘ ` # % \ *пробел* *перевод строки*

Длина **&-***цепочки* символов не должна превышать **495**. Сам ***Maple-***язык использует такого типа оператор **&*** для представления некоммутативного произведения, тогда как все другие идентификаторы **&<*символ*>** описанного формата рассматриваются пакетом 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 указанных случаев производится вызов процедуры, чье определение имеет следующий вид**:**

`&<*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, ...)**. Более того, ***Maple-***язык не накладывает на пользовательские операторы специальной семантики и рассматривает идентификатор оператора в качестве имени сооответствующей пользовательской процедуры. В общем случае, пользовательский оператор является **n-***арным префиксным* оператором либо вызовом **n-**арной функции, т.е. следующие две конструкции эквивалентны в среде ***Maple-***языка, а именно**:**

$$\&<Символ>\ (x_1, x_2, ..., x_n) \equiv \&<Символ>(x_1, x_2, ..., x_n) \qquad (n \ge 1)$$

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)];
```

$$\left[-\frac{(a-b\,I)\,I}{c+d\,I}-h\,I, -\frac{(a-b\,I)\,I}{c+d\,I}-h\,I\right]$$

```
> `&Kr`:= proc(x::numeric, y::numeric)  evalf(sqrt(x*y)/(x + y) + sin(x*y))  end proc:
> [1942  &Kr  2006, &Kr(1942, 2006)];   ⇒ [1.490076388, 1.490076388]
> `&Art`:= proc() sqrt(product(args['k'], 'k'=1..nargs))/sum(args['k'], 'k'=1..nargs) end proc:
> [&Art  (59, 64, 39, 10, 17, 44), &Art(59, 64, 39, 10, 17, 44)];  ⇒
```

$$\left[\frac{16\sqrt{4302870}}{233}, \frac{16\sqrt{4302870}}{233}\right]$$

Первый пример представляет *унарный префиксный* **&Cs-**оператор, применение которого к комплесному числу возвращает сопряженное ему число. Второй пример представляет бинарный инфиксный **&Kr-**оператор, определенный над двумя *числовыми* значениями и возвращающий значение, вычисляемое по указанной формуле. Наконец, третий пример представляет **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 ему процедуры. Таким образом**,** в среде ***Maple-***языка пользовательский (*нейтральный*) **&-**оператор представляется вызовом соответствующей процедуры. При этом, *инфиксная* нотация допустима лишь для случая *двух* операндов, тогда как *префиксная* – для *любого* числа операндов**.** Описанный метод определения *пользовательских* операторов с учетом механизма процедур ***Maple-***языка довольно прозрачен и имеет важное прикладное значение, позволяя вводить собственные **&-**операторы для специальных операций**.** В качестве одного из таких приложений рассмотренного метода уже был приведен пример **&ma** оператора, приведем еще пару 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("inc\
        orrect 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);   ⇒ Ar(x+a, y+b, z+c, t+d, u+g, r+s, h+q)
> &Shift (Ar, k, x, y, 64, 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
```

Вышеприведенный фрагмент иллюстрирует применение описанного способа для реализации пользовательского оператора сдвига **`&Shift`** для функции от нескольких переменных, определяемого следующим соотношением**:**

$$\&Shift\ (G, [h_1, h_2, ..., h_n], x_1, x_2, ..., x_n) \Rightarrow G(x_1 + h_1, x_2 + h_2, ..., x_n + h_n)$$

Его первый операнд определяет *имя* функции, второй **-** 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 данного пользовательского **&-**оператора, иллюстрируется пример анализа операндов, над которыми определен оператор, на корректность. Предыдущий пример представляет определение оператора и некоторые результаты его применения для реализации указанного выше оператора *функционального сдвига* **&Shift.**

Тогдда как второй пример фрагмента представляет процедуру ***lop3*(O, x, y),** поддерживающую следующие четыре операции (**O**) 3-значной логики над переменными (**x, y**) из множества {0, 1, 2}, а именно: ***Inv*** – инверсия, ***Dis*** – дизъюнкция, ***Con*** – конъюнкция и ***Webb*** – функция Вебба. Данная процедура может быть довольно несложно расширена и на другие операции 3-значной логики (*а в общем случае и* **к**-*значной*), что позволит использовать ее для представления любых многозначных функций алгебры логики. Читателю в качестве весьма полезного упражнения рекомендуется**,** используя описанный метод**,** 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.

В завершении раздела целесообразно сделать *одно* полезное в практическом отношении замечание. Определения процедур можно объединять в модули, помещая их в качестве элементов (*имя* **-** *вход*, *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([

$$Ds = (\textbf{proc}()\ \textbf{local}\ k;\ \mathrm{sum}((\mathrm{args}[k] - \mathrm{Sr}(\mathrm{args}))\text{^}2,\ k = 1\ ..\ \mathrm{nargs})/\mathrm{nargs}\ \textbf{end proc}),$$

$$Sr = \left((\ ) \to \frac{\text{`+`}(\mathrm{args})}{\mathrm{nargs}}\right),\ H = ((\ ) \to \sqrt{\text{`*`}(\mathrm{args})}),$$

$$G = ((\ ) \to \text{`+`}(\mathrm{args})^2)$$

])

```
> with(PT); PT[G](64, 59, 39, 10, 17, 44), Sr(64, 59, 39, 10, 17, 44), PT[H](64, 59, 39, 10, 17, 44),
  PT[Ds](64, 59, 39, 10, 17, 44);
```

$$[Ds, G, H, Sr]$$

$$54289, \frac{233}{6}, 16\sqrt{4302870}, \frac{14249}{36}$$

Данный 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 [8-14,29,41,103].

3.9. *Иллюстративные примеры оформления Maple-процедур*

Представленное выше описание структурной организации ***Maple-****процедуры* иллюстрируется нижеследующими примерами**,** отражающими основные ее элементы и принципы**,** что позволяет непосредственно приступать с учетом ранее рассмотренного материала к созданию, на первых порах, относительно несложных пользовательских процедур различного назначения. В *основу* иллюстративных примеров положим ряд процедур из нашей *Библиотеки* [41,103,109], содержащих практически полезные приемы программирования в среде ***Maple-****языка*. Наряду с этим, примеры представляют процедуры, имеющие непосредственный прикладной интерес при программировании приложений.

В ряде случаев для *упрощения* разработки ***Maple-***приложений (*например*, *для обеспечения их совместимости относительно релизов*) мы вынуждены *одновременно* открывать несколько релизов в текущем сеансе ***Windows.*** В этой связи вполне естественным представляется вопрос программного тестирования наличия релизов ***Maple*** в *текущем* сеансе работы с операционной средой. Данная задача решается процедурой ***mapleacs,*** исходный текст и примеры применения которой представляет следующий фрагмент. Успешный вызов процедуры ***mapleacs*()** возвращает вложенный список, подсписки которого имеют {2|3} элемента в зависимости от релиза пакета и его клона.

```
mapleacs := proc()
local a, b, c, d, q, h, k, p, t, r, v;
    assign( a = interface( warnlevel ),
        b = cat( [ libname ][ 1 ][ 1 .. 2 ], "\$$$Avz_Agn$$$"), h = [ ], p = [ ] );
    assign67( interface( warnlevel = 0 ) ), com_exe2( { `tlist.exe` } ),
        system( cat( "tlist > ", Path( b ) ) );
    do
        d := readline( b );
        if d = 0 then break
        elif Search2( Case( d ),
        map( cat, { 'maplew8', 'wmaple', 'maplew', 'cwmaple9', 'java' }, `.exe` ) ) ≠ [ ]
        then h := [ op( h ), d ]
        end if
    end do;
    for k in h do
        if search( k, "java.exe", 't' ) then
            if not search( k, "[Server ", 'q' ) then
                p := [ op( p ), [ 'Maple9s', [ Suffix( k[ t + 8 .. -1 ], " ", v, del ), v ][ 2 ] ] ]
            else p := [ op( p ), [ 'Maple9s',
                cat( ``, [ Suffix( k[ t + 8 .. q – 4 ], " ", v, del ), v ][ 2 ] ),
                cat( ``, k[ q + 1 .. -2 ] ) ] ]
            end if
        else
            search( k, "-", 't' ), assign( 'd' = SLD( Red_n( k[ 1 .. t – 3 ], " ", 2 ), " " ) ),
                assign( 'd' = cat( d[ 3 ], d[ 4 ] ) );
```

```
            assign('r' = k[t + 2 .. -1]), `if`(d[-1] = "6",
                assign67('r' = r[2 .. -2], 'q' = NULL), [
                search(r, "[Server ", 't'), assign('r' = r[2 .. t - 4], 'q' = r[t + 1 .. -3])
                ]);
            p := [op(p), map(convert, [d, r, q], symbol)]
        end if
    end do;
    null(interface(warnlevel = a)), fremove(b), p
end proc
> mapleacs();  ⇒ [[Maple7, uplib.mws, Server 1], [Maple8, mapleacs.mws, Server 1], [Maple6,
  MkDir.mws], [Maple9, helpman.mws, Server 1], [Maple10, Lprot.mws, Server 1]]
> mapleacs();  ⇒ [[Maple8, mapleacs.mws, Server 1], [Maple8, ProcLib_6_7_8_9.mws, Server 1]]
```

Первый элемент такого подсписка определяет *релиз* и его *клон*. Например, *Maple9s* определяет релиз ***Maple 9*** для ядра ***'maplew9.exe'*** (*стандартный режим*) и *Maple9* для ядра пакета ***'cwmaple9.exe'*** (*классический режим*). Тогда как *второй* элемент подсписка определяет имя либо полный путь к текущему ***Maple***-документу соответствующего релиза, тогда как *третий* элемент (*при его наличии*) подсписка определяет номер сервера. Процедура ***mapleacs*** представляется достаточно полезным средством при нескольких *одновременно* загруженных релизах (*могут быть и одинаковыми*) в текущий сеанс ***Windows.***

В качестве еще одного примера приведем процедуру ***`convert/proc`***, иллюстрирующую как подход к расширению стандартных средств пакета, так и полезную в практическом отношении. Процедура обеспечивает *конвертирование* равенства или их списка/множества в процедуру или список/множество. При этом, левые части равенств должны быть функциями вида **f(x, y, z, ...),** тогда как правые – 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. Следующий фрагмент представляет *исходный* текст процедуры и примеры ее применения.

```
convert/proc := proc(x::{equation, set(equation), list(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(64, 59)), convert(a = b, `proc`);  ⇒ 87.04596487, 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`*** полезна в целом ряде *приложений*, например, при работе с дифференциальными уравнениями. В качестве достаточно полезного упражнения рекомендуется рассмотреть организацию этой 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 ряде случаев и нестандартных*) приемов программирования в ***Maple,*** можно найти в *архиве*, поставляемом с книгой [41, 103] либо бесплатно скачать архив с *Библиотекой* с адреса, указанного в [109].

В целом ряде случаев возникает необходимость определения имен, которым в текущем сеансе были просвоены заданные выражения. 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(  ), "/$Art_Kr$" ) );
    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( [ "
            " = 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( [ "
                " = NULL ], readbytes( f, ∞, '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:=64: b:=64: c:= "ransian64": c9:= "ransian64": d:= table([1=64, 2=59, 3=39]): `): 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, [64, "ransian64", table([1=64, 2=59, 3=39]), svegal, 19.42, [x, y, z], a+b*I,
  (a1+b1)/(c1+d1), cos(x), proc () `+`(args)/nargs end proc, x..y]), nvalue(); ⇒ [{a,b}, {c,c9}, {d},
  {h}, {t47, t42}, {L}, {R, B}, {Z}, {f}, {g}, {r, n, m}], {Lasnamae, Tallinn, Grodno, Vilnius, Moscow}
```

Задача решается процедурой, базирующейся на стандартной процедуре ***anames*** с использованием некоторых особенностей ***Maple*** [41,103,109]. Вызов процедуры ***nvalue*(v)** возвращает список глобальных имен, которым в текущем сеансе было присвоено выраже-

ние **v,** 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 таких назначений не производилось.

При наличии в текущем сеансе ***Windows*** нескольких активных ***Maple***-сеансов в целом ряде случаев возникает необходимость их идентификации. Решает эту задачу ***Kernels***-процедура, чей вызов ***Kernels*()** возвращает последовательность 2-элементных списков, первый элемент который определяет ID процесса и второй – соответствующий ему активный ***Maple***-сеанс, точнее загруженное ядро этого сеанса. При этом, порядок элементов последовательности соответствует порядку загрузки ***Maple***-сеансов в ***Windows.***

```
Kernels := proc()
local a, b, c, d, h, t, k;
    assign('c' = "$ArtKr$", 't' = interface(warnlevel)),
        assign('a' = system(cat("tlist.exe > ", c)));
    interface(warnlevel = 0), com_exe2({`tlist.exe`}), interface(warnlevel = t);
    assign67(b = fopen(c, 'READ'), d = NULL, h = "maple");
    do
        a := readline(c);
        if a = 0 then break
        else
            if search(Case(a), h) then
                k := SLD(Red_n(a, " ", 2), " ");
                d := d, `if`(search(k[2], "maple"), [came(k[1]), k[2]], NULL)
            end if
        end if
    end do;
    d, delf(c)
end proc
> Kernels();  ⇒ [3404, "maplew8.exe"], [2068, "cwmaple.exe"], [2692, "maplew.exe"],
  [2736, "cwmaple9.exe"], [2464, "wmaple.exe"]
```

Вызов процедуры ***Release*()** возвращает целое число 6, 7, 8, 9, 9.5, 10 или 11 для семи последних релизов пакета и значение`*Other release*` в противном случае. Если же вызов процедуры ***Release*(h)** использует необязательный **h**-аргумент (*имя*), через него возвращается полный путь к главному каталогу пакета, как иллюстрирует следующий фрагмент. В отличие от ***Release,*** вызов процедуры ***Release1*()** возвращает число 6 для **6**-го релиза и 7 для релизов **7 – 11** пакета. Это обусловлено причинами, изложенными ниже.

```
Release := proc()
local k, cd, L, R;
    assign(cd = currentdir(  ), L = [libname]);
    for k to nops(L) do
        try currentdir(cat(L[k][1 .. searchtext("/lib", L[k])], "license"))
        catch "file or directory does not exist": NULL
        catch "file or directory, %1, does not exist": NULL
        end try
    end do;
    assign(R = readbytes("license.dat", 'TEXT', ∞)), close("license.dat"),
```

```
        `if`(nargs = 1, assign([args][1] = currentdir( )[1 .. -9]), NULL),
        currentdir(cd);
    `if`(search(R, "Maple7With"), 7, `if`(search(R, "MapleWith"), 6, `if`(
        search(R, "Maple8With"), 8, `if`(search(R, "Maple9With"), 9, `if`(
        search(R, "Maple9.5With"), 9.5, `if`(search(R, "Maple10With"), 10,
        `if`(search(R, "Maple11With"), 11, `Other release`)))))))
end proc
> Release(h), h;   ⇒ 10, "C:\Program Files\Maple 10"
Release1 := proc()
local v, f, k;
    assign(f = cat(currentdir( ), "/$Kr_Art$.m")),
        (proc(x) local v; v := 63; save v, x end proc )(f),
        `if`(nargs = 0, 9, assign(args[1] = [type(mkdir, 'libobj'), _libobj][2])),
        assign('v' = readbytes(f, 'TEXT', ∞)), fremove(f);
    `if`(v[4] = "5", 5, `if`(v[4] = "4", 4, [search(v, "R0", 'f'), `if`(
        member(parse(v[2 .. f − 1]), {k $ (k = 6 .. 10)}), parse(v[2 .. f − 1]),
        Release( ))][2]))
end proc
> Release1(h), h;   ⇒ 7, "C:\Program Files\Maple 10/lib"
```

Тогда как вызов ***Release1*(h)** через **h** дополнительно возвращает полный путь к главной библиотеке пакета. При этом, процедура идентифицирует и более ранние релизы **4** и **5** пакета. Данная процедура, прежде всего, представляет несомненный интерес в связи с отсутствием *полной* совместимости (*как «сверху-вниз», так и «снизу-вверх»*) релизов **6,** с одной стороны, и релизов **7–11,** с другой стороны. Детально этот вопрос рассмотрен в [41].

В ряде случаев возникает задача определения наличия *вызовов* заданной функции/процедуры **N** в некоторой процедуре ***Proc.*** 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;
        [Res]
    end if
end proc
> Proc:= proc() local a,b,c,d,y,x,v,x1,y1,w,g; a:=56; assign(x=56, 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'=64, '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 = 56,y = (a+b)/(c+d))", "assign(v = 42,('h') = (a+b)^2*(x1+y1)/(c+d))",
 "assign(('w') = 64, ('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*(*Proc*, N)** возвращает список вызовов функции **N,** используемых процедурой **P.** Вызовы возвращаются в строчном формате, дабы не инициировать реальных вызовов вне тела процедуры. В случае отсутствия искомых вызовов инициируется ошибочная ситуация с диагностикой «*call of the form* <%1> *does not exist*». Данная процедура имеет целый ряд полезных применений при разработке приложений.

Как уже отмечалось [103], ***Maple*-**язык допускает использование встроенной ***goto*-**функции безусловных переходов. Поэтому в ряде случаев возникает задача проверки на наличие в процедуре такого 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 дополнительным **T**-аргументом обеспечивает возврат через него множества имен всех меток, обусловленных ***goto*-**вызовами в процедуре **P.** При этом, процедура обеспечивает проверку меток процедуры **P** на допустимость. В случае обнаружения недопустимости выводится соответствующее сообщение. Для обеспечения корректности использования меток может быть использована процедура ***Lprot*** [41,103], обеспечивающая присвоение *protected*-атрибута всем меткам заданной процедуры. Ниже приводится исходный текст ***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("}), 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("proc\
            edure <%1> contains invalid labels %2; error is possible at the proce
            dure 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}
```

Вызов процедуры ***swmpat*(S, m, p, d {, h})** возвращает *true*-значение тогда и только тогда, когда строка **S** содержит вхождения подстрок, соответствующих шаблону **m** с *группирующими* символами, определенными четвертым **d**-аргументом.

```
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, ν, n, ω, t, ε, x, y;
    assign67(c = {args} minus {S, insensitive, m, 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(ν = ((x, n) → cat(x $ (b = 1 .. n))), ω = (t → `if`(t = 0, 0, 1)));
        ε := 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, ν(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 ε(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
> swmpat(S, "art1986kr", [7, 14], "*", `insensitive`, t), t;   ⇒ true, [15, 36]
```

Тогда как *третий* фактический **p**-аргумент задает список кратностей соответствующих вхождений группирующих символов в **m**-шаблон. Если 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 *подсписки* определяют первую и последнюю позиции непересекающихся подстрок **S,** соответствующих **m**-шаблону. В случае отсутствия в **m**-шаблоне *группирующих* символов через **h**-аргумент возвращается целочисленный список, чьи элементы определяют первые позиции непересекающихся *подстрок* **S,** соответствующих **m**-шаблону. Детально с процедурой ***swmpat*** и ее модификацией ***swmpat1*** можно ознакомиться в [41,103,109].

Вызов **Red_n(S, G, N)** возвращает результат *сведения* кратных вхождений в строку/символ **S** символов/строк, определенных **G**-аргументом, к кратности не большей, чем **N.**

```
Red_n := proc(S::{string, symbol}, G::{string, symbol, list({string, symbol})},
N::{posint, list(posint)})
local k, h, Λ, 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')));
    Λ := 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 := Λ(S, g, n, g)
    else h := Subs_All(z = g, Λ(Subs_All(g = z, S, 2), z, n, g), 2)
```

```
    end if;
    convert( h, whattype( S ) )
end proc
> Red_n(`aaaaaaaa10aaaaaagnaaaa17aaaaaaaahhhaaaartaakr`, `a`, 6);
                              aaaaa10aaaaagnaaaa17aaaaahhhaaaartaakr
> Red_n("aaccccbbccccccccckccccckcccccccc", "cccc", 1);    ⇒ "aabbkk"
```

В частности, при **N** = {**1** | **2**} строка/символ **G** удаляется из строки/символа **S** или сохраняется с кратностью **1** соответственно. При этом, тип возвращаемого процедурой **Red_n** результата соответствует типу **S**-аргумента. Процедура **Red_n** производит регистро-зависимый поиск. Если **G** не входит в **S,** вызов процедуры **Red_n(S, G, N)** возвращает первый аргумент **S** без обработки с выводом соответствующего сообщения. Процедура обрабатывает основные особые и ошибочные ситуации.

Если *второй* и *третий* аргументы определяют списки с взаимно однозначным соответствием их элементов, то *сведение* кратностей производится по всем элементам списка **G** с соответствующими им кратностями из списка **N.** При этом, если **nops(G) > nops(N),** то последние **nops(G) - nops(N)** элементов из **G** получают кратность **1** в возвращаемом результате. Если же **G** – список и **N** – положительное целое, то все элементы **G** получают одинаковую кратность **N.** Наконец, если **G** – символ либо строка и **N** – список, то **G** получает кратность **N[1]** в возвращаемом результате. Процедура ***Red_n*** является довольно полезным средством при работе со *строками* и *символами*, используется она и целым рядом средств нашей *Библиотеки* [41,103,109].

Вызов процедуры ***deltab*(T, r, a)** возвращает *NULL*-значение, т.е. *ничего*, обеспечивая удаление из таблицы **T** всех элементов, имеющих соответствие по отношению к *выражению* **a,** в зависимости от заданного режима удаления **r,** а именно: **r=0** – равенство *входов*, **r=1** – равенство *выходов* и **r=2** – наличие любого из указанных равенств. Процедура – достаточно полезное средство в ряде приложений, имеющих дело с табличными структурами.

```
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=17, Kr=10]);
                          T := table([z = Art, Art = 17, Kr = 10, x = a, y = b])
> deltab(T, 0, Art),  eval(T); deltab(T, 1, a),  eval(T);  deltab(T, 2, Art), eval(T);
                                table([z = Art, Kr = 10, x = a, y = b])
                                  table([z = Art, Kr = 10, y = b])
                                    table([Kr = 7, y = b])
```

В качестве *замены* для стандартной ***assign***-процедуры для ***Maple 6-11*** может вполне выступить процедура ***assign67.*** Своими возможностями процедура ***assign67*** достаточно существенно расширяет процедуру ***assign,*** выполняя те назначения, на которых даже релиз **10** пакета вызывает 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 r \
        eserved.`;
    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, { 'list', 'set' } ) then
            seq( procname( op( k, X ) ), k = 1 .. nops( X ) )
        elif type( X, `=` ) and Kr = [ 1 ] or type( X, { 'list', 'set' } ) then
            procname( op( X ), args[ 2 .. -1 ] )
        elif Kr ≠ [ ] and type( X, { `::`, 'symbol', 'equation', '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=64, y=59, 39, z=17, 10);                                    # Maple 10
Error, (in assign) invalid arguments
> assign67(x=64, y=59, 39, z=17, 10); x, [y], [z];    ⇒ 64, [59], [17, 10]
> 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(Art_Kr=NULL), [Art_Kr];
                                                    []
> assign67(('q', 'w', 'e') = (1, 2, 3)); assign67(('x', 'y', 'z') = (64, 59, 39)), [x, y ,z], [q], [w], [e];
                                                [64, 59, 39], [1], [2], [3]
> assign67(V=42,64,G=47,59,S=67,39,Art=89,17,Kr=96,9,Ar=62,44),[V],[G],[S],[Art],[Kr],[Ar];
                                        [42, 64], [47, 59], [67, 39], [89, 17], [96, 9], [62, 44]
```

Процедура ***assign67,*** расширяя возможности стандартной ***assign***-процедуры релизов **6 – 10,** оперспечивает дополнительно такие операции как множественные присвоения последовательностей значений переменным, множественные присвоения *NULL*-значений

и др. При этом, вызов процедуры ***assign67*** корректен 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 [41,103,109]. Например, вызов процедуры ***assign67*(x1, x2, x3,…),** если фактическими аргументами являются символы, делает присвоение **x1:= x2, x3, … .** Тогда как вызов ***assign67*(x1, x2,…,xk, y=a,…)** при том же предположении производит присвоения **x1:=*NULL*,…, xk:=*NULL*.** Уже в настоящей *реализации* процедура ***assign67*** во многих случаях упрощает программирование в среде ***Maple*-**языка, однако она допускает и дальнейшие интересные расширения.

Вызов процедуры ***type*(P, *fpath*)** возвращает *true*-значение, если аргумент **P** определяет *допустимый* полный путь, и *false*-значение в противном случае. При этом, следует иметь в виду, что возврат tr*u*e-значения лишь говорит, что проверяемый каталог **P,** вообще говоря, может быть элементом файловой системы компьютера.

```
type/fpath := proc(P::{string, symbol})
local a, b, f, fl, 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'))),
        ERROR("<%1> - illegal logical drive name", Case(a[1], 'upper'))));
    assign(f = cat(WT( ), "\_$$$14_06_2005$$$" ), b = currentdir( ));
    fl := 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(fl)
    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("G:/aaa\\bbb/ccc\\ddd/salcombe.txt", 'fpath'); ⇒ *true*
Warning, <G> is idle disk drive
> type("@:/aaa/bbb/ccc/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/ccc\\ddd/salcombe.txt", 'fpath'); ⇒ *true*
> type("C:/aaa\\bbb/ccc\\ddd/salcombe.txt", 'path'); ⇒ *false*

Данная процедура хорошо дополняет процедуру ***`type/path`*** [41,103] и представляется довольно полезной в ряде задач, имеющих дело с обработкой файлов данных.

В ряле случаев возникает необходимость программной проверки имени текущего документа. Нижеследующая процедура ***mwsname*** успешно решает данную задачу.

```
mwsname := proc()
local a, b, c, t, k;
    unassign('VGS_vanaduspension_14062005 ');
    VGS_vanaduspension_14062005 ;
    assign('c' = "$Art16_Kr9$", 't' = interface(warnlevel)),
        assign('a' = system(cat("tlist.exe  >  ", c)));
    if a ≠ 0 then
        try null(interface(warnlevel = 0)), com_exe2({`tlist.exe`})
        catch "programs appropriate":
            null(interface(warnlevel = t)), RETURN(delf(c), false)
        end try;
        null(interface(warnlevel = t)), goto(VGS_vanaduspension_14062005 )
    end if;
    assign(b = fopen(c, 'READ'));
    while not Fend(c) do
        a := readline(c);
        if search(a, " Maple ") then
            k := Search2(a, {"]", "["});
            delf(c), assign('a' = a[k[1] .. k[-1]]);
            if search(a, "[Untitled", 'c') then
                RETURN(cat(``, a[c + 1 .. Search(a, ")")[1]]))
            else
                c := Search(a, ".mws");
                b := Search(a, "[");
                RETURN(cat(``, a[b[1] + 1 .. c[1] + 3]))
            end if
        end if
    end do
end proc
> mwsname();  ⇒ ProcLib_6_7_8_9.mws
```

Успешный вызов процедуры ***mwsname*()** возвращает имя текущего документа в *symbol*-формате. При этом предполагается, что в текущем ***Windows***-сеансе инициирован только один ***Maple***-сеанс. Процедура ***mwsname*** достаточно полезна при программной обработке текущих ***Maple***-документов.

Для целого ряда задач весьма полезным оказывается вызов процедуры ***type*(P, *boolproc*).**

```
type/boolproc := proc(P::anything)
local _Art16_Kr9_, _avz63_, ω, ν, z;
option `Copyright International Academy of Noosphere - Tallinn, November, 2003` ,
remember;
    `if`(type(eval(P), 'procedure'), assign(`type/_avz63_` = eval(P)),
        RETURN(false));
    assign(ω = (x → interface(warnlevel = x)),
        z = (( ) → unassign(``type/_avz63_`')));
    try
        ν := interface(warnlevel);
```

```
        ω(0);
        type(_Art16_Kr9_, '_avz63_');
        true, null(ω(ν)), z( )
    catch "result from type `%1` must be true or false":
        RETURN(false, null(ω(ν)), z( ))
    catch "%1 is not a procedure": RETURN(false, null(ω(ν)), z( ))
    catch "%1 uses ": RETURN(false, null(ω(ν)), z( ))
    catch "invalid input: %1 expects": RETURN(false, null(ω(ν)), z( ))
    catch "quantity of actual arguments": RETURN(false, null(ω(ν)), z( ))
    catch : RETURN(true, null(ω(ν)), z( ))
    end try
end proc
> map(type, [IsOpen, `type/file`, PP, isDir, IsFempty, MkDir, lexorder, Empty, mwsname,
  `type/boolproc`, `type/package`, `type/dir`, save2], 'boolproc');
                [true, true, false, true, true, false, true, true, false, true, true, true, false]
```

Вызов процедуры ***type*(P, *boolproc*)** возвращает *true*-значение, если **P** является булевской процедурой (*т.е. возвращает только значения* ***true*** *или* ***false***), и *false*-значение в противном случае. Процедура производит тестирование с высокой степенью достоверности. В случае необходимости процедура ***type*(P, *boolproc*)** легко адаптируется на особые типы тестируемых процедур посредством подключения дополнительных **catch**-блоков.

Вызов процедуры ***type*(F, file)** возвращает *true*-значение, если **K –** *реально* существующий файл данных, и *false*-значение в противном случае. При этом, следует иметь в виду, что возврат *false*-значения говорит лишь о том, что тестируемый файл *не:* (**1**) последний элемент в цепочке каталогов **F,** (**2**) в текущем каталоге и (**3**) среди открытых файлов. Ниже представлен исходный текст процедуры и примеры ее применения.

```
type/file := proc(F::anything)
local a, b, c, k, f, SveGal;
global _datafilestate;
    if not type(F, {'symbol', 'string'}) then return false
    else c := interface(warnlevel); null(interface(warnlevel = 0))
    end if;
    SveGal := proc(f)
            try open(f, 'READ'); close(f)
            catch "file or directory does not exist":
                RETURN(false, unassign('_datafilestate'))
            catch "file or directory, %1, does not exist":
                RETURN(false, unassign('_datafilestate'))
            catch "file I/O error": RETURN(false, unassign('_datafilestate'))
            catch "permission denied": RETURN(false, unassign('_datafilestate'))
            end try;
            true, assign67('_datafilestate' = 'close', f)
        end proc;
    if Empty(Red_n(F, " ", 1)) then
        null(interface(warnlevel = c)); ERROR("argument <%1> is invalid", F)
    else assign67(a = iostatus( ), b = NULL, f = CF(F)),
```

```
        null( interface( warnlevel = c ) )
    end if;
    if nops( a ) = 3 then SveGal( f ), null( interface( warnlevel = c ) )
    else
        for k in a[ 4 .. -1 ] do if CF( k[ 2 ] ) = CF( f ) then b := b, [ k[ 1 ], k[ 2 ] ] end if
        end do;
        if b = NULL then SveGal( f ), null( interface( warnlevel = c ) )
        else true, assign67( '_datafilestate' = 'open', b ),
            null( interface( warnlevel = c ) )
        end if
    end if
end proc
> map(type, ["C:/Temp/http.htm", "D:\\Grodno\\Maple1.doc", `C:/Program Files`,
  `C:/Temp`, "D:\\Academy\\Books/GrGU/Program.pdf", "C:/Print\\STV.doc"], 'file');
                                [true, true, false, false, true, true]
```

Между тем, тестируемый файл, вообще говоря, может быть элементом файловой системы компьютера. При этом, через *глобальную* переменную ***_datafilestate*** в случае возврата *true*-значения возвращается последовательность, чей первый элемент определяет состояние файла {*close*, *open*}, тогда как второй элемент для *close*-состояния определяет полный путь к найденному файлу, а для *open*-состояния – 2-элементный *список*, чей первый элемент определяет используемый файлом *логический* канал в/в, тогда как *второй – полный* путь к файлу. Средства нашей *Библиотеки* [41], имеющие дело с *доступом* к файлам данных, используют именно эту процедуру. Между тем, для более развитого тестирования ***Maple***-объектов на предмет их принадлежности к *file*-типу предназначена процедура ***type*(F, 'file1'),** представленная ниже исходным текстом и примерами применения.

```
type/file1 := proc(F::anything)
local a, b, k, p, t, t1, t2, t3, h, u, ω;
    `if`( type( eval( F ), {'symbol', 'string', 'name'} ), assign( b = iostatus( ) ),
        RETURN( false ) );
    ω, u := ( ) → ERROR( "<%1> is invalid path to a directory or a file", F ),
        interface( warnlevel );
    `if`( Red_n( cat( " ", F ), " ", 2 ) = " " or member( F, { ``, "" } ), ω( ),
        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
> map(type, ["C:/Print/Info.rtf", "D:/Photo/Aladjev8.jpg"], 'file1');  ⇒ [true, true]
```

Вызов процедуры **type(F, 'file1')** возвращает *true*-значение, если **F** определяет реальный файл данных и *false*-значение в противном случае. При этом, следует иметь в виду, что возврат *false*-значения говорит лишь о том, что тестируемый файл *не:* (**1**) последний элемент в цепочке каталогов **F,** (**2**) в главном каталоге ***Maple,*** (**3**) в каталоге с ***Maple***-ядром, (**4**) в каталоге **Users** и (**5**) среди открытых файлов. Между тем, тестируемый файл, вообще говоря, может быть элементом файловой системы компьютера. Вызов процедуры не именяет текущих атрибутов и состояния {*open* | *close*} тестируемых файлов. Процедура полезна при разработке приложений, имеющих дело с обработкой файлов данных.

Стандартные функции ***system*** и ***ssystem*** позволяют выполнять целый ряд довольно важных процедур, обеспечивая более эффективное программирование. Однако на некоторых операционных платформах не все системные команды доступны через данный механизм. Как правило, указанные ***Maple***-функции имеют дело с файлами типов {***'.exe'***, ***'.com'***}, стандартно поставляемых с операционной системой. Между тем, множество таких файлов весьма обширно и содержит целый ряд весьма полезных средств, представляющих немалый интерес и при разработке *приложений* в ***Maple***-среде. Следующие две процедуры ***com_exe1*** и ***com_exe2*** в полной мере решают вопрос пополнения операционной *среды* требуемыми системными командами. С описанием алгоритмов, реализуемых данными процедурами, можно ознакомиться в [41,103,109].

```
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') then
        assign(f = "$Art16Kr9$.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, {".com", ".exe"}) then
        _TABprog[cat(``, b)] := readbytes(P, ∞); 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_exe2 := proc()
local a, b, c, d, p, k, h;
    if type(_TABprog, table) then
        assign(a = WD(), d = [], p = [], b = map(op, [indices(_TABprog)]));
        unassign('AVZ14061942')
    else ERROR("_TABprog is inactive in the current Maple session")
    end if;
    if nargs = 0 then h := b; goto(AVZ14061942)
    else
        if type(args[1], {'list'({'string', 'symbol'}), 'set'({'string', 'symbol'})}) then
```

```
            if Empty( args[ 1 ] ) then
                h := map( op, [ indices( _TABprog ) ] ); goto( AVZ14061942 )
            else for k in args[ 1 ] do
                    if member( cat( "    ", k )[ -4 .. -1 ], { ".com", ".exe" } ) then
                        p := [ op( p ), cat( ``, k ) ]
                    else WARNING( "<%1> is an inadmissible filename", k )
                    end if
                end do
            end if
        else ERROR( "the first factual argument %1 is invalid", args[ 1 ] )
        end if;
        h := SoLists:-`intersect`( p, b );
        if p = [ ] or h = [ ] then
            ERROR( "programs appropriate for uploading are absent" )
        end if
    end if;
    AVZ14061942;
    for k in h do
        c := cat( a, "\", k );
        if not type( c, 'file' ) then
            writebytes( c, _TABprog[ k ] ); close( c ); d := [ op( d ), k ]
        end if
    end do;
    if d ≠ [ ] then d, WARNING(
        "programs %1 have been uploaded into the main Windows directory", d )
    else WARNING( "the required programs are already in the Windows directory" )
    end if
end proc
> com_exe1("C:\\Windows\\crc.exe","C:\\lbz\\userlib");
Warning, _TABprog of library <"C:\lbz\userlib"> has been updated by program <"crc.exe">n
> com_exe1(0, "C:\\Temp/UserLib9/userlib");   ⇒ [tasklist.exe, kill.exe, tlist.exe, taskkill.exe]
```

Первым **P-**аргументом вызова ***com_exe1*(P, L)** может быть полный путь к файлу данных типа {***'.exe', '.com'***} с системной командой или {**0**|**1**}-значение, определяющее *режим* проверки глобальной переменной ***_TABprog,*** созданной/обновленной процедурой. Вызов ***com_exe1*(0, L)** возвращает полные имена программ, сохраненных в таблице ***_TABprog,*** находящейся в ***Maple***-библиотеке, адрес которой указан в **L.** Тогда как вызов процедуры ***com_exe1*(1, L)** возвращает список полных имен программ, находящихся в ***_TABprog*** и доступных в текущем сеансе. Наконец, вызов ***com_exe1*(P, L)** обновляет ***Maple***-библиотеку **L** программой, полный путь к которой определяется **P-**аргументом.

Процедура ***com_exe2,*** дополняя предыдущую, предназначена для *дозагрузки* требуемых программ в главный каталог ***Windows.*** Детальнее с данными средствами можно ознакомиться в [41,103] либо в справке по нашей *Библиотеке* [108,109].

Вызов процедуры ***E_mail*(F, H)** обеспечивает выбор корректных *email*-адресов из текстового файла, указанного первым аргументом **F,** с помещением их в выходной файл, указанный вторым аргументом **H.** При этом, при отсутствии пути к принимающему файлу **H** он создается с выводом соответствующего сообщения о *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, ω, ν;
    `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(ω = OutF, L = [ ], t = 0), [
        assign(ν = interface(warnlevel), L = [ ], t = 0),
        null(interface(warnlevel = 0)), assign(ω = MkDir(OutF, 1)),
        null(interface(warnlevel = ν))])));
    T := {45, 46, (64 + k) $ (k = 0 .. 58), (48 + k) $ (k = 0 .. 9)} minus
        {64, 91, 92, 93, 94, 96};
    [assign(Art = time( )), S = readbytes(InF, 'TEXT', ∞), 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(ω, L[k]), writebytes(ω, `if`(2 < nargs, [10], [44, 32]))
    end do;
    close(`if`(Z = 12, ω, NULL),
```

```
        `if`( Y = 4, cat( substring( InF, 1 .. -4 ), "$$$" ), NULL ) );
    WARNING( "%1 - total of constructions `@` in datafile <%2>
        %3 - quantity of correct email addresses in datafile <%4>
        %5 - quantity of multiple email addresses
        %6 - quantity of suspicious email addresses in datafile <%7>
        run time of the procedure = %8 minute(s)", nops( V ), InF, r, ω, t – r, R,
        cat( substring( InF, 1 .. -4 ), "$$$" ), round( 1/60×time( ) – 1/60×Art ) )
end proc
> E_mail("C:\\RANS\\My_email.txt", "C:\\RANS\\Email\\Email.txt", 10);
Warning, 20 - total of constructions `@` in file <C:\RANS\My_email.txt>
20 - quantity of correct email addresses in file <C:\RANS\Email\Email.txt>
0 - quantity of multiple email addresses
0 - quantity of suspicious email addresses in file <C:\RANS\My_email.$$$>
run time of the procedure = 0 minute(s)
```

Путь к файлу **H** создается ранее рассмотренной процедурой ***MkDir*** [41,103,109]. Результатом выполнения процедуры является не только создание текстового файла с искомыми *email*-адресами, пригодными для немедленного использования средствами *email*, но и обеспечение основной статистики, чье назначение легче проясняет фрагмент, представленный выше. При этом, если вызов процедуры ***E_mail*(F, H, h)** использует необязательный третий **h**-аргумент (*любое допустимое* ***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, чье главное имя идентично имени исходного **F**-файла. Процедура выводит и 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*}, и в более общем случае на файлы *rlb*-типа. Фрагмент ниже представляет текст процедуры.

```
email := proc( F::file, G::{ string, symbol } )
local h, k, t, ω, ν;
    `if`( not type( eval( G ), { 'symbol', 'string' } ),
        ERROR( "<%1> is not a datafile path", G ), `if`( type( G, 'file' ), assign( ω = G ),
        [ assign( ν = interface( warnlevel ) ), null( interface( warnlevel = 0 ) ),
        assign( ω = MkDir( G, 1 ) ), null( interface( warnlevel = ν ) ) ] ) ),
        assign( 'ν' = cat( F, ".$$$" ) ) );
    assign( h = subs( [ seq( t = NULL, t = [ k $ ( k = 0 .. 31 ) ] ) ], readbytes( F, ∞ ) ) ),
        writebytes( ν, h ), close( F, ν );
    try E_mail( ν, ω, 169 ), close( ν )
    catch "datafile <%1> does not contain email addresses": WARNING( "email a\
        ddresses have not been found, however they can be encoded in datafile <\
        1>", F )
    end try
end proc
```

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 и процедура ***extexp***, обеспечи-

вающая вполне удовлетворительное решение данной задачи. Нижеприведенный фрагмент представляет исходный текст процедуры ***extexp*** 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 must be no more than %2 b
                ut has received %3", j, b, n[j]
            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(convert, d, whattype(S)), 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 () x:= proc () end proc end", "x:= proc () end"]
> 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 **n** содержит позиции начала подвыражений в **S**-строке и/или их префиксы в строчном формате. Тип возвращаемых элементов списка определяется типом первого фактического аргумента вызова процедуры. Процедура обрабатывает основные особые и ошибочные ситуации. *Прием*, используемый процедурой для извлечения *корректных* подвыражений, может быть достаточно полезным при программировании других приложений в среде ***Maple.***

Задачи, имеющие дело с поиском минимума и/или максимума (*минимакса*) выражения имеют не только теоретическое значение для исследования *качественных* аспектов большого числа математических разделов, но и в большей степени в многочисленных приложениях анализа. Для решения задач данного вида пакет ***Maple*** имеет целый ряд инструментальных средств различного уровня. Здесь мы представим **aAMV** процедуру, до

некоторой степени дополняющую стандартные инструментальные средства ***Maple*** для поиска *минимакса* алгебраических выражений, в качестве которых выступают элементы массивов различного типа.

```
aAMV := proc(
A::{ Array( numeric ), Vector( numeric ), Matrix( numeric ), array( numeric ) })
local a, b, c, d, h, p, k, x, min, max;
global _Art_Kr_;
    assign( a = (x::range → [ seq( k, k = lhs( x ) .. rhs( x ) ) ] ), '_Art_Kr_' = [ ] );
    assign( b = map( a, `if`( type( A, { Matrix, Vector } ),
        [ seq( 1 .. k, k = [ op( 1, eval( A ) ) ] ) ], [ op( 2, eval( A ) ) ] ) ) ),
        assign( d = nops( b ) ), assign( c = [ seq( cat( x, p ), p = 1 .. d ) ] );
    eval( parse( cat( seq( "seq(", p = 1 .. d ), "assign('_Art_Kr_'=[op(_Art_Kr_),",
        convert( c, 'string' ), "])", seq(
        cat( ",", convert( c[ k ], 'string' ), "=", convert( b[ k ], 'string' ), ")" ), k = 1 .. d ) ) )
        ;
    assign( h = sort( [ seq( A[ op( k ) ], k = _Art_Kr_ ) ] ) ),
        assign( min = [ h[ 1 ] ], max = [ h[ -1 ] ] );
    for k in _Art_Kr_ do
        if A[ op( k ) ] = min[ 1 ] then min := [ op( min ), k ]
        elif A[ op( k ) ] = max[ 1 ] then max := [ op( max ), k ]
        end if
    end do;
    min, max, unassign( '_Art_Kr_' )
end proc
> A := array(1..2,1..2,1..4,[]): A[1,1,1]:=68: A[2,1,1]:=65: A[1,2,1]:=42: A[2,2,1]:=42: A[1,1,2]:=65:
  A[2,1,2]:=42: A[1,2,2]:=350: A[2,2,2]:=350: A[1,1,3]:=144: A[2,1,3]:=67: A[1,2,3]:=85:
  A[2,2,3]:=350: A[1,1,4]:=88: A[2,1,4]:=78: A[1,2,4]:=89: A[2,2,4]:=56:
  A1:=LinearAlgebra[RandomMatrix](10, 18):
> aAMV(A);   ⇒  [42, [1, 2, 1], [2, 2, 1], [2, 1, 2]], [350, [1, 2, 2], [2, 2, 2], [2, 2, 3]]
> aAMV(A1);  ⇒  [-99, [10, 10], [8, 14]], [98, [3, 13]]
```

Вызов процедуры **aAMV(A)** возвращает двухэлементную последовательность *вложенных* списков. Первый элемент первого списка определяет значение минимального элемента массива **A** типов {*array*, *Array*, *Matrix*, *Vector*}, чьи элементы имеют *numeric*-тип, тогда как остальные его элементы определяют списки координат таких минимальных элементов. Тогда как первый элемент второго списка определяет значение максимального элемента массива **A,** в то время как остальные его элементы определяют списки координат таких максимальных элементов. Процедура **aAMV** применима к массивам указанного типа и любой размерности. Данная процедура весьма полезна во многих приложениях.

Представленные в данном разделе средства вполне обозримы и достаточно прозрачны, составляя небольшую часть нашей *Библиотеки*. Они предназначены для иллюстрации как организации процедур в среде ***Maple,*** так и для используемых ими ряда полезных приемов программирования. В заключение настоящей главы вкратце остановимся на вопросах отладки ***Maple***-процедур.

3.10. *Элементы отладки Maple-процедур и функций*

Проблема *отладки* может возникать или при появлении рассмотренных выше, или других особых и аварийных ситуаций, либо при получении некорректных с точки зрения решаемой задачи результатов. В настоящее время проблема отладки **ПС** достаточно хорошо разработана и на данном вопросе нет *особого* смысла останавливаться. Ибо имеющий определенный компьютерный навык пользователь *вполне* знаком с данным вопросом. Из простых средств отладки можно отметить методы *контрольных точек*, *трассировки* (*прокрутки*) и др. ***Maple***-язык в качестве такого средства предлагает метод *трассировки* вычислений, поддерживаемый процедурами {***trace, debug***}, представляющими на самом деле одну и ту же процедуру, но с альтернативными идентификаторами. Поэтому, говоря о ***trace***-процедуре, будем иметь в виду и альтернативную ей ***debug***-процедуру, и наоборот.

В тестирующем режиме, определенном процедурой {***trace*|*debug***}**(P1, P2,...,Pn),** производится трассировка каждого вызова **Pк**-процедур/функций, указанных в списке фактических аргументов функции до тех пор, пока не будет {***untrace*|*undebug***}**(P1, P2, ..., Pn)-**вызова процедуры соответственно**,** отменяющего режим тестирования всех либо части тестируемых **Pk**-процедур/функций. Для трассируемой процедуры/функции на печать выводятся точки их вызова, результаты всех выполняемых промежуточных вычислений и предложений, а также точки выхода. В точках входа указываются фактические значения аргументов, а в выходных – возвращаемые ими результаты. Детально вопросы использования данных средств для отладки процедур рассмотрены в [12], *исходную* версию книги можно бесплатно получить по адресу [91].

Для *отладки* механизма вызовов процедур из других процедур весьма полезным средством может оказаться и процедура ***where*(|<*Число*>),** по которой выводится содержимое стэка вызовов процедур на глубину, определяемую значением (*целого числа*) фактического аргумента функции. Вызов процедуры ***where*()** определяет *трассировку вызовов* процедур, начиная с верхнего уровня; выводятся выполняемые предложения процедур и значения передаваемых процедурам фактических аргументов. При определении *глубины* стэка выводятся только элементы заданного числа его *нижних* уровней. С учетом сказанного, вызов ***where***-процедуры должен указываться внутри процедуры, трассировка вызовов которой должна отслеживаться, как это иллюстрирует простой фрагмент трассировки вызовов ***Kr***-процедуры на глубину **5** стэка вызовов процедуры:

```
> 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(64,59,39,10,17,44);
TopLevel: G(64,59,39,10,17,44)
        [64, 59, 39, 10, 17, 44]
G: S(args)
        [64, 59, 39, 10, 17, 44]
S: V(args,95)
        [64, 59, 39, 10, 17, 44, 95]
V: Kr(args,99)
        [64, 59, 39, 10, 17, 44, 95, 99]
Currently in Kr.
        [8, 427]
> Kr(Kr(64, 59, 39, 10, 17, 44));    ⇒ [1, [6, 233]]
```

```
TopLevel: Kr(Kr(64,59,39,10,17,44))
        [64, 59, 39, 10, 17, 44]
Currently in Kr.
TopLevel: Kr(Kr(64,59,39,10,17,44))
        [[6, 233]]
Currently in Kr.
```

Следует отметить, что *пакетный отладчик* (**Debugger**) располагает **where**-командой, аналогичной ***where***-процедуре, за исключением того, что первую можно выполнять интерактивно в отладочном режиме, инициируемом отладчиком. При этом, наибольший эффект от использования ***where***-процедуры можно получить при трассировке *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 посредством функционального (**->**)-оператора:

```
> Fnc:= () -> [where(5), evalf(sqrt(sum(args['k']^2, 'k' = 1..nargs)), 3)];
```

$$Fnc := (\ ) \rightarrow \left[\text{where}(5), \text{evalf}\left(\sqrt{\sum_{'k'=1}^{\text{nargs}} \text{args}_{'k'}^{\ 2}}, 3 \right) \right]$$

```
> V(Kr(Fnc((42, 47, 67, 89, 96, 62))));  ⇒ [2, 99 + [1, [171.]]]
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.]]
Currently in Kr.
TopLevel: V(Kr(Fnc(42,47,67,89,96,62)))
        [[1, [171.]]]
V: Kr(args,99)
        [[1, [171.]], 99]
Currently in Kr.
```

Данный фрагмент использует две процедуры ***V*** и ***Kr,*** определенные в *предыдущем* фрагменте, и определяет ***Fnc***-функцию, содержащую вызов ***where***-процедуры, что позволяет использовать для нее описанный механизм тестирования вызовов через стэк.

Встроенный **DEBUG**-отладчик ориентирован на отладку достаточно сложных процедур в *интерактивном* режиме и базируется на механизме *контрольных точек* (*check points*), допуская два варианта исполнения. По *первому* варианту режим отладки активируется по функции **DEBUG(**{<*Комментарий*>}**),** идентификация ***Input***-параграфа которого производится "**DBG>**"-метками. Тогда как отмена режима отладки производится по команде {**quit**|**stop**|**done**}, при этом следует иметь в виду, что при вводе команд режима *отладки* кодирование после них разделителя не допускается. В **DEBUG**-режиме отладки допускается использование более **16** команд: **cont, next, step, into, stopwhen, return, stop, list, where, showstat, showstop, stopat, stoperror** и др., обеспечивающих целый ряд важных отладочных функций, с которыми детально можно ознакомиться в книгах [8,9] либо в определенной степени по справке пакета, а также по поставляемой с пакетом документации [78-85]. Там же достаточно детально можно ознакомиться и с ограничениями по применению **DEBUG**-отладчика. Здесь мы вкратце остановимся лишь на некоторых из наиболее используемых командах режима отладки. Прежде всего, отметим, что кодирование команд **DEBUG**-режима отладки завершается { **|;|:**}-разделителем, что отличает

их синтаксис от синтаксиса предложений ***Maple*-языка**, а также то, что вид разделителя не влияет на возврат результата выполнения команд, как это имеет место в случае традиционных предложений ***Maple*-языка**. Новые релизы пакета часто расширяют функции отладчика.

Прежде всего, в целях удобства установки в процедурах контрольных точек рекомендуется пронумеровать составляющие их предложения, что обеспечивает команда отладчика **showstat(*Proc* {, <*Диапазон*>}),** возвращающая определение ***Proc*-процедуры** с приписанными ее предложениям номерами, позволяя использовать их при установке контрольных точек. По команде отладчика **stopat(*Proc* {, <{*номер* | *диапазон*}> {, <ЛУ>}})** производится установка контрольных точек в предложения ***Proc*-процедуры** с номерами, определяемыми ее вторым параметром. При этом, *третий* необязательный **ЛУ**-параметр определяет логическое условие, *true*-значение которого разрешает производить останов в данной контрольной точке. Данное условие может связывать как *глобальные*, так и *локальные* переменные, а также и формальные аргументы процедуры. Если предложения процедуры не нумеровались, то производится установка контрольной точки в начало процедуры. По **stopat**-команде возвращается список всех установленных контрольных точек, а отмена *конкретной* контрольной точки производится по команде **unstopat(*Proc*, <*номер*>),** тогда как команда **unstopat()** отменяет все контрольные точки. Расстановка контрольных точек производится в соответствии с логикой отлаживаемой процедуры. В отличие от метода *трассировки*, метод *контрольных точек* позволяет исключать из анализа прозрачные участки программ, существенно уменьшая вывод *избыточной* отладочной информации. В процессе выполнения процедуры производится *останов перед* предложением, в котором была установлена контрольная точка, позволяя провести отладочные операции. По **cont**-команде можно продолжить выполнение процедуры до следующей установленной контрольной точки.

По команде **stopwhen(<*Id*>)** определяется режим мониторинга значений для указанной ее аргументом локальной или глобальной переменной. В случае *глобальной* переменной указывается только ее имя, тогда как *локальная* требует указания в качестве фактического параметра команды списка, первый элемент которого определяет имя процедуры, а второй – имя собственно ее *локальной* переменной. Отмена мониторинга производится по **unstopwhen**-команде, относительно которой остается в силе *сказанное* в адрес команды **unstopat.** Наконец, по команде **stoperror(**<*Сообщение*>**)** определяется мониторинг появления заданного ее фактическим аргументом диагностического сообщения об ошибках. При возникновении в процессе выполнения процедуры *ошибочной* ситуации с указанным *сообщением* (*если она не обрабатывается* ***traperror*-функцией**) активируется режим отладки, выводятся сообщение об ошибке и вызвавшее ее предложение процедуры. По команде **showstop()** можно получать информацию о всех функциях и процедурах, содержащих предложения **stopat, stoperror** и **stopwhen**. Тогда как по функции ***debugopts*** предоставляется возможность как проверять, так и модифицировать параметры, управляющие режимом **DEBUG**-отладки процедур.

При этом, следует иметь в виду, что в случае использования в процедуре *нескольких* одноименных переменных контрольная точка для их мониторинга по **stopwhen**-команде устанавливается только для *первого* их вхождения в тело процедуры, как это хорошо иллюстрирует следующий достаточно простой фрагмент:

```
> F:= proc(x) local a,b; a:= 64: b:= sqrt(x + a): a:= `if`(x >= 0, a, `End`) end proc:
> stopwhen([F, a]);   ⇒  [[F, a]]
> F(39);
a := 64
```

```
F:
  2  b := sqrt(x+a);
[DBG> cont   ⇒ 64
> F(-12);
a := 64
F:
  2  b := sqrt(x+a);
[DBG> cont   ⇒ End
> V:= proc(x) local a,b; a:= 64: b:= sqrt(x + a): a:= `if`(x >= 0, evalf(b, 4),`K`) end proc:
  V(2006); stopwhen([V, a]):   ⇒ 45.51
> V(2006);
a := 64
V:
  2  b := sqrt(x+a);
[DBG> a:= 1942
V:
  2  b := sqrt(x+a);
[DBG> cont   ⇒ 62.84
```

Не имеет особого смысла установка в процедуре контрольной точки после последнего предложения ее тела: если предложение было отлично от **return**-предложения, то процедура возвращает *NULL*-значение, в противном случае – результат выполнения **return**-предложения. По **stopat**-процедуре вообще невозможна установка контрольной точки после последнего предложения процедуры, ибо **end**-предложение не нумеруется.

При этом, следует иметь в виду, что в **DEBUG**-*отладочном* режиме допускается производить вычисления с участием *переменных* процедуры, выводить промежуточные результаты и присваивать значения переменным процедуры, производя мониторинг вариантов вычислений. Перед выходом из отладочного **DEBUG**-режима следует 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 же случае производится выход из отладочного режима, но ядро пакета остается в так называемом *оперативном* **DEBUG**-режиме, активируемом только в момент вызова соответствующих процедур и функций, определенных указанными командами, которые в этом случае выступают уже на уровне функций языка и требуют соответствующего синтаксиса (*должны завершаться* {**;**|**:**}-*разделителем*). В *оперативном* режиме **DEBUG** допускается использование всех вышерассмотренных команд/функций, тогда как внутри режима *отладки* определенных ими процедур/функций *допускается* использование *всех* команд отладочного режима. Установка контрольных точек любого из рассмотренных трех типов может производиться и внутри самих процедур, *инициируя* режим *отладки* в моменты их вызовов, однако, на наш взгляд, это не самая лучшая технология отладки. В [12] и [91] можно найти ряд весьма *поучительных* примеров применения вышерассмотренных **DEBUG**-средств языка пакета для отладки достаточно простых процедур.

Из данных примеров не только четко прослеживаются основные принципы тестирования процедур на основе *контрольных* точек, но из них также следует, что средство функции **DEBUG** уже для относительно несложных процедур является малообозримым и может быть рекомендовано только в достаточно сложных для отладки обычными средствами случаях.

В качестве простого средства отладки ***Maple***-процедур можно использовать и модифицированный метод контрольных точек. Этот метод состоит в следующем. Для генерации процедур, обеспечивающих возвращение значений заданных выражений в установленных контрольных точках, служит процедура ***ChkPnt*** [103], кодируемая в самом начале тела тестируемой процедуры ***Proc*** в виде вызова ***ChkPnt*(*args*)**. Затем в требуемых местах процедуры ***Proc*** кодируются вызовы следующего формата:

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 **xj (j=1 .. n)** – *имена выражений*, значения которых требуется получить в данной точке процедуры.

Вызов тестируемой процедуры ***Proc*(*args*)** на кортеже ее *основных* фактических аргументов не включает механизма контрольных точек, тогда как вызов процедуры ***Proc*(*args*, *chkpnt*=N)** для **N** из диапазона **[1 .. 61]** обеспечивает *вывод* значений требуемых выражений в контрольной точке с номером **N** (*формат* **1**) либо вывод значений с их возвратом (*формат* **2**), как это наглядно иллюстрирует достаточно прозрачный фрагмент.

```
ChkPnt := proc()
local a, k;
    unassign( seq( cat('chkpnt', k ), k = 1 .. 61 ));
    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 := 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(64.42, 59.47, 39.67);
Warning, Results: a=96.23, b=734.8, c=1.862
> Proc(64.42, 59.47, 39.67, chkpnt = 2);   ⇒ 96.23, 734.8
Warning,  in chkpnt2 variables [_a, _b] have the following values [96.23, 734.8]
```

```
> Proc(64.42, 59.47, 39.67, chkpnt = 1);
Warning,  in chkpnt1 variables [_x, _y, _z, _a] have the following values
[64.42, 59.47, 39.67, 96.23]
> Proc(64.42, 59.47, 39.67, chkpnt = 3);
Warning,  in chkpnt3 variables [_a, _b, _c] have the following values [96.23, 734.8, 1.862]
```

Представленный механизм контрольных точек позволяет легко устанавливать контрольные точки и получать в них значения требуемых выражений, а также возвращать эти значения, что обеспечивает простой механизм запрограммированного выхода из *любой* точки процедуры с возвращением значений требуемых выражений. Данный механизм довольно эффективен ввиду *структурированности* процедур ***Maple,*** которые не используют ***goto***-механизм.

Наконец, по вызову процедуры ***maplemint*(*P*)** производится вывод протокола результатов семантического анализа ***P***-процедуры, включая ее *никогда* не выполняемые предложения. Следующий простой фрагмент иллюстрирует результат применения процедуры ***maplemint*** на примере анализа простой процедуры:

```
> AGN:= proc(_x)  local a,b;  a:= 64(x):  b:= ln(x + a):  a:= `if`(x >= 10, 3(b^2), `G`)  end proc:
> maplemint(AGN);
   This expression may be missing an operator like '*':  64(x)
   This expression may be missing an operator like '*':  3(b^2)
   These names were used as global names, but were not declared:   x, G
   These parameters were never used explicitly:   _x
```

В качестве полезного упражнения читателю рекомендуется проверить различные варианты отладки ***Maple***-процедур на основе рассмотренных *отладочных* средств. Наш опыт работы с **ПС** различных уровня и назначения *вполне* позволяет констатировать, что при достаточно *полном* соответствии описания функционирования конструкций **ПС** их реализации, хороших знаниях сущности погружаемой в их среду задачи и возможностей данного средства существует целый ряд *более* эффективных средств отладки, чем методы *трассировки* и *контрольных* точек в их чистом виде. Наша практика создания различного рода **ПС** не использовала данной методологии в ее классическом виде, как не отвечающей основной задаче эффективной отладки. В целом же, даже достаточно сложные процедуры возможно отлаживать и без указанных выше средств, имея ввиду интерактивный характер ***Maple***-языка, позволяющий вполне успешно вести пошаговую интуитивную отладку процедур параллельно с их написанием, т.е. использовать методику и приемы т.н. *эвристического программирования* (*точнее*, *его элементы*). Именно подобным образом создавалось большинство наших программных средств.

В заключение настоящей главы кратко остановимся еще на одном вопросе, связанном с оптимизацией процедур. В качестве определенного подспорья здесь может оказаться и процедура ***maplemint,*** чей вызов ***maplemint*(*Proc*)** генерирует отчет с семантической информацией по заданной процедуре ***Proc*** (*активной в текущем сеансе либо находящейся в* ***Maple****-библиотеке, логически связанной с главной библиотекой пакета*) и выводит *коды*, которые *невыполнимы* при вызове процедуры. Вызов ***maplemint*(*Proc*)** генерирует информацию по таким аспектам процедуры ***Proc*** как: *константы*, присваиваемые в качестве значений; декларированные *глобальные* переменные, начинающиеся с символа '_' (*подчеркивания*); неиспользуемые декларированные *глобальные* переменные; используемые, но не декларированные *глобальные* переменные; *неспользуемые* декларированные *локальные* переменные; *переменные цикла*, повторно используемые во вложенных циклах; недостижимый код; ключевые слова **break** и/или **next,** обнаруженные вне цикла; отсутствие знака

умножения '*' и др. Несколько детальнее с данным средством можно ознакомиться по конструкции **?maplemint,** тогда как с исходным текстом процедуры можно ознакомиться по следующему предложению:

> interface(verboseproc = 3); eval(`maplemint/recurse`);

Между тем, текущая реализация процедуры ***maplemint*** не поддерживает получения семантической информации по процедурам, содержищим *модульные* объекты и средства обработки особых и ошибочных ситуаций. Но и перечисленное может оказаться достаточно полезным при решении ряда *вопросов оптимизации* ***Maple***-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(Find);
  These names were used as global names, but were not declared:  __filesize
> maplemint(Sts);
  These names were used as global names, but were not declared:  k
  These local variables were used before they were assigned a value:  a
> maplemint(Case);
  These local variables were used before they were assigned a value:  k
> maplemint(Kvantil);
  These names were used as global names, but were not declared:  x, t
> 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:  a, b, zeta, s, nu
  These parameters were never used explicitly: E
> maplemint(save2);
  These names were used as global names, but were not declared:  omega, c, c, c
  These local variables were used before they were assigned a value:  x, b, nu
> maplemint(Proc);
Error, (in maplemint/statement) exception handling is not yet supported
> Proc(64, 59, 39, 10, 17, 44); Proc();   ⇒ 233/6
Error, (in Proc) procedure call has no arguments
```

Из приведенного фрагмента можно сделать ряд выводов, а именно: (**1**) сложные процедуры, как правило, используют средства обработки особых и ошибочных ситуаций, поэтому они не доступны для анализа процедурой ***maplemint*** (*даже такая примитивная как процедура* ***Proc*** *из последнего примера недоступна для* ***maplemint***), (**2**) *далеко* не всегда результаты семантического анализа соответствуют конкретному алгоритму тестируемой процедуры, например, для формальных аргументов подпроцедур, *глобальных* переменных, индексов суммирования и т.д., и (**3**) в ряде случаев выводимая информация не совсем

корректна с точки зрения ***Maple***-языка, как это иллюстрирует нижеследующий весьма простой фрагмент:

```
> P:= proc() assign('libname' = op([libname, "D:/RANS/IAN"])); `+`(args)/nargs end proc:
> maplemint(P);
   These names were used as global names, but were not declared:
      "C:\\Program Files\\Maple 8/lib",
      "c:/program files/maple 8/lib/userlib"
```

В данном фрагменте в качестве имен *глобальных* переменных указывается не предопределенная ***libname***-переменная, а ее *значение*, что совсем не одно и то же. Имеются и другие некорректности. Однако, для случая достаточно простых процедур вышеуказанное средство ***Maple*** может быть вполне полезным, прежде всего, для не столь искушенного пользователя.

Наконец, синтаксический контролер **Mint** анализирует программу ***Maple*** и генерирует сообщение о возможных ошибках в ***mpl***-файле данных с исходным ***Maple***-текстом. Если файл не задан, то для чтения исходного ***Maple***-текста используется стандартный ввод. Анализ завершается по достижении конца текста. Более детально с данным средством синтаксической проверки ***Maple***-программ можно ознакомиться по конструкции **?mint.**

Между тем, наш опыт и опыт наших коллег показывают, что при *достаточной* квалификации использование для отладки и оптимизации средств, созданных в программной среде пакета ***Maple,*** вышеперечисленных стандартных средств *совершенно* не обязательно, тем более, что они имеют целый ряд весьма существенных ограничений. Достаточный программистский опыт, возможность *интерактивно-эвристически программировать*, хорошее знание самой сути программируемых задач вполне достаточны для создания достаточно сложных и эффективных программных средств в среде пакета.

Действительно, ***Maple***-язык позволяет в интерактивном режиме программировать алгоритм вашей задачи поэтапно, также поэтапно производя отладку полученных документов либо процедур. Он же позволяет относительно несложно производить их постепенную доводку до нужных требований; при этом, сам процесс такого программирования обеспечивает уточнение отдельных составляющих программируемого алгоритма. При этом, отладка всего *комплекса* сводится к отладке последовательно связанных его составляющих, которая производится одновременно с их программированием. А так как составляющие с учетом весьма 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 данная методика и использовалась нами при создании средств нашей *Библиотеки* [41,103,109]. Опыт ее применения со всей очевидностью доказал ее жизнеспособность и во многих случаях большую эффективность, чем имеющиеся для этих целей стандартные средства отладки. Правда, описанный подход *априори* предполагает как хорошее знание сути программируемого приложения, так и саму программную среду пакета.

Рассмотрев процедурные объекты ***Maple***-языка, обеспечивающие достаточно высокий уровень модульности программирования в среде пакета, представим теперь новый тип (*начиная с* **6**-*го релиза*) объектов ***Maple***-языка, не только повышающих уровень *модульности* разрабатываемых в среде ***Maple*** программных средств, но и в *определенной* мере обеспечивающих *объектно-ориентированную* технологию программирования в его среде. Такими объектами являются программные *модули*, представление которых целесообразно предварить небольшим, но полезным экскурсом в недавнее прошлое *программирования*.

Глава 4. *Организация программных модулей в Maple*

4.1. *Вводная часть*

В настоящей главе рассматриваются модульные объекты ***Maple-***языка, предполагая, что читатель в определенной мере имеет представление о работе в среде ***Maple*** в пределах, например, книг [9-14] либо подобных им изданий. Все используемые понятия и определения полностью соответствуют вышеупомянутой книге [13]. Акцент сделан на модульных объектах языка, определяющих возможность использования элементов объектно-ориентированного программирования в среде ***Maple-***языка.

После **50**-летнего периода прогресса в развитии инструментального **ПО** его возможности начали существенно отставать от возможностей аппаратных средств и эта разница увеличивается с каждым годом. Одной из причин такого состояния является то обстоятельство, что **ПС** создается последовательно строка за строкой, тогда как большинство современных **ЭВМ** разрабатывается и создается по *интегральной* технологии, на основе печатных плат и т.д., позволяя использовать уже хорошо апробированные решения большинства компонент **ЭВМ**. С появлением *объектно-ориентированной технологии* (**ООТ**) появилась возможность разработки **ПС** на основе готовых базовых программных конструкций и компонент [1-3]. Данный подход позволяет создавать **ПС** из существующих компонент значительно быстрее и дешевле, обеспечивая существенное повышение их надежности, гибкости и мобильности.

В **ООТ** пользователь имеет дело с тремя базовыми элементами: *объектами*, *сообщениями* и *классами*. *Объекты* представляют собой многократно используемые программные модули, содержащие связанные *данные* и *процедуры*. Структурно объекты состоят из двух частей: *переменных* и *методов*. *Методы* представляют собой *наборы* процедур и функций, определяющих алгоритм функционирования объекта. Подобно переменным в традиционных языках программирования объектные переменные могут содержать как простые данные (*числа*, *массивы*, *текст и т.д.*), так и сложной структуры информацию (*графика*, *звуковые образы и т. д.*). Более того, объектные переменные могут содержать другие объекты и т.д. Такие объекты называются *сложными*. Таким образом, объекты являются автономными модулями, содержащими всю необходимую для их выполнения информацию, что делает их идеальным блочным строительным материалом для создания сложных **ПС** различного назначения.

Для связи между объектами используются *сообщения*, состоящие из *трех* частей: идентификатора объекта-адресата, имени метода (*процедуры*), который должен выполняться в искомом объекте, а также любой дополнительной информации (*фактические значения для формальных параметров*), необходимой для настройки режима выполнения выбранного метода. Использование сообщений позволяет вводить четкую систему протоколов для взаимодействия объектов в системе, не акцентируя внимания на их внутренней организации. Данный подход не только защищает (*скрывает*) внутреннюю структуру объекта, но и позволяет легко изменять ее при условии, что новый объект будет воспринимать те же сообщения, что и предыдущий. Это позволяет весьма гибко изменять структурную организацию сложных многомодульных систем, не изменяя общего алгоритма их функционирования.

Во многих случаях сложная программная система нуждается в *большом* количестве *однотипных* объектов. В такой ситуации весьма неэффективно для каждого отдельного объекта содержать всю информацию о методах и переменных. С этой целью вводится по-

нятие *класса* объектов. *Классы* напоминают собой своего рода шаблоны для однотипных объектов, содержащие информацию, необходимую для генерации однотипных объектов, включая определения их методов и типов объектных переменных. Отдельные объекты *каждого* класса содержат только ту часть информации, которая отличает один объект от другого объекта одного и того же класса, а именно - значения *объектных переменных*. В классе допускается определение *иерархии* подклассов. То, что подклассы всех высших в иерархии классов наследуют их свойства, делает подобные иерархические структуры мощным средством проектирования многомодульных **ПС**. Таким образом, *классы* являются действительно мощной составляющей **ООТ**, определяя *шаблоны* для объектов, использующихся многократно, и допуская однократное определение каждого метода и объектной переменной, даже при условии использования их в различных классах. Так как **ООТ** исповедует принцип максимального использования готовых объектов для создания новых **ПС,** то **ООТ** предполагает использование ряда новых подходов к проектированию программного обеспечения, как системного, так и прикладного.

Идеология *объектно-ориентированного программирования* (**ООП**) восходит к **60**-м годам в связи с исследованиями по проблематике искусственного интеллекта. Понятие *программных* объектов впервые было введено в языке ***Simula-67,*** выросшем из ***Algol-60*** и ориентированном на создание **ПС,** предназначенных для имитации принятия решений в условиях управляемого множества обстоятельств. Однако **ООП**-идеология не привлекала широкого внимания вплоть до создания в 1970 г. ***SmallTalk***-языка, состоящего исключительно из объектно-ориентированных конструкций и представляющегося нам *наиболее* развитым языком **ООП**. Затем данная идеология была адаптирована разработчиками графических интерфейсов и интегрирована в ***SmallTalk***-подобные языки. Позднее она была адаптирована и в гибридные языки, подобные ***C+,*** которые явились *первой* попыткой навести мосты *между* более процедурными и **ООП**-языками. При **ООП**-подходе понятия процедуры и данных, используемые в обычных системах программирования, заменяются понятиями объект и сообщение; *объект* – это пакет информации вместе с алгоритмом ее обработки, а *сообщение* – это спецификация условий одной из операций обработки объекта. В отличие от процедуры, которая описывает алгоритм обработки, *сообщение* только определяет, что хочет выполнить его отправитель, а получатель точно определяет, как это сделать.

Методология **ООП,** являясь дальнейшим естественным развитием традиционного программирования, предполагает большую степень структурированности (*чем в структурном программировании*), модульности и абстрактности (*чем предыдущие попытки абстрагирования данных и сокрытия деталей*) **ПС**. Новая методология определяется тремя функциональными характеристиками **ООП,** а именно:

- ***инкапсуляция:*** *объединение* записей с процедурами и функциями, что превращает их в новый тип данных – *объекты*. *Объекты* сохраняют *структуру*, *значение* и *поведение* структуры данных, допуская намного более завершенную абстракцию и модульность в программировании;
- ***наследование:*** определение объекта с последующим использованием его для построения иерархии порожденных объектов с наследованием доступа каждого из *порожденных* объектов к процедурам и данным своего предка;
- ***полиморфизм:*** присвоение *единого* имени процедуре, которая передается вверх и вниз по иерархии объектов, с выполнением этой процедуры *способом*, соответствующим каждому объекту в иерархии.

За более детальной информацией по ***ООП*** и средствам поддержки ***ООТ*** можно обратиться к книгам [1-3]. Здесь же мы рассмотрим элементы **ООТ,** поддерживаемые ***Maple*** в лице его программных модулей.

Начиная с **6**-го релиза, пакет включает средства по обеспечению ряда *базовых* элементов объектно-ориентированного программирования, которые поддерживаются *механизмом* программных модулей языка (*или в дальнейшем просто программных модулей* – **ПМ**). Данные модули не следует ассоциировать с более широким понятием *пакетных модулей*, которые употреблялись на протяжении книг [8-12], рассматривающих пятый релиз пакета. Вместе с тем, можно считать, что *программные модули* при оформлении их в библиотечные структуры составляют *подмножество* всех пакетных модулей. Таким образом, если *пакетные модули* можно рассматривать как *внешние* по отношению к его ядру хранилища определений *функциональных* (*модульных*) средств, то механизм *программных модулей*, в первую очередь, ориентирован на обеспечение определенного уровня объектно-ориентированного программирования в среде ***Maple***-языка пакета.

Если процедуры позволяют определять последовательность предложений ***Maple***-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 *более* высокий уровень абстрагирования, позволяя "скрывать" от пользователя уже целые наборы тематически связанных процедур и данных. Относительно ***Maple***-языка **ПМ** являются *новым* типом *выражений* подобно числам, уравнениям, отношениям и процедурам. Тестирование *модульных объектов* производится функциями ***typematch, type*** и процедурой ***whattype,*** как иллюстрирует следующий достаточно простой фрагмент:

```
> Grsu:= module()   end module:
> type(Grsu,`module`), typematch(Grsu,`module`), whattype(eval(Grsu));
                                  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 последние примера иллюстрируют возможность тестирования не только готового модульного объекта языка, но и его определения. Для этого определение модуля должно указываться *невычисленным*, в противном случае возвращается *false*-значение. Данная организация **ПМ** позволяет использовать их при программировании параметрических алгоритмов, создании пакетных модулей, а также предоставляет возможность использования в ***Maple***-программах ***Pascal***-подобных записей [1] или **C+/C**-подобных структур [12]. 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 и (**4**) параметрическое программирование. Детально эти *способы* с рекомендациями по их применению рассмотрены в книгах [13-14,29-33,41].

Инкапсуляция в значительной степени гарантирует, что уровень абстрагирования процедур и данных **ПМ** строго соответствует определенному для него интерфейсу. Это позволяет пользователю программировать сложные мобильные и многократно используемые программные системы с хорошо определенными пользовательскими интерфейсами. Более того, обеспечиваются лучшие *сопровождение* и понимание исходных програм-

мных текстов, что является 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 его.

Пакетные модули (*в отличие от* **ПМ**) обеспечивают механизм *совместного* хранения тематически *связанных* ***Maple***-процедур. 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. Большое количество функциональных средств пакета находится именно в его *модулях* как внутренних, так и внешних. Основной организацией модулей пакета предыдущих релизов являлась табличная структура [12], в которой входами являлись имена процедур, а выходами их определения. **ПМ** обеспечивают иной механизм организации модулей пакета, который детально рассмотрен в [13,41]. В частности, пакетный модуль **LinearAlgebra,** обеспечивающий функции линейной алгебры, был имплантирован в среду пакета релизов **6** и выше именно как *программный модуль*.

Посредством **ПМ** легко программируются объекты. В программной среде под *объектом* понимается некоторая программная единица, определяемая как своим состоянием, так и поведением. Вычисления над такими объектами производятся передачей им некоторых управляющих сообщений, на которые они отвечают соответствующими действиями (*вычисление*, *управление*, *изменение состояния и др.*). Параметрические программы пишутся без знания того, как организованы объекты, обработку которых они производят. Параметрическая программа будет работать с любым *объектом*, имеющим с ней общий интерфейсный протокол, безотносительно того, как объект удовлетворяет этому протоколу. Перечисленные выше **4** аспекта, поддерживаемые механизмом **ПМ,** определяют самое непосредственное практическое применение ***Maple***-технологии, которая широко иллюстрируется в [32] на ряде практически полезных примеров. *Основы* механизма **ПМ** и их использования в ***Maple***-среде иллюстрируются соответствующими примерами.

4.2. *Организация программных модулей Maple-языка*

Организация *программного модуля* (**ПМ**) весьма напоминает организацию ***Maple***-процедуры, довольно детально рассмотренной в предыдущей главе. 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*}
ТЕЛО МОДУЛЯ
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 явно заданных формальных аргументов. Завершается *определение* модуля закрывающей

скобкой **`end module`**. Все остальные *компоненты* структуры модуля, находящиеся между **module()** и **`end module`**, являются необязательными. При этом, взаимный порядок расположения указанных пяти деклараций в *определении* модуля несущественен, но все они должны предварять (*при их наличии*) *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. Данное 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 (*при этом, специальные опции могут различаться*). **ПМ** можно представлять себе как собрание 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 (**local**) или неявно *локальными* и доступны только в рамках определения модуля в период его реализации. Они используются только алгоритмом, реализуемым предложениями ***Maple***-языка, составляющими *тело* 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. Все другие переменные, встречающиеся 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 или через неявные правила синтаксического и семантического анализа пакета в период упрощения определения 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 *пустым* либо содержать ***Maple***-предложения и/или допустимые ***Maple***-выражения. Данные конструкции тела обеспечивают как необходимые вычисления в процессе реализации 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. Переменным модуля (*локальным*, *глобальным или экспортируемым*) могут присваиваться любые допустимые ***Maple***-выражения, включая *процедуры* и *модули*. Простейший модульный объект имеет следующий вид: **module() ... end module**. Данный модуль не имеет особого смысла, т.к. не экспортирует переменных, не имеет ни *локальных*, ни *глобальных* переменных, ни даже 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 *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 существуют серьезные различия. *Поименованный* первым способом модуль может *многократно* переименовываться, создавая свои копии. Тогда как поименованный *вторым способом* модуль не может быть переименован, а его *имя* имеет атрибут ***protected***. 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 достаточно разумное решение, т.к. модули могут быть *непоименованными* либо иметь разноименные копии, тогда как *второй* способ именования делает модуль с *фиксированным* именем. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> 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
> A:= module B ()  end module:
> type(A, `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. Это связано с тем, что *имя* поименованного вторым способом модуля имеет ***protected***-атрибут. Рассмотрим компоненты определения модуля несколько детальнее.

Ряд неудобств работы с программными модулями *второго* типа, которые в ряде случаев требуют более сложных алгоритмов их обработки при создании программного обеспечения с их использованием, поднимает задачу создания процедуры, конвертирующей их в модули *первого* типа. При этом, под программным модулем *первого* типа мы будем понимать модуль, именованный конструкцией вида **"Name := module () … ",** тогда как программный модуль *второго* типа характеризуется именующей конструкцией следующего вида **"module Name () … "**. Более того, имеется и *третий* способ именования программных модулей, позволяющий за одно определение задавать функционально *эквивалентные* разноименные модули *второго* типа, как это иллюстрирует следующий весьма простой фрагмент:

```
> M:= module M1 () export x; x:= () -> `+`(args)/nargs end module:  5*M:- x(64,59,39,17,10);
                                        189
> 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(59, 39),  3*M1:- x(64, 17, 10);
                                  [], [true, true], 49, 91
```

Некоторые полезные соображения по использованию программных *модулей второго* типа могут быть найдены в наших книгах [12-14,29-33,39,41,45,103]. Достаточно полезная процедура ***mod21*** решает данную проблему конвертирования программных модулей. Ранее мы уже представляли ее аналог, реализованный нашим методом "*дисковых транзитов*", здесь же была использована упоминаемая выше конструкция типа

***eval*({*parse* | *came*}(<*сгенерированная строка*>)).**

Вызов процедуры ***mod21*(M)** возвращает *NULL*-значение, т.е. *ничего*, обеспечивая в текущем сеансе конвертацию программного модуля второго типа, заданного фактическим аргументом **M,** в эквивалентный ему программный модуль *первого* типа с тем же самым именем. 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 каталог* **f** *отсутствует, то он будет создан с произвольным уровнем вложенности*). В этом случае вызов процедуры ***mod21*** возвращает полный путь к файлу данных с сохраненным отконвертированным модулем **M** и с выводом соответствующего сообщения, информирующем о полном пути к созданному файлу. Файл с сохраненным программным модулем имеет входной ***Maple*-**формат. Примеры нижеследующего фрагмента достаточно прозрачно иллюстрируют сказанное.

```
mod21 := proc(M::`module`)
local a, b, c, d, t, p, k, v, sf, h, ν;
    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( ν = interface( warnlevel) ), null( interface( warnlevel = 0) ),
                assign('h' = cat( MkDir( args[ 2]), "\", M, ".mod1") );
            ( proc() null( interface( warnlevel = ν) ); save M, h end proc )(  ), h;
            WARNING( "module <%1> has been converted into the first type, an
                d saved in datafile <%2>", M, h)
        end if
    end if
end proc
```

```
> restart; 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:/temp/aaa/ccc"),
mod21(A1,"C:/temp/aaa/ccc"), mod21(A2,"C:/temp/aaa/ccc"), mod21(A3,"C:/temp/aaa/ccc"),
mod21(M,"C:/temp/aaa/ccc"), mod21(M1,"C:/temp/aaa/ccc"), mod21(N1,"C:/temp/aaa/ccc"),
mod21(N, "C:/temp/aaa/ccc");
```
Warning, module <A> has been converted into the first type, and saved in datafile
<c:\temp\aaa\ccc\A.mod1>
"C:/temp/aaa/ccc\A1.mod1", "C:/temp/aaa/ccc\A2.mod1", "C:/temp/aaa/ccc\A3.mod1",
"C:/temp/aaa/ccc\M.mod1", "C:/temp/aaa/ccc\M1.mod1", "C:/temp/aaa/ccc\N1.mod1",
"C:/temp/aaa/ccc\N.mod1"
```
> map(type, [A, A1, A2, A3, M, M1, N, N1], 'mod1');
```
[*true*, *true*, *true*, *true*, *true*, *true*, *true*, *true*]

Следует отметить, что именно на данной процедуре мы обнаружили недостатки работы пакетного стэка и нам пришлось редактировать соответствующим образом процедуру, чтобы обеспечить совместимость в рамках ***Maple*** релизов **6-10**. Вопросы работы с модулями *обоих* типов, а также соответствующие для этого средства рассмотрены в наших книгах [12-14,41,42,45,103].

Декларация **description**. Определение программного модуля может содержать декларацию **description,** включающую текстовую строку, являющуюся своего рода кратким документированием данного модуля. Например, строка может содержать краткое описание модуля. Данная компонента определения модуля аналогична *одноименной* компоненте описания процедуры и в результате автоматического упрощения модуля становится самой последней среди всех его деклараций. Декларация определяет одну либо несколько строк, составляющих единое описание модуля. Следующий весьма простой фрагмент иллюстрирует сказанное.

```
> 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, тогда как описание характеризует его экспортируемую **Sr**-переменную.

Декларация **options**. Определение программного *модуля* может содержать подобно случая процедуры опции, определяющие режим работы с модулем. Однако, в отличие от процедур опции ***remember***, ***system***, ***arrow***, ***operator*** и ***inline*** не имеют смысла. Опции кодируются или в форме переменной, или уравнения с переменной в качестве его левой части. Если в **options**-декларации указаны нераспознаваемые языком опции, то они игнорируются, что позволяет пользователю указывать *опции* для собственных нужд, которые распознаются языком как атрибуты.

Из опций модуль допускает ***trace, package*** и **`*Copyright …*`**, аналогичные случаю ***Maple***-процедур [12], и специальные опции ***load = name*** и ***unload= name,*** где ***name*** – имя *локальной* или *экспортируемой* переменной модуля, определяющей процедуру. Данная процедура вызывается при первоначальном создании модуля или при чтении его из системы хранения *функциональных* средств пакета. Как правило, данная опция используется для какой-либо инициализационной цели модуля. Следующий простой фрагмент иллюстрирует применение ***load***-опции для инициализации **Sr**-процедуры:

```
> module Q () local Sr;  export Ds; options load=Sr; Ds:=() -> sqrt(sum((args[k]-Sr(args))^2,
  k = 1..nargs)/nargs);   Sr:= () -> sum(args[k], k = 1..nargs)/nargs;  end module:
> 6*Q:- Ds(39, 44, 10, 17, 64, 59);   ⇒  √14249
```

Опция ***load*** позволяет создавать наборы тематически связанных *экспортируемых* процедур, для инициализации которых используются необходимые локальные неэкспортируемые процедуры и/или функции Данная опция выполняет своего рода инициализационные функции модуля. Опция ***unload = name*** определяет имя *локальной* либо *экспортируемой* процедуры модуля, которую следует 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. Более детально с опциями модуля можно ознакомиться в справке пакета по **?module,option**.

Локальные и глобальные переменные **ПМ**. Аналогично процедурам, **ПМ** поддерживают механизм *локальных* (**local**) и *глобальных* (**global**) переменных, используемых в определении модуля. *Глобальные* переменные имеют областью определения *текущий* сеанс, тогда как область определения *локальных* переменных ограничивается *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 подобного рода приводит к ошибочной ситуации. Результатом вычисления определения модуля является модуль, к экспортируемым членам (*переменным*) которого можно программно обращаться вне области модуля. Следующий простой фрагмент иллюстрирует результат определения *локальных* и *глобальных переменных* программного модуля:

```
> module() local a,b; global c; assign(a=64, b=59); c:= proc() `+`(args) end proc end module:
> a, b, c(42, 47, 89, 96, 67, 62);   ⇒  a, b, 403
> N:= module() export d;  d:= proc()  `+`(args)  end proc end module:
> d(64, 59, 39, 44, 10, 17),  N: -d(64, 59, 39, 44, 10, 17);   ⇒  d(64, 59, 39, 44, 10, 17), 233
> module() local a,b; export a,b,c; assign(a=64, b=59); c:=proc() nargs end proc end module;
Error, export and local `a` have the same name
```

Из приведенного фрагмента видно, что между *глобальными* и *экспортируемыми переменными* модуля имеется существенное различие. Если *глобальная* переменная доступна вне области программного модуля, то для доступа к *экспортируемой* переменной механизм более сложен и в общем случае требует *ссылки* на экспортирующий данную *переменную* модуль 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,** тогда как по вызову процедуры ***with*(M)** возвращается список всех *экспортов* модуля **M;** при этом, во втором случае экспорты модуля **M** становятся доступ-

ными в текущем сеансе и к ним можно обращаться *без ссылок* на содержащий их *модуль*, как это иллюстрирует следующий простой фрагмент:

```
> 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]
> 5*M[Sr](64, 59, 39, 10, 17), 5*M[Dis](64, 59, 39, 10, 17);  ⇒ 189, √11714
> 5*M:- Dis(64, 59, 39, 10, 17), with(M);  ⇒ √11714, [Dis, Sr]
> 5*Sr(64, 59, 39, 10, 17), 5*Dis(64, 59, 39, 10, 17);  ⇒ 189, √11714
> G:= module() global X; export Y; Y:=() -> `+`(args)/nargs; X:=64 end module:
> X, Y(1, 2, 3), G:- Y(1, 2, 3);  ⇒ 64, 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***. По вызову ***exports*(*M*)** получаем последовательность всех экспортов модуля **M,** однако это только имена. Затем по (**:-**)-связке получаем доступ к экспорту ***Dis*** с передачей ему фактических аргументов. Наконец, по вызову **with(M)** получаем список всех экспортов модуля **M,** определяя их доступными в текущем сеансе. Таким образом, глобальность той или иной переменной модуля можно определять или через **global**-декларацию, или через **export**-декларацию модуля. Однако, если в *первом* случае мы к такой 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 **V1**-процедуры. Таким образом, можно 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 (*пакетной или пользовательской*) *последующий* доступ к его *глобальным* переменным стандартными средствами становится невозможным, т. е. указанный прием работает лишь при условии вычисления *определения* модуля в текущем сеансе. Следовательно, описанный выше прием достаточно искусственен и ограничен, и носит в основе своей лишь иллюстративный характер.

Как уже отмечалось, минимальным объектом, распознаваемым пакетом в качестве *модуля*, является определение одного из следующих видов:

M:= module() end module или **module M1 () end module**

о чем говорит и их тестирование, 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]
```

Тогда как попытка получить их *экспорты* посредством вызова **with**-процедуры вызывает ошибочную ситуацию со следующей релизо-зависимой диагностикой:

```
> M:= module () end module: module M1 () end module:
> with(M);                                                    Maple 6 - 8
Error, (in pacman:-with) module `M` has no exports
> with(M1);
Error, (in pacman:-with) module `M1` has no exports
```

```
> with(M);                                                    Maple 9 - 10
Error, (in with) module `M` has no exports
> with(M1);
Error, (in with) module `M1` has no exports
> lasterror;   ⇒  "module `%1` has no exports"                Maple 6 - 10
> map(exports, [M, M1]);   ⇒  []
```

Тогда как во всех релизах переменная ***lasterror*** получает *идентичное* значение. Поэтому для обеспечения более простой работы с программными модулями рекомендуется для получения их экспортов использовать встроенную функцию ***exports,*** как иллюстрирует последний пример фрагмента, либо использовать нашу процедуру [103], которая является *расширением* стандартной процедуры ***with*** и которая не только в данной ситуации возвращает корректный результат, но и позволяет использовать себя внутри процедур в отличие от ***with,*** например:

```
With := proc(P::{ `module`, package }, 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 ], "/_$Art17_Kr9$_" ) )
        , 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(M), With(M1);   ⇒  [], []
```

Это обстоятельство следует учитывать при работе с модулями в программном режиме.

Так как модули, как правило, содержат процедуры, а те и другие поддерживают механизм *локальных* переменных, то при наличии неявно определенных *локальных* переменных язык при выводе об этом предупреждений привязывает их к содержащим их *объектам*, как это иллюстрирует следующий достаточно простой фрагмент:

```
> GRSU:= module() local a; export b; global t; a:= proc() c:= 64; h:= 10 end proc;
  b:= proc() d:= 17 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`
> e:= proc() g:= 59 end proc: t:= proc() v:= 64 end proc:
Warning, `g` is implicitly declared local to procedure `e`
Warning, `v` is implicitly declared local to procedure `t`
```

***Декларация* export** определяет последовательность локальных имен объектов модуля, к которым возможен доступ извне модуля, т.е. определяет локальные переменные (*присущие сугубо модулю и скрываемые от внешней среды*) *глобальными*. В отличие от *локальных экспортируемые* переменные модуля не могут быть определены таковыми *неявно*. Их следует определять явно через **export**-декларацию. Важно помнить, что при *экспортировании* модулем неопределенной локальной переменной она не тождественна одноименной глобальной переменной текущего сеанса, что иллюстрирует следующий пример:

> module R() export VG; VG:= () -> `+`(args) end module: evalb(VG= R:- VG); ⇒ *false*
> VG:= 64: VG, R:- VG(64, 59, 39, 10, 17); ⇒ 64, 189

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 иллюстрирует следующий простой фрагмент (*см. прилож.* **6** [13]):

```
> GU:=module() local k; export ln; ln:=()->evalf(sum(log(args[k]),k=1..nargs)) end module:
> ln(42, 47, 67, 89, 96, 64);
Error, (in ln) expecting 1 argument, got 6
> GU[ln](42, 47, 67, 89, 96, 64);    ⇒ 25.00437748
```

Данное свойство, в частности, достаточно полезно при создании собственных функций пользователя, *подобных* пакетным функциям, но отличающимися какими-либо особенностями. При этом, имеется хорошая возможность сохранять за ними пакетные имена.

Вне программного модуля обращение к *экспортируемым* переменным модуля производится по конструкциям следующего общего вида:

***Имя_модуля*:- *Имя_экспортируемой_переменной*{(*Аргументы*)}**
***Имя_модуля*[*Имя_экспортируемой_переменной*]{(*Аргументы*)}**

Более того, выполнение предложения ***with*(*Имя_модуля*)** позволяет обращаться ко всем экспортируемым переменным модуля только по их именам, делая их доступными в текущем сеансе для любого активного либо находящегося в очереди готовых документов, как это иллюстрирует следующий весьма простой фрагмент:

```
> 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 данный механизм в значительной степени позволяет облегчить имплантирование в среду пакета функциональных средств из других программных систем.

Каждое определение ***Maple***-процедуры ассоциируется с неявными переменными ***args***, ***nargs*** и ***procname***, рассмотренными выше. Тогда как с определением **ПМ** ассоциируется только одна неявная ***thismodule***-переменная. В рамках тела модуля данной переменной присваивается содержащий ее модуль. Это позволяет ссылаться на модуль в рамках его собственного определения. Следующий простой фрагмент иллюстрирует применение переменной ***thismodule:***

```
> module AVGSv() export a, b; a:= () -> sum(args[k], k=1..nargs); b:= thismodule:- a(42, 47,
  67, 62, 89, 96) end module: AVGSv:- a(64, 59, 39, 44, 17, 10), AVGSv:- b;   ⇒ 233, 403
```

Посредством ***thismodule***-переменной предоставляется возможность организации рекурсивных выполнений модуля внутри самого модуля, что *существенно* расширяет возможности модульного программирования в среде ***Maple***-языка пакета.

По функции ***op*** можно получать доступ к трем компонентам модуля **M,** а именно:

1) op(1, eval(M)) – последовательность *экспортов* модуля **M**
2) op(2, eval(M)) – оболочка *определения* 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:= 64, 59; 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. Она используется только для качественной печати модуля.

Дополнительно к ранее рассмотренным, для модулей в релизе **10** введен ряд дополнительных *переменных*, а именно. Если модуль **M** экспортирует переменную ***ModuleApply,*** то вызов вида **M(*args*)** обеспечивает *вызов* процедуры **M:-ModuleApply(*args*),** например:

```
> M:=module() export ModuleApply; ModuleApply:=()->evalf(`+`(args)/nargs) end module:
> M(64, 59, 39, 17, 10, 44);   ⇒ 38.8333333300
```

Если модуль содержит *локальную* либо *экспортируемую* переменную ***ModuleLoad,*** то определенная ею процедура вызывается, когда модуль читается из содержащей *его* ***Maple***-библиотеки. Если модуль имеет локальную либо экспортируемую ***ModuleUnload-***переменную, то определенная ею процедура вызывается, когда модуль больше недоступен или в случае завершения сеанса с ***Maple***. Две последние переменные являются зеркальными средствами опций ***load*** и ***unload*** модуля. Если модуль имеет локальную или экспортируемую ***ModulePrint****-переменную,* то вместо модуля возвращается результат вызова ***ModulePrint*****().** Более детально с данными переменными модуля можно ознакомиться в справочной системе ***Maple*** по запросу **?module.** Как следует из их описания, данные переменные ничего особо существенного не несут, однако в ряде случаев могут оказаться достаточно полезными.

Механизм вызовов ***Maple*** не позволяет использовать динамически генерируемые имена для экспортов модуля. Данную задачу решает процедура ***dcemod*** [41,103,109]**.**

```
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; fremove(a) end proc )(  ), _v63g58,
    unassign('_v63g58')
end proc
```

Параметризация модулей. В отличие от процедур, программные *модули* не используют механизма *формальных* аргументов. Поэтому для использования **ПМ** в задачах параметрического программирования используется следующая *общего* характера 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
> R95_06:= SveGal(95, 99):  R95_06:- Gal(59, 64, 39),  R95_06:- Sv(39, 44, 10, 17);
                                        27/64, 291720
```

Параметризация позволяет создавать модули, легко настраиваемые на конкретные условия применения, что обеспечивает их гибкость и мобильность при программировании задач из различных приложений. Вышеприведенный простой фрагмент иллюстрирует применение данной конструкции для параметризации конкретного программного модуля, определенного в процедуре ***SveGal.*** Процедура ***SveGal*** в качестве *формальных* аргументов **a** и **b** использует параметры вложенного в нее модуля, экспортирующего две функции ***Gal*** и ***Sv***. Присвоение вызова данной процедуры с конкретными фактическими аргументами некоторой переменной генерирует поименованный первым способом программный модуль, параметры **a** и **b** которого получают конкретные значения. Следовательно, мы получаем программный модуль, настроенный на определенные значения его параметров. В дальнейшем такой модуль используется описанным выше способом. Описанный механизм позволяет производить *параметризацию* программных модулей, что обеспечивает решение разнообразных задач параметрического программирования в среде пакета ***Maple***.

Представленные в книге [103] процедуры нашей *Библиотеки* представляют целый ряд весьма полезных средств для работы с программными и пакетными модулями, существенно дополняющими имеющиеся стандартные средства пакета. Вместе с тем, данные средства позволяют детализировать сами модульные ***Maple***-объекты и использовать их особенности для программирования задач с использованием подобных объектов. Так, в главе **3** [41] представлена группа средств, расширяющих возможности пакета ***Maple*** релизов **6 – 10** при работе с процедурами и программными модулями. Данные средства поддерживают такие виды обработки как преобразование модулей в процедуры, проверка наличия в файлах некорректных модулей, проверка аргументов процедур и модулей, проверка активности (*пригодности к непосредственному использованию*) процедуры или модуля, проверка типа модульной таблицы, преобразование файлов *входного* формата ***Maple,*** содержащего модули, преобразование модуля второго типа в первый тип, преобразование файла входного формата ***Maple*** в файл внутреннего формата ***Maple,*** и наоборот, и т.д. Представленные инструментальные средства обеспечивают набор разнообразных полезных операций с процедурными и модульными объектами ***Maple.*** Эти инструментальные средства используются достаточно широко при расширенном программировании различных приложений в ***Maple*** и в целом ряде случаев весьма существенно упрощают программирование.

4.3. *Сохранение процедур и модулей в файлах*

Maple-язык располагает средствами *сохранения* в файлах процедур и программных модулей с возможностью их последующего *чтения* как в текущем сеансе, так и после перезагрузки пакета. Для сохранения *процедур* и программных *модулей* в файле служит **save**-предложение языка, имеющее следующие простые форматы кодирования:

save N1, N2, ..., Nk, <*Файл*> или **save(N1, N2, ..., Nk,** <*Файл*>**)**

где **N1, N2, ..., Nk** – последовательность *имен* сохраняемых объектов и *Файл* – имя файла или полный путь к нему типа {*string*, *symbol*}. Объекты ***Maple-***языка сохраняются в файлах в одном из двух форматов (*входном* ***Maple-****формате*, *в терминах* ***DOS*** *эквивалентном формату* ***ASCII****, и внутреннем* ***m-****формате*). Вызов **save(N1, N2, ..., Nk,** <*Файл*>**)** либо выполнение предложения **save N1, N2, ..., Nk,** <*Файл*> пишет в заданный файл определения имен **N1, N2, ..., Nk** в виде последовательности предложений присвоения. При попытке сохранения неопределенного имени **A** в файл пишется предложение **A:= A**. Успешный вызов **save(…)** возвращает *NULL*-значение, т.е. ничего.

При этом, если в *файле* определено ***m-***расширение имени, то файл сохраняется во *внутреннем* ***m-***формате пакета, в противном случае используется *входной* формат ***Maple-***языка, т. е. (***ASCII-****формат*). *Внутренний* ***m-***формат используется, чтобы сохранять процедуры, модули и другие объекты в более компактном, простом для чтения пакетом формате. Объекты, сохраненные во *внутреннем* формате, могут читаться быстрее (*прежде всего*, *при файлах большого объема*) чем объекты, сохраненные во *входном* ***Maple***-формате языка. Следующий весьма простой пример иллюстрирует сохранение *целочисленного* **L**-списка из 1000 элементов в файлах обоих форматов, из которого следует, что файл ***"file"*** входного формата занимает 5024 байта, тогда как файл ***"file.m"*** *внутреннего* формата *только* 4026 байтов.

> L:= [k$k=1 .. 1000]: save(L, "C:\\temp\\file"); save(L, "C:\\temp\\file.m");

При этом, для больших сохраняемых объектов эта разница может быть *довольно* существенной. В случае указания в **save-**предложении неопределенных ***Id-***идентификаторов, они сохраняются в виде **Id := 'Id'** с выводом соответствующих предупреждающих сообщений, например:

```
> save(h, g, s, `D:\\Academy\\file`);
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*) представляют ***Maple***-объекты, сохраняемые в библиотеках пакета (*т.н.* ***Maple****-библиотеках*) и *идентичных* с ними по организации. Однако, здесь имеется и одно существенное «но». Файлы с ***Maple***-объектами, сохраненными во *входном* ***Maple***-формате посредством предложения **save**, мобильны относительно всех релизов пакета (*хотя они и могут быть синтаксически зависимы от релиза*), тогда как файлы внутреннего ***m***-формата всегда релизо-зависимы. Попытка чтения ***m***-файла (*не соответствующего текущему релизу*) посредством предложения **read** вызывает *ошибочную* ситуацию, тогда как чтение файла во входном ***Maple***-формате корректно в *любом* релизе, если *определения* процедур и модулей, расположенных в нем, не содержат каких-либо *релизо*-зависимых синтаксических элементов. В нынешних реалиях ***m***-файлы, созданные в среде пакета ***Maple 6***, *несовместимы* с ***m***-файлами, созданными в среде релизов ***7 – 10***, тогда как в рамках релизов ***7 – 10*** имеет место *полная* совместимость. Обусловлено это изменением соответствия между *идентификационными* номерами внутренних структур

данных и их именами. В дальнейшем мы будем называть файлы, корректно читаемые предложением **read,** файлами пакета или ***Maple***-файлами.

Следующий простой фрагмент иллюстрирует применение **save-**предложения для создания файлов обоих указанных форматов (*входного и внутреннего*)**:**

```
> Sr:= () -> `+`(args)/nargs:  save(Sr, "C:/Temp/Sr");  save(Sr, "C:/Temp/Sr.m");
Sr := proc () options operator, arrow; `+`(args)/nargs end proc;        - входной формат
M7R0                                                                    - m-формат
I#Srf*6"F$6$%)operatorG%&arrowGF$*&-%"+G6#9""""9#!""F$F$F$F$
```

В данном фрагменте определяется простая ***Sr***-процедура и по **save-**предложению сохраняется в ***Sr-файлах*** во *входном* (**"Sr"**) и *внутреннем* (**"Sr.m"**) форматах. Содержимое обеих файлов выводится. На небольшого объема файлах не ощущается преимущества от того либо иного формата, однако при возрастании объема предпочтение внутреннего ***m-***формата становится все ощутимее, позволяя создавать более компактные и быстрочитаемые пакетом файлы данных.

Загрузка сохраненного по **save-**предложению файла данных выполняется по предложению **read,** которое имеет следующие простые форматы кодирования:

read <*Файл*> или **read(**<*Файл*>**)**

где *Файл* – имя файла или *полный* путь к нему типа {*string*, *symbol*}; при этом, если файл указывается только именем, а не полным путем к нему, то предполагается, что он находится в текущем каталоге. Между тем, результат **save-**предложения зависит от формата *загружаемого* файла. Общим является 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**,** то 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. Примеры следующего фрагмента иллюстрируют применение **read-**предложения для *загрузки* файлов данных обоих форматов (*входного и внутреннего*):

```
> restart;  read("C:/Temp/Sr");  5*Sr(64, 59, 39, 10, 17);      ⇒ 189
```

$$Sr := (\ ) \rightarrow \frac{\text{`+`(args)}}{\text{nargs}}$$

```
> restart;  read("C:/Temp/Sr"):  5*Sr(64, 59, 39, 10, 17);      ⇒ 189
> restart;  read("C:/Temp/Sr.m");  5*Sr(64, 59, 39, 10, 17);    ⇒ 189
> restart;  read("C:/Temp/Sr.m"):  5*Sr(64, 59, 39, 10, 17);    ⇒ 189
```

На первых порах работы с ***Maple-***языком средства доступа к *внутренним* ***m-***файлам наиболее полезны при необходимости создания различного рода библиотек пользовательских процедур или функций, а также сохранения часто используемых ***Maple-***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 (*если до выгрузки они были вычисленными*)**,** делая их доступными в *текущем* сеансе работы с пакетом.

В ***Maple 6*** **save**-предложение допускает формат **save(F),** по которому пишет все вычисленные в текущем сеансе имена в файл **F** как последовательность предложений присвоения. Кроме того, данная возможность предоставляется только для **F**-файлов данных,

чьи имена завершаются "**.m**", т.е. файлы *внутреннего* ***Maple***-формата. Однако, начиная с ***Maple 7,*** такая возможность отсутствует. Между тем, в ряде случаев подобное средство может упростить программирование и облегчить работу с ***Maple***. В этом плане может оказаться достаточно полезной наша процедура ***saveall*** [41-43,103,108,109].

```
saveall := proc(F::{string, symbol})
local _avz_agn_asv_, a, k, j, t, ψ, p, f;
    _avz_agn_asv_ := ((
        ({anames( )} minus {anames('environment')}) minus {anames('builtin')}
        ) minus {anames('package')}) minus {'`type/interfaceargs`', 'lasterror',
        'lastexception', '`Context/InitProcs`', 'ContextKey', '`context/InitProcs`',
        'ContextData', 'stack', 'interface'};
    _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(ψ = (a → convert(eval(a), 'string'))));
    _avz_agn_asv_ := {seq(`if`(
        Search2(ψ(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\\aaa\\bbb\\file");
Warning, current session does not contain user definite objects suitable for saving
> saveall("c:\\temp\\grodno\\bbb\\ccc\\save169.m");
                              "c:\temp\grodno\bbb\ccc\save169.m"
```

Вызов процедуры ***saveall*(F)** возвращает *реальный* путь к принимающему файлу данных (*имя которого либо путь к нему определены фактическим аргументом* **F**), содержащему все вычисленные в текущем сеансе *имена* в виде последовательности предложений присво-

ения. При этом, процедура не сохраняет встроенные средства, переменные среды пакета, пакетные модули, библиотечные средства в рамках всех библиотек, определенных предопределенной ***libname***-переменной ***Maple***. При отсутствии сохраненных имен вызов процедуры возвращает *NULL*-значение, т. е. *ничего*, с выводом соответствующего сообщения, как это иллюстрирует предыдущий фрагмент. В целом ряде случаев ***saveall***-процедура имеет достаточно полезные приложения.

Еще на одном весьма существенном аспекте следует заострить внимание. Предложение **save** некорректно сохраняет программные модули, 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(64, 59, 39, 17, 10, 44),  x(64, 59, 39, 17, 10, 44);
                          x(64, 59, 39, 17, 10, 44), x(64, 59, 39, 17, 10, 44)
> M1:- y(64, 59, 39, 17, 10, 44),  y(64, 59, 39, 17, 10, 44);
                          y(64, 59, 39, 17, 10, 44), y(64, 59, 39, 17, 10, 44)
> M2:- z64, 59, 39, 17, 10, 44),  z(64, 59, 39, 17, 10, 44);
                          z(64, 59, 39, 17, 10, 44), z(64, 59, 39, 17, 10, 44)
```

Из приведенного фрагмента видно, что сохраненные по **save**-предложению в ***m***-файле модули **M, M1** и **M2,** затем читаются **read**-предложением в текущий сеанс и распознаются ***type***-функцией действительно как *модули*. Более того, вызов **with**-процедуры вполне корректно возвращает списки экспортируемых модулями переменных. Тогда как стандартные *вызовы* этих переменных возвращаются невычисленными, т.е. экспортируемые такими модулями переменные оказываются неопределенными. Причина лежит в том, что **save**-предложение не сохраняет в ***m***-файлах внутреннего ***Maple***-формата тела программных модулей.

Весьма детальное обсуждение данного вопроса может быть найдено в наших книгах [29-33,39,42-44,103]. Для устранения подобного недостатка нами был предложен ряд интересных средств, *существенно* расширяющих функциональные возможности стандартных предложений **save** и **read,** с которыми можно ознакомиться в нашей книге [103] и в прилагаемой к ней *Библиотеке*. В частности, предложение **save** не позволяет сохранять в файле динамически вычисляемые имена, что в целом ряде случаев представляется существенным недостатком. Наша процедура ***save1*** [103] устраняет данный 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 присваивалось значения, 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 данных **F,** обеспечивающему присвоения выражений **E** соответствующим символам из **N** в текущем сеансе ***Maple*** с выводом необходимых сообщений.

Данная процедура существенно расширяет возможности предложения **save,** обеспечивая сохранение в файлах данных и входного формата ***Maple,*** и внутреннего ***Maple*** формата присвоений переменным с динамически генерируемыми именами. В целом ряде

задач это – весьма важная возможность. Более того, процедура поддерживает *сохранение* ***Maple***-объектов в файлах данных с *произвольными* путями к ним. Примеры фрагмента, представленного ниже, иллюстрируют некоторые из наиболее типичных применений ***save1***-процедуры.

```
save1 := proc(N::{symbol, list(symbol)}, E::anything, F::{string, symbol})
local a, b, c, k, ν, ω, ζ, ψ, r, s, x;
    assign( b = (x → null( interface( warnlevel = x ) ) ), ζ = "_$Euro$_", ν = "save(",
        s = "symbols %1 are protected");
    if not type( eval(F), {'string', 'symbol'} ) 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;
    ψ := 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, ∞ ) ), close(f), fremove( a ) )
        end proc;
    ω := proc(N::{symbol, list(symbol)}, E::anything, F::{string, symbol})
        local k;
            `if`( type( N, 'symbol'), assign( 'ν' = cat( ν, N, "," ) ),
                seq( assign( 'ν' = cat( ν, N[k], "," ) ), k = 1 .. nops(N) ) ),
                writeline( ζ, cat( ν, """, r, """, "):") ), close( ζ );
            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(ζ) read ζ; fremove( ζ ) end proc )( ζ )
        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([
            ω(a, args[2], r), ψ(r), `if`(nops(a) = nops(N), NULL,
            WARNING(s, {op(N)} minus {op(a)}))]))
    else ω(args[1 .. 2], r), ψ(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)], [59, 64, 39,
  43, 10, 17], "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",  59, 64, 39, 43, 10, 17
> restart; read("C:\\Academy/RANS\\IAN.m"); G, xyz, abc, ht, RANS, Sv;
                                    59, 64, 39, 43, 10, 17
> save1([cat(x, y, z), cat(y, z), cat(z, 6)], [proc() `+`(args)/nargs end proc,59,39], "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, 59, 39
```

Приведем еще пару процедур, *расширяющих* предложения **save** и **read** по работе с программными модулями пакета. Обе процедуры ***savem1*** и ***readm1*** имеют форматы кодирования следующего вида:

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 или их последовательность. При этом, модули не должны иметь *protected*-атрибута. Процедура ***savem1*** предназначена для *сохранения* программных модулей первого типа в файлах внутреннего ***Maple***-формата (***m****-файлах*); если в качестве **F**-аргумента указан не ***m***-файл, то к его имени добавляется расширение "**.m**". Успешный вызов процедуры 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 ранее в нем по ***savem1***-процедуре программных модулей. Успешный вызов процедуры возвращает *NULL*-значение, т.е. *ничего*. Ниже представлен фрагмент с исходными текстами обеих процедур и примерами их применения для сохранения/чтения программных 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
```

```
> 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"
```

```
readm1 := proc(F::file)
local b, c, k;
    assign(b = {  }, c = 10/17);
    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
```

```
> restart;  readm1("C:/temp/pdf.htm");
Error, (in readm1) file should has internal Maple format, but had received `.htm`-type
> readm1("C:/temp/grsu.m");
Error, (in readm1) file <C:/temp/grsu.m> should be saved by procedure `savem1`
> 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(64, 59, 39, 17, 10, 44),  x(64, 59, 39, 17, 10, 44);
                                  183589120, 183589120
> 6*M1:- y(64, 59, 39, 17, 10, 44),  6*y(64, 59, 39, 17, 10, 44);   ⇒ 233, 233
> M2:- z(64, 59, 39, 17, 10, 44),  z(64, 59, 39, 17, 10, 44);       ⇒ 233, 233
> M3:- h(64, 59, 39, 17, 10, 44),  h(64, 59, 39, 17, 10, 44);       ⇒ 15.264, 15.264
```

Обе процедуры обрабатывают основные особые и ошибочные ситуации. Алгоритм *первой* процедуры базируется на *строчном* представлении определения программного модуля, сохраняемого в *специальном* формате в файле *внутреннего* формата. Тогда как *вторая* процедура использует представление сохраненного в ***m***-файле модуля, используя структуру файлов такого типа и специальную процедуру ***Iddn1***, обеспечивающую возврат имен ***Maple***-объектов, сохраненных в файлах *внутреннего* ***Maple***-формата. В качестве весьма полезного упражнения читателю рекомендуется рассмотреть организацию обоих процедур.

В ряде случаев возникает необходимость анализировать ***m***-файл внутреннего формата ***Maple*** на предмет находящихся в нем объектов. Данную задачу решает ***mnames***-процедура, исходный текст и примеры применения которой приведены ниже. Вызов процедуры ***mnames*(F)** для случая анализа в среде ***Maple 6*** файла **F**, подготовленного в среде ***Maple 7 - 10***, возвращает только множество имен объектов с выводом 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 nam\
            es 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] := {h, op(tab[t])}
                else tab[t] := {h, op(tab[t])}
                end if
            end do;
            `if`(nargs = 1, NULL, Unassign(op(a))), eval(tab)
        end if
    end if
end proc
```

> mnames("C:/Temp/Test8.m", 10); 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* = {*Grod*}, *procedure* = {P}])

AGN, *L1*, *S*, *Akr*, *Grod*, *AVZ*, *m*

Тогда как аналогичный вызов ***mnames*(F)** для случая анализа в среде ***Maple 7 - 10,*** файла **F,** подготовленного в среде ***Maple 6 - 10,*** не только возвращает таблицу, чьими входами являются типы объектов файла, а выходами множества соответствующих им имен, но и делает эти объекты активными в текущем сеансе. Тогда как вызов ***mnames*(F, P)** при тех же предположениях, где **P –** произвольное ***Maple***-выражение, оставляет имена объектов файла **F** неопределенными, возвращая при этом вышеуказанную таблицу. Процедура неоднократно успешно использовалась для анализа незнакомых файлов внутреннего ***Maple***-формата. Целый ряд других полезных средств для сохранения программных модулей в файлах *внутреннего* ***Maple***-формата, а также для работы с файлами указанного формата представлен в наших книгах [41,42,103] и в прилагаемой к ним *Библиотеке* для пакета ***Maple*** релизов **6 – 10.** Наряду с практическим интересом данные средства представляют определенный интерес и для практического программирования в ***Maple,*** предлагая целый ряд полезных и эффективных приемов программирования.

Глава 5. *Средства Maple-языка для работы с данными и структурами строчного, символьного, списочного, множественного и табличного типов*

В предыдущей главе был представлен ряд *базовых функциональных* средств ***Maple***-языка по обеспечению работы с основными типами данных и структур данных. Здесь мы дадим более полное, хотя и не исчерпывающее представление данных средств, которые наиболее часто используемы в программировании различных ***Maple***-приложений.

5.1. *Средства работы Maple-языка с выражениями строчного и символьного типов*

В настоящей главе рассмотрим *базовые* средства работы в среде ***Maple***-языка с данными *строчного* (***string***) и *символьного* (***symbol***) типов более детально. Для тестирования выражений типа ***string*** используется уже упоминаемая ***whattype*(S)-**процедура, возвращающая *string*-значение, если результат вычисления **S-**выражения имеет *string*-тип. Тестирование можно осуществлять и по ***type*(S, 'string')-**функции, возвращающей ***true***-значение в случае **S-**строки и ***false*** в противном случае. Для тестирования выражений типа ***symbol*** используется уже упоминаемая ***whattype*(S)-**процедура, возвращающая *symbol*-значение, если результат вычисления **S-**выражения имеет *symbol*-тип. Тестирование можно производить и по ***type*(S, '*symbol*')-**функции, возвращающей ***true***-значение в случае **S-**символа и ***false*** в противном случае. Данные *строчного* и *символьного* типов играют весьма *важную* роль при операциях ввода/вывода, работе с текстовой информацией, символьных вычислениях и преобразованиях и др. При этом, данные этих типов наиболее широко используются, в первую очередь, в задачах символьных обработки и вычислений [8-14,41,78-90,55,58-62,103,108,109].

Символьные значения представляются как простыми идентификаторами, так и составными, ограниченными верхники кавычками (`); при этом, любая цепочка символов, ограниченная кавычками, рассматривается в качестве символьного значения. Тестирующими средствами ***Maple***-языка символьные выражения распознаются как значения типа {*symbol*, *name*}, как это иллюстрирует следующий весьма простой пример:

```
> whattype(AV), whattype(`A V Z`), whattype(Academy_Noosphere);
                        symbol, symbol, symbol
> type(AV, 'symbol'), type(`A V`, 'symbol'), type(Academy_Noosphere, 'name');
                          true, true, true
```

Строчные выражения представляются значениями, ограниченными двойными кавычками ("); при этом, *любая* цепочка символов, ограниченная такими кавычками, рассматривается в качестве *строчного* значения. Тестирующими средствами ***Maple***-языка *строчные* значения распознаются как значения *string*-типа, как это иллюстрирует следующий простой пример:

```
> whattype("AV"), whattype("A V Z"), whattype("Academy");   ⇒ string, string, stringl
> type("AV",'string'), type("A V",'string'), type("Academy_2006", 'string');  ⇒ true, true, true
```

При этом, со строчной информацией на уровне *отдельного* символа ***Maple-***язык работает как со строками длины **1,** например ***length*("A")=1,** что не требует специального опре-

деления типа для отдельных символов. В дальнейшем для *строчных* и *символьных* выражений будем использовать групповое имя *«символьные»* выражения. Для конвертации *строчных* выражений в *символьные* и наоборот служит ***convert***-функция, как это иллюстрирует следующий простой фрагмент:

```
> convert("AVZ", 'symbol'), convert(AVZ, 'string');     ⇒ AVZ, "AVZ"
> ``||"AVZ", ""||AVZ, cat(``, "AVZ"), cat("", AVZ);    ⇒ AVZ, "AVZ", AVZ, "AVZ"
> cat(a + b, 'string'), ""||(a + b);   ⇒ ||(a + b)||string, ""||(a + b)
```

Конвертацию можно проводить и посредством ***cat***-функции и **||**-оператора конкатенации, как иллюстрирует второй пример фрагмента. Тогда как последний пример подтверждает, что именно ***convert***-функция является наиболее общим средством подобной конвертации символьно/строчных выражений.

Для обеспечения работы со *строчными* выражениями ***Maple***-язык располагает рядом *базовых* средств манипулирования со строками, пригодными, в свою очередь, и для обработки символьных значений. Немногочисленные средства собственно ***Maple***-языка по обеспечению работы с данными указанных двух типов достаточно прозрачны и в том или ином виде имеются во всех *современных* системах программирования, поэтому особых пояснений не требуют. Отметим лишь средства, наиболее часто используемые в практическом программировании в среде ***Maple.*** Их специфические свойства и особенности достаточно детально рассмотрены в [8-14,29,30,41,103,108].

Прежде всего, для объединения (*конкатенации*) *строк* и/или *символьных* значений предназначены ***cat***-функция и **||**-оператор конкатенации, имеющие следующие довольно простые форматы кодирования, а именно:

cat(S1, S2, ..., Sn) и **S1||S2||** ... **||Sn**

возвращающие результат *конкатенации* **Sk**-строк/символов. В случае использования бинарного **||**-оператора в качестве первого операнда должно быть *символьное* либо *строчное* выражение. Тип возвращаемого результата в обоих случаях определяется типом *первого* соответственно фактического аргумента и операнда, например:

```
> map(whattype, map2(cat, "TRG", ["IAN", RANS]));      ⇒ [string, string]
> map(whattype, map2(cat, TRG, ["IAN", RANS]));        ⇒ [symbol, symbol]
> whattype("TRG"||RANS), whattype(TRG||"RANS");  ⇒ string, symbol
```

При этом, следует отметить, что в качестве *первого* операнда **||**-оператора могут выступать и более общего вида выражения, однако как для ***cat***-функции, так и для оператора *конкатенации* имеет место ряд особенностей (*детально представленных в* [12-14,39]), позволяющих рассматривать их, прежде всего, *именно* как средства работы со *строчными* данными и структурами. По ***substring***-функции, имеющей простой формат кодирования:

substring({"<*Строка*>"|`<*Символ*>`}**, m|, m..n**)

возвращается соответственно {*строка*|*символ*}, находящиеся на **m**-й позиции заданного первым фактическим аргументом значения {*string*|*symbol*}-типа или начинающиеся с **m**-й и заканчивающиеся **n**-й позицией. В случае отрицательных значений позиций отсчет производится, начиная с *правой* границы исходных {*строки*|*символа*}. Тогда как вызов ***length***(<*Выражение*>)-функции возвращает *длину* результата вычисления выражения и в случае его типа {*string*| {*symbol*|*name*}} – длину {*строки*|*символа*}. Для *целых* числовых значений возвращается *число цифр*, исключая знак; тогда как для других ***Maple***-выражений по ***length***-функции возвращается результат, определяемый как сумма длин каждого операнда выражения, вычисляемых *рекурсивно*, и числа слов, используемых для пред-

ставления исходного выражения. Данный результат определяет своего рода числовую меру сложности выражения. Следующий простой фрагмент иллюстрирует сказанное**:**

```
> H:= `Russian Academy of Natural Sciences`:  M:= "Tallinn Research Group":
> map(length, [H, M, 42.64, -59, cat(H, M), ``||H||M, sin(x)+y^2]);  ⇒ [35,22,8,2,57,57,21]
> cat(substring(H, 9 .. 15),`, `, substring(M, 9 .. length(M)));  ⇒ Academy, Research Group
> cat(sin(x), TRU), cat(convert(sin(x), 'string'), TRU);  ⇒ || (sin(x)) || TRU, "sin(x)TRU"
```

Таким образом, если относительно функций ***cat*** и ***length*** допускаются значения фактических аргументов типов {*string*, *symbol, name*} с учетом того, что тип возвращаемого ***cat***-функцией результата определяется типом ее первого аргумента, то уже в общем случае **||**-оператора *конкатенации* требуется специальный формат его кодирования. Тогда как последний пример фрагмента хорошо иллюстрирует тот факт, что в *общем* случае *произвольное* ***Maple***-выражение, выступающее в качестве аргумента (*операнда*) ***cat***-функции (**||**-*оператора*), предварительно следует конвертировать в строку либо символ.

По функции {***searchtext*|*SearchText***}, имеющей следующий формат кодирования**:**

{***searchtext*|*SearchText***}**(**`<*Контекст*>`**,** {**"**<*Строка*>**"|**`<*Символ*>`}{**, m..n**}**)**

производится *регистро*-{*независимая*|*зависимая*} идентификация факта *вхождения* заданного *контекста* в заданные {*строку*|*символ*}. Если закодирован и третий аргумент (*позиции*), то поиск осуществляется относительно указанного местоположения *контекста* в {*строке*|*символе*}**.** Сказанное о втором аргументе ***substring***-функции полностью сохраняет силу и для третьего аргумента {***searchtext*|*SearchText***}-функции. Обе функции поиска возвращают *позицию* первого вхождения *контекста* в *строку***;** *нулевое* значение говорит об отсутствии в строке искомого *контекста*. При этом, следует иметь в виду, что *возвращаемый* по ***searchtext***(*Контекст***, S, n..p)**-вызову номер позиции (***m***) *первого* вхождения искомого *контекста* в **S-**строку в диапазоне [**n..p**] отсчитывается не от начала строки, а от **n-**позиции, как *базовой*; т.е. относительно начала строки номер позиции определяется как (***n+m***), что следует учитывать при программировании с использованием данного средства ***Maple***-языка.

При этом, на основе ***SearchText***-функции ***Maple***-язык предоставляет и недокументированную процедуру ***search*****(S, s** {**, 't'**}**),** которая тестирует факт вхождения **s**-подстроки в **S**-строку (*в качестве* **s** *и* **S** *могут выступать символы и/или строки*), возвращая соответственно *true* или *false*. Если при вызове процедуры быд определен и третий необязательный **t**-аргумент, то через него возвращается позиция первого вхождения **s** в **S;** при отсутствии такого вхождения **t**-аргумент возвращается *невычисленным*. Следующий простой пример хорошо иллюстрирует вышесказанное:

```
> searchtext(`_`, Academy_Nooshpere), [search(Academy_Nooshpere, `_`, 't'), p];
                                  8, [true, 8]
> SearchText(n, Academy_Nooshpere), SearchText(N, Academy_Nooshpere);  ⇒ 0, 9
```

По конструкции вида {**\n**} можно определять режим переноса строчного выражения на новую строку, кодируя его в нужных местах длинных *строчных* или *символьных* выражений, например:

```
> `Address: \nRaadiku 13 - 75\nTallinn 13817\nEstonia\nwww.aladjev.newmail.ru`;
```

Address:
Raadiku 13 – 75
Tallinn 13817
Estonia
www.aladjev.newmail.ru

Для обеспечения синтаксического анализа строчной структуры, заключающей в себе ***Maple-***выражение/предложение, служит ***parse-***функция, имеющая форматы**:**

***parse*(**{ |"<*Строка*>"|"<*Строка*>"**,** <*Параметры*>}**)**

***parse*(**{ |`<*Символ*>`|`<*Символ*>`**,** <*Параметры*>}**)**

Предполагается, что содержимое аргумента "*Строка*" определяет *одно* ***Maple-****выражение* либо *предложение*, если указан ***statement-***параметр в качестве второго аргумента функции. Данная функция производит *синтаксический* анализ заданной своим первым аргументом *строки*, как если бы она была введена в ***Input-***параграфе или считана из файла**.** Выражение/предложение, *составляющее* строку, анализируется на соответствие синтаксису ***Maple-***языка и возвращается *невычисленным***.** Если же в результате анализа обнаружена ошибка, то выводится сообщение вида**:** "Error, incorrect syntax in **parse:** <*Тип ошибки*> (***m***)"**,** где ***m-***число определяет *позицию* строки, с которой ассоциируется обнаруженная ошибка указанного типа. Данное сообщение возможно использовать для обработки ошибочных и особых ситуаций. Даже в случая обнаружения ***parse-***ошибки по (%)-конструкции возвращается значение *строки невычисленным***.** В случае второго формата ***parse-***функции *сохраняет* силу все сказанное относительно ее *первого* формата, т.е. функция обрабатывает значения как *string***,** так и *symbol-*типа.

Если содержимое *строки* **–** ***Maple-***предложения не завершается {**;**|**:**}-разделителем либо *символом перевода строки* (**СПС**), то автоматически устанавливается (**;**)-*разделитель*, если не определен ***nonsemicolon-****параметр*. С другой стороны, если содержимое *строки* завершается {**;**|**:**}-разделителем, но не содержит **СПС,** то он добавляется автоматически. Наконец, по вызову функции ***parse*()** возвращается {*true*|*false*}-значение в *зависимости* соответственно от того, завершалось ли содержимое *последней* успешно обработанной ***parse-***функцией строки (**;**)-разделителем либо нет. Следующий фрагмент иллюстрирует разные варианты вызова ***parse-***функции**:**

```
> parse(`exp(Art*x) - Kr*Pi*sqrt(gamma/Catalan)`);
```

$$e^{(Art\,x)} - Kr\,\pi\,\sqrt{\frac{\gamma}{Catalan}}$$

```
> [evalf(%), parse()];   ⇒  [e^(Art x) - 2.493901789 Kr, false]
> parse("Z:= sqrt(Art + Kr) - a/Pi!Pi + 64*x^2;");
Error, incorrect syntax in parse: missing operator or `;` (29)
> [%, parse()];   ⇒  [[e^(Art x) - 2.493901789 Kr, false], false]
```

При этом, следует иметь в виду, что функция ***parse*** не всегда корректно диагностирует ошибочную ситуацию *синтаксического* анализа *строчной* конструкции**,** содержащей выражение/предложение ***Maple***. Следующий фрагмент иллюстрирует вышесказанное**:**

```
> AGN:= "(x^3 + 5.9*sin(x)/(exp(10*x) - Kr*Pi*sqrt(gamma/x))": length(AGN);   ⇒ 51
> parse(AGN, 'statement');
Error, incorrect syntax in parse: `;` unexpected (53)
```

В данном фрагменте ***parse-***функция диагностирует *отсутствие* (**;**)-разделителя в несуществующей позиции ***AGN-***строки {***length*(AGN)=51**}. Более того, такая ситуация должна *автоматически* обрабатываться ***parse-***функцией, как отмечено выше. Тогда как в данном случае имеет место несоответствие числа *открывающих* и *закрывающих* скобок выражения/предложения ***Maple*,** составляющего ***AGN-***строку**.** Во многих случаях значение ***m-***параметра *диагностического* сообщения ***parse-***функции весьма приблизительно идентифицирует место ошибки (*как правило, с точностью до операнда*).

Полезные рекомендации по использованию parse-функции. *И*спользуя *возможности*, предоставляемые ***parse***-функцией, можно применять их для организации весьма простого механизма динамической генерации вычисляемых ***Maple***-выражений, включая достаточно сложные конструкции такие, как *процедуры* и программные *модули*. Пример тому дает следующая простая процедура, возвращающая в зависимости от значения ее первого фактического аргумента одну из двух активных в текущем сеансе процедур:

```
                    A := proc(x::{1, 2}, y::symbol)
                        if x = 1 then parse("" || y || ":=() -> `+`(args):", 'statement')
                        else parse("" || y || ":=() -> `*`(args):", 'statement')
                        end if
                    end proc
> A(2, Sr), Sr(64, 59, 39, 44, 10, 17);   ⇒  () -> `*`(args), 1101534720
> A(1, Sr), Sr(64, 59, 39, 44, 10, 17);   ⇒  () -> `+`(args), 233
> A(1, Summa), Summa(64, 59, 39, 44, 10, 17);   ⇒  () -> `+`(args), 233
        F := (f::symbol, x::symbol, n::posint) → parse(
            "" ||f|| ":=(" || (seqstr(seq(x || k, k = 1 .. n))) || ") -> " || "`+`(args)/nargs;", 'statement')
> F(Kr, y, 10);
                                (y1, y2, y3, y4, y5, y6, y7, y8, y9, y10) → `+`(args)/nargs
> Kr(42, 47, 67, 89, 95, 62);   ⇒  67
```

В зависимости от **x**-значения ***A***-процедура возвращает *одну* из процедур (*активную в текущем сеансе*) с *именем*, определенным вторым фактическим **y**-аргументом. Тогда как **F**-процедура, базирующаяся на ***parse***-функции и реализованная однострочным экстракодом, возвращает **n**-арную процедуру (*активную в текущем сеансе*) с заданным именем **f** и ведущими переменными, начинающимися с **x**. В общем случае пусть **P –** *исходная* конструкция, подлежащая созданию/модификации с последующим ее *вычислением* (*активизацией*) в текущем сеансе. На первом этапе обеспечивается вызов следующего формата:

P1:= *convert*({P | *eval*(P)}, 'string')

После этого, согласно требуемому алгоритму модификации строка **P1** обрабатывается средствами, ориентированными на обработку строк, включая и эффективные процедуры, представленные нашей библиотекой [103,109], давая в результате **P2**-строку. И на заключительном этапе по вызову ***eval*(*parse*(P2))** получаем результат вычисления **P2**-выражения, *доступный* в текущем сеансе. Нижеприведенный текст процедуры ***Aproc*** [109], генерирующей расширенные процедуры, весьма прозрачен и превосходно иллюстрирует описанный выше механизм герерации выражений посредством ***parse***-функции:

```
Aproc := proc(F::symbol, A::list({symbol, `::`}), p::integer)
local a, b, c, d;
    assign(a = convert(eval(F), 'string')), assign(b = op(1, eval(F))),
        assign(c = "(" || (seqstr(b)) || ")");
    if A = [ ] then parse(a)
    elif b = NULL then d := "(" || (seqstr(op(A))) || ")"; eval(parse(sub_1(c = d, a)))
    elif p ≤ 0 then
        d := "(" || (seqstr(op(A), b)) || ")";
        c := "(" || (seqstr(b)) || ")";
        eval(parse(sub_1(c = d, a)))
```

```
    elif nops([b]) ≤ p then
        d := "(" || (seqstr(b, op(A))) || ")";
        c := "(" || (seqstr(b)) || ")";
        eval(parse(sub_1(c = d, a)))
    else
        d := "(" || (seqstr(op([b][1 .. p]), op(A), op([b][p + 1 .. -1])))) || ")";
        c := "(" || (seqstr(b)) || ")";
        eval(parse(sub_1(c = d, a)))
    end if
end proc
> Kr:= proc(x, y, z, h) `+`(args) end proc:  Aproc(Kr, [a, b, c, t::string, v, r::list(symbol), w], 2);
                proc(x, y, a, b, c, t::string, v, r::list(symbol), w, z, h)  `+`(args)  end proc
```

Для возможности представления конструкций в **P1**-строке можно эффективно использовать для их программирования такие средства как ***assign, assign67, seq, add*** и целый ряд других. Достаточно эффективным средством выступает использование непоименованных процедур. В частности, посредством этих средств мы имеем возможность программировать *однострочные экстракоды*, реализующие достаточно сложные алгоритмы. Немало примеров этому можно найти в книге [41,103]. Сложность таких конструкций определяется как опытом, так и навыками пользователя. Между тем, указанный подход имеет целый ряд ограничений и в этом отношении более *универсальным* является предложенный нами метод *«дисковых транзитов»* [41,42,103], широко используемый нами и в других системах программирования.

Несомненным преимуществом метода *«дисковых транзитов»* является и то, что он распространяется на многие программные системы, не располагающие эффективными *аналогами* ***parse***-функции, в частности, ***Mathematica, MathCAD*** и др. Поэтому, многие процедуры, использующие данный метод, *существенно* проще погружаемы в программную среду таких средств. Таким образом, средства нашей *Библиотеки* используют *оба* указанных метода при *явном* приоритете *второго*. Учитывая характеристики современных **ПК** (*RAM*, *HDD*, *тактовая частота*), а также их ближайшие потенции, оценивать эффективность обоих методов, на мой взгляд, задача достаточно неблагодарная. Однако, пользователь в качестве *весьма* полезных упражнений может поставить перед собой *задачу* привести *все* (*или выбранные*) процедуры нашей библиотеки [41,103,109] к единому методу. В принципе, представленные в ней средства далеко не всегда *оптимизировались* в строгом понимании этого понятия, хотя *оценки* эффективности для целого ряда из них и проводились. Нами таких целей не ставилось, ибо многие средства писались (*что называется с листа*) за один присест. Между тем, **4**-летний период использования всех трех версий *Библиотеки* во многих научно-исследовательских организациях и университетах подтвердили вполне достаточную эффективность библиотеки как в качестве дополнения к пакету ***Maple***, так и в качестве полезного учебного материала по курсу программирования в среде пакета. Надеемся, что и читатель найдет для себя кое-что полезное при освоении программной среды пакета ***Maple.***

К ***parse***-функции непосредственно примыкает и *группа* из *двух* функций ***sscanf***, ***sscanf*** и процедуры ***scanf***, имеющих следующие три формата кодирования**:**

(1) ***sscanf*****(**"<*Строка*>", "<*Формат*>"**)** **(2)** ***fscanf*****(**<*Файл*>, "<*Формат*>"**)**

(3) ***scanf*****(**"<*Формат*>"**)**

и предназначенная для обеспечения *синтаксического* анализа сканируемого содержимого *строки*, базирующегося на заданной *форматирующей строке* (*формат*), в соответствии

с *синтаксисом* ***Maple***. В этом отношении ***sscanf*-**функция сочетает возможности рассмотренной функции ***parse*** и ***printf*-**функции, рассматриваемой ниже. При этом, в качестве фактических значений *string*-типа для аргументов всех трех средств могут выступать и значения типа {*symbol*, *name*}.

Функция ***sscanf*** сканирует указанную *Строку* [*конвертируя входящие в нее числа и подстроки согласно заданной форматирующей строки-конвертора* (*Формат*)] и осуществляет их грамматический анализ в соответствии с ***Maple*-**синтаксисом, возвращая список сканированных компонент указанной *первым* аргументом *строки*. *Форматирующая строка* состоит из конвертирующих **%-***спецификаторов*, имеющих следующую довольно простую структуру кодирования:

%{*****} {**<***Длина***>**} **<***Код***>**

напоминающую структуру форматирующих **%-***спецификаторов* рассматриваемой ниже процедуры ***printf***. Здесь необязательный (*)-параметр указывает на то, что сканируемая компонента *строки* не должна появляться в составе возвращаемого ***sscanf*-**функцией результата. Параметр *Длина* определяет максимальную длину сканируемой части компоненты *строки*, которой соответствует данный **%-**спецификатор. Это позволяет выделять для конвертации и анализа подкомпоненты данной компоненты. Наконец, параметр *"Код"* определяет как *тип* сканируемой компоненты строки, так и *тип* возвращаемого элемента в выходном списке; его допустимые значения определяет следующая табл. **9.**

Таблица 9

Код	***Смысл: следующий сканируемый символ (не пробел) относится к:***
d	десятичному целому числу со знаком или без; возвращается целое число
o	целому **8-**ричному числу без знака; возвращается как десятичное целое
x	целому **16-**ричному число без знака; возвращается как десятичное целое
{**e** | **f** | **g**}	десятичному числу со знаком или без; возможно с десятичной точкой либо в научной нотации с {**E** | **e**}-основанием; возвращается как число ***float*-**типа
s	строчному типу (*внутри не допустимы пробелы*); возвращается ***Maple*-**строка
a	невычисляемому ***Maple*-**выражению (*не должно включать пробелы*)
c	строчному значению; *длина*-параметр определяет его длину при возврате
[...]	символы между **[..]-**скобками рассматриваются как элементы списка и возвращаются в виде строки; если список начинается с (**^**)-символа, все элементы списка игнорируются; если список содержит **]-**скобку, она должна следовать сразу за **[-**скобкой, если список не начинается с (**^**)-символа; (**-**)-символ может использоваться в качестве указателя диапазона символов, например: **A-H**
m	сканируется целиком ***Maple*-**выражение, заданное в формате ***m*-**файла; *длина*-параметр игнорируется; выражение возвращается *невычисленным*
{**he** | **hf** | **hg**}	1- или 2-мерный массив {*float*, *integer*}-чисел; возвращается *hfarray*-значение
hx	1- или 2-мерный массив чисел *float*-типа в 16-ричном **IEEE-**формате; возвращается 1- или 2-мерный массив *hfarray*-значений
n	возвращается как целое общее число сканируемых до "**%n**" символов

Следует иметь в виду, что непустые символы между **%-**спецификаторами форматирующей строки игнорируются, но должны по типу отвечать соответствующим символам сканируемой (*считываемой/вводимой*) строки. Функция ***fscanf*** (*второй формат*) отлична от ***sscanf*-**функции только тем, что *сканируемая* строка читается из файла, заданного его спецификатором (*путь к нему в файловой системе* **ПК**) в качестве фактического *значения* ее *первого* аргумента и открываемого как файл текстового формата. Функция ***fscanf*** читает строки файла целиком, но использует ровно столько символов, сколько требуется

для обеспечения всех конвертирующих %-спецификаторов ее *форматирующей* строки. Остающиеся символы доступны для следующего вызова функции, так как файл остается открытым и указатель установлен на следующий считываемый символ.

При использовании {**he**|**hf**|**hg**}-кода следует иметь в виду, что сканируемые символы классифицируются на три типа**:** числовые, разделители и ограничители. К числовым относятся**:** цифры, десятичная точка, знаки (**±**) и символы {**e, E, d, D**}. Символы пробела**,** запятые или квадратные скобки полагаются *разделителями***,** остальные **-** *ограничителями***.** При этом, символ слэша "**/**" является идентификатором конца сканирования, а не символ обратного слэша "****", как указано в документации. Более того, в условиях ***Windows***-платформы недопустимыми являются форматирующие {**D, O, X**}-коды.

Наконец, *третий* формат (***scanf***) читает символы из *стандартного входа* и эквивалентен функции ***fscanf*(**<*Стандартный вход*>**,** <*Формат*>**).** В качестве *стандартного входа* по *умолчанию* полагается ввод с консоли (*клавиатуры*) **ПК**. Только успешно сканированные компоненты *строки* возвращаются в качестве элементов *выходного* списка, в противном случае возвращается *пустой* список,т.е. []. Если при сканировании *компонент* не обнаружено соответствующих форматирующей строке и реально достигнут *конец* ввода, то возвращается нулевое значение. Это же значение возвращается, если считывается пустой файл. Ряд сделанных ниже замечаний относительно форматирующей ***printf***-процедуры сохраняет силу и для ***sscanf, fscanf*** и ***scanf***. Следующий комплексный фрагмент иллюстрирует применение средств ***sscanf***-группы для сканирования и синтаксического анализа *строчных* и *символьных* ***Maple***-конструкций**:**

```
> sscanf(`2006 64 Abc 19.4264E+2 :Example `, ` %5d\%o\%x\%12e\%8s`);
                    [2006, 52, 2748, 1942.64, ":Example"]
> sscanf(`2006 64 Abc 19.4264E+2 :Пример`, ` %5d\%o\%x\%12e\%8s`);
                    [2006, 52, 2748, 1942.64, ":П"]
> sscanf("sqrt((sin(x) - gamma)/(tan(x) - Pi)) AVZ 64 Abc 19.4257E+2 :Example 21.09.2006",
  `%30a\%o\%x\%12e\%8s`);
Error, (in sscanf) incorrect syntax in parse: `;` unexpected (14)
> sscanf(`sqrt((sin(x)-gamma)/(tan(x)-Pi))AVZABCDEFG`, "%32a\%3c\%7[A-G]\%n");
                    [sqrt((sin(x)-gamma)/(tan(x)-Pi)), "AVZ", "ABCDEFG", 42]
> fscanf("C:\\ARM_Book\\Academy\\Salcombe.IAN", "%16s\%5f\%a\%d\%n");
                    ["Input_from_file:", 19.42, AVZ, 350, 65]
> Kr:= "Paldiski 23 april 2006":  sscanf(Kr, "%s%d%s%d");   ⇒ ["Paldiski", 23, "april", 2006]
> parse("sqrt((Art+Kr)/(VsV+VaA)+abs(Agn+Avz)-GAMMA*(Pi*Catalan)/20.06)");
                    sqrt((Art + Kr)/(VsV + VaA) + |Agn + Avz| - 0.04985044865 Γ π Catalan)
> fscanf("C:/ARM_Book/Academy/Lasnamae/Galina.52", "%15s %s %a");
                    ["Input_Of_File", "25.03.99", RAC-IAN-REA-RANS]
> sscanf("RANS IAN", "%8s"), sscanf("x*A + y*B", "%9a"), sscanf("RANS IAN", "%8c");
                    ["RANS"], [x A], ["RANS IAN"]
> sscanf("64, 47.59, 10.17, / 20.069", "%he"), sscanf("64, 47.59,\\ 10.17 20.06", "%he");
        [[64., 47.5900000000000034, 10.1699999999999999]], [[64., 47.5900000000000034,
                    10.1699999999999999, 20.0599999999999987]]
> map(type, [op(%[1]), op(%[2])], 'hfarray');   ⇒ [true, true]
> sscanf("[[64, 47.59], [ 10.17, 17.59]])", "%he"): type(op(%), 'hfarray');   ⇒ true
> sscanf("2006", "%4D");
Error, (in sscanf) unknown format code `D`
```

В частности, использование средств данной группы с русскими *текстами* может приводить к *некорректным* результатам, как это иллюстрирует второй пример фрагмента для ***Maple 8***. Средства для работы с выражениями *string*–типа *полностью* применимы и к выражениям *symbol*-типа, ибо один тип *легко* конвертируется в другой, и наоборот. Поэтому, в отношении функциональных средств данные типы можно считать эквивалентными. При этом, так как, начиая с **6**-го релиза, строчное выражение является индексируемым, т.е. к отдельному его символу можно адресоваться по номеру его позиции (например, **"abcddfg"[3], "abcddfg"[3..5];** ⇒ "c", "cdd"), то к такого типа выражениям *применимы* и многие стандартные средства пакета, предназначенные для работы с индексированными структурами данных.

Ниже средства ***Maple***-языка пакета для работы со строчными и символьными данными и их структурами будут рассматриваться в различных *контекстах* при представлении иллюстративных примеров. При этом, следует отметить, что по *представимости* работы со строчными структурами ***Maple***-язык располагает меньшим количеством функциональных средств, чем упоминавшийся выше пакет ***Mathematica,*** что предполагает *несколько* большую искушенность пользователя в этом направлении. Правда, с последними релизами поставляется пакетный модуль **StringTools,** содержащий набор средств для работы со строчными структурами. Его появление, по-видимому, было *навеяно* нашим набором средств подобного типа, созданным еще для ***Maple V*** (*все наши издания по* ***Maple****-тематике хорошо известны разработчикам ввиду имевшего места сотрудничества в процессе подготовки этих изданий*). Между тем, представленные в нашей *Библиотеке* [103,108,109] средства работы со строчными и символьными выражениями существенно дополняют имеющиеся средства пакета для задач подобного типа.

Например, во многих приложениях, имеющих дело со *строчными* выражениями, широко используется процедура ***Red_n,*** обеспечивающая *сведение* кратности вхождений символов или подстрок в строку или символ. Вызов процедуры имеет формат **Red_n(S, G, N),** где: **S** – строка или символ, **G** – строка или символ длины **>= 1** или их список и **N** – положительное целое (*posint*) либо список целых положительных чисел.

Вызов процедуры ***Red_n*****(S, G, N)** возвращает результат сведения кратности вхождений символов или строк, заданных вторым фактическим аргументом **G** в *строку* или *символ*, указанный первым фактическим **S** аргументом, к количеству не большему, чем третий аргумент **N.** В частности, если **N={1|2}**, то строка/символ **G** удаляется из строки **S** или остается с кратностью **1** соответственно. Кроме того, тип возвращаемого результата соответствует типу исходного фактического **S** аргумента. Процедура ***Red_n*** производит регистро-зависимый поиск. Если символ **G** не принадлежит строке **S,** то процедура возвращает *первый* фактический аргумент без обработки с выводом соответствующего предупреждения. Нулевая длина второго фактического аргумента **G** вызывает ошибочную ситуацию с диагностикой типа «length of <> should be more than 1».

Если второй и третий фактические аргументы определяют *списки*, между которыми существует *взаимно-однозначное* соответствие, то сведение кратности делается по всем элементам списка **G** с соответствующими кратностями из *списка* **N.** Если ***nops*****(G) > *nops*****(N),** последние **nops(G) - nops(N)** элементов **G** будут иметь *кратности* **1** в возвращаемом результате. Если **G** – список и **N** – положительное целое число, то все элементы **G** получают *одинаковую* кратность **N**. Наконец, если **G** – символ или строка и **N** – список, то **G** получает *кратность* **N[1]** в возвращаемом результате. Процедура ***Red_n*** представляет собой достаточно полезный инструмент для обработки строк и символов при *символьном* решении задач [103]. Ниже представлены исходный текст процедуры и некоторые примеры ее применения.

```
Red_n := proc(S::{string, symbol}, G::{string, symbol, list({string, symbol})},
N::{posint, list(posint)})
local k, h, Λ, 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')));
    Λ := 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 := Λ(S, g, n, g)
    else h := Subs_All(z = g, Λ(Subs_All(g = z, S, 2), z, n, g), 2)
    end if;
    convert(h, whattype(S))
end proc
```

```
> Red_n("aaccccbccccccccccccbccccccccckcccckcccccccccccc", "cccc", 2);
                                "aaccccbccccbcccckcccckcccccccc"
> Red_n("aaccccbbccccccccckcccckcccccccc", "cccc", 1);   ⇒ "aabbkk"
> Red_n("1111122222233333334444444445555555556666666666", [`1`,`2`,`4`,`5`,`6`], [2,3,4,5,6]);
                                "12233333334444555666666"
```

Не меньший интерес представляет и ***seqstr***-группа процедур, обеспечивающих различные режимы *конвертации* последовательностей выражений в строку [41,103]. Например, простая процедура ***seqstr1,*** реализованная однострочным экстракодом вида:

```
seqstr1 := () -> cat("", op(map(convert, [args], 'string')));
```

обеспечивает *конвертацию* последовательности выражений в строку конкатенации этих выражений, например:

```
> seqstr1(), seqstr1(Z(x), x*y, (a + b)/(c + d), "ArtKr", 10/17)
                        "", "Z(x)x*y(a + b)/(c + d)ArtKr10/17"
```

Определенный интерес представляет и ***swmpat***-группа процедур, обеспечивающих поиск образцов, содержащих *wildcard-символы* [103]. В качестве примера приведем ***swmpat***-процедуру. Вызов процедуры ***swmpat*(S ,m, p, d {, h})** возвращает значение *true*, если и только если строка или символ, указанный фактическим аргументом **S,** содержат вхождения подстрок, которые соответствуют образцу **m** с группирующими символами, указанными *четвертым* аргументом **d,** тогда как *третий* фактический аргумент **p** определяет список кратностей соответствующих вхождений *группирующих* символов **d** в **m**.

Например, пусть триплет **<"a*b*c", [4,7], "*">** определяет образец **m**="**** b ******* c". Вызов процедуры ***swmpat*(S, m, [4, 7], "*")** определяет факт вхождения в строку **S** непересекающихся *подстрок*, которые имеют вид образца **m** с произвольными символами вместо всех вхождений *группирующего* символа "*". При этом, если вызов процедуры ***swmpat*(S, m, p, d, h)** использовал необязательный пятый аргумент **h** и было возвращено значение *true*, то через **h** возвращается вложенный список, чьи **2**-элементные *подсписки* определяют первые и последние позиции непересекающихся подстрок **S,** которые соответствуют образцу **m**. Кроме того, если образец **m** не содержит *группирующие* символы, то через **h** возвращается целочисленный список, чьи элементы определяют первые позиции непересекающихся подстрок **S,** которые соответствуют *образцу* **m**. Если же вызов процедуры возвращает значение *false*, то через **h** возвращается *пустой* список, т. е. [].

Если *четвертый* фактический аргумент **d** определяет строку или символ длины *большей* **1**, то первый ее символ используется как *группирующий* символ. Если же список **p** имеет меньше элементов, чем количество вхождений *группирующих* символов в образец **m,** то избыточные вхождения получают кратность **1**. По умолчанию, процедура ***swmpat*** поддерживает *регистро-зависимый* поиск, если *вызов* процедуры использует дополнительное ключевое ***insensitive*-**слово, то выполняется *регистро-независимый* поиск. Данная процедура имеет целый ряд весьма полезных приложений в задачах обработки *строк* и *символов* [103]. Ниже представлены исходный текст процедуры и примеры ее применения.

```
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, ν, n, ω, t, ε, x, y;
    assign67(c = {args} minus {S, d, insensitive, m, p}, 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( ν = ((x, n) → cat(x $ (b = 1 .. n))), ω = (t → `if`(t = 0, 0, 1)));
        ε := 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, ν(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 ε(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; ⇒ true, [[15, 31], [36, 52]]
> swmpat(S, "art1986kr", [7, 14], "*", r), r; ⇒ true, [15, 36]
> swmpat(S, m, p, "*", 'z'); ⇒ true
> S:= "avz1942agn1947Art1986kr1996Svet1967Art1986kr1996Svet": m:= "a*1986*svet":
  p:= [2, 6]: swmpat(S, m, p, "*", a), a; ⇒ false, []
> S:= "avz1942agn1947Art1986Kr1996Svet1967Art1986Kr1996Svet": m:= "a*1986*svet":
  p:=[2, 6]: swmpat(S, m, p, "*", b, `insensitive`), b; ⇒ true, [[15, 31], [36, 52]]
> swmpat(S, "art1986kr", [7, 14], "*", t), t; ⇒ false, t
> swmpat(S, "art1986kr", [7, 14], "*", `insensitive`, 't'), t; ⇒ true, [15, 36]
```

Отметим еще одну довольно полезную процедуру работы со строчными выражениями. Процедура ***nexts*(S,A,B {,r})** обеспечивает *поиск* в *строке* или *символе*, определенных фактическим аргументом **S,** образцов **B,** ближайших к образцам **A** справа или слева. Вызов процедуры с тремя аргументами определяет *поиск вправо* от **A,** тогда как вызов с *четырьмя* или более аргументами определяет поиск влево от **A**. Вызов процедуры ***nexts*(S, A, B {, r})** возвращает вложенный список, элементами которого являются **2**-элементные списки (*если вложенный список содержит только один элемент, то возвращается обычный список*). Первый элемент такого подсписка определяет позицию образца **A** в **S,** тогда как второй определяет позицию образца **B,** ближайшего к **A** *вправо* или *влево* соответственно. Более

того, если *ближайший* к образцу **A** образец **B** не был найден, то возвращаемый процедурой список будет содержать только позицию самого образца. Кроме того, в качестве аргумента **A** может выступать как *образец*, так и *позиция* отдельного символа в **S**. Если *второй* аргумент **A** отсутствует в **S,** то вызов процедуры возвращает *false*. Если какой-либо фактический аргумент пуст, то вызов процедуры вызывает ошибочную ситуацию. Во многих задачах обработки символов и строк данная процедура оказалась довольно полезным средством и используется рядом процедур нашей *Библиотеки* [103,109]. Ниже представлены исходный текст процедуры и некоторые примеры ее применения.

```
nexts := proc(S::{string, symbol}, A::{posint, string, symbol}, B::{string, symbol})
local a, b, s, k, t, n;
    if map(member, {S, A, B}, {``, ""}) = {false} then
        try
            assign(s = cat("", S), b = [ ], n = `if`(type(A, 'posint'), 1, length(A)));
            `if`(type(A, {'symbol', 'string'}), assign(a = Search2(s, {A})), assign(
                `if`(A ≤ length(s), assign(a = [A]),
                ERROR("2nd argument must be <=%1", length(s)))))
        catch : ERROR("wrong type of arguments in nexts call - %1",[args])
        end try
    else ERROR("arguments cannot be empty",[S, A, B])
    end if;
    if a = [ ] then false
    elif nargs = 3 then
        for k in a do `if`(search(s[k + n .. -1], B, 't'),
            assign('b' = [op(b), [k, t + k + n − 1]]), assign('b' = [op(b), [k]]))
        end do;
        `if`(nops(b) = 1, op(b), b)
    else
        for k in a do assign('a' = Search2(s[1 .. k − 1], {B})), `if`(a = [ ],
            assign('b' = [op(b), [k]]), assign('b' = [op(b), [k, a[-1]]]))
        end do;
        `if`(nops(b) = 1, op(b), b)
    end if
end proc
```

> S:= "aaaccccacaaaccaaacaacddddccrtdrtbbaaabaaaabaaahtyrebaaa": nexts(S, 29, b), nexts(S, 29, b, 8); ⇒ [29, 33], [29]

> nexts(S, "", b);

Error, (in **nexts**) arguments cannot be empty,
[aaaccccacaaaccaaacaacddddccrtdrtbbaaabaaaabaaahtyrebaaa, , b]

> nexts(S, aaa, b), nexts(S, aaa, bb);

[[1,33], [10,33], [15,33], [35,38], [39,42], [43,51], [52]], [[1,33], [10,33], [15,33], [35], [39], [43],[52]]

> nexts(S, aaa, ba); ⇒ [[1, 34], [10, 34], [15, 34], [35, 42], [39, 51], [43, 51], [52]]

> nexts(S, xyz, ba); ⇒ *false*

> nexts(S, "cr", rt); ⇒ [27, 31]

> nexts(S, aaa, b, 1), nexts(S, aaa, bb, 2);

[[1], [10], [15], [35, 34], [39, 38], [43, 42], [52,51]], [[1], [10], [15], [35,33], [39,33], [43,33], [52,33]]

> nexts(S, ccc, t, 1), nexts(S, crt, bb, 2), nexts(S, crt, bb); ⇒ [[4], [5]], [27], [27, 33]

Наконец, вызов следующей процедуры ***SUB_S*(L, S)** возвращает результат выполнения подстановок, определенных аргументом **L** (*список уравнений-подстановок*), в строку либо символ **S;** при этом, очередная подстановка из **L** выполняется до ее *полного* исчерпания в строке **S.** Тип возвращаемого результата соответствует типу исходного аргумента **S.** В случае неприменимости подстановок **L** второй аргумент **S** возвращается без обработки.

```
SUB_S := proc(L::list(equation), S::{string, symbol})
local k, d, G, h, P, R, l;
    if L = [ ] then S
    elif seq(`if`("" || lhs(k) = "" || rhs(k),
    ERROR("substitution <%1> initiates infinite process","k), NULL), k = L) = NULL
    then
        assign(l = [seq(convert(lhs(L[k]), 'string') = convert(rhs(L[k]), 'string'),
            k = 1 .. nops(L))], R = "" || S), `if`(nargs = 3 and args[3] = 'sensitive',
            assign(P = 'SearchText'), assign(P = 'searchtext'));
        for k to nops(L) do
            assign('d' = P(lhs(l[k]), R), 'h' = lhs(l[k]), 'G' = rhs(l[k]));
            if d ≠ 0 then
                R := cat(substring(R, 1 .. d − 1), G,
                    substring(R, d + length(h) .. length(R)));
                k := k − 1
            end if
        end do;
        convert(R, whattype(S))
    end if
end proc
> SUB_S([ab=cdgh,aa=hhsg,cd=cprhkt],"aaababccabaacdcdaaasvrhcdabaa");
    "hhsgcprhktghcprhktghcccprhktghhhsgcprhktcprhkthhsgasvrhcprhktcprhktghhhsg"
> SUB_S([42=64,47=59,67=39],"aaab4242ccaba4747daaasv6767abaa");
                                "aaab6464ccaba5959daaasv3939abaa"
```

В главе **5** [103] рассматриваются программные средства, расширяющие возможности пакета ***Maple*** релизов **6 - 10** при работе с выражениями типов {*string*, *symbol*}. Представленные в ней средства включены и в нашу *Библиотеку* [108,109], и обеспечивают множество полезных процедур таких как специальные виды преобразования, сравнение строк или символов, различные виды поиска в строках, обращение символов, строк или списков, исчерпывающие замены в строках или символах, сведение кратности вхождения символа в строку, определение вхождения специальных символов в строку и целый ряд *иных*. Во многих случаев данные средства существенно упрощают программирование с использованием объектов ***Maple*** типов {*string*, *symbol*, *name*} в среде пакета. Это актуально, в первую очередь потому, что символьные выражения составляют как основу важнейших структур пакета, так и основной объект символьных вычислений и обработки. Переходим теперь к рассмотрению средств *обеспечения* работы со структурами и данными *списочного* и *множественного* типов {*list*, *set*}, весьма широко используемых ***Maple*-**языком.

5.2. *Средства работы Maple-языка с множествами, списками и таблицами*

Списочные и *множественные* структуры и данные (*или для краткости просто списки и множества*) имеют следующий чрезвычайно простой вид кодирования:

***List*:= [X$_1$, X$_2$, X$_3$, ..., Xn]** и ***Set*:= {X$_1$, X$_2$, X$_3$, ..., Xn}**

где в качестве **X$_j$**-элемента (**j=1, 2, 3, ..., n**) могут выступать любые допустимые ***Maple***-выражения, включая и *сами* структуры типов {*list*, *set*}, т.е. такие структуры допускают различные *уровни вложенности*, глубина которых ограничивается только доступным объемом оперативной памяти **ПК.** *Пустой* список (*множество*) обозначается как "[]" ("{}"). Следующий простой фрагмент иллюстрирует типичные структуры типов {*list*, *set*}, допускаемые ***Maple***-языком пакета:

```
> SL:= [`Example`, array(1..2, 1..2, [[V, G], [S, A]]), F(k)$k=1..3, Int(G(x), x = a..b)];
```

$$SL := \left[Example, \begin{bmatrix} V & G \\ S & A \end{bmatrix}, \mathrm{F}(1), \mathrm{F}(2), \mathrm{F}(3), \int_a^b \mathrm{G}(x)\,dx \right]$$

```
> SS:= {`Example`, array(1..2, 1..2, [[V, G], [S, A]]), F(k)$k=1..3, Int(G(x), x = a..b)};
```

$$SS := \{ Example, \int_a^b \mathrm{G}(x)\,dx, \mathrm{F}(3), \mathrm{F}(1), \mathrm{F}(2), \begin{bmatrix} V & G \\ S & A \end{bmatrix} \}$$

Результатом вычисления ***List***-*структуры* (***Set***-*структуры*), как правило, является выражение *list*-типа (*set*-*типа*), тестируемое функциями ***type***, ***typematch*** и ***whattype***-процедурой, рассмотренными выше: по *тестирующим* функциям {***type*** | ***typematch***}**(L, 'list')** и ***whattype*****(L)** возвращается соответственно значение ***true*** и ***list***, если **L** – *списочная* структура. Аналогично обстоит дело и со структурами *set*-типа. Например, многие встроенные, библиотечные и модульные функции ***Maple***-языка пакета возвращают результат типа {*list*, *set*}, как это весьма хорошо иллюстрирует следующий простой фрагмент:

```
> L:= [64, [sqrt(25), 2006], [a, b], {x, y}, 10.17\2006];
                    L := [64, [5, 2006], [a, b], {x, y}, 10.172006]
> type(L, 'list'), typematch(L, 'list'), whattype(L);   ⇒ true, true, list
> Art:= "IAN 7 april 2006":  sscanf(Art, "%a%d%a%d");   ⇒ [IAN, 7, april, 2006]
> convert(G*sin(x)+S*sin(y)-ln(a+b)*TRG, 'list');   ⇒ [G sin(x), S sin(y), -ln(a+b) TRG]
> S:= "123456789":  map(F, map(parse, [S[k]$k=1..9]));
                    [F(1), F(2), F(3), F(4), F(5), F(6), F(7), F(8), F(9)]
> fsolve({10*x^2 + 64*y - 59, 17*y^2 + 64*x -59}, { x, y});
                    {x = 0.7356456027, y = 0.8373164917}
> type(%, 'set'), typematch(%, 'set'), whattype(%);   ⇒ true, true, set
> H:= "awertewasderaryretuur":  Search2(H, {a, r, t});   ⇒ [1, 4, 5, 8, 12, 13, 14, 16, 18, 21]
```

В частном случае результатом вычисления или данными может быть и *пустой* []-*список*. По функции ***convert*****(B, '*list*')** можно конвертировать в *списочную* структуру вектор, таблицу или произвольное **B**-выражение, *операнды* которого будут элементами списка, как это иллюстрирует четвертый пример предыдущего фрагмента. Это же имеет место и в случае конвертации в *set*-структуру. С особенностями структур типов {*list*,*set*} можно ознакомиться в [39]. Структуры *list*-типа и *set*-типа во многих отношениях подобны, используя один и тот же *базовый* набор средств для их обработки. Ниже для удобства мы будем говорить о списочных структурах, везде (*если не оговорено противного*) предполагая и

структуры *set*-типа. Наиболее *существенная разница* обоих типов структур состоит в следующем:

(**1**) если *список* – структура с четко определенным порядком элементов, заданным при его определении (*при этом, ее элементы могут дублироваться*), то для *множество* не имеет места четкое упорядочение элементов;
(**2**) элементы *множества* не дублируются

Множество представляет собой структуру, элементы которой, в принципе, неупорядочены в принятом смысле (*порядок ее элементов при создании в общем случае отличен от результата ее вычисления*). Данная структура является аналогом математического понятия множества объектов. В отличие от списка, порядок ввода элементов множества при его определении отличается от порядка выходного множества, устанавливаемого согласно соглашениям пакета. Если в результате определения *списочной* структуры ее элементы только вычисляются, то в случае *множества* они дополнительно еще и упорядочиваются согласно *адресов* занимаемых ими ячеек памяти **ЭВМ**, а также производится *приведение* кратности вхождений идентичных элементов к единице. Однако можно показать, что между порядками элементов **L**-списка и **M**-множества, имеющих одинаковые длину и состав элементов, имеет место следующее определяющее соотношение (*при этом, предполагается также, что* **L**-*список не имеет дублирования элементов, ибо во множестве* **M** *каждый элемент представляется в единственном экземпляре*):

sort(L, 'address')[k] = M[k], где **k –** номер элемента объекта, например:

> L:= [h, g, s, a, x, [64, b], {y, c, z}, 59, Sv, 39, Art, 17, Kr, 10]: M:={ h, g, s, a, x, [64, b], {y, c, z}, 59, Sv, 39, Art, 17, Kr, 10}: (sort(L, 'address')[k] = M[k])$k=1..14;

10 = 10, 17 = 17, 39 = 39, 59 = 59, [64, b] = [64, b], {c, z, y} = {c, z, y}, Sv = Sv, Art = Art, Kr = Kr, h = h, g = g, s = s, a = a, x = x

> sort(L, 'address') = M;

[10, 17, 39, 59, [64, b], {c, z, y}, Sv, Art, Kr, h, g, s, a, x] = {10, 17, 39, 59, [64, b], {c, z, y}, Sv, Art, Kr, h, g, s, a, x}

> M:= { h, g, s, a, x, [64, b], {y, c, z}, 59, Sv, 39, Art, 17, Kr, 10};

M := {10, 17, 39, 59, [64, b], {c, z, y}, Sv, Art, Kr, h, g, s, a, x}

В определенных обстоятельствах данное соотношение может оказаться полезным при работе со структурами типов {*list*, *set*}. В свете сказанного во многих случаях можно рассматривать множества как упорядоченные объекты и использовать данное обстоятельство в различных приложениях. В общем же случае это не имеет места, поэтому использовать его следует весьма осмотрительно. Структуры и данные *списочного* типа являются весьма общим объектом и могут использоваться как самостоятельные *типы* данных, для которых язык располагает целым рядом средств представления и обработки, так и для организации составных вычислительных конструкций, о которых вскользь речь шла выше и по которым дополнительная информация дается ниже. Так как объекты, подобные векторам и матрицам, в среде пакета представляются списочными структурами, то ряд средств поддержки работы с первыми может быть успешно использован и непосредственно для списков, как и наоборот. В свете определения списочной структуры, она представляется весьма удобной для более *компактного* представления результатов вычисления выражений, как это иллюстрировалось выше. Более того, результат вычисления списка возвращает вновь *списочную* структуру с сохранением порядка элементов исходного списка. Рассмотрим детальнее *базовые* средства работы со списками. При этом, для удобства и не оговаривая отдельно, в представляемых форматах функций в качестве их **L**-аргументов понимается *списочная* структура (*список*) и, если не оговорено

противного, также и *множество* с учетом указанных *различий* между ними. Там, где имеются различия, они будут отмечаться.

По упоминаемой выше ***op*(L)**-функции можно "*снимать*" списочную структуру, превращая **L-***список* (*множество*) в *последовательность* (*exprseq*) его элементов, а по вызову функции ***nops*(L) –** получать число его *элементов* (*длину списка/множества*). По конструкциям вида**: L[k]** и **L[k]:=** <*Выражение*> соответственно возвращается **к-**й элемент и изменяется значение **к-**го элемента **L-***списка* путем присвоения ему значения указанного *выражения*, тогда как по ***NULL*-**значению (**L[k]:=*NULL***) удалять **к-**й элемент невозможно, т.к. возникает ошибочная ситуация с диагностикой «Error, expression sequences cannot be assigned to lists». Для этой цели следует использовать вызов ***subsop*(k=NULL, L)** или прием, представленный в разделе **1.5**. Тогда как для случая **M**-множества вторая конструкция заменяется, например, следующими: **S:= [op(M)]: S[k]:=** <*Выражение*>**: M:= {op(S)}**. Наоборот, по конструкции {**[**<*exprseq*>**]**|{<*exprseq*>**}**} производится конвертация *последовательности* (*exprseq*) в структуру типа {*list*|*set*} соответственно или это *можно* сделать по рассмотренной ***convert*-**функции.

Два случая применения ***op*-**функции следует рассмотреть *особо*. Как уже отмечалось выше, по конструкции ***op*(0, B)** в *общем* случае возвращается тип **B-**выражения. Однако для *индексированных* структур и функций возвращаются их идентификаторы, тогда как для некоторых других структур, например, *последовательностей* (*exprseq*), может инициироваться *ошибочная* ситуация. Наконец, в случае третьего формата ***op*-**функции (*раздел* **1.3**) первый ее аргумент ***op*(p1, B)** определяет **p1-**позицию выделяемого элемента **1-**го (*внешнего*) уровня вложенности **B-**выражения {**B1=*op*(p1,** <*Выражение*>**)**}, **p2 –** позицию выделяемого элемента из **B1** {**B2 = *op*(p2, B1)**} и т.д., т.е. имеет место следующее определяющее соотношение**:**

op([p1, p2, ..., pn], <*Выражение*>**) ≡ op(pn, ... op(p3, op(p2, op(p1,** <*Выражение*>**)))...)**

Следовательно**,** третий формат ***op*-**функции ориентирован на работу с *вложенными* списочными структурами, а в общем случае с ***Maple*-**выражениями, имеющими несколько уровней *вложенности***.** Следует иметь в виду, что вычисления в *списочной* структуре ориентированы и производятся не независимо**,** а *поэлементно* слева направо (*включая уровни вложенности*), именно поэтому результаты вычислений *можно передавать* между элементами списка в указанном направлении, что весьма существенно при организации вычислений и будет использоваться нами в дальнейшем. Функции {***op*, *nops***} очень широко используются и для произвольных выражений, но для списочных структур и множеств они являются одними из *базовых* средств по манипулированию их элементами, включая и *уровни* их *вложенности*. Например, по конструкции **L:=[*op*(L),** <*Элемент*>**]** производится пополнение **L-**списка новым указанным *элементом*, например**:**

> L:=[64,59,39,10,117]: L:=[op(L),V,G,Sv,Kr,Art]; ⇒ *L*:= [64, 59, 39, 10, 117, V, G, Sv, Kr, Art]

Данная конструкция широко используется в практическом программировании задач.

Для возможности *символьной* обработки уравнений, неравенств, отношений и диапазонов важно иметь средства выделения их {*левой*|*правой*} части или {*начального*|*конечного*} выражения диапазона, что обеспечивается соответственно {***lhs*|*rhs***}-функцией, имеющей простой формат кодирования, а именно: {***lhs*|*rhs***}**(V),** где **V – *Maple*-**выражение одного из указанных типов {*relation*, *range*}. Следующий фрагмент достаточно прозрачен и каких-либо особых пояснений не требует:

```
> [map(rhs, [A = B, A <>B, A <+ B, A .. B]), op(A <>B)];   ⇒ [[B, B, B, B], A, B]
> map(op, [A = B, A <>B, A <+ B, A .. B]);   ⇒ [A, B, A, B, A, B, A, B]
> [op(k, A = B), op(k, A <> B), op(k, A <= B), op(k, A..B)]$k=1..2;   ⇒ [A,A,A,A], [B,B,B,B]
```

Приведенный фрагмент иллюстрирует не только эквивалентность (*легко следующую из определения функций* **op** *и* {***lhs*** | ***rhs***}, *конструкций* {***lhs*** | ***rhs***}**(V)** *и* ***op***({**1** | **2**}, **V)**, но и возможность по конструкции ***op*****(V)** получать список, элементами которого являются левая и правая части **V**-выражения (*уравнение*, *неравенство*, *отношение*, *диапазон*).

Функция ***member***, имеющая в общем случае следующий формат кодирования:

member(<*Выражение*> {, <*Список*> | , <*Множество*>} {, '<*Идентификатор*>'})

выполняет тестирование указанного *списка/множества* на предмет *вхождения* в него указанного *первым* аргументом *выражения*. При кодировании необязательного третьего аргумента (*должен быть невычисленный идентификатор*) ему присваивается номер позиции *первого* вхождения заданного *выражения* в *список/множество*, если ***member***-функция возвращает *true*-значение; иначе идентификатор остается *неопределенным*. Данная функция весьма широко используется при практическом программировании. В качестве ее расширения была определена процедура ***belong*** [103,109], обеспечивающая тестирование принадлежности указанного выражения *множеству*, *списку*, *модулю*, *процедуре*, *символу*, *строке* и другим типам выражений. Ниже представлены ее исходный текст и некоторые примеры применения.

```
belong := proc(a::anything, V::{set, equation, procedure, relation, Matrix, Vector, Array,
array, table, list, `module`, range, string, symbol})
    `if`(a = V, true, `if`(type(V, 'table'),
    member(a, expLS({indices(V), entries(V)})), `if`(
    type(V, {'equation', 'relation'}),
    procname(a, {OP(lhs(V))}) or procname(a, {OP(rhs(V))}), `if`(member(
    whattype(eval(V)), {'array', 'Matrix', 'Array', 'Vector'['row'], 'Vector'['column']})
    , RETURN(procname(a, expLS(convert(V, 'listlist1')))), `if`(
    type(V, 'procedure'), procname(a, map(op, [2, 6], eval(V))), `if`(
    type(eval(V), 'symbol') or type(eval(V), 'string'), search(V, convert(a, 'string'))),
    `if`(type(V, `module`), member(a, V), `if`(
    type(V, 'range') and type(a, 'numeric'),
    `if`(a ≤ rhs(V) and lhs(V) ≤ a, true, false), `if`(
    type(a, 'set') and type(V, {'list', 'set'}),
    evalb({op(a)} intersect {op(V)} = {op(a)}), `if`(
    type(a, 'list') and type(V, 'list'), Sub_list(a, V), `if`(
    type(V, 'range') and type(a, {'list'('numeric'), 'set'('numeric')}),
    `if`(map(procname, {op(a)}, V) = {true}, true, false), `if`(
    type(V, {'symbol', 'string'}),
    `if`(map2(search, V, map(cat, {op(a)}, ``)) = {true}, true, false),
    member(a, V))))))))))))))
end proc
```

```
> m:= matrix(2, 3, [a,b,c,42,96,47]): v:= vector([42,G,47,g]): a:= array(1..2, 1..3,[[42,47,67],
  [62,89,96]]): t:= table([c = z]): V:= Vector([61, 56, 36, 40, 7, 14]): V1:= Vector[row]([61, 56,
  36, 40, 7, 14]): M:= Matrix(1..2, 1..2, [[T,S],[G,K]]): A:=Array(1..2,1..3,1..2,[]): A[2,2,2]:=42:
  A[1,2,2]:=47: A[2,3,2]:=67: A[2,1,2]:=89: A[2, 2, 1]:=96: gs:=[1, [67, 38],[47, 58], 6]:
  sv:=42 < 63: belong(47, m), belong(47, a), belong(61, V), belong(61, V1), belong(K, M),
  belong(67, A), belong(z, t), belong([67, 38], gs), belong(63, sv);
                    true, true, true, true, true, true, true, true, true
```

Данная процедура существенно расширяет встроенную функцию ***member***, упрощая в целом ряде случаев программирование приложений в среде пакета.

Начиная с **8**-го релиза, ***Maple***-язык расширен новым **in**-оператором, имеющим формат:

el **in** **L** или **`in`(*el*, L)**

где ***el*** - произвольное ***Maple***-выражение и **L** - список либо множество. Назначением данного оператора является тестирование на принадлежность ***el*** к **L**. Результатом применения данного оператора является возврат исходного вызова в принятой математической нотации, тогда как для получения собственно результата тестирования к полученному результату следует применять ***evalb***-функцию. Весьма простые примеры иллюстрируют вышесказанное:

> 64 in [59, 39, 64, 10, 17, 44]; evalb(%); ⇒ 64 ∈ [59, 39, 64, 10, 17, 44] true
> evalb(`in`(64, [59, 39, 64, 10, 17, 44])); ⇒ *true*

Детальнее с данным оператором **in** можно ознакомиться в справке по пакету.

К ***member***-функции по *смыслу*, но с более широкими возможностями, примыкает и процедура ***charfcn*[A](X)**, являющаяся *характеристической* для множеств и алгебраических ***Maple***-выражений. В качестве **X** выступает произвольное алгебраическое выражение, а в качестве **A** - *спецификатор* множества. Кусочно-определенная ***charfcn***-процедура задается следующим определяющим соотношением:

$$\mathbf{charfcn[A](X)} = \begin{cases} \mathbf{1},\ \textit{если X удовлетворяет A-спецификатору} \\ \mathbf{0},\ \textit{если X не удовлетворяет A-спецификатору} \\ \mathbf{'charfcn[A](X)'},\ \textit{в противном случае} \end{cases}$$

В качестве **A**-*спецификатора* может выступать множество, действительное или комплексное значение, диапазон действительных или комплексных значений либо последовательность выражений любого из перечисленных выше *типов*. Тогда как смысл выражения "*удовлетворяет* ***A***-*спецификатору*" определяется следующей табл. **10.**

Таблица 10

Спецификатор A	***Смысл: X удовлетворяет A-спецификатору***
Множество	***member*(X, A)** ⇒ ***true***
A - действительное или комплексное	**X = A**
a .. b; a, b - действительные числа	**a <= X <= b;** допускаются **±*infinity*-**значения
a .. b; a, b - комплексные значения	**Re(a) <= Re(X) <= Re(b)** и **Im(a) <= Im(X) <= Im(b)**
A1, A2, … ,An; Ak - *выражение* одного из предыдущих типов	удовлетворяет одному из **Ak-**спецификаторов

Примеры нижеследующего фрагмента иллюстрируют применение ***charfcn***-процедуры, в частности, удобной в числовых вычислениях, как иллюстрирует последний пример.

```
> charfcn[64, 59, 39, 10, 17](17), charfcn[64, 59, 39, 10, 17](44);  ⇒ 1, 0
> charfcn[1 .. 64](59), charfcn[3.1 .. 3.3](Pi), charfcn[2+I .. 17+2*I](2+I);  ⇒ 1, 1, 1
> G:= X -> 64*charfcn[1..9](X)+59*charfcn[10..17](X)+39*charfcn[18..25](X)+17*
  charfcn[-infinity .. 0, 26..infinity](X): map(G, [2, 12, 20, -3, 32]);  ⇒ [64, 59, 39, 17, 17]
```

Последний пример фрагмента иллюстрирует использование *характеристической* функции ***charfcn*** для определения кусочно-определенной **G**-функции.

Следующие две функции ***select*** и ***remove***, имеющие идентичный формат кодирования:

{***select*** | ***remove***}**(ТФ,** <*Выражение*> {**,** <*Фактические аргументы*>}**)**

позволяют {*выбирать* | *удалять*} из указанного *выражения* только те его операнды (*элементы*), которые на заданной *тестирующей* функции (**ТФ**) возвращают *true*-значение. При

этом, необязательный третий аргумент {***select***|***remove***}-функции позволяет передавать для **ТФ** необходимые фактические аргументы, например, фактические значения для второго аргумента функции {***type***|***typematch***}. Функция {***select***|***remove***} применима к *спискам*, *множествам*, *суммам*, *произведениям* и *функциям*, возвращая конструкции, *тип* которых определяется типом исходного *выражения*, например:

> select(type, [64,59,39,10,17],'even'), remove(type, [64,59,39,10,17],'odd'); ⇒ [64,10], [64,10]

По ***subsop-***функции, имеющей следующий простой формат кодирования**:**

subsop(p1 = B1, p2 = B2, ... ,pn = Bn, <*Список/Множество*>**)**

производится замена всех элементов указанного *списка/множества***,** находящихся на его $\mathbf{p_j}$-позициях, на указанные $\mathbf{B_j}$-выражения**;** в случае вложенного *списка/множества* в качестве $\mathbf{p_j}$-позиций указываются *списки* (*подобно рассмотренному для случая* ***op-****функции*). Если ***subsop-***функция не содержит уравнений замены ($\mathbf{p_j = B_j}$), то указанный *список/множество* возвращается без изменения. В общем случае, функция ***subsop*** применима к любому ***Maple-***выражению, содержащему операнды. При этом, нулевые значения для $\mathbf{p_j}$-позиций *допустимы* только для выражений типа *функция*, *индексированное выражение* или *ряд*. Замечания, сделанные относительно ***op-***функции, сохраняют силу и для функции ***subsop*****,** например:

> subsop(1=42, 2=47, 3=76, 4=96, 5 = 89, [64, 59, 39, 10, 17]); ⇒ [42, 47, 76, 96, 89]

Близка к ***subsop-***функции по *идеологии* и ***applyop-***процедура с форматом кодирования**:**

applyop(F, p, <*Выражение*> {**,** <*Фактические аргументы*>}**)**

и применяющая указанную своим идентификатором **F-**функцию к *операнду*, находящемуся на заданной **p-**позиции *выражения***;** при необходимости можно определять фактические аргументы для передачи их **F-**функции. Аргумент *позиция* (**p**) ***applyop-***процедуры определяется *номером позиции* операнда (*элемента*), *списком позиций* (*аналогично случая* ***subsop-****функции*) либо *множеством позиций* операндов, к которым **F-**функция применяется одновременно. Особенности выполнения процедуры см. в *прилож.* **1** [12]. Приведем простой пример применения процедуры ***applyop:***

> applyop(F, {1, 2, 3}, [x, y, z], a, b, c, d); ⇒ [F(x, a, b, c, d), F(y, a, b, c, d), F(z, a, b, c, d)]

В отличие от ***applyop-***процедуры ранее упоминавшаяся ***map-***функция с форматом:

map(F, <*Выражение*> {**,** <*Фактические аргументы*>}**)**

позволяет применять **F-**функцию/процедуру, заданную своим идентификатором, к каждому операнду указанного *выражения* с возможностью передачи **F-**функции необходимых *фактических* аргументов**,** которые в общем случае кодирования ***map-***функции необязательны. По ***map-***функции указанная **F-**функция применяется ко всем элементам структур типа {*list*, *set*, *array*, *table*}. Дальнейшим *расширением* данной функции является ***map2-***функция, также рассмотренная нами выше. Следующий прозрачный фрагмент иллюстрирует применение обоих функций с акцентом на их различиях**:**

> map(G, [x, y, z, u], a, b, c);	⇒ [G(x,a,b,c), G(y,a,b,c), G(z,a,b,c), G(u,a,b,c)]
> map2(G, [x, y, z, u], a, b, c);	⇒ G([x, y, z, u],a,b,c)
> map(G, t, [x, y, z, u], a, b, c);	⇒ G(t, [x, y, z, u], a, b, c)
> map(map2, [x, y, z], a, d, c);	⇒ [x(a,d,c), y(a,d,c), z(a,d,c)]
> map2(map, [x, y, z], a, d, c);	⇒ [x(a,d,c), y(a,d,c), z(a,d,c)]
> map(map2, h, [x, y, z], a, d, c);	⇒ h([x, y, z],a,d,c)
> map2(map, h, [x, y, z], a, d, c);	⇒ [h(x,a,d,c), h(y,a,d,c), h(z,a,d,c)]

В порядке расширения возможностей функций ***map*** и ***map2*** нами создан ряд подобных процедур ***map3 .. map6*** и ***mapN*** [41,103]. В частности, вызов процедуры ***mapN*****(F, p, x1, x2, …, xn, [a, b, c, …])** возвращает результат вычисления функции **F** от аргументов **<x1, x2, …, xn>** при условии получения его **p**-м аргументом значений из списка или множества, определенного последним фактическим аргументом процедуры. Процедура представляется достаточно полезным дополнением к вышеупомянутым двум стандартным функциям пакета. Ниже представлены исходный текст процедуры ***mapN*** и пример ее применения.

```
mapN := proc(F::symbol, p::posint)
local k;
    `if`(nargs – 3 < p, ERROR(`position number <%1> is invalid`, p), `if`(p = 1,
    convert([seq(F(args[-1][k], args[4 .. -2]), k = 1 .. nops(args[-1]))],
    whattype(args[-1])), `if`(p = nargs – 2, cnvtSL(
    [seq(F(args[3 .. -2], args[-1][k]), k = 1 .. nops(args[-1]))], whattype(args[-1]))
    , cnvtSL(
    [seq(F(args[3 .. p + 1], args[-1][k], args[p + 3 .. -2]), k = 1 .. nops(args[-1]))],
    whattype(args[-1])))))
end proc
> F:= (x, y, z, r, t, h) -> G(x, y, z, r, t, h):  mapN(F, 6, x, y, z, r, t, h, [a, b, c, d, e, f, n, m, p, u]);
      [G(x,y,z,r,t,a), G(x,y,z,r,t,b), G(x,y,z,r,t,c), G(x,y,z,r,t,d), G(x,y,z,r,t,e), G(x,y,z,r,t,f),
                G(x,y,z,r,t,n), G(x,y,z,r,t,m), G(x,y,z,r,t,p), G(x,y,z,r,t,u)]
```

По вызову функции ***subs*****(L, <***Выражение***>)** возвращается результат *подстановки* в указанное вторым аргументом *выражение* всех вхождений левых частей уравнений, определяемых списком/множеством **L,** на их правые части. При этом, в качестве **L**-аргумента может выступать и отдельное уравнений (*отдельная подстановка*). Подстановка правых частей уравнений производится *одновременно*, используя естественный порядок уравнений в **L**-списке/множестве. Более того, *подстановке* подвергаются только операнды исходного *выражения*, распознаваемые ***op***-функцией, независимо от уровня их *вложенности*. Однако, подстановка производится только для строгого вхождения левых частей уравнений, определяя так называемую *синтаксическую подстановку*. При этом, подстановка не влечет за собой непосредственного вычисления *выражения* и для этих целей следует использовать затем ***eval***-функцию, которая, однако, в общем случае не обеспечивает полного вычисления результата подстановки, как это иллюстрируют примеры следующего простого фрагмента:

```
> subs([x=a,y=b,z=c], [x,y,z,a*x, b*y, c*z, x+y+z, x*y*z]);  ⇒ [a, b, c, a^2, b^2, c^2, a + b + c, a b c]
> subs(x=Pi, sin(x)+cos(x)), eval(subs(x=Pi, sin(x)+cos(x)));   ⇒  sin(π) + cos(π),  -1
> subs([ooo=a,aao=b,oaa=c,aoa=d], [ooo,aao,oaa,ooo,aoa,aao,aao,oaa]);  ⇒ [a, b, c, a, d, b, b, c]
> subs([oo=NULL, aa=NULL, ao=NULL, oa=NULL], [oo, aa, aa, oo, ao, aa, ao, oa]);   ⇒ []
> subs({oo=NULL, aa=NULL, ao=NULL, oa=NULL}, {oo, aa, aa, oo, ao, aa, ao, oa});   ⇒ {}
```

Как следует из примеров фрагмента, по ***subs***-функции можно не только заменять элементы списка/множества на основе их значений, но и удалять требуемые элементы.

Под структурой *listlist*-типа понимается *вложенный* список, все элементы первого (*внешнего*) уровня которого являются списками той же длины**,** что и содержащий их список**,** например: **[[a, b, c], [d, e, f], [g, h, i]]**. Такой объект распознается ***type***-функцией как объект *listlist*-типа, например, ***type*****([[a,b,c], [d,e,f], [g,h,i]], 'listlist');** ⇒ *true*. По ***convert***-функции можно конвертировать {*список*|*массив*} в *список списков* (*listlist*)**;** при этом, сам *список*

должен задаваться уравнениями вида <*позиция списка*> **=** <*элемент списка*>. Однако в ряде версий **6**-го релиза вызов ***convert*(R::array, listlist)** вызывает ошибочную ситуацию с диагностикой "Error, in convert/old _ array_to_listlist) Seq expects its 2nd argument …. ". Для устранения данной ситуации может использоваться модификация стандартной функции – процедура ***'convert/listlist1'*** [41,103,109], которая обеспечивает конвертацию *списка*, *множества*, *массива* либо *rtable*-объекта в *listlist*-объект, например**:**

> convert([2=64, 3=10, 4=39, 5=59, 1=17, 6=44], 'listlist1'); ⇒ [17, 64, 10, 39, 59, 44]

В ряде задач, имеющих дело со структурами *list*-типа, возникает потребность использования так называемых *вложенных* структур списков (*nestlist*). Скажем, список – *вложенный список*, если по крайней мере *один* его элемент имеет *list*-тип. Для проверки списка быть *вложенным* списком служит расширение стандартной ***type***-функции [103]. Вызов процедуры ***type*(*expr*, 'nestlist')** возвращает *true*, если список, определенный *первым* фактическим ***expr*** аргументом является *вложенным списком*, и *false* в противном случае, например:

> L:= [a, c, [h], x, [a, b, []]]: type([], nestlist), type([[]], nestlist), type(L, nestlist), type([[[]]], nestlist), type([[61], [56], [36], [7], [14]], nestlist), type([a, b, c, {}], nestlist);

false, *true*, *true*, *true*, *true*, *false*

В ряде задач, имеющих дело со структурами *set*-типа, возникает потребность использования так называемых *setset*-структур (*аналогично listlist*). Скажем, множество имеет тип *'setset'*, если его элементы – множества *того же самого* количества элементов. Для проверки множества на *setset*-тип служит расширение стандартной ***type***-функции [41,103], т.е. вызов процедуры ***type*(S, 'setset')** возвращает *true*, если множество, определенное *первым* фактическим **S** аргументом, имеет *setset*-тип, и *false* в противном случае, например:

> map(type, [{{}}, {{},{}}, {{{7}}}, {{a, b}, {c, d}}, {{a, b}, {c}}, {{10, 17}, {64, 59}, {39, 44}}], 'setset');

[*true*, *true*, *true*, *true*, *false*, *true*]

По функции ***sort*(L {, СФ})** производится *сортировка* элементов **L-**списка либо согласно принятому в пакете соглашению (*определяемому типом элементов списка*), либо согласно заданной *сортирующей функции* (**СФ**), устанавливающей приоритет между любыми *двумя* элементами списка. А именно, **СФ G(x, y)** возвращает {*true*|*false*}-значение в зависимости от {*соответствия*|*несоответствия*} двух смежных {**x, y**}**-**элементов **L-**списка и в зависимости от него {*не меняет*|*меняет*} их местами в выходном **L`-**списке. Наиболее типичными примерами **СФ,** поддерживаемыми ***Maple***-языком, являются**: *address*, *numeric*, `<` и *lexorder*;** при этом, первая и последняя определяют сортировку соответственно адресную (*в соответствии со значениями адресов, занимаемыми в памяти элементами списка*) и лексикографическую. Если **СФ-**аргумент ***sort*-**функции не указан, то *числовой* список сортируется согласно значениям *чисел* в порядке их возрастания, *символьный* (*строчный*) список – в *лексикографическом* порядке, в остальных случаях производится *адресная* сортировка, которая зависит от текущего сеанса работы с пакетом (*в свете планирования для пакета памяти* **ПК**). Пользователь имеет возможность определять собственные **СФ,** решающие *специфические* задачи сортировки**;** один из примеров такой **СФ** (**SF**) приведен в нижеследующем простом фрагменте:

> SF:=(X::{string,symbol}, Y::{string,symbol}) -> `if`(""||X = "" and ""||Y = "" or ""||X="", true, `if`(""||Y = "", false, `if`(sort([(""||X)[-1], (""||Y)[-1]], 'lexorder')[1] = (""||X)[-1], true,false))): sort(["12345", xyz, `123d`, "123a", `123c`, ``, abc, "", ta,avz64, agn59, vsv39],SF);

[``, "", *avz64*, "12345", *agn59*, *vsv39*, "123a", *ta*, *123c*, *abc*, *123d*, *xyz*]

Для работы со *списками* и *множествами* нами был предложен целый ряд достаточно полезных процедур [103], в частности, полезной во многих приложениях является проце-

дура ***SLj,*** обеспечивающая сортировку *вложенных* списков на основе заданных позиций элементов их *подсписков*. Ниже представлены ее исходный текст и пример применения.

```
SLj := proc(L::nestlist, n::posint)
local a, b, c, k, p, R;
    assign( b = map( nops, { op( L ) } )[ 1 ], c = nops( L ) ), `if`( b < n,
        ERROR( "invalid 2nd argument <%1>; must be >= 1 and <= %2", n, b ),
        assign( R = [ ] )), assign( a = sort( [ op( { seq( L[ k ][ n ], k = 1 .. c ) } ) ],
        `if`( 2 < nargs and type( args[ 3 ], 'boolproc' ), args[ 3 ], NULL ) ) );
    for k to nops( a ) do
        for p to c do if a[ k ] = L[ p ][ n ] then R := [ op( R ), L[ p ] ] end if end do
    end do;
    R
end proc
> K:= [[a,b,c], [g,d,c], [a,g,c,c,c,l], [l,s,a], [f,d,k,k], [s,a,d,b,w,q,s,d,a]]:  SLj(K, 2, 'lexorder');
        [[s, a, d, b, w, q, s, d, a], [a, b, c], [g, d, c], [f, d, k, k], [a, g, c, c, c, l], [l, s, a]]
```

В качестве *обобщения* процедуры ***SLj*** можно рассматривать процедуру ***snl,*** чей *исходный* текст и примеры применения приводяюся в нижеследующем фрагменте. При этом, в первом примере приводится расширение типа ‘relation’, полезное в целом ряде приложений, имеющих дело с выражениями *relation*-типа.

```
type/relation1 := proc (x::symbol)
    if member( x, { `=`, `>`, `>=`, `<`, `<=`, `<>` } ) then true else false end if
end proc
> map(type, [`<`, `<=`, `=`, `<>`, `>=`, `>`], 'relation');
                                [false, false, false, false, false, false]
> map(type, [`<`, `<=`, `=`, `<>`, `>=`, `>`], 'relation1');
                                 [true, true, true, true, true, true]
snl := proc (L::nestlist, f::procedure, p::list(posint))
local a, b, sf, k;
    seq( `if`( not type( map( evalf, k ), list( numeric ) ),
        ERROR( "element %1 has a type different from list or is non-numeric list", k )
        , `if`( sort( p )[ -1 ] ≤ nops( k ), NULL,
        ERROR( "sublist %1 has length more than maximal element of list %2", k, p ) ) )
        , k = L );
    if nargs = 3 then a := `<=`
    elif 4 ≤ nargs and type( args[ 4 ], 'relation1' ) then a := args[ 4 ]
    else
        WARNING( "4th argument has a type different from relation1; for it defau\
            lt value had been set");
        a := `<=`
    end if;
    sf := proc (L1, L2)
        local k, j;
            evalb( evalf( a( f( seq( L1[ k ], k = p ) ), f( seq( L2[ j ], j = p ) ) ) ) )
        end proc ;
    sort( L, sf )
end proc
```

```
> f:= () -> `+`(args):  snl([[2,6,9], [11,34,47], [67,2,10,18]], f, [1, 2, 3], `>`);
                                [[11, 34, 47], [67, 2, 10, 18], [2, 6, 9]]
> snl([[2,6,9], [11, 34, sin(18)], [67, 2, 10, 18]], f, [1, 2, 3]);
                                [[2, 6, 9], [11, 34, sin(18)], [67, 2, 10, 18]]
> snl([[2,6,9], [11,34,47], [67,2,10,18]], f, [1, 2, 5], `>`);
Error, (in snl) sublist [2, 6, 9] have length more than maximal element of list [1, 2, 5]
> snl([[2,6,9], 2007, [67,2,10,18]], f, [1, 2, 3]);
Error, (in snl) element 2007 has a type different from list or is non-numeric list
```

Первый аргумент вызова процедуры ***snl*(L, f, p)** определяет вложенный список, тогда как второй определяет процедуру, чьи фактические аргументы получают в качестве значений элементы подсписков **L,** стоящие на указанных аргументом **p** позициях. *Подсписки* **L** по умолчанию сортируются в зависимости от значений **f(x1, x2, …)** согласно функции упорядочивания **`<=`;** четвертый необязательный аргумент определяет упорядочивающую функцию из множества {**`<`, `<=`, `>`, `>=`**}. Сортировке подвергаются только числовые вложенные списки, в противном случае инициируется ошибочная ситуация.

Нижеследующий фрагмент первым примером представляет процедуру, обеспечивающую один вид полезной конвертации произвольного ***Maple***-выражения **L** на основе составляющих его операндов. В частности, данный тип конвертации достаточно полезен в случае необходимости конвертации списка/множества в выражение, представляющее собой «*склейку*» входящих в исходной объект элементов. Тогда как *второй* пример фрагмента представляет один полезный вид сортировки списочных структур на основе процедуры ***slsa,*** использующей представленные выше процедуры.

```
`convert/opexpr` := proc(L::anything)  if type(evalf(L), 'numeric') then L else
        parse(cat(op(map(convert, [op(L)], 'string'))))  end if  end proc
> convert([A, V, 1942, "AVZ+AGN", RANS_IAN, 42, 47, 67, 10, 18], 'opexpr');
                                AV1942AVZ + AGNRANS_IAN4247671018
slsa := proc (L::list( { algebraic , string, symbol } ))
local  a, b, c, k, ss;
      assign( a = [  ], b = [  ]), seq( `if`( type( evalf( k ), 'numeric' ),
           assign('a' = [ op( a ), k ]), assign('b' = [ op( b ), k ] )), k = L );
      ss := ( a::anything , b::anything ) → `if`( c( convert( convert(
           convert( evalf( a ), 'string' ), `if`( type( evalf( a ), 'numeric' ), 'string', 'bytes' ) )
           , opexpr ), convert( convert( convert( evalf( b ), 'string' ),
           `if`( type( evalf( b ), 'numeric' ), 'string', 'bytes' ) ), opexpr ) ), true , false );
      assign( c = `<=`),
           `if`( 1 < nargs and  type( args[ 2 ], 'relation1' ), assign('c' = args[ 2 ]), NULL )
           , [ op( sort( a, ss ) ), op( sort( b, ss ) ) ]
end proc
> slsa(["a", 678,"arf",1942,"yrt",sqrt(2),"aqw", 2007,64,59, (avz+agn)/(Art+Kr), sin(18), 10]);
                [ sin( 18 ), √2 , 10, 59, 64, 678, 1942, 2007, "a", "aqw", "arf", "yrt", (avz + agn)/(Art + Kr) ]
> slsa(["a",678,"arf",1942,"yrt",sqrt(2),"aqw",2007,64,59,(avz+agn)/(Art+Kr),sin(18),10], `>=`);
                [ 2007, 1942, 678, 64, 59, 10, √2 , sin( 18 ), (avz + agn)/(Art + Kr), "yrt", "arf", "aqw", "a" ]
> slsa(["a","arf","yrt","aqw", (avz+agn)/(Art+Kr), RANS_RAC_REA_IAN], `>=`);
                        [ (avz + agn)/(Art + Kr), RANS_RAC_REA_IAN, "yrt", "arf", "aqw", "a" ]
```

```
> slsa(["a","arf","yrt","aqw", (avz+agn)/(Art+Kr), RANS_RAC_REA_IAN]);
```

$$\left[\text{"a"}, \text{"aqw"}, \text{"arf"}, \text{"yrt"}, RANS_RAC_REA_IAN, \frac{avz + agn}{Art + Kr}\right]$$

Вызов процедуры ***slsa*(L)** возвращает результат *сортировки* списка **L** с элементами типов {*algebraic*, *string*, *symbol*} в порядке возрастания элементов. При наличии *второго* необязательного аргумента *relation1*-типа *сортировка* определяется заданным отношением. При этом, в начале отсортированного списка находятся (*если имеются*) элементы с *числовыми* значениями **a** (*для которых* ***type*(*evalf*(a), 'numeric') ⇒** *true*), за которыми следуют (*при наличии*) элементы других допустимых типов.

В свете сказанного, во многих случаях можно рассматривать *множества* как упорядоченные *объекты* и использовать данное обстоятельство в различных приложениях. В общем же случае это не имеет места, поэтому использовать его следует весьма осмотрительно. Именно по этой причине для *удаления* элемента из множества недопустимо использование конструкций, рассмотренных выше для случая *списочных* структур. Этих целей нами был предложен ряд средств [41,103,108,109]. В частности, нижеследующая процедура

```
insituls := proc(L::uneval, a::{equation, list(equation), set(equation)})
    if not type(L, 'symbol') then
        error "1st argument must 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, seq(k, k = a))], eval(L)))
    else error "1st argument must has type {list, set} but had received %1-type"
        whattype(eval(L))
    end if
end proc
> L:= [64,59,39,44,10,17]: insituls(L, [64=42, 59=47, 17=89]), L, insituls(L, 44=NULL), L;
                                [42, 47, 39, 44, 10, 89], [42, 47, 39, 10, 89]
> S:= {64,59,39,44,10,17}: insituls(S, [64=42, 59=47, 17=89]), S, insituls(S, 44=NULL), S;
                                {10, 39, 42, 44, 47, 89}, {10, 39, 42, 47, 89}
```

Вызов процедуры ***insituls*(L, a)** обеспечивает *обновление* элементов списка/множества **L** «на месте»; при этом, **a**-аргумент может быть уравнением либо списком/множеством уравнений, определяющих замены в **L.** Левые части уравнений определяют текущие элементы списка/множества **L,** тогда как его правые – замены для текущих элементов. В частности, ***NULL*** в качестве правой части удаляет соответствующий элемент из **L**. Успешный вызов процедуры возвращает *NULL*-значение, выполняя требуемое *обновление* списка/множества **L** «на месте».

```
insitudelL := (L::uneval, P::{set(posint), list(posint)}) → parse(
    "" || L || ":=[seq(`if`(member(`_0`,"|| (convert(P, 'string')) || "),NULL," || L ||
    "[`_0`]),`_0`=1..nops("|| L || "))]:", 'statement')
> H := [a,b,c,h,g,k,d,r,t,y,e]: insitudelL(H, {2,4,6,8,10}), H;  ⇒ [a, c, g, d, t, e], [a, c, g, d, t, e]
```

Реализованная однострочным экстракодом процедура ***insitudelL*,** обеспечивает *удаление* «на месте» элементов списка **L,** заданных списком/множеством их позиций **P** [103,109].

В ряде задач работы со вложенными списками/множествами требуется определять элементы с максимальным/минимальным *уровнем вложенности*. Здесь может быть полезна процедура ***mlsnest*.** Вызов процедуры ***mlsnest*(L)** возвращает 2-элементный вложенный список, чей первый подсписок содержит номер элемента списка/множества **L** и мини-

мальный уровнь его вложенности, тогда как второй подсписок определяет элемент **L** и максимальный уровнь его вложенности. При вызове ***mlsnest*(L, 't')** через **t** возвращается дополнительно множество всех уровней вложенности списка/множества **L.**

```
mlsnest := proc(L::{list, set})
local a, n, k, j, t, r, P, N, G;
    `if`(Empty(L), RETURN(L), [
        assign(n = [ ], t = 0, r = 0, N = [ ], a = whattype(L)),
        assign(G = convert(L, 'list'))]);
    for k to nops(G) do
        if not type(G[k], a) then n := [op(n), [k, t]]
        else
            assign('t' = 1, 'P' = G[k]);
            do
                for j in P do
                    if not type(j, a) then next
                    elif type(j, a) and r = 0 then
                        N := [op(N), op(j)]; assign('t' = t + 1, 'r' = 16)
                    else N := [op(N), op(j)]
                    end if
                end do;
                if r = 0 then
                    assign('n' = [op(n), [k, t]], 'N' = [ ], 'r' = 0, 't' = 0); break
                else assign('P' = N, 'r' = 0), assign('N' = [ ])
                end if
            end do
        end if
    end do;
    assign('r' = SLj(n, 2)), [r[1], r[-1]], `if`(1 < nargs and type(args[2], 'symbol'),
        assign('args'[2] = {seq(r[k][2], k = 1 .. nops(r))}), NULL)
end proc
> L:=[a,b,[x,[a,b],[c,b,x,y],[x,y]],c,[y,[a,b,[x,y,[2,z],z],[[8,16]]]],[[[[[a,b,c]]]]]: mlsnest(L, 'z'), z;
                                [[1, 0], [6, 5]], {0, 2, 4, 5}
```

По вызову функции ***convert*(B, 'set')** можно конвертировать в структуру *set*-типа массив, таблицу или произвольное **B**-выражение, чьи *операнды* становятся элементами *множества*. При этом, операция конвертации *список* ⇒ *множество* ⇒ *список* в общем случае не является *обратимой*, хотя оба типа структур и базируются на общей *последовательностной* (*exprseq*) структуре.

Из рассмотренных выше средств по работе со *списочными* структурами кроме ***sort***-функции остальные распространяются и на *множества*, не требуя дополнительных пояснений. Однако в случае необходимости любая функция, работающая со *списочными* структурами, может быть использована и с *множествами*, конвертировав их предварительно в списки. Этому способствует и то обстоятельства, что конвертация типа: *множество* ⇒ *список* ⇒ *множество* является *обратимой*, в отличие от конвертации типа: *список* ⇒ *множество* ⇒ *список*. Вместе с тем, структуры типа *множество* обеспечены ***Maple***-языком рядом специальных функций для поддержки теоретико-множественных операций.

Для обеспечения работы с *множествами* ***Maple-***язык пакета поддерживает три базовых теоретико-множественных оператора/функции, имеющих форматы кодирования**:**

M1 {**union**|**intersect**|**minus**} **M2**
{**`union`**|**`intersect`**|**`minus`**}**(M1, M2, …, Mn)**

и возвращающие соответственно результаты *объединения* (***union***), *разности* (***minus***) и *пересечения* (***intersect***) **Mk-**множеств (**k = 1..n**). Операторы {**union, intersect**} являются **n-**местными**,** *инфиксными***,** *ассоциативными* и *коммутативными*, а **minus -** *инфиксным* бинарным оператором. Приведем простой пример на применение данных средств:

> `union`({a,b,c}, {x}), `intersect`({a,b,c}, {a,c}), `minus`({a,b,c}, {b}); ⇒ {a, b, c, x}, {a, c}, {a, c}

Рассмотренные в настоящем разделе средства ***Maple-***языка для работы со списочными структурами и множествами будут в различных контекстах (*глубже проясняющих их суть и назначение*) использоваться при представлении разнообразных иллюстративных примеров. Между тем, следует иметь в виду, что наряду с рассмотренными пакет в рамках модульной части главной библиотеки предоставляет ряд функций по работе с рассмотренными структурами (*списки и множества*), поддерживаемых, в первую очередь, модулями **combinat** и **totorder,** позволяющими соответственно использовать комбинаторные операции со списками и упорядоченными множествами**.** Начиная с **7**-го релиза, поставляется и модуль **ListTools,** содержащий набор средств для работы со *списочными* структурами. Его включение, по-видимому, было навеяно нашим набором средств подобного типа, созданных еще для ***Maple V*** (*все наши издания по* ***Maple****-тематике хорошо известны разработчикам*). Однако, не взирая на это, представленные в {103] средства работы со *списками* и *множествами* существенно дополняют имеющиеся средства пакета. Наконец, из-за пересечения свойств ряда типов структур, поддерживаемых пакетом, многие его функции в этом отношении многоаспектны, поэтому со *списочными* структурами и *множествами* допустимо использование и не только специфических для них средств.

Табличные объекты. Табличный объект (*либо просто таблица*) создается по встроенной функции ***table*(F, L),** где **F** – *индексная* функция и **L** – список/множество начальных входов таблицы. При этом, оба аргумента *необязательны*. Более того, создавать *таблицу* можно как явно по ***table***-функции, так и *неявно* путем присвоения выражений индексированному идентификатору. В обоих случаях созданный объект опознается тестирующими средствами ***Maple***-языка как *таблица*, например:

> T:= table([V=64, G=59, S=39, Art=17, Kr=10]): H[V]:=64: H[G]:=59: H[S]:=39: H[Art]:=17: H[Kr]:=10: map(type, [T, H], 'table'), map(typematch, [T, H], 'table'), map(hastype, [T, H], 'table'), map(whattype,map(eval, [T,H])); ⇒ [*true, true*], [*true, true*], [*true, true*], [*table, table*]

В отличие от *массивов*, у которых *индексы* должны быть целочисленны, *входы* (*или индексы*) таблицы могут быть любыми ***Maple***-выражениями. Данное обстоятельство определяет важность и распространенность использования *табличных объектов*. В частности, в качестве *входов* таблицы могут выступать имена процедур, а *выходов* – соответствующие им определения. Ряд пакетных модулей организованы именно таким образом.

Функция ***table*** создает таблицу с начальными значениями, определяемыми **L**. Если **L** – *список* или *множество уравнений*, то левые части **L** становятся *входами* таблицы, а правые части ее *выходами*. В противном случае, элементы **L** становятся *выходами* таблицы, а *входами* становятся целые числа, начиная с **1**. Использование множества начальных значений **L** в последнем случае *неоднозначно*, т.к. отсутствует порядок элементов множества, и следовательно *порядок* вхождений в результате может не соответствовать порядку, в котором *вхождения* были сделаны. Если аргумент **L** не определен, то создается пустая таблица, т.е. table([]). Новый вход в таблицу **T** можно добавлять по конструкции **T[***Вход***]:=**

<*Выход*>. По таким же конструкциям можно и создавать таблицу неявно. *Удаление* входов из таблицы **T** можно выполнять по конструкции **T[**<*Вход*>**]:= 'T[**<*Вход*>**]'**, например, **T[a]:= 'T[a]'** удаляет из таблицы **T** элемент со входом **a**. При этом, обновление таблицы производится *«на месте»*. Между тем, в ряде случаев более удобным является применение для этих целей нашей процедуры ***detab*** [103,108,109].

```
detab := proc(T::table, a::anything)
local k;
    if nargs = 2 then table( subs( `if`( type( a, {'set', 'list'} ),
        [ seq( ( a[k] = T[a[k]] ) = NULL, k = 1 .. nops( a ) ) ], ( a = T[a] ) = NULL ),
        op( eval( T ) ) ) )
    else assign( 'T' = table( subs( `if`( type( a, {'set', 'list'} ),
        [ seq( ( a[k] = T[a[k]] ) = NULL, k = 1 .. nops( a ) ) ], ( a = T[a] ) = NULL ),
        op( eval( T ) ) ) ) )
    end if
end proc

detab := proc(T::uneval, a::anything)
local k;
    if type( eval( T ), 'table' ) then
        if nargs = 2 then table( subs( `if`( type( a, {'set', 'list'} ),
            [ seq( ( a[k] = T[a[k]] ) = NULL, k = 1 .. nops( a ) ) ],
            ( a = T[a] ) = NULL ), op( eval( T ) ) ) )
        else assign( 'T' = table( subs( `if`( type( a, {'set', 'list'} ),
            [ seq( ( a[k] = T[a[k]] ) = NULL, k = 1 .. nops( a ) ) ],
            ( a = T[a] ) = NULL ), op( eval( T ) ) ) ) )
        end if
    else error
        "1st argument must has table-type but received %1-type",whattype( eval( T ) )
    end if
end proc
> T:= table([V=64, G=59, S=39, Art=17, Kr=10]): detab(T, {G, Art}), eval(T);
        table([V = 64, S = 39, Kr = 10]), table([V = 64, G = 59, S = 39, Art = 17, Kr = 10])
> detab(T, {G, Art}, 10), eval(T);    ⇒ table([V = 64, S = 39, Kr = 10])
```

Достаточно простая процедура ***detab*****(T, a {, b})** возвращает результат *удаления* из таблицы **T** элементов с входами, определенными вторым **a**-аргументом, в качестве которого может выступать как отдельный *вход*, так и их *список/множество*. В случае кодирования третьего необязательного аргумента (*произвольное* ***Maple****-выражение*) *обновление* таблицы **T** удаляемыми элементами производится *«на месте»*, т.е. обновляется *сама исходная* таблица **T**. Выше представлены исходные тексты двух реализаций процедуры и некоторые примеры ее применения.

Замечание. *Подход*, реализованный в данной процедуре, может быть успешно применен в тех случаях, когда требуется вызовом процедуры обновлять *вычисленные* ***Maple***-объекты, находящиеся вне тела самой процедуры, т.е. *глобальные* относительно ее и выступающие даже в качестве фактических аргументов. При этом, следует иметь в виду, что если в качестве обновляемых выступают объекты, не использующие специальных вычислительных правил подобно таблицам и процедурам (*т.е. обращение к ним возвращает их*

идентификаторы, *а не значения*), то при указании их в качестве формального аргумента им следует присваивать *uneval*-тип. *Второй* способ реализации ***detab***-процедуры иллюстрирует вышесказанное. Между тем, операцию обновления *«на месте»* следует выполнять довольно осмотрительно во избежание возможной *рассинхронизации* вычислительного процесса в целом.

Функция индексации **F** может быть процедурой или идентификатором, определяющим каким образом должна выполняться индексация; по умолчанию полагается обычная индексация. В качестве встроенных функций индексации пакет допускает: ***symmetric***, ***antisymmetric***, ***sparse***, ***diagonal*** и ***identity***. Для дополнительной информации об этих функциях см. **?indexfcn**.

Таблицы имеют специальные правила вычисления (*подобно процедурам*) такие, что если имя **T** было присвоено таблице, то ***type*(T, 'symbol') = *true*** и ***type*(eval(T), 'table') = *true***. Вызов ***op*(T)** либо ***op*(1, T)** возвращает фактическую структуру таблицы **T** и ***op*(*op*(T))** либо ***op*(2, eval(T))** возвращает компоненты таблицы в составе функции индексации (*если существует*) и списка уравнений для значений таблицы, как это иллюстрирут следующий весьма простой фрагмент:

```
> T:=table([x=a, y=b, z=c]): type(T, 'symbol'), type(eval(T), 'table'), whattype(T),
  whattype(eval(T));   ⇒ true, true, symbol, table
> op(T), op(1, T);   ⇒ table([z = c, y = b, x = a]), table([z = c, y = b, x = a])
> op(op(T)), op(2, eval(T));   ⇒ [z = c, y = b, x = a], [z = c, y = b, x = a]
```

Функция ***indices*(T)** возвращает *входы* таблицы **T**, тогда как функция ***entries*(T)** – ее *выходы*. В то же время формат возврата не всегда удобен для его последующего использования. Поэтому *второй пример* нижеследующего фрагмента приводит более приемлемый формат, в котором *между* обоими возвращаемыми списками имеет место взаимно-однозначное соответствие:

```
> indices = indices(T), entries = entries(T);   ⇒ indices = ([z], [y], [x]), entries = ([c], [b], [a])
> indices = map(op, [indices(T)]), entries = map(op, [entries(T)]);
                                  indices = [z, y, x], entries = [c, b, a]
```

Табличная структура – одна из *наиболее* используемых пакетом. Она используется не только для вычислений, но и для хранения процедур при организации, например, пакетных модулей. Следующий несложный фрагмент иллюстрирует организацию пакетного модуля на основе табличной структуры:

$$ST := \mathrm{table}\left([Sr = \left((\,) \to \frac{\grave{}+\grave{}(\mathrm{args})}{\mathrm{nargs}} \right),\right.$$

$$\left. Dis = \left((\,) \to \sqrt{\frac{\mathrm{add}((\mathrm{args}_k - \mathrm{Sr}(\mathrm{args}))^2, k = 1 \,..\, \mathrm{nargs})}{\mathrm{nargs}}} \right) \right])$$

```
> with(ST), 6*ST[Sr](64, 59, 39, 44, 10, 17), 6*ST[Dis](64, 59, 39, 44, 10, 17);
```

$$[Dis, Sr], 233, \sqrt{14249}$$

```
> UpLib("C:\\Program Files\\Maple 8\\Llib\\UserLib", [ST]);
Warning, Library update has been done!
> restart;  ST:- Sr(64, 59, 39, 44, 10, 17), ST:- Dis(64, 59, 39, 44, 10, 17);
Error, `ST` does not evaluate to a module
> ParProc(ST);
Error, (in ParProc) <ST> is not a procedure and not a module
> with(ST), 6*ST[Sr](64, 59, 39, 44, 10, 17), 6*ST[Dis](64, 59, 39, 44, 10, 17);
```

$$[Dis, Sr], 233, \sqrt{14249}$$

> type(ST, 'package'), whattype(ST), whattype(eval(ST)), M_Type(ST);

true, *symbol*, *table*, *Tab*

В приведенном фрагменте в табличную **ST**-структуру погружаются две простые процедуры. Вызов **with(ST)** для которой возвращает *список* находящихся в ней процедур, тогда как *индексированные* вызовы этих процедур на кортежах фактических аргументов возвращают *искомые* результаты – их *среднюю* и *дисперсию*. После чего процедура ***UpLib*** [41] сохраняет **ST**-таблицу в библиотеке пользователя **UserLib,** которая логически сцеплена с *главной* библиотекой ***Maple***. После выполнения **restart**-предложения делается попытка обратиться к сохраненной таблице аналогично модуля, что вызывает соответствующую ошибочную ситуацию. Ошибку вызывает и процедура ***ParProc*(ST)** [103], тестирующая параметры *процедур*, *программных* и *пакетных* модулей. Тогда как *индексированный* вызов дает корректные результаты. Наконец, сохраненная **ST**-*таблица* распознается как *пакет* (*пакетный модуль в нашей терминологии* {*табличной организации*}) и *таблица*, а также процедурой ***M_Type*(ST)** [103,109] как пакет *табличной* организации. Следует отметить, например, что ***Maple 8*** содержит **34** пакета табличной организации, ***Maple 9* – 23**, а вот уже ***Maple 10*** только **16**. Статистика говорит о *снижении* количества пакетов табличной организации с ростом номера релиза ***Maple***. Между тем, они все еще играют весьма существенную роль, о чем говорит следующий пример:

> map(M_Type, [Slode, context, plots, simplex, student, tensor, DEtools, diffalg, LREtools, PDEtools, algcurves, orthopoly, combstruct, difforms, inttrans, networks]); # Maple 10

[*Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*, *Tab*]

Обсуждение причин такого явление не входит в задачи данной книги. Заинтересованный же читатель может обратиться к нашим книгам [41,42,45,46,103]. Для работы с выражениями типов {*list*, *table*, *set*} ***Maple*-я**зык располагает целым рядом средств, основные из которых были представлены либо упомянуты в настоящем разделе. Ряд достаточно полезных средств для *обработки* такого типа выражений представляет и наша *Библиотека* [103], демо-версию которой и саму *Библиотеку* можно загрузить с адресов [108,109].

Так, глава **6** [103] представляет средства, расширяющие возможности ***Maple*-я**зыка при работе с объектами типов {*list*, *set*, *table*}. *Списочные* структуры играют чрезвычайно важную роль, определяя упорядоченные последовательности элементов. Начиная с ***Maple* 6,** появилась возможность существенного расширения операций со списочными структурами. В качестве примера, имеющего интересные практические приложения, мы рассматриваем определение алгебры на множестве всех списков, имеющих одну и ту же длину. Алгебраические операции над списками обеспечивают соответствующие процедуры. Ряд процедур главы поддерживает полезные виды обработки типа: специальное преобразование списков в множества, и наоборот, операции с *разреженными* списками, динамические присваивания значений элементам списка или множества, оценка *входов* таблицы по ее *выходу*, представление специального типа таблиц, специальные виды исчерпывающих *подстановок* в списки или множества, целый ряд важных видов сортировки вложенных списков, и много других. Данные инструментальные средства оказались достаточно полезными при работе с объектами вышеупомянутых типов в среде ***Maple***.

5.3. *Алгебраические правила подстановок для символьных вычислений*

Для решения задач символьной обработки, имеющих дело с формальными системами подстановок, ***Maple-***язык располагает средствами обеспечения работы с *алгебраическими* правилами *подстановок*. Данные средства представляют основной аппарат при исследованиях формальных систем обработки слов в конечных алфавитах и абстрактных моделей вычислений. Основным *понятием* здесь является *правило подстановки*, определяемое группой функций ***subs, subsop*** и процедур ***algsubs, asubs.***

По первой уже рассматриваемой функции ***subs*****({x = a|<УР>}, V)** производится подстановка **a-**выражения вместо каждого *вхождения* **x-**выражения в **V-**выражение или одновременная подстановка правых частей *уравнений* (**УР**), заданных списком/множеством, вместо *всех вхождений* в **V-**выражение соответствующих им левых частей *уравнений*. При этом, по ***subs-***функции делаются подстановки лишь для вхождений левых частей уравнений (*правил подстановки*) в качестве *операндов* **V-**выражения**.** Такого типа подстановки носят своего рода синтаксический характер, *глубоко* не затрагивая структуры **V-**выражения. Результатом подстановки не является вычисление и при необходимости выполнения *полного* вычисления результата подстановки требуется применение ***eval-***функции, как показано на фрагменте, представленном несколько ниже.

Для возможности обеспечения *выборочных* подстановок служит специальная ***subsop*****($n_1 = V_1, n_2=V_2, ..., n_p=V_p$, W)-**функция, по которой производится замена на *правые* V_j**-**части уравнений операндов **W-**выражения, определяемых их n_j**-***номерами*. При этом, в качестве *левых* n_j**-**частей допустимо использование списков целочисленных выражений, определяя *подоперанды* **W-**выражения в порядке понижения их уровней вложенности. Целочисленные значения должны находиться в диапазоне **[-*nops*(), *nops*(W)]**, а *нуль*-значение допустимо только для функций, индексированных выражений и рядов. В случае n_j отрицательного номер полагается равным ***nops*****(W) + n_j + 1.**

По процедуре ***asubs*****(∑ = V, W {, x|, x, *always*|, *always*})** производится подстановка в **W-**выражение аналогично случаю ***subs-***функции, однако она носит скорее *алгебраический*, чем *синтаксический* характер, допуская в качестве *левой* **∑-**части уравнения *(правила подстановки*) использование сумм операндов *полиномиального* типа, замещающих соответствующие им суммы в исходном **W-**выражении. При этом, *замена* сумм производится только в том случае, если левая **∑-**часть правила подстановки и соответствующее ей *подвыражение* **W-**выражения являются развернутыми полиномами по ведущей **x-**переменной. Необязательная ***always-***опция позволяет представлять каждое заменяемое **П-***подвыражение* **W-**выражения в виде **П-∑+V.** Аналогично рассмотренному выше случаю ***subs-***функции результатом подстановки на основе ***asubs-***процедуры не является *вычисление* и при необходимости выполнения *полного вычисления* результата подстановки требуется применение ***eval-***функции, как это проиллюстрировано в нижеследующем фрагменте.

Наконец, по процедуре ***algsubs*****(a=b, V {, x|, x,** <*Опция*>**})** производится алгебраическая подстановка **b-***подвыражения* вместо каждого вхождения **a-**подвыражения в **V-***выражение*. В данном отношении ***algsubs-***процедура является *обобщением* вышерассмотренной ***subs-***функции, осуществляющей *синтаксического характера* подстановки. Расширенные возможности первой функции относительно второй хорошо иллюстрируют примеры нижеследующего фрагмента. Более того, в отличие от ***subs-***функции, процедура ***algsubs*** выполняет подстановку в **V-**выражение *рекурсивно*, не делая подстановок внутри *индек-*

сированных подвыражений. Между тем, она также перед подстановкой не производит раскрытия степеней и произведений, что требует в ряде случаев *предварительного* применения к **V-**выражению ***expand-***функции.

В случае выполнения подстановок в **W-**выражение от нескольких ведущих переменных возможно появление неопределенностей**,** связанных с неоднозначностью толкования правила применения подстановки**.** Для устранения подобных ситуаций при проведении подстановок в **W-**выражение используются необязательные третий и четвертый аргументы ***algsubs-***процедуры**.** Прежде всего**,** третий **x-**аргумент функции**,** кодируемый в виде списка, устанавливает порядок ведущих переменных, определяющий сам режим подстановки. При отсутствии данного аргумента порядок переменных устанавливается на основе правила подстановки**,** заданного первым фактическим аргументом функции. Совместно с третьим аргументом может использоваться и опция, допускающая два значения**:** ***remainder*** (*по умолчанию*) и ***exact*****,** определяющие режим выполнения *подстановки* в обрабатываемое **W-**выражение. В частности, при отсутствии четвертого аргумента в результате подстановки в рациональное **W-**выражение вычисляется обобщенный остаток. Тогда как по ***exact-***опции в случае, если в правиле подстановки **a=b** левая ее **a-**часть является суммой термов, то подстановка производится только тогда, когда **a-**часть является точным делителем замещаемого *ею* подвыражения в **W-**выражении.

В отличие от вышерассмотренных функции ***subs*** и процедуры ***asubs*****,** результатом подстановки на основе ***algsubs-***процедуры является *вычисление*, поэтому не требуется последующего применения ***eval-***функции для обеспечения полного вычисления результата подстановки**.** Данное свойство функции позволяет успешно использовать ее для обеспечения символьно-численных вычислений**,** включающих как символьные преобразования**,** так и численные вычисления. Рассмотренные функциональные средства **Subs-**группы играют весьма важную роль во многих задачах символьной обработки алгебраических выражений в среде ***Maple***

В отличие от рассмотренных средств **Subs-**группы, обеспечивающих, в первую очередь, символьную обработку выражений на основе *синтаксически-алгебраических* подстановок, ***applyop-***процедура**,** имеющая следующий простой формат кодирования**:**

applyop*(F,** <*Операнды*>**, W** {**,** <**F-аргументы*>**})**

обеспечивает *выборочное* применение **F-**функции к указанным вторым фактическим аргументом ***applyop-***процедуры *операндам* **W-**выражения с возможностью *передачи* ей фактических **F-***аргументов*, определяемых *необязательным* четвертым аргументом. При указании в качестве второго фактического аргумента целочисленного **p-**выражения имеет место соотношение**:** ***applyop*(F, p, W) = *subsop*(p=F(*op*(p,W)), W),** которое сохраняет силу и в случае списка *целочисленных* выражений в качестве второго фактического аргумента ***applyop-***поцедуры, позволяя производить *выборочную* **F-***обработку* подвыражений **W-**выражения. В случае указания в качестве *второго* фактического аргумента ***applyop-***процедуры множества *целочисленных* выражений **F-***обработке* одновременно подвергаются соответствующие элементам множества *операнды* **W-**выражения. Необязательный четвертый аргумент ***applyop-***процедуры позволяет передавать **F-**функции дополнительные фактические аргументы в порядке их кодирования. Данная функция позволяет выборочно обрабытывать операнды выражений.

С детализирующими замечаниями по всем представленным выше средствам подстановок можно ознакомиться в наших книгах [9-14]. Следующий *сводный* фрагмент иллюстрирует примеры применения рассмотренных средств обеспечения *подстановок* различных типов**:**

> subs({ab=d, ac=h, cd=g}, [ac, ab+cd, d*ab, h+cd]); ⇒ [h, d + g, d^2, h + g]

> [subs(x=0, exp(x) + cos(x)), eval(subs(x=0, exp(x) + cos(x)))]; ⇒ [e^0 + cos(0), 2]

> subsop(1=GS, 3=Art, 4=99, z^2 + 64*x + 59*y + 99); ⇒ *GS* + 64 x + *Art* + 99

> subsop(1=G, 2=NULL, 3=[a, b, c, d, e, g], [x, y, z]); ⇒ [*G*, [a, b, c, d, e, g]]

> subsop(0=G, 1=g(x, y), 2=exp(x), 3=s(t), H(x, y, z)); ⇒ G(g(x,y), e^x, s(t))

> subsop(0=G, [2, 0]=W, [3, 1]=t, H(x, g(x, y, z), x)); ⇒ G(x, W(x, y, z), t)

> subs(y^2 + 64*y + 59 = G(x), sin(y^2 + 64*y + 59) - 10); ⇒ sin(G(x)) - 10

> asubs(x^3+y+17 = W-17*x, (x^3+x^2+3*x+17+y-64*y^2)^2 + 10*x + 99, 'always');

$$(\mathrm{W} - 14\,x + x^2 - 64\,y^2)^2 + 10\,x + 99$$

> [asubs(x^3+x=0, exp(x^3+x)), eval(asubs(x^3+x=0, exp(x^3 + x)))]; ⇒ [1, 1]

> [subs(x^3=h+x^2, x^5+x^2), algsubs(x^3=h+x^2, x^5+x^2)]; ⇒ [$x^5 + x^2$, x^4 + (h + 1) x^2]

> [subs(a+b=c, 10+a+b+3*c), algsubs(a+b=c, 10+a+b+3*c)]; ⇒ [10 + a + b + 3 c, 10 + 4 c]

> [subs(a*x*y = b, 10*a*x*y^2 + 3*b*x*y), algsubs(a*x*y = b, 10*a*x*y^2 + 3*b*x*y)];

$$[10\,a\,x\,y^2 + 3\,b\,x\,y,\ 10\,y\,b + 3\,b\,x\,y]$$

> [subs(a*b/c=d, 3*a^2*b^2/c + sqrt(a*b*d/c)), algsubs(a*b/c=d, 3*a^2*b^2 + sqrt(a*b*d/c))];

$$\left[\frac{3\,a^2\,b^2}{c} + \sqrt{\frac{a\,b\,d}{c}},\ 3\,a^2\,b^2 + \sqrt{d^2}\right]$$

> subs(x^3=Pi, exp(10 – Pi + x^3 + x^5 - x^6)), algsubs(x^3=Pi, exp(10–Pi + x^3+x^5-x^6));

$$e^{(10 + x^5 - x^6)},\ e^{(\pi\,x^2 + 10 - \pi^2)}$$

> [algsubs(x^2 + 3*x = 64, (x + 2)^2 + 59), algsubs(x^2 + 3*x = 64, expand((x + 2)^2 + 59))];

$$[(x + 2)^2 + 59,\ x + 127]$$

> algsubs(x*y^2=G, x^2*y^4 + x^3*y^2 + y^4*x), algsubs(x*y^2=G, x^2*y^4 + x^3*y^6, x);

$$G\,x^2 + y^2\,G + G^2,\ G^3 + G^2$$

> algsubs(x^2 + 10*y=S, 3*x^2 + 17*y), algsubs(x^2 + 10*y=S, 3*x^2 + 17*y, 'exact');

$$-13\,y + 3\,S,\ 3\,x^2 + 17\,y$$

> G:= s^3 + 2*s^2*h - 3*s^2/h + 3*s*h^2 - 6*s + 3*s/h^2 + h^3 - 3*h + 3/h - 9/h^3:

> algsubs(s^2=1, G), algsubs(s^2=1, G, [h, s]), algsubs(1/h=h, G, [s, h]), algsubs(s*h=1, G);

$$\frac{(3\,h^4 - 5\,h^2 + 3)\,s}{h^2} + \frac{-h^4 - 9 + h^6}{h^3},\ h^3 + 3\,s\,h^2 - h - 5\,s + \frac{3\,s}{h^2} - \frac{9}{h^3},$$

$$s^3 - s^2\,h - 6\,s + 6\,s\,h^2 - 8\,h^3,\ s^3 - \frac{3\,s^2}{h} - 4\,s + \frac{3\,s}{h^2} - \frac{9}{h^3} + \frac{3}{h} + h^3$$

> SV:=(a + b - 3)*x/(a + b) + a*y/(a + b) - b*(a + b)/z; ⇒

$$SV := \frac{(a + b - 3)\,x}{a + b} + \frac{a\,y}{a + b} - \frac{b\,(a + b)}{z}$$

> simplify([algsubs(a + b = 1/h^2, SV), algsubs(a + b = 1/h^2, SV, [a, b], 'exact')]);

$$\left[-\frac{-x\,z\,h^2 + 3\,x\,z\,h^4 + y\,h^4\,z\,b - y\,h^2\,z + b}{z\,h^2},\ -\frac{-x\,z\,h^2 + 3\,x\,z\,h^4 - y\,a\,h^4\,z + b}{z\,h^2}\right]$$

> restart: applyop(G, {[1, 2], [3, 2]}, x^y + x + y^h, z); ⇒ $x^{G(y,\,z)} + x + y^{G(h,\,z)}$

> applyop(G, 3, x + y + z, t, h) = subsop(3 = G(op(3, x + y + z), t, h), x + y + z);

> applyop(evalf, {[1, 2], [3, 2]}, x^sin(59) + R + y^ln(17), 2); ⇒ $x^{0.64} + R + y^{2.8}$

> n, h:= 1, 2: applyop(Art, [n + 1, h^n], H(x) + G(x, y), t); ⇒ H(x) + G(x, Art(y, t))

С учетом сказанного особых пояснений примеры фрагмента не требуют. Вместе с тем, наряду с рассмотренными функциями поддержки подстановок в целом ряде случаев *эффективно* использовать *две* ранее рассмотренные функции ***numboccur*(V,h)** и ***has*(V,h),** возвращающие число вхождений и сам факт вхождения {*true*|*false*} **h-**подвыражения в **V-**выражение соответственно. В качестве примера ниже приводится ***PSubs-***процедура,

существенно использующая первую из указанных функций ***Maple-***языка. Слелующий фрагмент представляет исходный текст процедуры ***PSubs*** и примеры ее применения.

```
PSubs := proc()
local k, h, L, W;
    `if`(nargs = 0, RETURN( ), assign(W = args[nargs], L = 63));
    for k to nargs – 1 do L := L, `if`(type(args[k], 'equation'), args[k], `if`(
        type(args[k], 'set'('equation')) or type(args[k], 'list'('equation')),
        op(args[k]), RETURN("Substitution rules are incorrect")))
    end do;
    `if`(nops([L]) – 1 = nargs, RETURN(WARNING(
        "Substitution rules %1 are not applicable to absent expression", [args])),
        NULL);
    for k from 2 to nops([L]) do
        h := numboccur(W, lhs(L[k]));
        `if`(h = 0, print(cat(`Rule (`, convert(L[k], 'symbol'), `) is inactive`)), [print(
            cat(`Rule (`, convert(L[k], 'symbol'), `) was applied `,
            convert(h, 'symbol'), ` times`)), assign('W' = subs(L[k], W))])
    end do;
    W
end proc
```

> PSubs(x=a, {y=b, z=c}, [t=h, k=v], (Kr(x^2+y^2+z^2))/(t+k + sqrt(x*y-t*k))-Art(x+z+t+k));

Ryle (x = a) *was applied 3 times*
Ryle (y = b) *was applied 2 times*
Ryle (z = c) *was applied 2 times*
Ryle (t = h) *was applied 3 times*
Ryle (k = v) *was applied 3 times*

$$\frac{\mathrm{Kr}(a^2+b^2+c^2)}{h+v+\sqrt{a\,b-h\,v}}-\mathrm{Art}(a+c+h+v)$$

> PSubs(h=a*b*c, Kr(x+y+z)*Art(x+z)+VSV(y+z)); $\Rightarrow$ Kr(x + y + z) Art(x + z) + VSV(y + z)

Ryle (h = a b c) *is no active*

В качестве первого аргумента ***PSubs-***процедуры выступает последовательность, элементами которой могут быть отдельные уравнения (*правила подстановок*) либо их множества/списки. Вторым аргументом процедуры выступает собственно само обрабатываемое выражение. Процедура возвращает результат последовательного *применения* заданных первым фактическим аргументом правил в выражение, заданное ее вторым фактическим аргументом. Одновременно выводится информация по применению каждого из правил подстановки. Читателю рекомендуется разобраться в организации процедуры. С целью лучшего усвоения *принципов* и *особенностей* выполнения вышерассмотренных средств, имеющих важные приложения, читателю рекомендуется провести с ними определенную наработку. В частности, в качестве *весьма* полезного упражнения читателю рекомендуется в терминах алгебраических правил подстановок запрограммировать в среде ***Maple-***языка хорошо известные абстрактные модели *вычислителей*, например, машину Тьюринга (*последовательная модель вычислений*) и классические однородные структуры (*параллельная модель вычислений*) [1-3,36,40,92-96,98,100-102].

Учитывая важность различного рода подстановок при работе с *символьными* и *строчными* выражениями – одними из основных составляющих символьных вычислений и об-

работки – нами был определен целый ряд средств данного типа, ориентированных на различные случаи приложений [41,103,108,109]. В частности, процедура ***Sub_st*(E, S, R {, `*insensitive*`})** возвращает результат подстановки правых частей уравнений, определенных первым фактическим аргументом **E,** в строку или символ, указанный вторым аргументом **S,** вместо *всех* вхождений в нее левых частей уравнений **E.** При этом, обработке подвергаются и *пересекающиеся* вхождения левых частей уравнений подстановок. Более того, тип возвращаемого результата соответствует типу второго аргумента **S.**

```
Sub_st := proc(E::list(equation), S::{string, symbol}, R::evaln)
local k, h, G, v, ω, p;
    assign(p = {args},
        ω = ((s, v) → `if`(member(insensitive, v), Case(s, 'lower'), s)));
    `if`(nops(E) = 0, WARNING("Substitutions system <%1> is absent", E), assign(
        G = cat("", S), R = true, v = array(1 .. nops(E), [0 $ (k = 1 .. nops(E))])));
    `if`(Search2(ω(S, p), {seq(ω(lhs(E[k]), p), k = 1 .. nops(E))}) = [ ],
        RETURN(assign('R' = false), S,
        WARNING("Substitutions system %1 is not applicable to <%2>", E, S)),
        NULL);
    for k to nops(E) do
        `if`(ω(lhs(E[k]), p) = ω(rhs(E[k]), p), [assign('R' = G), RETURN(S,
            WARNING("Substitution <%1> generates an infinite process", E[k]))]
            , assign('v[k]' = 0));
        while search(ω(G, p), ω(lhs(E[k]), p)) do
            assign('h' = searchtext(ω(lhs(E[k]), p), ω(G, p)), 'v[k]' = v[k] + 1);
            G := cat(G[1 .. h − 1], rhs(E[k]),
                G[h + length(lhs(E[k])) .. length(G)])
        end do
    end do;
    convert(G, whattype(S)), evalm(v)
end proc
> S:="ARANS95IANGIANRANS95RAEIAN99RACREAIANRANSIANR99ANSRANS":
  Sv:=["RANS"="Art", "IAN"=" ", "95"="S", "99"="Kr"]: Sub_st(Sv, S, Z);
                "AArtS G ArtSRAE KrRACREA Art RKrANSArt", [4, 5, 2, 2]
> S:="ARANS95IANGIANRANS95RAEIAN99RACREAIANRANSIANR99ANSRANS":
  Sv:=["RANS"="Art", "IAN"=" ", "95"="S", "99"="Kr"]: Sub_st(Sv, S, Z), Z;
                "AArtS G ArtSRAE KrRACREA Art RKrANSArt", [4, 5, 2, 2], true
```

Результат обработки строки или символа **S** посредством системы подстановок **E** есть только *первый* элемент возвращаемой **2**-элементной последовательности, тогда как в качестве ее второго элемента выступает целочисленный вектор, определяющий количество выполненных применений к **S** соответствующих подстановок из **E**. Если же левые части подстановок из **E** не принадлежат **S,** или по меньшей мере *одна* подстановка инициирует *бесконечный* процесс, то процедура выводит соответствующее информационное предупреждение.

Правила подстановок **E** задаются списком уравнений вида **[Y1 = X1, Y2 = X2, …, Yn = Xn];** где для обеих частей уравнений предполагаются выражения типов {*string*, *symbol*}. При этом, алгоритм применения системы подстановок **E** состоит в следующем: первая подстановка **E** применяется к **S** до ее полного исчерпания в **S,** затем аналогичная операция

проделывается со второй подстановкой из **E,** и так далее до *полного исчерпания* всех подстановок из **E**.

При кодировании четвертого дополнительного аргумента ***insensitive*** процедура ***Sub_st*** поддерживает регистро-независимый, в противном случае выполняется регистро-зависимый поиск *вхождений* левых частей подстановок. Наконец, через третий фактический аргумент **R** возвращается следующее значение: (**1**) *true*, если обработка **S** была завершена *успешно*, (**2**) результат обработки **S** до ситуации обнаружения подстановки, ведущей к *бесконечному* процессу (*циклическая работа*). Выше представлен фрагмент с исходным текстом процедуры ***Sub_st*** и результатами ее применения.

Наряду с целым рядом средств, обеспечивающих разнообразную обработку *символьных* и *строчных* выражений, наша библиотека [41,103] содержит и средства поддержки ряда полезных в практическом отношении *подстановок*. Читатель и сам может запрограммировать требуемые для своих конкретных приложений интересные средства, используемые многократно.

В определенной мере к средствам, обеспечивающим подстановки в выражения, примыкает и специальное **use**-предложение языка, имеющее формат кодирования:

use <ПВ> **in** <ПП> **end use**

где ПВ – последовательность выражений типа {**`=`,** *equation*} и ПП – последовательность предложений ***Maple*** (*тело* **use**-*предложения*). ПВ определяет *последовательность связывающих форм*. В простейшем виде *связывающая* форма представляет собой *уравнение* (*точнее*, *правило подстановки*), чья левая сторона представляет *имя*, тогда как правой стороной уравнения может быть любое выражение языка, но не их последовательность. Другие виды *связывающих форм* определяются в терминах *эквациональных* связывающих форм. В качестве связывающих форм могут выступать выражения выбора члена модуля; связывающая форма вида **M:- e** эквивалентна эквациональной связывающей форме **e = M:- e**. Наконец, в качестве связывающей формы может выступать модульное выражение либо его имя. Использование модуля **M** в качестве связывающей формы эквивалентно определению уравнения **e = M:- e** для всех экспортов **e** модуля **M**. Предложение **use** отличается от всех других предложений ***Maple***-языка тем, что оно разрешается в течение автоматического упрощения, а не в процессе вычислений. Предложение **use** не может быть вычислено.

Предложение **use** вызывает синтаксическое преобразование его тела согласно подстановкам, указанным в последовательности *связывающих* форм. Однако, оно в отличие от простых подстановок производит их в соответствии со статическими правилами просмотра ***Maple***-языка. Каждое **use**-предложение вводит новый контур связывания, в пределах которого в течение упрощения *имена* левых частей каждого из уравнений связывания заменяются соответствую-ими *выражениями* правых частей уравнений. Тело **use**-предложения *"перезаписывается"***,** выполняя указанные замены.

Каждое вхождение связующего *имени* (*левая часть*) заменяется соответствующим ему *выражением* (*правая часть*) всюду по телу **use**-предложения. Когда тело **use**-предложения вычисляется, значение связующего *имени* вычисляется *один* раз для каждого его вхождения. Это означает, что в то время как само **use**-предложение не налагает никаких ограничений в процессе вычисления, следует проявлять *внимательность* в случае, когда *правые* части уравнений связывания могут выполнять важные вычисления. Правые части *связывающих* форм не вычисляются при обработке **use**-предложения. Детальнее с **use**-предложением можно ознакомиться по конструкции **?use,** здесь же мы представим примеры, иллюстрирующие применение **use**-предложения в различных ситуациях:

```
> use a = 64, b = 59, c = 39, d = 17 in (a + b)/(c+d) end use;   ⇒ 123/56
> P:= proc(n::posint) add(a+k, k=1..n) end proc: P(17);   ⇒ 153 + 17 a
> use a = 64  in  proc(n::posint) add(a + k, k=1..n) end proc  end use;
                        proc(n::posint) add(64 + k, k = 1 .. n) end proc
> %(17);   ⇒ 1241
> use b = 64 in proc(n::posint) local b; add(b+k, k=1..n) end proc end use;
                    proc(n::posint) local b; add(b + k, k = 1 .. n) end proc
> use add = seq in proc(n::posint) local b; add(b+k, k=1..n) end proc end use;
                    proc(n::posint) local b; :-seq(b + k, k = 1 .. n) end proc
> %(10);   ⇒ b + 1, b + 2, b + 3, b + 4, b + 5, b + 6, b + 7, b + 8, b + 9, b + 10
> use `+` = `*` in proc(n::posint) local b; add(k, k=1..n); b:=proc() global c; c*`+`(args)
  end proc  end proc end use;
  proc (n::posint) local b; add(k,k = 1 .. n); b := proc () global c; c*`+`(args) end proc end proc
> use  `+` = ((a, b) -> a * b)  in 64 + 56 end use;   ⇒ 3584
> use a = b in module() export a; a:=() -> add(args[k], k=1..nargs) end module end use;
                                module() export a;  end module
> %:- a(64, 59, 39, 10, 17);   ⇒ 189
> use `and` = `or` in proc() if nargs > 6 and args[1] = 64 then 2006 end if end proc  end use;
                proc()  if :-`or`(6 < nargs, args[1] = 64) then 2006 end if end proc
> %(64);   ⇒ 2006
```

Из примеров фрагмента следует, что **use**-предложение в *процедурах* и *модулях* игнорирует замены *локальных*, *глобальных* и *экспортируемых* переменных, возвращая свое тело лишь упрощенным без выполнения тех замен, левые части которых определяют имена указанного типа переменных. Более того, **use**-предложение не производит прямых замен, в частности, *бинарных* инфиксных операторов (*например,* **`+`** *и* **`*`**) или *унарных* префиксных или постфиксных операторов, как это иллюстрирует **8**-й пример фрагмента. Тогда как такие операторы можно заменять, если правые части связывающей формы определяют процедуры или модули, как это иллюстрирует **9**-й пример фрагмента. По **use**-предложению можно заменять такие операторы как: **`+`, `*`, `-`, `/`, `^`, `!`, `and`, `or`, `not`, `=`, `<>`, `<`, `<=`.**

```
Use := proc(P::anything)
    if nargs = 1 then P
    elif type(P, `module`) then
        WARNING("1st
                argument is a module; apply the `use`-clause");
        return P
    else
        try eval(parse(SUB_S([seq(`if`(
            type(args[k], 'equation') and type(lhs(args[k]), 'symbol'),
            lhs(args[k]) = convert(rhs(args[k]), 'string'), NULL), k = 2 .. nargs)],
            convert(eval(P), 'string'))))
        catch : P
        end try
    end if
end proc
> Use(proc(n::posint) local b; add(k, k=1..n); b:= proc() global c; c*`+`(args) end proc
  end proc, `+` = `*`);
```

```
  proc(n::posint) local b;  add(k, k = 1 .. n); b := proc() global c; c*`*`(args) end proc end proc
> Use(proc(n::posint) local b; add(k, k=1..n); b:=proc() global c; c*`+`(args) end proc
  end proc, c = d);
  proc(n::posint) local b; add(k, k = 1 .. n); b := proc() global c; c*`+`(args) end proc end proc
> Use(proc(n::posint) local b; add(k, k=1..n); b:=proc() global c; c*`+`(args) end proc
  end proc,  a = c);
  proc(n::posint) local b; add(k, k = 1 .. n); b := proc() global c; c*`+`(args) end proc end proc
> Use(proc(n::posint) local b; add(k, k=1..n); b:=proc() global c; c*`+`(args) end proc
  end proc,  k = j);
  proc(n::posint) local b;  add(j, j = 1 .. n);  b := proc() global c; c*`+`(args) end proc end proc
> Use((a+b)/d, d = 0);
```

$$\frac{a+b}{d}$$

```
> use d = 0 in (a+b)/d end use;
Error, numeric exception: division by zero
> Use(proc(n::posint) local b; add(b+k, k=1..n) end proc, b = 64);
                        proc(n::posint) local b; add(b + k, k = 1 .. n) end proc
> Use(module() export a; a:=() -> add(args[k], k=1..nargs) end module, a=b);
Warning, 1st argument is a module; apply the `use`-clause
                                    module() export a;  end module
> %:- a(64, 59, 39, 10, 17);   ⇒ 189
> Use(proc() if nargs > 6 and args[1] = 64 then 2006 end if end proc, `and` = `or`);
                    proc() if 6 < nargs or args[1] = 64 then 2006 end if end proc
> Use(proc() Digits:= 40; a*1/10000000000000000000001.0, (64 + Digits^a) end proc, a = 2);
                  proc () Digits := 40; 0.2000000000*10^(-22), 64 + Digits^2 end proc
> %();   ⇒ 0.2000000000 10 -22, 1664
```

Представленная выше процедура ***Use*(P, x=x1, y=y1, …)** в ряде случаев оказывается неплохим дополнением **use**-предложению, обеспечивая подстановки правых частей уравнений, определяемых фактическими аргументами, начиная со второго, вместо вхождений в выражение **P** соответствующих им левых частей. При этом, в качестве первого аргумента **P** может выступать любое выражение, кроме *модулей*; в противном случае вызов процедуры возвращает исходное выражение с выводом соответствующего предупреждения. На остальных выражениях **P** вызов процедуры ***Use*(P, x = x1, y = y1,…)** во многом *подобен* результатам применения **use**-предложения, тогда как имеется и целый ряд особенностей. Например, **Use**-процедура позволяет делать *прямые* замены вышеуказанных операторов, обрабатывает особые и ошибочные *ситуации* и др. Предыдущий фрагмент представляет исходный текст процедуры ***Use*** и примеры ее применения. При этом, если необходимо произвести обработку ***Use*-процедурой** последовательности выражений или предложений, то их можно *обрамлять процедурой*, как иллюстрирует *последний* пример фрагмента. В ряде приложений ***Use*-процедура** *предпочтительнее* **use**-предложения.

Наша *Библиотека* [41,103,109] представляет целый ряд достаточно полезных средств для обеспечения различного рода как *символьных*, так и *алгебраических* подстановок, ряд из которых довольно неплохо дополняют стандартные средства подобного типа. Многие из них используются и другими средствами *Библиотеки*, в ряде случаев существенно *упрощая* программирование и делая реализуемые ими алгоритмы более прозрачными.

5.4. *Средства Maple-языка для обработки алгебраических выражений*

Выросший из системы *символьных* (*алгебраических*) вычислений, пакет ***Maple*** располагает достаточно развитыми средствами символьных вычислений и различного рода математических преобразований, что делает его *одним* из наиболее мощных **ПС** данного типа. Такого рода средства позволяют не только получать решения в строгом алгебраическом виде, но и производить различного рода математические преобразования, важные при решении многих качественных вопросов из различных приложений. В этом плане они могут оказаться весьма полезным инструментом при исследовании целого ряда вопросов и в чистой математике. Прежде всего остановимся на группе средств, позволяющих производить различного рода преобразования выражений из одной формы в другую, упрощающие их для последующей алгебраической обработки. Данные средства важны и в практическом программировании.

Упрощение выражений. Одной из важнейших задач при работе с *алгебраическими* выражениями является *упрощение* как конечных, так и основных промежуточных результатов символь-ых вычислений. Для этих целей ***Maple***-язык располагает специальной процедурой ***simplify***, имеющей следующий формат кодирования:

simplify(<*Выражение*> {, <*Тип упрощения*>} {, ***assume*** = <*Свойство*>})

и производящей упрощение заданного своим *первым* фактическим аргументом *выражения* путем применения к нему специальных процедур упрощения. Если процедура содержит единственный аргумент-*выражение*, то производится анализ *выражения* на предмет вхождения в него вызовов функций, квадратных корней, радикалов и степеней.

После этого к *выражению* применяются подходящие упрощающие процедуры, включающие функции: Бесселя, Гамма, Ламберта, e^x, ***ln***, $\sqrt{x}$, тригонометрические, гиперболические и др., т.е. производится возможное *упрощение* исходного *выражения* по *всему* спектру возможностей функции. Третий необязательный фактический аргумент функции позволяет приписывать всем переменным упрощаемого *выражения* определенные *свойства*, которые будут приписаны переменным *упрощенного* выражения, либо определять *параметры*, управляющие применением правил упрощения. *Второй* аргумент функции также является *необязательным* и определяет принцип упрощения исходного *выражения*, определяемого первым аргументом, согласно специальным типам *упрощающих* правил. В качестве второго аргумента ***simplify***-процедуры могут выступать отдельный идентификатор, список или множество идентификаторов, определяющих специальные типы упрощения исходного *выражения*. В качестве таких идентификаторов допускаются следующие, определяемые в табл. **11:**

Таблица 11

Тип	***Производится упрощение выражения, содержащего:***
`@`	*операторы*; как правило при работе с *обратными* функциями
Ei	*экспоненциальные* интегралы; ***Ei*(1, x*I) → -*Ci*(x) + *Sin*(x)*I - π*I/2**
GAMMA	***GAMMA***-функции
hypergeom	*гипергеометрические* функции; представление в виде степенного ряда
ln	*логарифмические* функции
piecewise	*кусочно-определенные* функции; например: ***abs, sign, Signum*** и др.
polar	*комплексные* выражения, представленные в полярных координатах
power	*степени*, *экспоненты* и *логарифмы*; допускается ***symbolic***-параметр

radical	*радикальные* конструкции различного типа
RootOf	***RootOf*-функцию**; упрощаются полиномы от нее и их обращения
sqrt	*корни квадратные* и/или их степени**;** допускается ***symbolic*-**параметр
siderel	комбинации *упрощающих* уравнений из заданного {*списка*|*множества*}
trig	*тригонометрические* и/или *гиперболические* функции
wronskian	подвыражения вида **xy' - yx',** где **x, y -** специальные функции

Смысл и назначение *каждого* из представленных в табл. **11** значений для второго фактического аргумента ***simplify*-**процедуры достаточно прозрачны и проиллюстрированы в нижеследующем фрагменте, однако по отдельным необходимы пояснения.

```
> [x@exp@ln, simplify(x@exp@ln, `@`)];   ⇒ [x@exp@ln, x]
> simplify(GAMMA(p+1)*(p^2+3*p+2)*x + GAMMA(p+3), GAMMA);   ⇒ (1 + x) Γ(p + 3)
> simplify(ln(95), ln), simplify(ln(64*x+42*r), ln);
```

$$\ln(5) + \ln(19),\ \ln(2) + \ln(32\,x + 21\,r)$$

```
> simplify(64*signum(x) + 59*abs(x) - 39*sign(x), 'piecewise');
```

$$\begin{cases} -59\,x - 103 & x < 0 \\ -39 & x = 0 \\ 59\,x + 25 & 0 < x \end{cases}$$

```
> map(simplify, [x*3**y*10, (x**y)^(x**z), exp(3*ln(x^3) + 10**x)], 'power');
```

$$[\,10\,x\,3^{y},\ (x^{y})^{(x^{z})},\ x^{9}\,\mathbf{e}^{(10^{x})}\,]$$

```
> G:= [64^(1/4), 180^(2/3), (-1999)^(1/3), sqrt(191)]:  simplify(G, 'radical');
```

$$\left[\,2\sqrt{2},\ 180^{(2/3)},\ \frac{(1+\sqrt{3}\,I)\,1999^{(1/3)}}{2},\ \sqrt{191}\,\right]$$

```
> simplify((x^6 + x^5*y - 2*x^4*y^2 - 2*x^3*y^3 + x^2*y^4 + x*y^5)^(1/3));
```

$$(x\,(x-y)^{2}\,(x+y)^{3})^{(1/3)}$$

```
> s:= RootOf(t^2 – 13 = 0, t):  [3*s^2 – s + 10, simplify(3*s^2 – s + 10, 'RootOf')];
```

$$[\,3\,\mathrm{RootOf}(_Z^{2}-13)^{2} - \mathrm{RootOf}(_Z^{2}-13) + 10,\ 49 - \mathrm{RootOf}(_Z^{2}-13)\,]$$

```
> h:= (64*s - 59)/(s^2 + 10*s + 17):  [h, simplify(h, 'RootOf')];
```

$$\left[\,\frac{64\,\mathrm{RootOf}(_Z^{2}-13) - 59}{\mathrm{RootOf}(_Z^{2}-13)^{2} + 10\,\mathrm{RootOf}(_Z^{2}-13) + 17},\ -\frac{251}{40}\,\mathrm{RootOf}(_Z^{2}-13) + \frac{1009}{40}\,\right]$$

```
> p:= (12*v^2 - 191*v + 243)^(1/2):  [p, simplify(p, 'sqrt', 'symbolic')];
```

$$[\,\sqrt{12\,v^{2} - 191\,v + 243}\,,\ \sqrt{12\,v^{2} - 191\,v + 243}\,]$$

```
> [(64/(x - 10)^2)^(1/2), simplify((64/(x - 10)^2)^(1/2), 'sqrt', 'symbolic')];
```

$$\left[\,\sqrt{64}\,\sqrt{\frac{1}{(x-10)^{2}}},\ \frac{8}{x-10}\,\right]$$

```
> [sin(x)^2 - cos(x)^2, simplify(sin(x)^2 - cos(x)^2, 'trig')];
```

$$[\,\sin(x)^{2} - \cos(x)^{2},\ -2\cos(x)^{2} + 1\,]$$

```
> [sinh(x)^2*cosh(x) - cosh(x)^3, sin(x)*tan(x) + sec(x)+cos(x)];
```

$$[\,\sinh(x)^{2}\cosh(x) - \cosh(x)^{3},\ \sin(x)\tan(x) + \sec(x) + \cos(x)\,]$$

```
> map(simplify, %, 'trig');
```

$$\left[\,-\cosh(x),\ \frac{2}{\cos(x)}\,\right]$$

```
> simplify(57*x^4-52*x^3+32*x^2-10*x+3, {13*x^2+3*x+10 = 0});
```

$$\frac{86523\,x}{2197} + \frac{1201}{2197}$$

```
> SV:= 6*z*y^2 + 6*z^4*y^3 - 18*z^2*y + 112 + 53*z^2 + 14*z*y - 306/5*y^2 + 3*x*z - 3*x*y +
  18*z^4 + 8*z^3*y - 4*z^2*y^2 - 8*z*y^3 + 4*y^4:
> simplify(SV, {x*z - y^2=x, x*y^2 + z^2=z, (z+y)^2=x, (x+y)^2=z}, {x,y,z});   ⇒ 112 + 74 x
> Sideral:= {sin(x)^2 + cos(x)^2 = 1, 2*sin(x)*cos(x) = 1 - (cos(x) - sin(x))}:
> Expression:= sin(x)^3 - 3*sin(x)^2*cos(x) - cos(x)^3 - sin(x)*cos(x) + sin(x)^2 + 2*cos(x):
> H1:= BesselJ:  H2:= BesselY:  simplify(Expression, 'Sideral');   ⇒ cos(x)^3
> simplify(Pi*y*(H1(x + 1, y)*H2(x, y) - H1(x,y)*H2(x + 1, y)), 'wronskian');   ⇒ 2
> simplify(cos(x)*diff(sin(x), x) - sin(x)*diff(cos(x), x), 'wronskian');   ⇒ sin(x)^2 + cos(x)^2
```

За более детальным описанием и особенностями применения ***simplify****-процедуры* можно обращаться к справке по пакету либо к нашей книге [12], принимая во внимание, что и релиз накладывает свои особенности. Вместе с тем, представленной информации вполне достаточно для проведения многих важных *упрощающих* процедур над довольно широким классом *алгебраических* и *числовых* ***Maple***-выражений различного типа.

К задаче упрощения *алгебраических* **W**-выражений непосредственно относится и процедура ***fnormal*(W {, D | , D, ε})**, возвращающая для алгебраического выражения, определяемого первым фактическим **W**-аргументом, *эквивалентное* ему **W***-выражение в том отношении, что *все* входящие в него числовые значения *float*-типа, меньшие **ε**-величины, определяемой третьим необязательным аргументом функции, полагаются нулевыми. При этом, все *float*-значения **W**-выражения вычисляются с **D**-точностью, определяемой вторым необязательным фактическим аргументом процедуры. По умолчанию для аргументов **D** и **ε** процедуры полагаются соответственно значения *глобальной* **Digits**-переменной и **Float(1, -D + 2)**-значение. В качестве первого фактического **W**-аргумента процедуры могут выступать как отдельное алгебраическое выражение, так и список, множество, отношение, ряд или диапазон такого типа выражений. Следующий весьма простой фрагмент иллюстрирует применение ***fnormal***-процедуры**:**

```
> map(fnormal, [.0000000001*A + B, A/10^9 + B, A/10^9 + .0000000001*B]);
```

$$\left[B, \frac{A}{1000000000} + B, \frac{A}{1000000000}\right]$$

```
> map(fnormal, evalf([A/10^9 + B, A/10^9 + B, A/10^9 + .0000000001*B]));   ⇒ [B, B, 0.]
```

Применение ***fnormal***-процедуры имеет смысл только для *алгебраических* **W**-выражений**,** содержащих числовые значения *float*-типа. Поэтому в общем случае может потребоваться предварительное применение к **V**-выражению ***evalf***-функции, как это иллюстрирует предыдущий фрагмент.

По процедуре ***radsimp*(V** {**,** ***ratdenom***}**)** производится *упрощение* **V**-выражения**,** содержащего радикалы**;** кодирование необязательного второго аргумента функции определяет необходимость избавления знаменателя выражения от радикалов, например**:**

```
> [radsimp(3^(1/2) + 2/(2 + sqrt(3))), radsimp(3^(1/2) + 2/(2 + sqrt(3)), 'ratdenom')];
```

$$\left[\frac{2\sqrt{3}+5}{2+\sqrt{3}}, -(2\sqrt{3}+5)(-2+\sqrt{3})\right]$$

```
> [radsimp(sqrt(x) + a/(b + sqrt(x))), radsimp(sqrt(x) + a/(b + sqrt(x)), 'ratdenom')];
```

$$\left[\frac{\sqrt{x}\,b+x+a}{b+\sqrt{x}}, -\frac{(\sqrt{x}\,b+x+a)(b-\sqrt{x})}{-b^2+x}\right]$$

В определенной мере к ***simplify***-процедуре примыкает и функция ***convert*(W, <*Форма*>)**, обеспечивающая конвертацию **W**-выражения из одной в другую *форму*, определяемую вторым аргументом функции. Некоторые ее дополнительные возможности по конвертации выражений будут рассмотрены ниже. Функция ***convert*** может использоваться в

сочетании как с рассмотренной ***simplify*-**процедурой, так и с рассматриваемыми ниже другими средствами ***Maple*-**языка для упрощения различного рода выражений.

По функции с форматом кодирования ***expand*(**<*Выражение*> {**, V1,V2,...,Vn**}**)** производится *раскрытие* указанного ее *первым* фактическим аргументом *выражения* на суммы и термы некоторых других типов**.** Первой попыткой функции является раскрытие произведений в суммы**,** что характерно**,** в первую очередь**,** для *полиномов***;** для полиномиальных дробей такому раскрытию подвергаются только их числители**.** Для многих математических функций (***sin, cos, tan, sinh, cosh, tanh, det, erf, exp, factorial, GAMMA, ln, int, max, min, Psi, binomial, sum, product, limit, bernoulli, euler, signum, abs,*** *кусочно-определенных и др.*) ***expand*-**функция обеспечивает стандартные формы *раскрытия*. При необходимости запрета на раскрытие каких-либо **Vk-***подвыражений* раскрываемого по ***expand*-**функции *выражения* следует кодировать их в качестве ее необязательного *второго* аргумента **–** *последовательности подвыражений***.** При необходимости запрета на *раскрытие* всех входящих в раскрываемое *выражение* функций используется ***frontend*-**функция в конструкции вида ***frontend*(*expand*, [***Выражение***]).** Следующий фрагмент иллюстрирует вышесказанное**:**

```
> expand(cos(x+y+z) - sin(x+y+z)), expand((n^3 + 52*n^2 - 57*n + 10)*GAMMA(n + 3));
```

$$\cos(x)\cos(y)\cos(z) - \cos(x)\sin(y)\sin(z) - \sin(x)\sin(y)\cos(z) - \sin(x)\cos(y)\sin(z) - \sin(x)\cos(y)\cos(z) + \sin(x)\sin(y)\sin(z) - \cos(x)\sin(y)\cos(z) - \cos(x)\cos(y)\sin(z),$$

$$n^6\,\Gamma(n) + 55\,n^5\,\Gamma(n) + 101\,n^4\,\Gamma(n) - 57\,\Gamma(n)\,n^3 - 84\,\Gamma(n)\,n^2 + 20\,\Gamma(n)\,n$$

```
> expand((x + y)^3 - 3*(x - y)^2 + 10*(x+y));
```

$\Rightarrow\ x^3+3\,x^2y+3xy^2\,y^3 - 3x^2 + 6xy - 3y^2 + 10x + 10y$

```
> expand((z + 2)^2*((x + y)^2 - 3*(x - y)^2 + 10*(x + y)), x + y, x - y);
```

$$z^2\,(x+y)^2 - 3\,z^2\,(x-y)^2 + 10\,z^2\,x + 10\,z^2\,y + 4\,z\,(x+y)^2 - 12\,z\,(x-y)^2 + 40\,z\,x + 40\,z\,y + 4\,(x+y)^2 - 12\,(x-y)^2 + 40\,x + 40\,y$$

```
> expand(cos(x + y)*(x + 2)^2), frontend(expand, [(cos(x + y)*(x + 2)^3)]);
```

$$\cos(x)\cos(y)\,x^2 + 4\cos(x)\cos(y)\,x + 4\cos(x)\cos(y) - \sin(x)\sin(y)\,x^2 - 4\sin(x)\sin(y)\,x - 4\sin(x)\sin(y),$$

$$\cos(x+y)\,x^3 + 6\cos(x+y)\,x^2 + 12\cos(x+y)\,x + 8\cos(x+y)$$

Приведенный фрагмент достаточно прозрачен и особых пояснений не требует. Следует отметить**,** что ***simplify*-**процедура является наиболее важным *упрощающим* средством только на первый взгляд. В процессе проводимого ею упрощения не всегда достигается желаемый результат. Более того, в ряде случаев *упрощенное* выражение оказывается даже менее приемлемым для последующей обработки**,** чем исходное**.** В этом плане более предпочтительной оказывается именно ***expand*-**функция, раскрывающая выражение на *термы*, совокупность которых значительно более удобна для последующих упрощений и преобразований.

Для обеспечения дифференцированного управления режимом раскрытия выражений предназначены две функции ***expandoff*** и ***expandon*****,** имеющие единый формат кодирования, а именно**:**

***expand*{*off* | *on*}({*Id*_**1, *Id*_2, ..., *Id*_n**})**

возвращающие *NULL*-значения и позволяющие соответственно {*запрещать* | *разрешать*} *раскрытие* функций, указанных последовательностью их идентификаторов (***Id*_k**) в качестве фактического аргумента функции. В случае же вызова функции ***expand*{*off* | *on*}()** описанное действие по санкционированию *раскрытия* функций ***expand*-**функцией распространяется на *все* функции текущего сеанса, исключая пользовательские функции. Следующий прозрачный фрагмент иллюстрирует применение указанных функций**:**

```
> P:= x -> (x + 10)^3 - sin(3*x): expand(P(y));
```
$$y^3 + 30\,y^2 + 300\,y + 1000 - 4\sin(y)\cos(y)^2 + \sin(y)$$
```
> expandoff(sin): expand(P(z));  ⇒ z^3 + 30 z^2 + 300 z + 1000 - sin(3 z)
> expand(sin(3*x)), expand(expandon()), expandon(), map(expand, [P(h), sin(3*t)]);
```
$$\sin(3\,x),\ \mathrm{expandon}(\ ),$$
$$[h^3 + 30\,h^2 + 300\,h + 1000 - 4\sin(h)\cos(h)^2 + \sin(h),\ 4\sin(t)\cos(t)^2 - \sin(t)]$$
```
> P:= x -> (x + 10)^3 + sin(3*x)*exp(x + y): expandoff(): expand(P(t));
```
$$t^3 + 30\,t^2 + 300\,t + 1000 + 4\,\mathbf{e}^{(t+y)}\sin(t)\cos(t)^2 - \mathbf{e}^{(t+y)}\sin(t)$$

Данный фрагмент иллюстрирует также *remember*-свойство ***expand*-функции**, состоящее в том, что раз возвратив *разложение* выражения, функция будет помнить его, даже если наложен запрет на разложение входящих в него функций пакета**.** В значительной мере это весьма полезное свойство**,** однако оно в определенной мере не согласуется с функцией ***expandoff*,** запрещающей разложение заданных функций ***Maple*-**языка.

При использовании {***expandoff*** | ***expandon***}-*функции* следует иметь в виду, что в текущем сеансе работы с ядром она имеет *однократное* действие (*причины этого лежат в свойстве remember функции* ***expand***) и попытки ее повторного использования или инициирующих ее конструкций приводят к ошибочной ситуации, как показано в следующем фрагменте. Ситуация разрешается, в частности, после выполнения **restart-**предложения.

```
> expand(sin(3*x)); expand(expandoff()): expandoff(sin): [expand(sin(3*x)),
  expand(sin(4*x)), expand(cos(3*x))];
```
$$4\sin(x)\cos(x)^2 - \sin(x)$$
$$[4\sin(x)\cos(x)^2 - \sin(x),\ \sin(4\,x),\ 4\cos(x)^3 - 3\cos(x)]$$
```
> expandoff(cos); expand(expandoff()):
Error, wrong number (or type) of parameters in function expand
> restart: expand(expandoff()): expandoff(cos): expand(cos(3*x));  ⇒ cos(3 x)
```

Поэтому в случае необходимости рекомендуется сразу определять функции, не подлежащие раскрытию по ***expand*-**функции, что устранит ошибочные ситуации.

Наконец, *пассивная* ***Expand*(v)-**функция представляет собой *шаблон* ***expand*-**функции, используемый в *конъюнкции* с ***evala*-**функцией или в **mod-**конструкциях. По конструкции вида ***evala*(*Expand*(v))** производится *раскрытие произведений* в произвольном **v-**выражении, которое может включать алгебраические числа и/или ***RootOf*-**процедуры. Тогда как по конструкции ***Expand*(v) (mod p)** производится *раскрытие произведений* над целыми по **(mod p)** и **v-**выражение может содержать вызовы ***RootOf*-**процедур. Например**:**

```
> Expand((57*x + 3)^2*(x^2 - 2*x + 52)) mod 5;  ⇒ 4 x^4 + 4 x^3 + 3 x^2 + x + 3
> evala(Expand(RootOf(x^2 - 3*x - 10)^2));      ⇒ 10 + 3 RootOf(_Z^2 - 3 _Z - 10)
```

Обратной к ***expand*** является ***factor*(V{, F})-**процедура, возвращающая результат *факторизации* **V-**выражения полиномиального типа с целыми**,** рациональными**,** комплексными или алгебраическими числовыми коэффициентами. При этом, если **V** – отношение полиномов, то производится факторизация отдельно для *числителя* и *знаменателя*. Второй необязательный **F-**аргумент функции определяет тип *числового* поля, над которым должна производиться факторизация**;** по умолчанию выбирается поле, определяемое *типом* коэффициентов полинома. При значениях для **F-**аргумента *real* или *complex* типа производится соответственно *float-* или *complex-факторизация*. В этом случае в качестве **V-**выражения должен выступать *полином* от одной переменной или их отношение. Если в качестве факторизуемого **V-**выражения выступает список**,** множество, диапазон выра-

жений, ряд, отношение либо функция, то факторизация применяется рекурсивно к его компонентам. Следующий прозрачный фрагмент иллюстрирует некоторые варианты применения ***factor*-**процедуры для факторизации выражений**:**

```
> Digits:= 6:  P:= x*4 - 57*x^3 + 52*x^2 - 10*x + 32:  factor(P);   ⇒ -6 x - 57 x^3 + 52 x^2 + 32
> map2(factor, P, [real, complex]);  factor([x^2 - 3*x + 52, x^2 - 10*x + 32], 'complex');
```

$$[-57.\,(x-1.20919)\,(x^2+0.296908\,x+0.464281),\ -57.\,(x+0.148454+0.665013\,I)\,(x+0.148454-0.665013\,I)\,(x-1.20919)]$$

$$[(x-1.50000+7.05337\,I)\,(x-1.50000-7.05337\,I),\ (x-5.00000+2.64575\,I)\,(x-5.-2.64575\,I)]$$

```
> P1:= x^2*y^2 - 18*x^2*y + 81*x^2 - 4*x*y^2 + 72*x*y - 324*x + 4*y^2 - 72*y + 324:
> P2:= x^2*y^2 - 4*x^2*y + 4*x^2 - 2*x*y^2 + 8*x*y - 8*x + y^2 - 4*y + 4:  factor(P1/P2);
```

$$\frac{(x-2)^2}{(y-2)^2}$$

```
> factor([G(P1/P2) .. H(P2/P1), P1 <> P2]);
```

$$\left[\mathrm{G}\left(\frac{(y-9)^2\,(x-2)^2}{(y-2)^2\,(x-1)^2}\right) .. \mathrm{H}\left(\frac{(y-2)^2\,(x-1)^2}{(y-9)^2\,(x-2)^2}\right),\ (y-9)^2\,(x-2)^2 \neq (y-2)^2\,(x-1)^2\right]$$

Если в качестве значения **F-**аргумента выступают ***RootOf*-**процедура, список либо множество ***RootOf*-**процедур, радикал, их список либо множество, то факторизация производится над соответствующим числовым алгебраическим полем.

Наконец, по процедуре ***factors*(V{, F})** производится *факторизация полинома* от нескольких переменных с действительными или комплексными коэффициентами над рациональным алгебраическим числовым полем. В отличие от ***factor*-**процедуры, допускающей *любое* **V-**выражение**,** для ***factors*-**процедуры в качестве ее *первого* аргумента должен выступать только **V-**полином, а возвращается списочная структура следующего вида**:**

[G, [[M[1], k[1]], [M[2], k[2]], ... [M[n], k[n]]]] ; $\mathbf{V} = \mathbf{G} * \prod_{j=1}^{n} \mathbf{M[j]}^{\mathbf{k[j]}}$

где **M[j]** и **k[j]** – соответственно фактор (*сомножитель*) и его кратность. Данная структура достаточно *удобна* при использовании результатов факторизации в задачах программирования, т.к. позволяет легко выделять составляющие ее компоненты. Сказанное о втором необязательном **F-**аргументе ***factor*-**процедуры переносится и на ***factors*-**процедуру**.** Пассивная функция ***Factors*(V{, F})** представляет собой *шаблон* ***factors*-**процедуры, используемый в конъюнкции с ***evala*-**функцией либо в **mod-**конструкции аналогично тому, как это было описано для ***Factor*-**функции с очевидными изменениями. Следующий фрагмент иллюстрирует использование рассмотренных средств *факторизации*:

```
> Digits:= 3:  {factors(P1), factors(P2)};
```

{[1, [[x-2, 2], [y-9, 2]]], [1, [[y-2, 2], [x-1, 2]]]}

```
> factors(x*4 - 64*x^3 + 59*x^2 - 10*x + 39, 'real');
```

[-64., [[x - 1.24, 1], [x^2 + 0.320 x + 0.491, 1]]]

```
> evala(Factors(P2)),  evala(Factors(x^5 - 2*x^4 + x - 2)),  factors(9*x^4 - 3, sqrt(3));
```

$$[1, [[x-1, 2], [y-2, 2]]], [1, [[x-2, 1], [x^4+1, 1]]],\ \left[9, \left[\left[x^2+\frac{\sqrt{3}}{3}, 1\right], \left[x^2-\frac{\sqrt{3}}{3}, 1\right]\right]\right]$$

В приведенном фрагменте полиномиальные выражения **P1** и **P2** соответствуют предыдущему фрагменту,а возвращаемый ***factors*-**процедурой результат не управляется *предопределенной* **Digits-**переменной окружения пакета.

С еще большим основанием *обратной* к ***expand-***функции можно назвать ***combine-***процедуру, имеющую формат кодирования, подобный ***simplify-***процедуре, а именно:

combine(<*Выражение*> {, <*Тип объединения*>} {, <*Параметры*>})

Процедура использует правила, объединяющие термы сумм, произведений и степеней в *единое* целое, упрощая исходное *выражение*. При этом, производится приведение подобных термов. Второй необязательный аргумент процедуры определяет *тип объединения* термов выражения, подобно случаю ***simplify-***процедуры. В качестве первого аргумента ***combine-****процедуры* могут выступать списки, множества и отношения выражений, к элементам которых функция применяется рекурсивно. Третий необязательный аргумент функции определяет специфические *параметры* для конкретного *типа объединения*, указаного *вторым* аргументом процедуры. Следует иметь в виду, что для многих функций ***Maple-***языка пакета ***expand*** и ***combine*** являются взаимно-обратными.

В качестве второго фактического аргумента ***combine-****процедуры* могут выступать отдельный идентификатор, список или множество идентификаторов, определяющих *специальные типы* объединения термов исходного *выражения*. В качестве таких идентификаторов допускаются следующие, определяемые в табл. **12.**

Таблица 12

Тип	***Производится объединение компонент выражения, содержащих:***
arctan	*суммы* ***arctan-***функций; допускается кодирование ***symbolic-***параметра
atatsign	***@@-****операторы*; в качестве идентификатора допустимо `@@`-значение
conjugate	комплексные *сопряженные* термы
exp	*экспоненциальные* термы
ln	суммы *логарифмических* термов; допускается ***symbolic-***параметр
piecewise	*кусочно-определенные* функции
polylog	*полилогарифмические* функции; могут потребоваться ***assume-***условия
power	*степенные* термы
Psi	*термы*, включающие ***Psi-***функции
radicalf	*радикальные* термы; допускается кодирование ***symbolic-***параметра
trig	*тригонометрические* и/или *гиперболические* функции

Смысл и назначение каждого из представленных в табл. **12** значений для второго фактического аргумента ***combine-***процедуры достаточно прозрачны и проиллюстрированы в следующем фрагменте, однако по отдельным необходимы пояснения [8-14,39].

> combine(arctan(1/2) - arctan(10/17) + arctan(59/47)); ⇒ $\arctan\left(\frac{491}{449}\right)$

> combine(arctan(x) + arctan(y) - arctan(z), arctan, 'symbolic'); ⇒ $\arctan\left(\frac{\frac{x+y}{1-x\,y}-z}{1+\frac{(x+y)\,z}{1-x\,y}}\right)$

> combine(G((G@@2)((G@@(-1))(x + (v@@2)(x)))), `@@`);

$$(G^{(2)})(x+(v^{(2)})(x))$$

> h:=a + b*I: v:=b + a*I: t:=a + c*I: combine(conjugate(h)^2 +9*conjugate(v)*conjugate(t));

$$\overline{(a+b\,I)}^2+9\,\overline{((b+a\,I)\,(a+c\,I))}$$

> combine(exp(3*x)*exp(y)*exp(x^2 + 10), 'exp'); ⇒ $e^{(3\,x+y+x^2+10)}$

> combine(a*ln(x)+b*ln(y)-ln(b-z)+ln(c+y)/2, ln, 'anything', 'symbolic'); ⇒ $\ln\left(\frac{x^a\,y^b\,\sqrt{c+y}}{b-z}\right)$

```
> P_w:= piecewise(x < 42, ln(x*exp(y)), (42 <= x) and (x >= 64), 28*x*sin(2*x), (64 < x) and
  (x >= 99), cos(2*x), exp(ln(x))):  combine(P_w);
```

$$\begin{cases} \ln(x\,\mathbf{e}^{y}) & x < 42 \\ x & x < 64 \\ 28\,x\sin(2\,x) & 64 \le x \end{cases}$$

```
> assume(x > 10);  combine(polylog(3, x) + polylog(3, 1/x) + polylog(1, x/(x - 10)), 'polylog');
```

$$2\,\mathrm{polylog}\left(3, \frac{1}{x\sim}\right) - \frac{1}{6}\ln(-x\sim)\,\pi^2 - \frac{1}{6}\ln(-x\sim)^3 - \ln\left(1 - \frac{x\sim}{x\sim - 10}\right)$$

```
> map(assume, [p, k, t], 'integer'): combine((3^n)^p*9^n*2^(64*p+k)*9^(59*p+t), 'power');
```

$$2^{(64\,p\sim + k\sim)}\,3^{(n\,p\sim)}\,9^{(n + 59\,p\sim + t\sim)}$$

```
> combine((Psi(1, s^a - 1/2) - Psi(1, s^a + 1/2))*(s^b - 1/2)^3, Psi);   ⇒
```

$$\frac{\left(s^b - \frac{1}{2}\right)^3}{\left(s^a - \frac{1}{2}\right)^2}$$

```
> combine((3 + sqrt(10))^(1/2)*(5 - sqrt(10))^(1/2)*sqrt((3 - 4^(1/2))), 'radical');
```

$$\sqrt{(3 + \sqrt{10})\,(5 - \sqrt{10})}$$

```
> combine(sqrt(a)*sqrt(b) + sqrt(3)*sqrt(a + 1)^10*sqrt(b), 'radical', 'symbolic');
```

$$\sqrt{a\,b} + \sqrt{3}\,(a + 1)^5\sqrt{b}$$

```
> unassign('a', 'h'): map(combine, [2^11*sin(a)^6*cos(a)^6, 2^12*sinh(h)^3*cosh(h)], 'trig');
```

$$[-15\cos(4\,a) + 10 - \cos(12\,a) + 6\cos(8\,a),\ 512\sinh(4\,h) - 1024\sinh(2\,h)]$$

Для ***arctan*-**аргумента процедуры ***symbolic*-**параметр задает необходимость проведения объединения ***combine*-**процедурой термов, даже если процедура не устанавливает условия для объединения**;** как правило**,** это требуется в случае *символьных* аргументов функций ***arctan*.** Для ***ln*-**аргумента допускается использование ***symbolic*-**параметра, имеющего смысл предыдущего замечания, и параметра {***integer*|***anything***}, определяющего *тип* {*целый*|*любой*} для коэффициентов логарифмических термов. При этом, кодирование этих параметров производится в порядке**:** {***integer*|***anything***}**,** ***symbolic*.** Для ***radical*-**аргумента допускается использование ***symbolic*-**параметра, полагающего подрадикальные термы неопределенного знака действительными и положительными.

Следует отметить, что особого смысла использование ***piecewise*-**аргумента для ***combine*-**процедуры не имеет, ибо во многих случаях уже простейшие выражения, содержащие кусочно-определенные функции, возвращаются без изменений либо с минимальными упрощениями. Детально данный вопрос рассматривается в наших книгах [10-11,14].

По вызову ***combine*(V, *ln* {, t {, *symbolic*}})** производится группировка логарифмических термов **V-**выражения**;** при этом, необязательные *третий* **t-**аргумент процедуры определяет тип коэффициентов в логарифмических термах выражения, тогда как *четвертый* **-** необходимость проведения группировки в *символьном* виде. В качестве как самостоятельного средства упрощения (*преобразования*) выражений, так и в *совокупности* с ***combine*-**процедурой может выступать и ***simplify*(V, *power* {, *symbolic*})-**процедура**,** обеспечивающая упрощение **V-**выражения, содержащего степени, экспоненты или логарифмы и их совокупности. Следующий простой фрагмент хорошо иллюстрирует вышесказанное**:**

```
> combine(a*ln(x+3) + 3*ln(x+c) - 10*ln(d-y) + b*ln(10+y), ln, 'anything', 'symbolic');
```

$$\ln\left(\frac{(x + 3)^a\,(x + c)^3\,(10 + y)^b}{(d - y)^{10}}\right)$$

```
> combine(a*ln(x + 3) + 3*ln(x + c) - 10*ln(d - y) + b*ln(10 + y) - ln(g - z), ln, 'symbolic');
```

$$a\ln(x+3)+b\ln(10+y)+\ln\left(\frac{(x+c)^3}{(d-y)^{10}(g-z)}\right)$$

```
> simplify(a^b*a^(c+d)*(ln(x+10) - ln(y+17) + ln(64*y+59*y))*exp(a*x)*exp(b*y), 'power');
```

$$a^{(b+c+d)}\left(\ln(x+10)-\ln(y+17)+\ln(123)+\ln(y)\right)\mathbf{e}^{(a\,x+b\,y)}$$

```
> combine(%, ln, 'anything', 'symbolic');
```

$$\ln\left(\left(\frac{123\,(x+10)\,y}{y+17}\right)^{\left(a^{(b+c+d)}\mathbf{e}^{(a\,x+b\,y)}\right)}\right)$$

Завершить данный пункт целесообразно *важной* процедурой ***collect***, имеющей формат:

collect(<*Выражение*>, <*Переменная*> {, <*Форма*> {, ***Proc***}})

и рассматривающей *выражение*, заданное ее первым фактическим аргументом, в качестве обобщенного полинома по определенной вторым аргументом ведущей переменной (*списком или множеством переменных*; *в качестве переменных допускаются и идентификаторы функций*). Согласно данному предположению ***collect***-процедура возвращает результат *приведения* всех коэффициентов при одинаковых рациональных степенях ведущих переменных. При этом, сортировки термов не производится и в случае такой необходимости используется ранее рассмотренная ***sort***-функция ***Maple***-языка.

Третий необязательный аргумент определяет *форму* приведенного *выражения*, допуская два значения ***recursive*** (*по умолчанию*) и ***distributed***. В *первом* случае производится рекурсивное *приведение* выражения относительно каждой из ведущих переменных {**x1, x2, ..., xn**}, во *втором* - относительно термов $\mathbf{x1^{p}*x2^{q}*...*xn^{k}}$. Наконец, *четвертый* необязательный аргумент ***collect***-процедуры позволяет задавать процедуру (***Proc***), применяемую к коэффициентам приведенного *выражения*. В качестве такой процедуры наиболее употребительны такие как ***simplify***, ***factor*** и ***sort***, рассмотренные выше. В качестве основных приложений ***collect***-процедуры отметим следующие.

Прежде всего, ***collect***-процедура применяется с целью *упрощения* выражений *обобщенной* полиномиальной формы путем приведения термов с одинаковыми степенями по обобщенным *ведущим* переменным. Второй целью является последующая работа с коэффициентами полиномов, для чего желательна их группировка при одинаковых степенях переменных. Для случая полиномов от нескольких переменных ***collect***-процедура позволяет представлять их в различных формах, удобных для конкретных приложений. Указанные случаи применения ***collect***-процедуры весьма наглядно иллюстрирует следующий достаточно простой фрагмент:

```
> collect(x*y + a*x*y + 9*y*x^2 - a*y*x^2 + x + a*x + b*x^3 - c*x^2, x);
```

$$b\,x^3+(9\,y-a\,y-c)\,x^2+(y+a\,y+1+a)\,x$$

```
> collect(x*y + a*x*y + 9*y*x^2 - a*y*x^2 + x + a*x + b*x^3 - c*x^2, x, 'sort');
```

$$b\,x^3+(-y\,a+9\,y-c)\,x^2+(y\,a+y+a+1)\,x$$

```
> collect(x*y + a*x*y + 9*y*x^2 - a*y*x^2 + x + a*x + b*x^3 - c*x^2, {x, y});
```

$$b\,x^3+((-a+9)\,y-c)\,x^2+((a+1)\,y+a+1)\,x$$

```
> collect(x*y + a*x*y + 9*y*x^2 - a*y*x^2 + x + a*x + b*x^3 - c*x^2, {x, y}, 'distributed');
```

$$(a+1)\,x+(-a+9)\,y\,x^2+(a+1)\,x\,y+b\,x^3-c\,x^2$$

```
> collect(3*x*y^2 + 3*x^2*y + y^3 - 3*x^2 + 6*x*y - 3*y^2 + 9*x + x^3 + 3*y, x);
```

$$x^3+(-3+3\,y)\,x^2+(3\,y^2+9+6\,y)\,x+3\,y-3\,y^2+y^3$$

```
> collect(3*x*y^2 + 3*x^2*y + y^3 - 3*x^2 + 6*x*y - 3*y^2 + 9*x + x^3 + 3*y, y);
```

$$y^3+(-3+3\,x)\,y^2+(6\,x+3\,x^2+3)\,y+9\,x-3\,x^2+x^3$$

```
> collect(3*x*y^2 + 3*x^2*y + y^3 - 3*x^2 + 6*x*y - 3*y^2 + 9*x + x^3 + 3*y, y, 'factor');
```

$$y^3 + (-3 + 3\,x)\,y^2 + 3\,(x + 1)^2\,y + x\,(9 - 3\,x + x^2)$$

```
> collect(64*x^3 + x^2*ln(y) + ln(y) + 4*x^3*ln(y)^2 - 2*x*ln(y), 'ln');
```

$$4\,x^3\,\ln(y)^2 + (x^2 + 1 - 2\,x)\,\ln(y) + 64\,x^3$$

```
> collect(64*x^3 + x^2*ln(y) + ln(y) + 4*x^3*ln(y)^2 - 2*x*ln(y), ln, 'factor');
```

$$4\,x^3\,\ln(y)^2 + (x - 1)^2\,\ln(y) + 64\,x^3$$

```
> collect(b*x*y^2 + b*x*y + c*x*y - a*x*y^2 + a*x*y, y, ['factor', 'sqrt']);
```

$$[-x\,(a - b), \sqrt{-a\,x + b\,x}\,]\,y^2 + [x\,(a + b + c), \sqrt{a\,x + b\,x + c\,x}\,]\,y$$

```
> collect(b*x*y^2 + b*x*y + c*x*y - a*x*y^2 + a*x*y, y, ln@sqrt@abs@factor);
```

$$\ln(\sqrt{|x\,(a - b)|}\,)\,y^2 + \ln(\sqrt{|x\,(a + b + c)|}\,)\,y$$

```
> collect(b*x*y^2 + b*x*y + c*x*y - a*x*y^2 + a*x*y, y, ln@(H^2)@sqrt@abs@factor);
```

$$\ln(\mathrm{H}(\sqrt{|x\,(a - b)|}\,)^2)\,y^2 + \ln(\mathrm{H}(\sqrt{|x\,(a + b + c)|}\,)^2)\,y$$

В частности, из трех последних примеров фрагмента следует, что в качестве четвертого аргумента ***collect-***процедуры могут выступать более сложные конструкции**,** чем отдельные функции/процедуры. В частности, можно использовать **@-**конструкции, позволяющие обеспечивать различного рода *рекурсивную* обработку коэффициентов приведенного выражения, определенного первым фактическим аргументом ***collect-***процедуры.

В данном пункте был рассмотрен ряд базовых средств**,** обеспечивающих важные задачи по *упрощению* символьных выражений. Среди них следует еще раз акцентировать ваше внимание на таких часто используемых средствах как: ***convert, simplify, factor, combine, collect, map, expand.*** При этом, следует иметь в виду, что в задачах упрощения (*а в более жестком понимании и канонизации*) выражений используются также и другие функции ***Maple-***языка, имеющие иную основную направленность или более узкую ориентацию, например, на задачи полиномиальной арифметики, представленные в языке достаточно хорошо. Однако и задачу упрощения выражений следует понимать в существенно более широком смысле, как представление выражений в наиболее приемлемом для конкретных приложений виде. В следующем пункте настоящего раздела рассматриваются базовые средства**,** обеспечивающие символьные преобразования выражений и их алгебраическую обработку в целом.

Символьная обработка выражений. Для обеспечения работы с символьными конструкциями, в первую очередь на формальном уровне, ***Maple-***язык располагает рядом достаточно развитых средств**,** из которых мы акцентируем внимание на тех**,** которые носят более прикладной характер**;** некоторые из них были рассмотрены нами выше в связи с другим контекстом. Наряду с этим для формальных преобразований используются и другие ранее рассмотренные средства ***Maple-***языка**;** здесь же представлены, в основном, средства, ориентированные**,** в первую очередь**,** на такого типа обработку символьных выражений, которые позволяют осознанно использовать их уже на ранней стадии работы в среде ***Maple-***языка. Данные средства относятся к группе средств *формального* манипулирования символьными выражениями.

Прежде всего**,** задачи символьной обработки предполагают проведение определенных видов анализа *символьных* выражений**:** определение типа выражения, определение функциональной зависимости и др. Из данных средств можно отметить наиболее важные на *начальной* стадии освоения средств символьной обработки. Рассмотренные ранее функции ***nops*** и ***op*** многофункциональны и с успехом могут использоваться для общего анализа *внутренней* структуры ряда типов объектов, тогда как по функциям ***typematch, type*** и ***whattype***-процедуре и можно получать *тип* анализируемого объекта.

Для выявления факта зависимости алгебраического выражения от переменных служит логическая процедура ***depends*(**<*Выражение*>**,** <*Переменные*>**),** возвращающая ***true*-**значение, если указанные вторым аргументом *ведущие переменные* (*отдельная*, *список либо множество*) входят в заданное первым фактическим аргументом *выражение*, и ***false*-**значение в противном случае. При этом, переменная полагается *ведущей*, если она не связана внутренним соотношением (*переменная индексная*, *суммирования*, *интегрирования*, *произведения и т.п.*). В качестве как первого, так и второго фактических аргументов ***depends*-**процедуры могут выступать *список* или *множество* соответственно выражений и *ведущих* переменных. Функция возвращает ***true*-**значение, если обнаруживает указанного типа зависимость *по меньшей мере* для одной из пар <*выражение*, *переменная*>. Следующий фрагмент иллюстрирует применение ***depends*-**процедуры для установления факта наличия зависимости выражения от заданных переменных:

```
> depends([ln(x) + sin(y), ln(t + sqrt(z)), sqrt(Art(x) + Kr(z))], {x, z});   ⇒ true
> [depends(ln(x) + sin(y), {x, z}), depends(ln(t + sqrt(z)), {z, h})];   ⇒ [true, true]
> [depends(F(x)$x = a..b, x), depends(F(x)$x = a..b, {b, h})];   ⇒ [true, true]
> [depends(int(GS(y), y = a..b), y), depends(int(GS(y), y = a..b), b)];   ⇒ [false, true]
> [depends(M[n]*x, n), depends(M[n]*x, x), depends(diff(H(y), y), y)];   ⇒ [true, true, true]
> F:= x -> (cosh(x)^2 + sinh(x)^2)*(cos(x)^2 + sin(x)^2)/(2*cosh(x)^2 - 1):
> [depends(F(x), x), depends(simplify(F(x)), x), simplify(F(x))];   ⇒ [true, false, 1]
```

В случае сложных выражений ***depends*-**процедура может возвращать некорректные результаты, поэтому в подозрительных случаях следует упрощать исходное выражение посредством ***simplify*-**процедуры, как это демонстрирует *последний* пример фрагмента. В ряде случаев при определении для ***depends*-**процедуры в качестве ведущей связанную переменную инициируется ошибка, например в случае ранжирования:

```
> depends(F(x)$x = 1 .. 64, x);
```

Error, invalid input: depends expects its 2nd argument, x, to be of type {name, list(name), set(name)}, but received F(2)

Для тестирования произвольного **V-**выражения на предмет *вхождения* в него заданного элементарного **H-**подвыражения можно использовать ***Etest*-**процедуру, созданную на основе рассмотренных выше функций {***op***, ***nops***} и процедуры ***charfcn***:

```
Etest := proc(V::anything, H::anything, R::evaln)
local k, S, L, Z;
    assign(S = {op(V)}, L = {}, Z = {});
    do
        for k to nops({op(S)}) do `if`(nops({op(S[k])}) ≠ 1,
            assign('L' = {op(S[k]), op(L)}), assign('Z' = {S[k], op(Z)}))
        end do;
        if L = {} then break else S := L; L := {} end if
    end do;
    assign('R' = Z), [false, true][1 + charfcn[Z](H)]
end proc
> Etest(x*cos(y) + y*(a + b*exp(x))/(ln(x) - tan(z)) - sqrt(Art + Kr)/(Sv + Ag^z), Art, R), R;
                    true, {-1, 1, 2, cos(y), e^x, ln(x), tan(z), x, y, a, b, z, Art, Kr, Sv, Ag}
> Etest(x*tan(y) + y*(gamma + b*G(x))/(ln(x) - S(z)) <> sqrt(Art + Av)/(c^2 + d^2), Kr, V), V;
                    false, {-1, 1, 2, Av, d, γ, ln(x), c, x, y, b, Art, tan(y), G(x), S(z)}
```

Данная процедура возвращает ***true*-**значение в случае установления факта вхождения *элементарного* **H-**терма в качестве *подвыражения* в **V-**выражение, и ***false*-**значение в противном случае, где под *элементарным* будем полагать **G-**терм, для которого имеет место *определяющее* соотношение ***nops*({*op*(G)}) ⇒ 1.** Предыдущий фрагмент представляет исходный текст ***Etest*-**процедуры с примерами ее конкретного вызова. Процедура ***Etest*(V, H, R)** тестирует наличие факта вхождения в **V-**выражение *элементарного* **H-**терма и через третий фактический **R**-аргумент возвращает множество всех элементарных термов тестируемого **V**-выражения без учета их *кратности*. Процедура может представить определенный практический интерес и читателю рекомендуется в качестве полезного упражнения рассмотреть ее организацию.

Функция ***indets*(W)** возвращает множество, содержащее все *независимые* переменные алгебраического **W-**выражения, включая такие константы как {*false*, *gamma*, *infinity*, *true*, *Pi*, *Catalan*, *FAIL*}. Между тем, во многих случаях требуется найти все именно независимые переменные алгебраического **W-**выражения, т.е. **X**-переменные, удовлетворяющие определяющему соотношению **type(eval(X), 'symbol'); ⇒** *true*. Процедура ***Indets*** [103,109] решает данную задачу. Вызов процедуры ***Indets*(W)** возвращает множество, содержащее все независимые переменные алгебраического **W**-выражения, удовлетворяющие вышеупомянутому условию, например**:**

```
> W:= Catalan*sin(x) + sqrt(Art^2 + Kr^2)/(AG(h, x) + VS(t) + VA(r)):
  map2(indets, W, 'symbol');   ⇒  {x, h, Catalan, t, Kr, Art, r}
> indets(W);   ⇒  { t, h, r, x, Art, Kr, sin(x), AG(h, x), VS(t), VA(r), sqrt(Art^2 + Kr^2) }
> Indets(W);   ⇒  {x, h, t, Kr, Art, r}
```

В отличие от предыдущей процедуры вызов процедуры ***indetval*(A)** [42,103] возвращает множество всех возможных *неопределенных* символов произвольного ***Maple***-выражения **A,** базируясь на его операндах всех уровней. В случае кодирования в качестве второго необязательного аргумента **t** допустимого ***Maple***-типа вызов процедуры ***indetval*(A, t)** возвращает *операнды* **A**-выражения, имебщие **t**-тип. Приведенные ниже примеры хорошо иллюстрируют сказанное:

```
> k:= 64: t:= table([x = 64, y = 59, z = 39]):  indetval(k), indetval(t), indetval(t, 'integer');
                                {}, {x, y, z}, {39, 59, 64}
> type(MkDir, 'procedure'), indetval(MkDir);
                        true, {f, F, K, d, u, g, r, v, t, z, k, L, h, cd, s, omega, Lambda}
> indetval(Pi*cos(x) +Catalan*sin(y) - 2*exp(1)*gamma*z - sqrt(6)*ln(14)/(t+h) +
  G*S*V*Art*Kr);
                                {S, V, t, x, y, z, G, h, Art, Kr}
> Indets(Pi*cos(x) + Catalan*sin(y) - 2*exp(1)*gamma*z - sqrt(6)*ln(14)/(t+h) +
  G*S*V*Art*Kr);
                                {S, V, t, x, y, z, G, h, Art, Kr}
> indets(MkDir), Indets(MkDir), indets(t), Indets(t), indetval(t), indetval(t, 'integer');
                        {MkDir}, {MkDir}, {t}, {t}, {x, y, z}, {39, 59, 64}
> indets(nmmlft), Indets(nmmlft), indetval(nmmlft), indetval(nmmlft, 'procedure');
                        {nmmlft}, {nmmlft}, {f, F, d, c, t, k, h, a, b, p, file}, {min, max}
```

По функции ***normal*(V {, *extended*})** производится упрощение рационального **V-**выражения и приведение его к *нормальной* форме, суть которой состоит в приведении к *общему* знаменателю на всех уровнях и сокращению числителя и знаменателя на *общие* множители**.** Вместе с тем**,** *факторизации* выражения не производится и *выделяются* только *тривиальные* сомножители**.** Для получения *канонической нормальной* формы следует исполь-

зовать ***factor*(*normal*(V))-**конструкцию. В качестве **V-**выражения могут выступать ряд, список, множество, диапазон, уравнение, отношение или функция, нормализация которых производится рекурсивно относительно их элементов. В случае кодирования необязательного ***extended-***аргумента в числителе и знаменателе полиномы остаются раскрытыми. Следующий простой фрагмент иллюстрирует вышесказанное**:**

```
> R:= H(x)/W(y):  v:= 350/64:  [numer(R), denom(R), numer(v), denom(v)];
                                [H(x), W(y), 175, 32]
> Pf:= (170*x^3 - 680*x^4 + 170*x^5 + 1700*x^2 - 680*x - 1360)/(112*x^4 - 392*x - 448*x^2 -
  112*x^3 + 56*x^5 - 112):  Pf1:= [normal(Pf), normal(Pf, 'expanded')];
```

$$Pf1 := \left[\frac{85\,(x^2 - 4\,x + 4)}{28\,(x^2 + 2\,x + 1)}, \frac{85\,x^2 - 340\,x + 340}{28\,x^2 + 56\,x + 28} \right]$$

```
> factor(normal(Pf));  ⇒
```

$$\frac{85\,(x-2)^2}{28\,(x+1)^2}$$

```
> Pf2:= 1 + 1/(1 + x/(1 + x)):  [numer(Pf2), denom(Pf2)];  ⇒  [3 x+2, 2 x+1]
```

$$\frac{3\,x+2}{2\,x+1}$$

```
> normal(Pf2);  ⇒
> map(normal, [F(1 + 1/y), [1 + 1/x, x + 1/x], z + 1/z .. z + z/(1 + z)]);
```

$$\left[\mathrm{F}\left(\frac{y+1}{y} \right), \left[\frac{x+1}{x}, \frac{x^2+1}{x} \right], \frac{z^2+1}{z} .. \frac{z\,(2+z)}{1+z} \right]$$

По процедуре ***radnormal*(V{*, rationalized*})** производится *нормализация* алгебраического **V-**выражения, содержащего радикальные числа**,** исключая случаи *одновременного* вхождения *радикальных* чисел и ***RootOf-***конструкций. В случае *числовых* **V-**значений по умолчанию не нормализуются их *знаменатели*, но это можно определять посредством необязательного *второго* ***rationalized-***аргумента процедуры. В качестве **V-**аргумента процедуры допускаются отношения, списки и множества. По вызову ***rationalize*(V)-**процедуры производится *рационализация* знаменателя алгебраического **V-**выражения, например**:**

```
> S:= x -> ((x^2 + 2*x*2^(1/2) - 2*x*3^(1/2) + 5 - 2*2^(1/2)*3^(1/2))/(x^2 - 2*x*3^(1/2) + 1)):
> [radnormal(S(z)), radnormal(S(1)), radnormal(S(1) <> S(0), 'rationalized')];
```

$$\left[\frac{z - \sqrt{3} + \sqrt{2}}{z - \sqrt{3} - \sqrt{2}}, \frac{-3 - \sqrt{2} + \sqrt{3} + \sqrt{2}\,\sqrt{3}}{-1 + \sqrt{3}}, -\sqrt{3} + \sqrt{2} \neq 5 - 2\,\sqrt{2}\,\sqrt{3} \right]$$

```
> [(x - y)/(sqrt(x) + sqrt(y)), x + y/(x - sqrt(x + sqrt(2)))]:  rationalize(%);
```

$$\left[-\sqrt{y} + \sqrt{x}, \frac{(x^2 - x\sqrt{x + \sqrt{2}} + y)\,(x + \sqrt{x + \sqrt{2}})\,(x^2 - x + \sqrt{2})}{x^4 - 2\,x^3 + x^2 - 2} \right]$$

Тут же уместно несколько детальнее упомянуть и о ***RootOf-***процедуре с форматом**:**

***RootOf*(**<*Уравнение*> {**,** <*Переменная*> {**, p | , m .. n**}}**)**

и определяющей *шаблон* (*конструкцию*) для представления всех *корней* заданного ее *первым* фактическим аргументом *уравнения* от одного неизвестного. При этом, в качестве первого аргумента допускается и **V-**выражение, рассматриваемое ***Maple-***языком в качестве *левой* части уравнения **V = 0.** ***Maple-***язык использует ***RootOf-***конструкции для стандартного представления алгебраических чисел, функций и конечных полей Галуа. Так, ***Maple-***язык распознает над ***Root0f-***конструкциями операции упрощения (***simplify***), интегрирования (***int***), дифференцирования (***diff***), численных вычислений (***evalf***), разложения в ряд (***series***) и некоторые другие. Следующий довольно простой фрагмент иллюстрирует вышесказанное**:**

```
> R:=[RootOf(x^2 - 64*x+59, x), RootOf(x^2 = 10*y + 17, x)]: [evalf(R[1], 3), diff(R[2], y),
  int(R[2], y)];
```

$$\left[0.936, \frac{5}{\mathrm{RootOf}(_Z^2 - 10\,y - 17)}, \left(\frac{2\,y}{3} + \frac{17}{15}\right)\mathrm{RootOf}(_Z^2 - 10\,y - 17)\right]$$

```
> evalf(RootOf(x^3 + 10*x + 20.06, x, -2 .. 10), 16);   ⇒ -1.597962674497701
> RootOf((x + Pi/2)*sin(x) = t, x):  Order:= 8:  series(%%, t);
```

$$-\frac{\pi}{2} - t - \frac{1}{2}t^3 - \frac{17}{24}t^5 - \frac{961}{720}t^7 + \mathrm{O}(t^8)$$

```
> [type(R[1], RootOf), typematch(R[2], 'RootOf')];   ⇒ [true, true]
> convert(I + 2^(1/2)*x + 5^(3/2)*y, 'RootOf');
```

$$\mathrm{RootOf}(_Z^2 + 1, index = 1) + \mathrm{RootOf}(_Z^2 - 2, index = 1)\,x + 5\,\mathrm{RootOf}(_Z^2 - 5, index = 1)\,y$$

Тип ***RootOf-***конструкций распознается по функциям {***type***, ***typematch***}, тогда как по функции ***convert*** можно конвертировать **I-**константы и радикалы в ***RootOf-***конструкции.

По ***root-***процедуре, имеющей два эквивалентных формата кодирования вида**:**

root*(V, n** {**,** ***symbolic}**)** или ***root*[n](V** {**,** ***symbolic***}**)**

вычисляется **n-**й *корень* из алгебраического **V-**выражения**,** в качестве которого могут выступать также *действительные* и *комплексные* константы**.** В случае кодирования необязательного ***symbolic-***аргумента подкоренное *выражение* полагается *положительным* и производятся его определенные упрощения**.** В ряде случаев целесообразно по ***assume-***процедуре налагать определенные условия на компоненты **V-**выражения, например**:**

```
> assume(a > 0, b > 0, y > 0):  root(a + b*y + 10*y^2, 3);   ⇒ (a~ + b~ y~ + 10 y~^2)^(1/3)
> assume(y < 2, y >= 0): root[5](x^5*(y - 2)^4, 'symbolic');   ⇒ x (2 - y~)^(4/5)
```

$$\frac{x\,((x^2 + n)\,(z^3 - m))^{(1/3)}}{z^3 - m}$$

```
> root[3](x^3*(x^2 + n)/(z^3 - m)^2, 'symbolic');   ⇒
> map(root, [64 + 42*I, 10, 20.06, 19.47 - 59*I, gamma, Catalan], 2);
```

$$[\sqrt{64 + 42\,I}, \sqrt{10}, 4.478839135, 6.387470130 - 4.618416900\,I, \sqrt{\gamma}, \sqrt{Catalan}]$$

Весьма полезной при работе с *символьными* тригонометрическими выражениями может оказаться и ***trigsubs-***процедура, работающая с *тригонометрической таблицей* (***trig-****таблицей*) пакета**,** входы которой содержат известные для функции тригонометрические конструкции**.** По вызову процедуры ***trigsubs*(0)** возвращается множество распознаваемых процедурой тригонометрических функций. Вызов процедуры ***trigsubs*(V)** возвращает список тригонометрических выражений, *эквивалентных* тригонометрическому **V-**выражению, находящемуся в ***trig-***таблице. При этом, следует иметь в виду, что на **V-**выражениях, не распознаваемых процедурой ***trigsubs*(V)** в качестве входов в ***trig-***таблицу, инициируются ошибочные ситуации, поэтому в качестве **V-**выражения допускаются только известные процедуре тригонометрические конструкции. При этом, упрощения фактического **V-**аргумента не производится, что может вызывать ошибочные ситуации на очевидных выражениях**.** Поэтому, во избежание *некорректных* даже с точки зрения языка ***Maple*** ситуаций нами рекомендуется использовать конструкции следующего вида ***trigsubs*(*simplify*(*V*, 'trig'))**, как это хорошо иллюстрирует нижеследующий достаточно простой фрагмент**:**

```
> trigsubs(0);  trigsubs(AVZ);   ⇒ {tan, sec, sin, cos, cot, csc}
Error, (in trigsubs) unknown expression
> trigsubs(x*sin(x));
```

Error, (in **trigsubs**) expecting a product of two functions but got x*sin(x)
> trigsubs(sin(x) + cos(x));
Error, (in **trigsubs**) sum not found in table
> trigsubs(simplify(F(x), 'trig'));
Error, (in **trigsubs**) unknown function **-** try **trigsubs(0)**
> trigsubs(simplify(tan(x)*cos(x), 'trig'));

$$\left[\sin(x), -\sin(-x), 2\sin\left(\frac{x}{2}\right)\cos\left(\frac{x}{2}\right), \frac{1}{\csc(x)}, -\frac{1}{\csc(-x)}, \frac{2\tan\left(\frac{x}{2}\right)}{1+\tan\left(\frac{x}{2}\right)^2}, \frac{-1}{2} I \left(\mathbf{e}^{(x\,I)} - \mathbf{e}^{(-I\,x)}\right) \right]$$

В общем случае ***trig-***таблица пакета характеризуется относительно *ограниченным* представительством тригонометрических соотношений, что существенно снижает возможности процедуры ***trigsubs*** по обработке символьных выражений.

По вызову процедуры ***trigsubs*****(**<*Уравнение*>**),** где *уравнение* должно быть строго тригонометрическим, определяется *факт* его вхождения в ***trig-***таблицу. В зависимости от {*наличия*|*отсутствия*} уравнения в таблице возвращается символьное значение {`*found*`|`*not found*`}. Процедура ***trigsubs*** работает с *таблицей* **`*trigsubs/TAB*`**, содержимое которой для текущего релиза можно получать по вызову ***eval*****(`*trigsubs/TAB*`).**

По вызову процедуры ***trigsubs*****(**<*уравнение*>**,** <*выражение*>**)** в случае принадлежности *уравнения***,** заданного первым аргументом функции**,** ***trig-***таблице производится подстановка его в заданное вторым аргументом *выражение* и возвращается результат такого редактирования**;** в противном случае выводится соответствующее диагностическое сообщение с рекомендациями**.** Следующий простой фрагмент иллюстрирует вышесказанное:

> trigsubs(cot(Pi + a*x));

$$\left[\cot(a\,x), -\cot(-a\,x), \frac{1}{2}\frac{\cot\left(\frac{a\,x}{2}\right)^2 - 1}{\cot\left(\frac{a\,x}{2}\right)}, \frac{\cos(a\,x)}{\sin(a\,x)}, \frac{1+\cos(2\,a\,x)}{\sin(2\,a\,x)}, \frac{\sin(2\,a\,x)}{1-\cos(2\,a\,x)}, \right.$$

$$\frac{1}{\tan(a\,x)}, -\frac{1}{\tan(-a\,x)}, \frac{1}{2}\frac{1-\tan\left(\frac{a\,x}{2}\right)^2}{\tan\left(\frac{a\,x}{2}\right)}, \frac{1}{2}\cot\left(\frac{a\,x}{2}\right) - \frac{1}{2}\tan\left(\frac{a\,x}{2}\right),$$

$$\frac{(\mathbf{e}^{(a\,x\,I)} + \mathbf{e}^{(-I\,a\,x)})\,I}{((\mathbf{e}^{(a\,x\,I)})(1+\cos(2\,a\,x)) - (\mathbf{e}^{(-I\,a\,x)})(1+\cos(2\,a\,x)))\sin(2\,a\,x)}, \frac{\sin(2\,a\,x)}{1-\cos(2\,a\,x)},$$

$$\left. \csc(2\,a\,x) + \cot(2\,a\,x) \right]$$

> map(trigsubs, [sec(x)^2 = 1 + tan(x)^2, sin(x)^2 = 1 - cos(x)^2]); ⇒ [`*found*`, `*found*`]
> map(trigsubs, [sin(x) = tan(x)*cos(x), cos(x) = cot(x)*sin(x)]);
[`*not found*`, `*not found*`]
> trigsubs(sin(2*t) = 2*cos(t)*sin(t), sin(2*t)*t + tan(t)); ⇒ $2\cos(t)\sin(t)\,t + \tan(t)$
> trigsubs(cos(x + y) = cos(x)*cos(y) - sin(x)*sin(y), cos(x + y)*t - 57*cos(x + y) + 2006);
Error, (in **trigsubs**) not found in table **-** use **subs** to over ride

Как видно из приведенного фрагмента, в случае невозможности выполнить преобразование выводится сообщение с рекомендацией воспользоваться ***subs-***функцией.

Относительно использования ***eval-***функции, имеющей простой формат кодирования**:**

***eval*(**<*Выражение*> {**, n**}**)**

следует сделать отдельное пояснение. Под *вычислением* понимается подстановка вместо идентификатора приписанного ему *значения*, а каждый шаг в данном процессе определяет отдельный вычислительный уровень. По функции ***eval*** производится вычисление определенного ее первым фактическим аргументом *выражения* на его **n-**ом вычислительном уровне или полное вычисление *выражения***,** если второй аргумент отсутствует. В интерактивном режиме с пакетом при вводе выражений производится их полное вычисление, как если бы к входящим в них идентификаторам применялась ***eval-***функция. Иная ситуация имеет место, в частности, для ***Maple-***процедур, для которых локальные переменные вычисляются только на *первом* уровне, как это иллюстрирует следующий весьма наглядный пример**:**

> Proc:=proc() local a, b, c; a:= b; b:= c; c:= d: a*b*c end proc: Proc(), eval(Proc()); $\Rightarrow b\,c\,d,\ d^3$

Из приведенного примера видно, что при вызове ***Proc-***процедуры вычисление локальных переменных {**a, b, c**} производится на *первом* уровне, о чем свидетельствует возвращаемый процедурой результат. Тогда как применение к вызову процедуры ***eval-***функции обеспечивает полное вычисление процедуры**.** При этом, следует иметь в виду, что для *полного* вычисления локальных переменных могут потребоваться достаточно длинные цепочки *присвоений*, требующих затрат основных ресурсов **ПК.**

По ***subs-***функции можно эффективно производить модифицирующие подстановки и в процедуры либо функции, однако данный механизм действует только на *подвыражения* тела процедуры, не содержащие *локальных* переменных, как это иллюстрирует следующий достаточно простой фрагмент**:**

> Proc:= proc(x, y) local a, b, c; sqrt(a*x^2 + b*y^2 + c)*sin(a*x) + cos(b*y) + 2006 end proc:
> Proc1:=proc(x, y) global a,b,c; sqrt(a*x^2 + b*y^2 + c)*sin(a*x) + cos(b*y) + 2006 end proc:

> subs([a=Pi, b=sigma, c=gamma], Proc1(x, y)); $\Rightarrow \sqrt{\pi\,x^2 + \sigma\,y^2 + \gamma}\,\sin(\pi\,x) + \cos(\sigma\,y) + 2006$

> subs([a=Pi, b=sigma, c=gamma], Proc(x, y)); $\Rightarrow \sqrt{a\,x^2 + b\,y^2 + c}\,\sin(a\,x) + \cos(b\,y) + 2006$

> subs([a=sin(z),b=x*Pi], (x,y) -> sqrt(a*x+b*y)+a*b); $\Rightarrow (x, y) \to \sqrt{\sin(z)\,x + \pi\,x\,y} + \sin(z)\,\pi\,x$

Данная возможность позволяет использовать ***subs-***функцию**,** а также рассматриваемые ниже другие средства подстановок, для целого ряда весьма полезных динамических модификаций процедур в прикладных задачах (*при данном подходе базовое определение продедуры остается неизменым*, *модифицируясь только в точке ее вызова*), для чего входящие в состав модифицируемых компонент идентификаторы при определении процедур следует указывать *глобальными*.

По вызову процедуры ***applyrule*(R, V)** обеспечивается применение к **V-**выражению правил подстановки, определяемых ее *первым* фактическим **R-**аргументом (*одно правило либо их список*) до тех пор, пока они допустимы. При этом, правила подстановки можно определять как уравнениями, так и в форме сложных *шаблонов*, допускаемых ***patmatch-***процедурой**.** Процедура ***applyrule*** эффективнее средств {***subs, algsubs***}**,** однако подобно ***algsubs-***процедуре не производит математических преобразований обрабатываемого выражения, требуя для этих целей ***eval-***функции. Следующий фрагмент иллюстрирует применение ***applyrule-***процедуры для преобразования выражений**:**

> applyrule(x*h(x) = a*x^2 + b*x^2, x*h(x)+b*(d + x*h(x))); ⇒ $a\,x^2 + b\,x^2 + b\,(d + a\,x^2 + b\,x^2)$
> algsubs(x*h(x)=a*x, x*h(x));
Error, (in **algsubs**) cannot compute degree of pattern in **x**
> applyrule((a::float*x+b::integer*y)/c::function=Avz(h), (10.17*x+64*y)/sin(x)); ⇒ Avz(h)
> applyrule((a::float*x + b::integer*y)/c::function = 4, sqrt((10.17*x + 64*y)/G(x))); ⇒ $\sqrt{4}$
> applyrule([a=b, b=c, c=d, d=e, e=f,f=g,g=h], sqrt(Art^a+Kr^b+V*c)); ⇒ $\sqrt{Art^h + Kr^h + V\,h}$
> T:= function: applyrule([a=b, b=a], a); ⇐ *Прерванный бесконечный цикл*
Warning, computation interrupted
> applyrule([a^2+b^2=Art+Kr, c::function =W(t)], sqrt(a^2+b^2)*G(x)); ⇒ $\sqrt{Art + Kr}\ \mathrm{W}(t)$
> applyrule([a::T^b::T = Art(x), c::T/d::T = Kr(t)], S(x)^G(y) + V(x)/H(y)); ⇒ Art(x) + Kr(t)

Для исключения зацикливаний, следует уделять особое внимание определению правил подстановок. Один из примеров фрагмента иллюстрирует простую ситуацию по *зацикливанию* вызова ***applyrule*-**процедуры. Вместе с тем, совместное использование средств ***subs, applyrule, asubs, powsubs*** и ***algsubs*** в сочетании с рядом других функциональных средств позволяют достаточно эффективно производить весьма интересные и важные символьные преобразования различного типа выражений.

Функциональные конструкции Maple-языка. Под функцией в ***Maple*-**языке понимается невычисляемая конструкция следующего общего вида**:**

<*Идентификатор*>**(**<*Последовательность формальных аргументов*>**)**

где *идентификатор* определяет уникальное *имя* функции, а *последовательность* допустимых ***Maple*-**выражений – список ее *формальных аргументов*. Примерами функций являются ***exp*(x), *sin*(x), *sqrt*(x), *AGN*(x, y, z)** и др. ***Maple*-**язык в качестве функций понимает любую из конструкций указанного типа**,** независимо от того**,** является ли она *определенной*. Рассмотренная выше функция ***type***(<*Выражение*>**,** ***function***) тестирует произвольное выражение ***Maple*** на предмет отношения его к типу **"***функция***".** Как видно уже из следующего весьма простого примера:

> map(type, [F(x,y,z), sin(x), H(x, G(y), z)], 'function'); ⇒ [*true, true, true*]

к данному (*функциональному*) типу ***Maple*-**язык относит как процедуру ***sin*,** так и неопределенную в его среде конструкцию указанного формата. Наряду с этим, тестирующая ***type*-**функция позволяет идентифицировать не только собственно *функциональный* тип выражения, указанного ее *первым* фактическим аргументом, но и детализировать математический тип выражения-*функции* согласно значению ее второго фактического аргумента, а именно**:**

algfun* –** *алгебраическая функция;** допускается тестировать по типу коэффициентов и ведущим переменным**;**

{***arctrig*** | ***arctrigh***} **–** *обратная* {*тригонометрическая* | *гиперболическая*} функция**;** допускается тестирование по ведущим переменным**;**

{***evenfunc*** | ***oddfunc***} **–** {*четная* | *нечетная*} функция**;** допускается тестировать по ведущим переменным**;**

radalgfun* –** *алгебраическая функция*, определенная в терминах ***RootOf и радикалов**;** допускается тестирование по ведущим переменным и типам коэффициентов**;**

radfunextn* –** алгебраическое функциональное расширение в терминах *радикалов;**

***radfun* –** *радикальная* функция**;** допускается тестирование по ведущим переменным и типам коэффициентов**;**

***mathfunc* –** *математическая* функция**;** тестируется *любая* функция из приложения **8** [12];

{***trig*** | ***trigh***} **–** {*тригонометрическая* | *гиперболическая*} функция**;** допускается тестирование по ведущим переменным**.**

При этом, *алгебраическим* полагается ***Maple-***выражение одного из следующих типов**:**

integer fraction float string indexed `**+**` `**\***` `**^**` `**\*\***` ***series function*** `**!**` `**.**` ***uneval***

а под ведущей $\mathbf{x_j}$**-**переменной произвольной $\mathbf{F(x_1, ..., x_n)}$**-**функции такая переменная, на которой имеет место следующее определяющее соотношение**:**

$$(\exists x`_j)\,(\exists x``_j)\,(x`_j \neq x``_j \rightarrow F(x_1, ..., x`_j, ..., x_n) \neq F(x_1, ..., x``_j, ..., x_n)) \qquad (j = 1 .. n)$$

Следующий фрагмент иллюстрирует применение этих значений в качестве **2-**го аргумента ***type-***функции для тестирования типов функций**,** указанных ее **1-**м аргументом**:**

```
> G:= 64*y*z + 10*RootOf(x^5 – y + 59*y - 99*x^3/y, x):
> [type(G, algfun(rational)), type(G, algfun(rational, z))];   ⇒ [true, false]
> S:= arcsin(x*y): [type(S, arctrig(x)), type(S, arctrig(y))];   ⇒ [true, true]
> [type(sin(x)*cos(y) ,oddfunc(x)), type(sin(x)*cos(y), oddfunc(y))];   ⇒ [true, false]
> type((64^(1/5)*z^3 + 10*sqrt(x))^(1/6), 'radfunext');   ⇒ true
> typematch(GS(x, y, z, Art(h), Kr(t)), function, 't');      ⇒ true
> map(type, [WeierstrassSigma,InverseJacobiSD,MeijerG], 'mathfunc'); ⇒ [true, true, true]
```

Как следует из последнего примера фрагмента, для тестирования *математических* функций указываются только их идентификаторы. Наряду с рассмотренными выше возможностями {***type*** | ***typematch***}-функции по тестированию ***Maple-***выражений, частным случаем которых является *функция***,** представленные здесь новые возможности позволяют проводить достаточно детальную (*насколько это вообще на сегодня возможно*) *апробацию* как функционального типа выражений, так и математического типа собственно самих тестируемых функций.

В качестве весьма полезного тестирующего средства представляется также процедура**:**

hasfun(<*Выражение*>**,** <***IdF***> {**,** <*Ведущая переменная*>}**)**

позволяющая определять факт вхождения (***true***) в заданное первым аргументом *выражение* функции (*или функций в случае их списка/множества*), заданной вторым ***IdF-***аргументом**,** возможно**,** по *ведущей* (*ведущим в случае их списка/множества*) переменной, заданной ее необязательным третьим аргументом, например**:**

```
> G:=(L(x)+AG(x+y)/Art(z)+Kr(t))/(V(x)+W(y)): hasfun(G, [W,AG,Art,Kr], {y, z, t}); ⇒ true
```

Данная функция часто используется для проверки *вхождения* заданных функций в подинтегральное выражение, а также в задачах символьной обработки выражений.

Функциональная структура анализируется рассмотренными выше функциями ***op*** и ***nops***, возвращающими результаты согласно следующим соотношениям**:**

$$\mathbf{nops}(F(x_1, ..., x_n)) \Rightarrow n \qquad \mathbf{op}(F(x_1, ..., x_n)) \Rightarrow x_1, ..., x_n$$
$$\mathbf{op}(0, F(x_1, ..., x_n)) \Rightarrow F \qquad \mathbf{op}(k, F(x_1, ..., x_n)) \Rightarrow x_k \quad (k > 0)$$

Следующий простой фрагмент иллюстрирует вышесказанное**:**

```
> [op(0, G(x,y,z,h))], [op(G(x, y, z, h))], [nops(G(x, y, z, h))];   ⇒ [G], [x, y, z, h], [4]
> [op(k, G(x, y, z, h))]$'k' = 0 .. nops(G(x, y, z, h));             ⇒ [G], [x], [y], [z], [h]
```

При решении целого ряда математическиих задач определенный интерес представляет тип *arity*, отсутствующий в ***Maple*** релизов **6-10**. Вызов процедуры ***type*(F, arity(n))** [41,103] возвращает *true*-значение, если **F** является функцией или процедурой **n**-арности; в противном случае возвращается *false*-значение. Более того, через *глобальную* переменную '***n-arity***' возвращается реальная арность произвольной функции/процедуры **F,** например:

```
proc(F::{procedure, function}, n::nonnegative)
local a, k;
global `n-arity`;
    if type(F, 'procedure') then a := [op(1, eval(F))]
    else
        assign(k = 0, a = [ ]);
        do k := k + 1; try a := [op(a), op(k, F)] catch : break end try end do
    end if;
    assign(`n-arity` = nops(a)), `if`(`n-arity` = n, true, false)
end proc
```

```
> F:= f(x1, x2, x3, x4, x5, x6): G:= g(x): S:= s(x1, x2, x3, x4, x5, x6): V:=v(a, b, c, d, e, f, h):
> Art:= proc(x, y, z) [y, x, z] end proc: Kr:= proc(a, b, c, d, e, h) {a, b, c, d, e, h} end proc:
> seq([type(k, arity(6)), `n-arity`], k = [F, G, S, V, Art, Kr]);
                    [true, 6], [false, 1], [true, 6], [false, 7], [false, 3], [true, 6]
```

Идентификаторы пакетных функций имеют *protected*-атрибут, защищающий их от модификации со стороны пользователя; для обеспечения подобной защиты пользовательских функций/процедур следует использовать ***protect***-процедуру, рассмотренную выше, либо следующие две функции, обладающие более широкими возможностями:

***attributes*(B) –** тестирование наличия и типа атрибутов у **B**-выражения;
***setattribute*({B, A | B}) –** присвоение **A**-атрибутов **B**-выражению либо их отмена.

где в качестве **B**-выражения допускаются: *идентификаторы*, *списки*, *множества*, *невычисляемые* вызовы функций либо *float*-значения, тогда как **A**-атрибутом может быть любое корректное ***Maple***-выражение либо их последовательность, определяющая набор атрибутов для **B**-выражения. По ***attributes***-функции возвращается последовательность приписанных **B**-*выражению* атрибутов либо ***NULL***-значение при их отсутствии. Тогда как по ***setattrubute***-функции или приписываются **A**-атрибуты **B**-выражению, или отменяются приписанные ему при отсутствии у функции второго аргумента. Приписанные **B**-выражению атрибуты *переносятся* и на объект **W** (**W:= B;**), которому оно присваивается; при этом отмена атрибутов для одного из этих объектов автоматически отменяет их и для другого. Следует иметь в виду, что по предложению **restart** отменяются все атрибуты, приписанные выражениям в текущем сеансе. Следующий фрагмент иллюстрирует использование механизма атрибутов пакета:

```
> [attributes(sin), protect(GS), attributes(GS)];  ⇒ [protected, _syslib, protected]
> Vasco = H(x, y): setattribute('Vasco', protected, 'float');  ⇒ Vasco
> attributes('Vasco');  ⇒ protected, float
> W:= Vasco: attributes(W);  ⇒ protected, float
> setattribute(W): [attributes(W), attributes(Vasco)];  ⇒ []   (пустой список атрибутов)
> [setattribute(V, 64), setattribute(G, 59)];  ⇒ [V, G]
> [attributes(V), attributes(G)];  ⇒ [64, 59]
> setattribute('Vasco', 'protected', 'float'): restart;
> [attributes(V), attributes(G), attributes(Vasco)];  ⇒ []   (пустой список атрибутов)
```

Механизм атрибутов позволяет не только проводить их пакетную обработку, например, по распознаваемому ядром *protected*-атрибуту, но и пользователь имеет возможность организовывать достаточно интересные *условные* обработки выражений на основе приписанных им атрибутов, пример чему может дать следующий весьма простой пример:

```
> V:= 64: setattribute('V', 'integer'): [V, attributes('V')];  ⇒ [64, integer]
```

> V:= `if`(attributes('V') <> 'integer', 1995, 2006): V; $\Rightarrow$ 2006

Вместе с тем следует отметить, что аналогичный механизм *атрибутов*, поддерживаемый пакетом ***Mathematica*** [6,7], представляется нам существенно более развитым.

Функциональный идентификатор может использоваться в алгебраических скобочных конструкциях в сочетании с **@-**оператором, определяющим *композицию* функций, образуя новые функциональные выражения**.** Следующие достаточно прозрачные функциональные правила поддерживаются ***Maple-***языком, а именно**:**

> (S + G)(x, y, z), (S - G)(x, y, z), (-G)(x, y, z), (S@G)(x, y, z);

$$S(x,y,z) + G(x,y,z),\ S(x,y,z) - G(x,y,z),\ -G(x,y,z),\ S(G(x,y,z))$$

> (S@G)(x, y, z), (G@@5)(x, y, z); $\Rightarrow$ $S(G(x,y,z)),\ (G^{(5)})(x,y,z)$

> expand(%[2]); $\Rightarrow$ $G(G(G(G(G(x, y, z)))))$

> (S + G)(x[k]$k=1 .. n), (S@G)(x[k]$k=1 .. n);

$$S(x_k\ \$\ (k = 1\ ..\ n)) + G(x_k\ \$\ (k = 1\ ..\ n)),\ S(\,G(x_k\ \$\ (k = 1\ ..\ n)))$$

> (10*G@S@@2 + 17*S^2@G - S@G@H)(x, y, z)/(G@S@V@@2 - M^3@V + A@N@O)(x, y, z);

$$\frac{10\ G((S^{(2)})(x,y,z)) + 17\ S(\,G(x,y,z))^2 - S(\,G(\,H(x,y,z)))}{G(\,S((V^{(2)})(x,y,z))) - M(\,V(x,y,z))^3 + A(\,N(\,O(x,y,z)))}$$

> (sqrt@cos^2@ln + H@exp - G@(sin + cos)/(Art + Kr))(x);

$$\sqrt{\cos(\,\ln(x))^2} + H(\mathbf{e}^x) - \frac{G(\,\sin(x) + \cos(x))}{\mathrm{Art}(x) + \mathrm{Kr}(x)}$$

где "**@@n**" обозначает **n-**кратную *композицию* функции, определяемую соотношением:

$$\mathbf{(G@@n)(...) = G^{(n)}(...) = G(G(G...G(...)...))}$$

иллюстрируемым примерами предыдущего фрагмента. В терминах **@-**оператора для функции ***seq*** имеют место (*в ряде случаев весьма полезные*) соотношения, а именно**:**

$$\mathbf{\mathit{seq}(A(k), k = [B(x)]) \equiv \mathit{seq}(A(k), k = \{B(x)\}) \equiv (A@B)(x) \equiv A(B(x))}$$
$$\mathbf{\mathit{seq}(A(k), k = x) \equiv \mathit{seq}(A(k), k = B(x)) \equiv A(x)}$$

Активные и пассивные формы функций. Наряду с вычисляемыми в отмеченном выше смысле функциями, чьи идентификаторы начинаются со строчных букв, ***Maple-***язык располагает т.н. *пассивными* (*inert*) функциями (*операторами*), идентификаторы которых начинаются с заглавной буквы. ***Maple-***язык различает более **50** таких пассивных процедур/функций (***Diff, Int, Limit, Sum, Produc, Normal*** *и др.*)**,** суть которых нетрудно усматривается из самого их названия. В общем случае ***Maple-***язык рассматривает идентификаторы видов **Abb...b** и **abb...b** (**b -** *любой допустимый для имени символ*) как имена соответственно *пассивной* и *вычисляемой* функции/процедуры одного и того же назначения. Пассивные функции/процедуры имеют два основных назначения**:** (**1**) для символьных вычислений и преобразований и (**2**) для вывода в ***Maple-***документах конструкций в математической нотации. Для вычисления пассивных функций/процедур и выражений в целом служит процедура ***value*****(***Выражение***),** например**:**

> Int(sqrt((1 + x)/(1 - y)), x); simplify(value(%));

$$\int \sqrt{\frac{1+x}{1-y}}\,dx \qquad \frac{2\,(1+x)\sqrt{-\dfrac{1+x}{y-1}}}{3}$$

> Product((1 + x)^k, k = 1 .. 10); value(%); $\Rightarrow$ $\displaystyle\prod_{k=1}^{10}(1+x)^k \qquad (1+x)^{55}$

```
> Limit((1 + 1/j)^j, j = infinity);  value(%);
```

$$\lim_{j \to \infty} \left(1 + \frac{1}{j}\right)^{j} \qquad e$$

```
> Sum(k^4*ithprime(k), k = 1 .. 17)/Product(ithprime(k), k = 1 .. 10);  value(%);
```

$$\frac{\sum_{k=1}^{17} k^4 \, \mathrm{ithprime}(k)}{\prod_{k=1}^{10} \mathrm{ithprime}(k)} \qquad \frac{2568338}{1078282205}$$

Активная функция/процедура, начинающаяся со строчного символа, инициирует немедленное вычисление результата, тогда как *пассивная* функция/процедура, начинающаяся с *прописного* символа, возвращает *невычисленный* результат в стандартной математической нотации, для полного вычисления которого требуется применение ***value***-процедуры. При этом, следует отметить, что ***value***-процедура может быть распространена и на случай *пользовательских* функций. В случае невозможности вычислить выражение функция возвращает его в исходном виде. Многие из пассивных функций/процедур ***Maple***-языка широко используются в конъюнкции с ***evala***-процедурой и операторами **mod, modp1** модульной арифметики пакета.

Пассивные функции/процедуры тестируются и нашей процедурой ***ParProc*** [41,103], как это иллюстрирует нижеследующий фрагмент:

```
ParProc := proc(P::symbol)
local k, L, R, h, a, t, p, g;
option
system, remember, `Copyright International Academy of Noosphere - Tallinn - 2000`
;
    if P = 'ParProc' then return matrix([[Arguments = P::symbol],
        [locals = (k, L, R, h, a, t, p, g)], [`options` = (system, remember,
        `Copyright International Academy of Noosphere - Tallinn - 2000`)]])
    else
        L := ['Arguments', 'locals', `options`, 'rem_tab', `description`, 'globals',
            'lex_tab', 'exit_type'];
        unassign('A194219471967'), assign(R = Release1( )),
            `if`(P = 'randomize', goto(A194219471967), `10_17`), `if`(
            type(P, `module`),
            RETURN(cat(P, ` is module with exports`), [exports(P)]), `if`(
            type(P, 'procedure'), assign(h = Builtin(P)),
            assign(p = "" || P, g = `10_17`)));
        if g = `10_17` then
            if belong(op(convert(p[1], 'bytes')), 65 .. 90) then
                Sproc(`if`(length(p) = 1, ``||p,
                    ``||(Case(p[1], 'lower'))||p[2 .. -1]), 'g');
                if type(g, 'list') and member(g[2], {'builtin', 'iolib', 'Maple'}) then
                    return (proc(x)
                        try value(x( ))
                        catch : return inert_function, WARNING(
```

```
                    "<%1> is inert version of procedure/function <%2>",
                    x, cat(Case(p[1], 'lower'), p[2 .. -1]))
                end try;
                error "<%1> is not a procedure and not a module", x
            end proc)(P)
        else error "<%1> is not a procedure and not a module", P
        end if
    else error "<%1> is not a procedure and not a module", P
    end if
end if;
if type(h, 'integer') then RETURN(`builtin function`, h)
else
    try
        h := IO_proc(P);
        if type(h, 'integer') then RETURN(`iolib function`, h) end if
    catch "": NULL
    end try
end if;
A194219471967;
if P = 'writedata' then
    a := op(parse(Sub_st(["array(1)" = "'array'(1)"],
        convert([op(1, eval(writedata))], 'string'), t)[1]));
    t := 9
end if;
array(1 .. add(`if`(op(k, eval(P)) = NULL, 0, 1), k = 1 .. R + 1), 1 .. 1, [seq(
    `if`(op(k, eval(P)) ≠ NULL,
    `if`(k = 1 and t = 9, [L[k] = a], [L[k] = op(k, eval(P))]), NULL),
    k = 1 .. R + 1)])
  end if
end proc
> map(ParProc, [Int, Diff, Product, Sum, Limit, Normal]);
Warning, <Int> is inert version of procedure/function <int>
Warning, <Diff> is inert version of procedure/function <diff>
Warning, <Product> is inert version of procedure/function <product>
Warning, <Sum> is inert version of procedure/function <sum>
Warning, <Limit> is inert version of procedure/function <limit>
Warning, <Normal> is inert version of procedure/function <normal>
    [inert_function, inert_function, inert_function, inert_function, inert_function, inert_function]
> ParProc(AVZ); ParProc(avz); ParProc(Avz); ParProc(Mkdir);
Error, (in ParProc) <AVZ> is not a procedure and not a module
Error, (in ParProc) <avz> is not a procedure and not a module
Error, (in ParProc) <Avz> is not a procedure and not a module
Error, (in unknown) <Mkdir> is not a procedure and not a module
```

По установке ***infolevel*[<*Функция*>]:=n** (**n=1..3**) можно определять уровень протокола выполнения указанной функции**,** а по ***infolevel*[<*Функция*>, *_debug*]:= 3** установке определять уровень вывода отладочной информации. По умолчанию ***infolevel*-**таблица пакета

содержит только один вход ***hints* = 1** {***print*(*infolevel*); ⇒ TABLE([*hints* = 1])**}, определяя вывод ошибочных и предупреждающих сообщений**.** По приведенным установкам для указанной функции в таблице ***infolevel*** определяется соответствующий вход, обеспечивающий необходимый уровень мониторинга выполнения функции, например**:**

> infolevel[int]:= 3: int(sqrt((1 + x^2)/(1 - x^2)), x);

int/indef1: first-stage indefinite integration
int/indef1: first-stage indefinite integration
int/indef1: first-stage indefinite integration
int/algebraic2/algebraic: algebraic integration
int/indef1: first-stage indefinite integration
int/indef1: first-stage indefinite integration
int/indef1: first-stage indefinite integration
int/algebraic2/algebraic: algebraic integration
int/elliptic: trying elliptic integration
int/ellalg/ellindefinite: indefinite elliptic integration

$$\int \frac{-_X^3 + _X^2 - _X + 1}{\sqrt{1 - _X^4}}\, d_X$$

int/ellalg/ellindefinite: Step 2 -- partial fraction decomposition of 0 giving

$$\int 0\, d_X = \int 0\, d_X$$

int/ellalg/ellindefinite: Step 3 -- Hermite Reduction.
int/ellalg/ellindefinite: Step 4 -- Symbolic partial fraction reduction to Hj1's.
int/ellalg/ellindefinite: Step 5 -- Reduction of polynomial part to I0, I1, I2.
int/ellalg/ellindefinite: result of indefinite elliptic integration 1/2*(1-_X^4)^(1/2)-(1-_X^2)^ (1/2)*(1+_X^2)^(1/2)/(1-_X^4)^(1/2)*EllipticF(_X,I)+1/2*I*ln(I*_X^2+(1-_X^4)^(1/2))+(1-_X^2)^ (1/2)*(1+_X^2)^(1/2)/(1-_X^4)^(1/2)*(EllipticF(_X,I)-EllipticE(_X, I))
int/indef1: first-stage indefinite integration
int/indef1: first-stage indefinite integration
int/algebraic2/algebraic: algebraic integration
int/elliptic: trying elliptic integration
int/ellalg/ellindefinite: indefinite elliptic integration

$$\int \frac{-_X^3 - _X^2 - _X - 1}{\sqrt{1 - _X^4}}\, d_X$$

int/ellalg/ellindefinite: Step 2 -- partial fraction decomposition of 0 giving

$$\int 0\, d_X = \int 0\, d_X$$

int/ellalg/ellindefinite: Step 3 -- Hermite Reduction.
int/ellalg/ellindefinite: Step 4 -- Symbolic partial fraction reduction to Hj1's.
int/ellalg/ellindefinite: Step 5 -- Reduction of polynomial part to I0, I1, I2.
int/ellalg/ellindefinite: result of indefinite elliptic integration 1/2*(1-_X^4)^(1/2)+(1-_X^2)^ (1/2)*(1+_X^2)^(1/2)/(1-_X^4)^(1/2)*EllipticF(_X,I)+1/2*I*ln(I*_X^2+(1-_X^4)^(1/2))-(1-_X^2)^ (1/2)*(1+_X^2)^(1/2)/(1-_X^4)^(1/2)*(EllipticF(_X,I)-EllipticE(_X,I))

$$-\frac{\sqrt{-\frac{1+x^2}{-1+x^2}}\,(-1+x^2)\sqrt{1-x^2}\,\sqrt{1+x^2}\,\text{EllipticE}(x, I)}{\sqrt{-(1+x^2)\,(-1+x^2)}\,\sqrt{1-x^4}}$$

> infolevel[simplify]:= 3: simplify(%%);

simplify/do: applying simplify/sqrt function to expression
simplify/do: applying commonpow function to expression
simplify/do: applying power function to expression

$$\frac{\text{EllipticE}(x, I)}{\sqrt{1+x^2}}$$

> infolevel[dsolve]:= 3: dsolve(diff(y(x), x$3) - y(x) + sin(x)*cos(x)^2);
Methods for third order ODEs:
--- Trying classification methods ---
trying a quadrature
trying high order exact linear fully integrable
trying differential order: 3; linear nonhomogeneous with symmetry [0,1]
trying high order linear exact nonhomogeneous
trying linear constant coefficient
checking if the LODE has constant coefficients
<- constant coefficients successful

$$y(x) = \frac{1}{2920}\sin(3x) - \frac{27}{2920}\cos(3x) - \frac{1}{8}\cos(x) + \frac{1}{8}\sin(x) + _C1\,\mathbf{e}^{x}$$
$$+ _C2\,\mathbf{e}^{\left(-\frac{x}{2}\right)}\sin\left(\frac{\sqrt{3}\,x}{2}\right) + _C3\,\mathbf{e}^{\left(-\frac{x}{2}\right)}\cos\left(\frac{\sqrt{3}\,x}{2}\right)$$

> eval(infolevel); $\Rightarrow$ table([*dsolve* = 3, *int* = 3, *simplify* = 3, *hints* = 1])

Возможность мониторинга имеет особый смысл в учебных целях, предоставляя протокол получения решения, который может анализироваться на предмет изучения хода вычислений, выполняемого пакетом для получения требуемого результата.

Преобразования математических функций. Различного рода преобразования выражений составляют один из *краеугольных* камней многих разделов современных математических дисциплин и в этом отношении пакет ***Maple*** обеспечивает пользователя достаточно развитыми средствами, в большинстве своем рассматриваемыми на протяжении книги. Здесь мы представим основные средства ***Maple-***языка по *преобразованию* известных пакету математических функций, играющему чрезвычайно важную роль, прежде всего, в прикладной математике и ряде физических дисциплин, имеющих важные физико-технические приложения [8-14,32].

Для целей преобразования математических функций (*известных ядру пакета*) из одного типа в другой предназначена многоаспектная ***convert-***функция**,** рассмотренная в разделе **1.7**. Здесь мы акцентируем наше внимание на ее возможностях по конвертации одного типа математических функций в другой**.** В этом случае формат кодирования функции ***convert*** имеет следующий вид**:**

***convert*(**<*Выражение*>**,** <*Тип функции*>**)**

и преобразует входящие в исходное *Выражение* функции одного типа в функции *типа*, определяемого ее вторым аргументом**.** В качестве основных функциональных преобразований, поддерживаемых ***convert-***функцией (*дополнительно к уже рассмотренным выше*), отметим следующие**:**

* ***Bessel*{ |I|J|K|Y} -** преобразование *волновых* Айри-функций в функции Бесселя указанного типа либо функций Бесселя одного типа в функции Бесселя другого типа**;**
* ***Hankel -*** преобразование *волновых* Айри-функций и/или функций Бесселя в функции Ханкеля. В обоих случаях преобразованию подвергаются как собственно сами функции Айри и Бесселя, так и их производные. *Простейший* случай ***Bessel-***аргумента всег-

да возвращает результат, тогда как в остальных случаях возможно возвращение невычисленного выражения, т. е. выражения в его исходном виде;

* ***binomial*** **-** преобразование ***GAMMA*****-**функций и факториалов в биномиалы;
* ***erf*** **-** преобразование дополнительной ***erfc*****-**функции ошибок, ее итеративных интегралов и интегралов Доусона в основную ***erf*****-**функцию ошибок;
* ***erfc*** **-** преобразование основной ***erf*****-**функции ошибок в ее ***erfc*****-**дополнительную;
* ***GAMMA*** **-** преобразование факториальных и биномиальных функций в функцию ***GAMMA***; при этом, кодирование необязательного третьего вргумента ***convert*****-**функции позволяет определять ведущие переменные, по которым производится преобразование;
* ***StandartFunctions*** **-** преобразование функций Мейера и гипергеометрических в стандартные специальные и элементарные функции по классификации [12] (*ссылка* [65]).

Прежде чем представить следующий тип преобразования, представим средство ***Maple***-языка по заданию в его среде кусочно-определенных функций, играющих чрезвычайно важную роль во многих прикладных областях математики. Для этой цели ***Maple*****-**язык располагает функцией ***piecewise***, имеющей следующий формат кодирования:

piecewise**(<ЛУ_1>, <B1>, <ЛУ_2>, <B2>, ..., <ЛУ_n>, <Bn>** {**, <Z>, <Bf>**})

сущность которого соответствует условной конструкции (*в терминах* ***"if_then_else"-****конструкций*) следующего весьма прозрачного вида:

if <ЛУ_1> then <B1> else if <ЛУ_2> then <B2> else ... if <ЛУ_n> then <Bn> else <Bf>

По умолчанию значение **Bf** полагается *нулевым*; в качестве *логических условий* (**ЛУ**) могут выступать отношения или булевы выражения из неравенств. Тогда как в случае **Z-**выражения, отличного от условного, оно помещается в возвращаемом результате в качестве условия типа "*иначе*" (***otherwise***), как это иллюстрирует пример задания простой кусочно-определенной функции:

> R:=piecewise(x <= 0, x*a^x, (x>0) and (x<=59), x*log(x), (x > 59) and (x <= 64), x*exp(x),
exp(x)* cos(x) + sqrt(Art + Kr)); ⇒

$$R := \begin{cases} x\,a^{x} & x \le 0 \\ x\ln(x) & -x < 0 \ \textbf{and}\ x - 59 \le 0 \\ x\,\mathbf{e}^{x} & -x < -59 \ \textbf{and}\ x - 64 \le 0 \\ \mathbf{e}^{x}\cos(x) + \sqrt{Art + Kr} & otherwise \end{cases}$$

> convert(R, Heaviside);

x a^x - x a^x Heaviside(x) + x ln(x) Heaviside(x) - x ln(x) Heaviside(-59 + x) + x **e**^x Heaviside(-59 + x) - x **e**^x Heaviside(x - 64) + Heaviside(x - 64) **e**^x cos(x) + Heaviside(x - 64) (Art + Kr)^(1/2)

По ***convert*****-**функции предоставляется возможность *преобразования* кусочно-определенных функций в функции Хэвисайда и наоборот; тогда как функции ***abs***, ***sign*** и ***signum*** можно преобразовывать в функции Хэвисайда и кусочно-определенные функции.

* ***Heaviside*** **–** преобразование кусочно-определенной функции, а также функций ***sign***, ***abs*** и процедуры ***signum*** в функцию Хэвисайда;
* ***piecewise*** **–** преобразование функций ***sign***, ***abs***, процедуры ***signum*** и функции Хэвисайда в кусочно-определенные функции (*см. прилож.* **1** [12]); однако, уже ***csgn*****-**процедура определения знака для действительных и комплексных чисел не преобразуется в функцию Хэвисайда или в кусочно-определенную, хотя именно таковой и является;
* ***hypergeom*** **–** преобразование *любого* суммирования по процедурам ***sum*** и ***Sum*** в гипергеометрические функции, однако ***Maple*** не гарантирует их сходимости.

Следующий фрагмент иллюстрирует использование представленных средств функции ***convert*** для преобразования математических функций из одного вида в другой**:**

```
> convert(3*Pi*x^3*AiryAi(x) + y*BesselY(x^2, 1), 'BesselK');
```

$$3\,\pi\,x^3\,\mathrm{AiryAi}(x) + y\,\mathrm{BesselY}(x^2, 1)$$

```
> convert(2*BesselK(2*x^2, 1) + BesselJ(x^2, 1), 'Hankel');
```

$$\pi\,\mathbf{e}^{(x^2\,\pi\,I)}\,\mathrm{HankelH1}(2\,x^2, I)\,I + \frac{1}{2}\,\mathrm{HankelH1}(x^2, 1) + \frac{1}{2}\,\mathrm{HankelH2}(x^2, 1)$$

```
> convert(GAMMA(k + 3/2)/(k!*sqrt(Pi)*GAMMA(k + 10)), 'binomial');
```

$$\frac{1}{2}\,\frac{\mathrm{binomial}\left(k + \frac{1}{2}, k\right)}{\Gamma(k + 10)}$$

```
> simplify(convert(sqrt(x^2 + erfc(x))/dawson(x)*sqrt(Pi), 'erf'));
```

$$\frac{2\,I\sqrt{x^2 + 1 - \mathrm{erf}(x)}\;\mathbf{e}^{(x^2)}}{\mathrm{erf}(x\,I)}$$

```
> convert(sqrt(x^2 + erf(x) - 1)/erf(x), 'erfc');
```

$$\frac{\sqrt{x^2 - \mathrm{erfc}(x)}}{1 - \mathrm{erfc}(x)}$$

```
> convert((x! + y! + z!)/(m! + (m - n)!/m!), GAMMA, {x, y, m, n});
```

$$\frac{\Gamma(x + 1) + \Gamma(y + 1) + z!}{\Gamma(m + 1) + \frac{\Gamma(m - n + 1)}{\Gamma(m + 1)}}$$

```
> simplify(convert(abs(x) + sign(y)*z + signum(z)*y, 'Heaviside'));
```

$$-x + 2\,x\,\mathrm{Heaviside}(x) + z - y + 2\,y\,\mathrm{Heaviside}(z)$$

```
> R:= piecewise(g <= 42,64, (g <= 47) and (g > 42),59, (g <= 67) and (g > 47), 39, 2006);
```

$$R := \begin{cases} 64 & g \le 42 \\ 59 & g - 47 \le 0 \ \textbf{and} \ -g < -42 \\ 39 & g - 67 \le 0 \ \textbf{and} \ -g < -47 \\ 2006 & \mathit{otherwise} \end{cases}$$

```
> convert(R, 'Heaviside');
```

$$64 - 5\,\mathrm{Heaviside}(-42+g) - 20\,\mathrm{Heaviside}(g-47) + 1967\,\mathrm{Heaviside}(g-67)$$

```
> convert(%, 'piecewise');
```

$$\begin{cases} 64 & g < 42 \\ \mathit{undefined} & g = 42 \\ 59 & g < 47 \\ \mathit{undefined} & g = 47 \\ 39 & g < 67 \\ \mathit{undefined} & g = 67 \\ 2006 & 67 < g \end{cases}$$

```
> convert(Heaviside(t), 'piecewise');
```

$$\begin{cases} 0 & t < 0 \\ \mathit{undefined} & t = 0 \\ 1 & 0 < t \end{cases}$$

```
> convert(hypergeom([3, 5], [3/2], h), 'StandardFunctions');
```

$$\frac{-16\,h^3 + 216\,h^2 + 2346\,h + 919}{1024\,(-1 + h)^6} + \frac{1}{1024}\,\frac{(-1680\,h^2 - 1680\,h - 105)\sqrt{1 - h}\,\arcsin(\sqrt{h})}{(-1 + h)^7\sqrt{h}}$$

```
> simplify(convert(Sum(17*k/(k + 10), k = 0 .. n), 'hypergeom'));
```

$$-17\,(-5\,\mathrm{hypergeom}([2,2],[3],1)\,n^3+25\,\mathrm{hypergeom}([2,2],[3],1)\,n^2-40\,\mathrm{hypergeom}([2,2],[3],1)\,n+20\,\mathrm{hypergeom}([2,2],[3],1)-2\,\mathrm{hypergeom}([1,n-1],[n],1)\,n^2+9\,\mathrm{hypergeom}([1,n-1],[n],1)\,n+\mathrm{hypergeom}([2,n-1],[n],1)\,n^2+10\,\mathrm{hypergeom}([2,n-1],[n],1)\,n-18\,\mathrm{hypergeom}([1,n-1],[n],1)-11\,\mathrm{hypergeom}([2,n-1],[n],1)+\mathrm{hypergeom}([1,n-1],[n],1)\,n^3)\,/\,((n^2-3\,n+2)\,(n-2))$$

```
> convert(MeijerG([[1], [2]], [[0], [0]], 10*ln(t) - 2*ln(17)), 'StandardFunctions');
```

$$1+\ln(2)+\ln(5\ln(t)-\ln(17))-10\ln(t)+2\ln(17)$$

```
> map2(convert, csgn(x), ['Heaviside', 'piecewise']);   ⇒ [csgn(x), csgn(x)]
> simplify(convert(Pi*x*AiryAi(1, x^2) + y*AiryBi(3, x), 'BesselK'));
```

$$\pi\,x\,\mathrm{AiryAi}(1,x^2)+y\,\mathrm{AiryBi}(x)+y\,x\,\mathrm{AiryBi}(1,x)$$

В результате преобразования одного типа функции в другой возвращаемый результат может иметь далекий от оптимального вид, поэтому на первых порах для его *упрощения* можно использовать ***simplify-***процедуру, которая подобно ***convert-***функции многоаспектна и детальнее рассмотрена выше в контексте средств пакета по преобразованию выражений в целом. На первых же порах вполне достаточно для этих целей воспользоваться конструкцией вида ***simplify*(*convert*(F,** *Тип*)**),** по которой производится стандартное упрощение преобразованной в заданный *тип* математической **F-**функции. Однако следует иметь в виду, что данный прием может привести и к *прямо* противоположному результату**,** требуя и более сложных преобразований. С рядом особенностей выполнения ***convert-***функции можно ознакомиться в *прилож.* **1** [12].

Представленные в настоящем разделе средства ***Maple-***языка в совокупности с рядом ранее рассмотренных позволяют довольно эффективно производить обработку абстрактных *символьных* конструкций на формальном алгебраическом уровне, что играет, в частности, весьма важную роль в различного рода задачах, связанных с формальными функциональными преобразованиями различного характера**.** Из нашего опыта работы с пакетами ***Maple*** и ***Mathematica*** можно сделать вполне однозначный вывод о предпочтительности *первого* для задач символьной обработки. В заключение же настоящего раздела проиллюстрируем использование рассмотренных выше функциональных средств ***Maple-***языка по обеспечению символьных вычислений для решения одной частной, но базовой задачи динамической *теории однородных структур* (**ТОС**), представляющих собой одну из *основных* вычислительных моделей параллельной обработки информации и вычислений [1,4,12,25-27,35,36,40,43].

Современная точка зрения на **ТОС**, как на отдельную ветвь теории абстрактных бесконечных автоматов, сформировалась в 70-х годах под влиянием основополагающих работ Х. Ямада, С. Аморозо, А. Смита, А. Беркса, Х. Нишио, Р. Фольмара, Т. Тоффоли, Д. Кодда, Н. Хонда, С. Вольфрама, Э. Бэнкса, Т. Китагава, В.З. Аладьева, Я.М. Барздиня и др. В работах [92-102] приведены соображения в подтверждение роли и места **ТОС**-проблематики в структуре современной математической кибернетики и связанных с нею естественно-научных направлений. Особый интерес к **ОС**-моделям возобновился в начале 80-х годов в связи с активными работами по созданию *новых* перспективных архитектур высокопроизводительной вычислительной техники, проблеме искусственного интеллекта, робототехникой, информатикой и другими мотивациями. Наконец, предполагается, что **ОС** могут сыграть чрезвычайно важную роль в качестве концептуальных и прикладных моделей пространственно-распределенных динамических систем, из которых физические и биологические клеточные системы предтавляют интерес в

первую очередь. В этом направлении уже налицо значительная активность целого ряда исследователей, получивших весьма обнадеживающие результаты [43,104].

Однородные структуры (**ОС**) являются формализацией понятия *бесконечных* регулярных решеток (*сетей*) из идентичных конечных автоматов, которые информационно связаны друг с другом одинаковым образом в том смысле, что каждый автомат решетки может непосредственно получать информацию от вполне определенного для него конечного множества *соседних* ему автоматов. При этом, соседство понимается не в *геометрическом*, а в информационном плане. *Соседство* автоматов устанавливается постоянным для каждого автомата решетки и определяется специальным вектором – *индексом соседства*. Как правило, рассматриваются **d**-мерные регулярные решетки в Евклидовом пространстве $\mathbf{E^d}$, в целочисленные точки которого помещены копии некоторого автомата Мура. В качестве простого примера **ОС**-модели можно представить себе бесконечную клеточную бумагу, в каждой клетке которой расположена копия автомата Мура, для которого соседними являются все непосредственно примыкающие к нему автоматы, включая и его самого.

ОС функционирует в дискретные моменты времени **t** (**t = 0,1,2,...**) так, что каждый автомат решетки может *синхронно* изменять свое состояние в дискретные моменты времени **t > 0** как функция состояний всех своих соседей в предыдущий момент времени (**t - 1**). Эта *локальная* функция перехода может со временем меняться, но остается всегда постоянной для каждого автомата решетки в любой конкретный момент времени **t > 0**. Одновременное применение *локальной* функции перехода *ко всем* автоматам решетки определяет *глобальную* функцию перехода в структуре, которая действует *на всей* решетке, изменяя текущую конфигурацию состояний автоматов решетки на новую конфигурацию. Изменение *конфигураций* структуры под действием *глобальной* функции определяет *динамику* функционирования **ОС**-модели с течением времени, которая играет основную роль в исследованиях ее поведенческих свойств.

Состояния единичных автоматов **ОС** можно ассоциировать с различными понятиями, такими как состояния биологических клеток, команды (*инструкции*) клеточных микропроцессоров, символы некоторых параллельных формальных систем и др. Тогда как сама *история* конфигураций в **ОС** ассоциируется с динамикой погружаемых в структуру различного рода *дискретных* моделей, процессов, алгоритмов и явлений. Подобные модели могут быть применены в таких различных областях как: распознавание образов, машинное самовоспроизведение, морфогенез, адаптивные и динамические системы, теория эволюции и развития, искусственный интеллект и робототехника, вычислительная техника и информатика, математика, кибернетика, синергетика, физика, космология и др. Мы можем интерпретировать **ОС** не только как абстракцию биологических *клеточных* систем, но также как теоретическую основу искусственных параллельных систем обработки информации и вычислений.

С логической точки зрения **ОС** являются бесконечными абстрактными автоматами со специфической внутренней структурой, определяющей целый ряд важных свойств и допускающей использование ее в качестве новой перспективной среды моделирования различных дискретных процессов, допускающих режим максимального распараллеливания. **ТОС,** в целом, может рассматриваться как *структурная* и *динамическая* теория бесконечных абстрактных автоматов, наделенных специфической внутренней организацией, носящей качественный характер. В настоящее время **ТОС** образует *вполне* самостоятельный раздел современной математической кибернетики со своими проблематикой, методами и приложениями, а сами структуры служат формальной средой для моделирования многих дискретных процессов и явлений в различных областях науки и

техники. В нашей монографии [36] представлена архитектура **ТОС** и ее приложений с учетом достигнутых на тот момент результатов, а также основных тенденций развития данного направления исследований. Представленные материалы позволяют получить единую картину составляющих частей **ТОС** и ее приложений со всеми основными их взаимосвязями. На фоне предложенной архитектуры рельефнее вырисовывается и общая структура данного предмета исследований.

Наше дальнейшее изложение материала будет базироваться на так называемом *классическом* понятии **1**-*мерных* (**d=1**) *однородных структур* (**1-ОС**), относительно которого здесь вводится ряд основных определений. *Классическое* понятие **1-ОС** определяется как упорядоченная четверка компонент следующего вида:

$$\textbf{1-ОС} = \langle Z^1, A, \tau^{(n)}, X \rangle$$

где **A** – конечное *непустое* множество, называемое *алфавитом внутренних состояний* единичных автоматов структуры и представляющее собой множество *состояний*, которые может принимать каждый *элементарный* (*единичный*) автомат структуры. Алфавит **A** содержит так называемое *состояние покоя*, обозначаемое символом "**0**"; суть этого особого состояния будет выяснена несколько позже. Не нарушая общности, в качестве **A**-алфавита будем использовать множество состояний **A = {0, 1, 2, 3, ..., a-1},** содержащее **a** элементов – чисел от **0** до (**a - 1**).

Компонента Z^1 представляет собой множество всех **1**-мерных кортежей – *целочисленных* координат точек в евклидовом E^1 пространстве, т. е. E^1 представляет собой целочисленную решетку в E^1, элементы которой служат для пространственной идентификации *единичных* автоматов структуры. Компонента E^1 определяет однородное пространство структуры, в котором она функционирует. Естественно, что в целом ряде прикладных аспектов **ОС** их геометрия играет, порой, существенную роль, однако здесь данный вопрос не рассматривается.

В каждую точку пространства Z^1 помещается копия конечного автомата Мура, алфавит *внутренних* состояний которого есть **A.** Как известно, автомат Мура представляет собой *конечный* автомат, *выход* которого в данный момент времени **t** зависит только от его внутреннего состояния в этот же момент времени **t** и не зависит от значения его входов. В этом случае каждая точка Z^1 определяет имя или координату единичного автомата, помещенного в данную точку. Для удобства в дальнейшем будем идентифицировать точки пространства Z^1 с расположенными в них единичными автоматами. Таким образом, термины "автомат **z**" и "автомат с координатой $z \in Z^1$" будем полагать идентичными. Компонента **X,** называемая *индексом соседства* структуры, есть упорядоченный кортеж **n** элементов из Z^1, который служит для определения автоматов-*соседей* любого *единичного* автомата структуры, т.е. тех ее автоматов, с которыми данный *единичный* автомат непосредственно связан информационными каналами.

Каждый единичный автомат структуры в любой дискретный момент времени **t** может получать информацию только от своих непосредственных соседей и передавать информацию о своем текущем состоянии также только им. Таким образом, непосредственными соседями единичного автомата $z \in Z^1$ являются автоматы **z+x1,z+x2 ,..., z+xn,** где $X=\{x1,x2, ..., xn\}; xj \in Z^1$ (**j=1..n**). Индекс соседства **X** описывает единый *шаблон* соседства (*геометрический образ соседей-автоматов*) для каждого единичного **z**-автомата структуры. Он определяет позиции автоматов-соседей относительно каждого конкретного единичного **z**-автомата, который имеет с ними непосредственный *информационный* интерфейс. В дальнейшем, не нарушая общности, будем полагать, что **X**-*индекс* соседства содержит

0^1-элемент, определяющий центральный автомат шаблона соседства. Далее рассматриваются **1-ОС** с индексами соседства **X = {0, 1, ..., n-1}.**

Первые три рассмотренные компоненты **1-ОС,** а именно, **A**-алфавит состояний единичных автоматов, однородное пространство $\mathbf{Z^1}$ и **X**-индекс соседства образуют *однородную* среду, являющуюся статической частью **ОС**-модели. Данная часть описывает физическую организацию структуры и ее геометрию, но не специфицирует *взаимодействия* (*динамики*) среди составляющих ее единичных автоматов. Для определения функционирования **1-ОС** необходимо иметь возможность описывать текущие состояния всех единичных автоматов структуры в любой дискретный момент времени $\mathbf{t \geq 0}$.

Состояние всей однородной среды называется *конфигурацией* (**КФ**) **1-ОС** и представляет собой набор текущих *состояний всех* составляющих ее единичных автоматов. А именно, конфигурация **1-ОС** есть *любое* отображение $\mathbf{КФ: Z^1 \rightarrow A}$ и **C(A, 1)** обозначает множество всех таких конфигураций относительно $\mathbf{Z^1}$ и **A,** т.е. $\mathbf{C(A, 1) = \{КФ \mid КФ: Z^1 \rightarrow A\}}$. Множество **C(A,1)** включает и *пассивную* конфигурацию, содержащую автоматы только в состоянии *покоя*.

Функционирование **1-ОС** осуществляется в дискретной шкале времени **t=0,1,2,...** и определяется *локальной функцией перехода* (**ЛФП**) $\mathbf{\sigma^{(n)}}$**,** которая задает состояние *каждого* единичного **z**-автомата структуры в момент времени **t** на основе состояний всех соседних ему автоматов (*согласно* **X**-*индекса соседства*) в момент времени (**t - 1**). Иными словами, **ЛФП** есть любое отображение $\mathbf{\sigma^{(n)}: A^n \rightarrow A;}$ в дальнейшем для **ЛФП** классических структур будем использовать следующие основные обозначения:

$\mathbf{a1a2 \ldots an \rightarrow a1`}$ – можество параллельных подстановок

где **aj** – состояния любого **z**-автомата **1-ОС** и его соседей (*согласно индекса соседства* **X = {x1, x2, ..., xn}**) в момент времени (**t - 1**), а **a1`** – состояние этого **z**-автомата в следующий момент времени **t > 0**. Множество *параллельных подстановок* определяет программу (*параллельный алгоритм*) функционирования классической **ОС**-модели; параллельные подстановки представляют собой параллельный язык программирования низшего уровня в среде **ОС**-моделей. Мы будем рассматривать структуры, чьи **ЛФП** подчиняются определяющему соотношению $\mathbf{\sigma^{(n)}: 0^n \rightarrow 0,}$ т.е. структуры с ограничением на *скорость* передачи информации в них.

Одновременное применение **ЛФП** $\mathbf{\sigma^{(n)}}$ к текущей конфигурации шаблона соседства каждого единичного **z**-автомата **1-ОС** определяет *глобальную* функцию перехода (**ГФП**) $\mathbf{\tau^{(n)}}$ структуры, переводящую *текущую* $\mathbf{КФ\ c \in C(A, 1)}$ в *следующую* $\mathbf{КФ\ c\ \tau^{(n)} \in C(A, 1)}$ структуры. Именно для программной реализации глобальной функции перехода на основе заданной локальной функции и была создана процедура ***HS_1*,** представленная ниже.

```
HS_1 := proc(Co::symbol, A::set(symbol), p::symbol, n::posint)
local a, b, c, d, k, f, r;
global _LTF;
    if not belong(convert(Co, 'set1'), A) then error
        "1st argument should contain symbols from %1 only, but had received <%2>
        , A, Co
    else f := proc(a, p)
        local k, j;
            for k to length(a) do if a[k] ≠ p then break end if end do;
            for j from length(a) by -1 to 1 do if a[j] ≠ p then break end if end do;
```

```
            if k = 0 or j = 0 then p else a[k .. j] end if
        end proc
    end if;
    if member("" || p, map(convert, A, 'string')) then
        assign(a = "", d = A, b = cat(p $ (k = 1 .. n))), assign(c = "" || b || Co || b);
        if not type(_LTF, 'table') then
            seq(assign('d' = [seq(seq(cat(k, j), k = d), j = A)]), k = 1 .. n - 1),
                assign(r = rand(1 .. nops(A))));
            seq(assign(_LTF[k] = A[r( )]), k = d), assign('_LTF[b]' = p)
        end if;
        seq(assign('a' = cat(a, _LTF[``||c[k .. k + n - 1]])), k = 1 .. length(c) - n),
            assign('a' = ``||(f(a, "" || p))), a
    else error "3rd argument should belong to set %1 but had received <%2>",A, p
    end if
end proc
```

> HS_1(abcabcabcabcabc, {a,b,c}, c, 3); ⇒ *baacaacaacaacaab*

> HS_1(ccccccccccccccccccccccccc, {a,b,c}, c, 3); ⇒ *c*

```
> eval(_LTF);
  table([ccb = a, cba = c, aac = a, aca = c, ccc = c, bac = c, bca = c, cac = a, cca = b, abc = a,
  aab = b, bbc = a, bab = b, cbc = b, cab = a, acc = a, abb = a, aaa = a, bcc = b, bbb = c, baa = c,
  cbb = c, caa = a, acb = b, aba = c, bcb = a, bba = b])
> S:= abbabacbabccab:  for k to 9 do  assign('S' = HS_1(S, {a, b, c}, c, 3)), S  end do;
```

baabbccbcbabbaab
accbaababacbabcbab
baaaccbcbccbcbaacbab
accaaaababbbabaccabcbab
baabaaabcbabbccabaaacbab
accbccabaacbaabbaccaabcbab
baaabbbaccabccbabcababaacbab
accabacbcabaabacbacacbccabcbab
baabaccbbcaccbccbcccabbbbaaacbab

Предыдущий фрагмент представляет исходный текст процедуры и примеры ее применения для *генерации* конечных конфигураций из некоторой начальной конфигурации структуры **1-ОС**. В качестве формальных аргументов процедуры **HS_1(Co,A,p,n)** выступают кратко рассмотренные выше такие элементы классических **1-ОС** как начальная **КФ** (**Co**), алфавит внутренних состояний единичного автомата (**A**), состояние покоя (**p**) и размер связного шаблона соседюства структуры (**n**). Для генерации последовательностей конфигураций (*историй развития начальных конфигураций*) *необходимо* определить *локальную* функцию перехода (**ЛФП**), которая определяется стохастическим образом на основе **rand**-генератора псевдослучайных целых чисел. **ЛФП** определяется глобальной таблицей ***_LTF,*** чьи *входы* определяют все допустимые конфигурации шаблона соседства над **A**-алфавитом внутренних состояний, а *выходы* – соответствующие им состояния, получаемые центральным автоматом шаблона в следующий момент времени. Сгенерированная один раз таблица ***_LTF,*** сохраняется в течение всего текущего сеанса до ее переопределения, например, по **restart**-предложению. Ее текущее состояние можно получать по вызову функции **eval(*_LTF*).** Следует отметить, что для обеспечения отмеченного выше определяющего соотношения $\sigma^{(n)}: 0^n \to 0$, после генерации таблицы производится соответствующее переопределение одного ее входа (*в приведенном выше фрагменте*

в таблице он выделен ***bold****-шрифтом*). Таким образом, задав начальную конфигурация на конечном отрезке единичных автоматов **1-ОС** (**Co**), алфавит внутренних состояний **A,** состояние покоя **p∈A** и размер шаблона соседства (**n**), мы вызовом процедуры **HS_1(Co, A, p, n)** получаем следующую **КФ** структуры, т.е. конфигурацию в следующий момент времени. Применяя многократно вызов процедуры (*как это показано фрагментом*), получаем историю начальной **КФ Co** с течением *времени* под действием **ГФП,** определяемой заданной **ЛФП**. При этом, на **Co,** состоящей только из состояний покоя, вызов процедуры возвращает состояние покоя.

Процедура составляет лишь ядро, обеспечивающее *генерацию* последовательностей **КФ,** которое, в свою очередь может быть расширено другими интересными функциями по исследованию *динамики историй* таких конечных конфигураций в зависимости от начальной **КФ,** глобальной функции перехода и т.д. В качестве весьма полезного упражнения читателю рекомендуется разобраться в организации процедуры ***HS_1,*** использующей ряд полезных приемов. Процедура использует две наши нестандартные процедуры ***belong*** и ***`convert/set1`*** из *Библиотеки* [103], с которыми можно ознакомиться в демоверсии дибо в самой библиотеке, доступными по адресам, указанным в [41,108,109].

```
convert/set1 := proc(expr::anything)
local a, k, j, L, ω, ψ;
    assign( ω = ( a → `if`( type( a, 'equation' ), rhs( a ), a ) ),
        ψ = ( a → `if`( type( a, 'numeric' ), a, Evalf( a ) ) ) );
    if type( expr, 'Vector' ) then map( ω, { op( op( 2, expr ) ) } )
    elif type( expr, 'Matrix' ) then
        { seq( [ seq( expr[ j, k ], k = 1 .. op( 1, expr )[ 2 ] ) ], j = 1 .. op( 1, expr )[ 1 ] ) }
    elif type( expr, 'Array' ) then { seq( [ seq( expr[ j, k ], k = 1 .. rhs( op( 2, expr )[ 2 ] ) ) ],
        j = 1 .. rhs( op( 2, expr )[ 1 ] ) ) }
    elif type( expr, { 'list', 'set' } ) then { op( expr ) }
    elif ttable( expr ) then { op( came( convert( eval( expr ), 'string' )[ 7 .. -2 ] ) ) }
    elif type( expr, 'vector' ) then { seq( expr[ k ], k = 1 .. rhs( op( 2, eval( expr ) ) ) ) }
    elif type( expr, 'array' ) then { seq(
        [ seq( expr[ j, k ], k = 1 .. rhs( op( 2, eval( expr ) )[ 2 ] ) ) ],
        j = 1 .. rhs( op( 2, eval( expr ) )[ 1 ] ) ) }
    elif type( expr, 'range'( 'realnum' ) ) then { op( [
        assign( L = [ seq( k, k = ψ( lhs( expr ) ) .. ψ( rhs( expr ) ) ) ] ),
        `if`( belong( ψ( rhs( expr ) ), L ), L, [ op( L ), ψ( rhs( expr ) ) ] ) ] ) }
    elif type( eval( expr ), { 'symbol', 'string' } ) then
        { op( [ assign( 'a' = cat( "", expr ) ), seq( convert( k, whattype( expr ) ), k = a ) ] ) }
    elif type( expr, 'procedure' ) then
        try convert( convert( expr, CompSeq ), 'set' )
        catch "": { seq( op( k, eval( expr ) ), k = 1 .. 8 ) }
        end try
    elif type( expr, `module` ) then convert( parmod( expr ), 'set' )
    else { expr }
    end if
end proc
> V:= Vector([a, b, c, d, e, f, h]): V1:=Vector[row]([a, b, c, d, e, f, h]): V, V1:
> convert(V, 'set1'), convert(V1, 'set1');   ⇒ {f, a, b, c, d, e, h}, {f, a, b, c, d, e, h}
```

5.5. *Управление форматом вывода результатов вычисления выражений*

Если не определено противного**,** то ***Maple*** выводит результаты вычисления выражений в нотации и формате, стандартных для ***Output****-параграфа***.** Однако по целому ряду причин, особенно при необходимости последующего использования результатов вычислений в среде других **ПС** (*системы программирования* ***C***, ***C++***, ***Fortran*** *и др.*) либо при подготовке материала к публикации принятыми издательскими системами (*например*, ***TeX****-процессором для математических изданий*), требуется переформатирование полученных в среде пакета результатов либо изменения нотации представления самих числовых значений. Для этих целей язык располагает рядом *специальных* форматирующих функций, рассматриваемых в настоящем разделе. Ниже предполагается, что говоря о **B-***выражении* (*как основном аргументе* ***P****-функции вывода*), будем иметь в виду, что *вызов* ***P*****(B)** вычисляет, при необходимости, **B-**выражение и форматирует именно результат этого вычисления, а не само выражение**,** за исключением случаев, когда **B-**выражение само является *результатом* вычисления либо *невычисляемо*.

Основной функцией вывода и *форматирования* результатов вычисления выражений является ***print-***функция, имеющая следующий простой формат кодирования**:**

print(<*Выражение*_1>**, <***Выражение*_2>**, ..., <***Выражение*_n>**)**

где в качестве "*Выражения_j*" может выступать произвольное ***Maple-****выражение*. Функция вычисляет все передаваемые ей фактические выражения в качестве элементов *последовательности*-аргумента, выводит на печать их значения в требуемом формате и в случае успешного завершения возвращает *NULL*-значение. По вызову функции ***print*****()** выводится *пустая* строка. Элементы выводимой последовательности значений разделяются запятой. Управление форматом вывода осуществляет *prettyplot*-параметр ***interface-***процедуры, рассмотренной выше. В зависимости от значения параметра *prettyprint* определяется следующий формат вывода результатов вычисления последовательности выражений, определенной в качестве ее аргумента (*табл.* **13**)**.**

Таблица 13

prettyprint=	***Формат вывода Maple-выражений:***
0	*линейный входной* ***Maple-***формат (*аналогично* ***lprint***)
1	*двумерный символьно-ориентированный* ***Maple-***формат
2	*двумерный выходной* ***Maple-***формат
3	*двумерный выходной* ***Maple-***формат редактирования

Во всех случаях в качестве разделителя выводимых по ***print-***функции значений принимается символ *запятой* (**,**). По установке *prettyprint* **= 1** производится вывод в двумерном символьно-ориентированном ***Maple-***формате. По установке *prettyprint* **=** 2 формат вывода соответствует формату ***Output-****параграфа*, а по установке *prettyprint* **=** 3 (*по умолчанию*) поддерживается также режим редактирования результата. В случае вывода *больших* выражений для удобства используются **%-***метки*, обозначающие отдельные кратные подвыражения *выходного* выражения. Режим формирования **%-***меток* задается установками {*labeling*, *labelwidth*}-параметров процедуры ***interface***. При выводе содержимого массивов, таблиц, процедур и операторов в ***print-***функции следует указывать только их идентификаторы без индексирования.

Действие ***lprint*-функции,** имеющей тот же формат кодирования, что и ***print*-функция**, аналогично действию *второй*, как если бы для нее была определена установка *prettyprint* = 0 для процедуры ***interface***. Следующий фрагмент иллюстрирует вышесказанное**:**

> interface(prettyprint = 0): print(sqrt(Art^2 + Kr^2), (S + A)/(V + G), Art/Kr);

(Art^2+Kr^2)^(1/2), (S+A)/(V+G), Art/Kr

> interface(prettyprint = 1): print(sqrt(Art^2 + Kr^2), (S + A)/(V + G), Art/Kr);

$$\left(Art^{2} + Kr^{2}\right)^{1/2}, \frac{S + A}{V + G}, \frac{Art}{Kr}$$

> interface(prettyprint = 2): print(sqrt(Art^2 + Kr^2), (S + A)/(V + G), Art/Kr);

$$\sqrt{Art^{2} + Kr^{2}}, \frac{S + A}{V + G}, \frac{Art}{Kr}$$

> interface(prettyprint = 3): print(sqrt(Art^2 + Kr^2), (S + A)/(V + G), Art/Kr);

$$\sqrt{Art^{2} + Kr^{2}}, \frac{S + A}{V + G}, \frac{Art}{Kr}$$

> x:= sqrt(a + Pi*b): y:= exp(x)*h - (S + A)/(V + G): z:= (a + b*gamma)/(c + d*I)^(Art + Kr):
> W:= array(1..2, 1..2, [[Art, V], [G, Kr]]): interface(prettyprint = 2): print(W, z), lprint(W, z);

$$\begin{bmatrix} Art & V \\ G & Kr \end{bmatrix}, \frac{a + b\,\gamma}{(c + d\,I)^{(Art + Kr)}}$$

W, (a+b*gamma)/((c+I*d)^(Art+Kr))

> interface(prettyprint = 2); print(x, y, zx), lprint(x, y, z);

$$\sqrt{a + \pi\,b}, \mathbf{e}^{\left(\sqrt{a + \pi\,b}\right)}\,h - \frac{S + A}{V + G}, zx$$

(a+Pi*b)^(1/2), exp((a+Pi*b)^(1/2))*h-(S+A)/(V+G), (a+b*gamma)/((c+I*d)^(Art+Kr))

> L:= [17, Art, Kr, 10, Sv, 39, Arn, 44, Agn, 59, Avz, 64]: print(L, L, L); lprint(L, L, L);

%1, %1, %1

%1 := [17, Art, Kr, 10, Sv, 39, Arn, 44, Agn, 59, Avz, 64]

[17, Art, Kr, 10, Sv, 39, Arn, 44, Agn, 59, Avz, 64], [17, Art, Kr, 10, Sv, 39, Arn, 44, Agn, 59, Avz, 64], [17, Art, Kr, 10, Sv, 39, Arn, 44, Agn, 59, Avz, 64]

> print(sqrt), lprint(sqrt);

proc(x::*algebraic*, f::identical(*symbolic*)) ... **end proc**

sqrt

В отличие от ***print*,** функция ***lprint*** не выводит текстов ***Maple*-процедур;** однако она, как и первая, позволяет выводить выражения в корректном входном ***Maple*-формате**, что в общем случае позволяет использовать их в ***Input*-параграфах.** Элементы выводимой по ***lprint*-функции** строки разделяются *запятыми*. При установке *prettyprint* **=** 0 ***Maple*-язык** выводит все выражения**,** используя ***lprint*-функцию.** При этом, ***lprint*-функция** подобно ***print*-функции** возвращает ***NULL*-значение**, что не позволяет ссылаться на результат ее вызова по {%|%%|%%%}-оператору. Вместе с тем, *в отличие* от ***lprint*-функции** по функции ***print*** длинные выражения выводятся в терминах **%-***меток***,** как это иллюстрирует предпоследний пример предыдущего фрагмента.

Наконец, наиболее развитым средством *форматирования* результатов вычисления выражений ***Maple*** является группа (***printf*-***группа*) из четырех функций, реализованных на ***C*-**языке и имеющих следующие форматы кодирования их вызова**:**

(1) ***printf*(**<*Формат*>**,** <ПВ>**)**
(2) ***fprintf*(**<*Файл*>**,** <*Формат*>**,** <ПВ>**)**
(3) ***sprintf*(**<*Формат*>**,** <ПВ>**)**
(4) ***nprintf*(**<*Формат*>**,** <ПВ>**)**

Все четыре средства **–** ***printf***-процедура и три *iolib*-функции ***fprintf***, ***sprintf*** и ***nprintf*** **–** выводят заданную последовательность ***Maple***-выражений (**ПВ**) в формате, определяемом специальной *форматирующей строкой* (*Формат*), соответственно на: (**1**) *стандартное* устройство вывода (*как правило на экран*; ***printf***), (**2**) в *файл* (***fprintf***), заданный своим *спецификатором*, и (**3**) в *строчную* структуру (***sprintf***, ***nprintf***) в качестве возвращаемого результата. *Форматирующая строка* состоит из %-*спецификаторов*, разделенных пробелами, запятыми и т.д., и имеющих структуру следующего формата:

%{<*Флажки*>}{<*Длина*>}{.<*Точность*>}<*Код*>

Кроме параметра "*Код*" остальные параметры %-спецификации необязательны. В качестве значений для *флажков* используются *символы*, имеющие следующую *смысловую* нагрузку, а именно:

(**+**) - вывод числового значения с лидирующими знаками {**+** | **-**};
(**-**) - выравнивание выводимого значения по левому краю поля;
(***пробел***) - вывод значения с лидирующим пробелом в зависимости от знака;
(**0**) - заполнение поля выводимого значения нулями {*слева* | *справа*}; при наличии флажка (**-**) действие **0**-флажка подавляется.

Параметр "*Длина*" %-спецификатора определяет целое число, задающее минимальное количество символов, отводимых под выводимый результат; если же вывод имеет меньшую длину, то производится дополнение его *пробелами* слева, или *нулями* при указании **0**-флажка. Параметр "*Точность*" %-спецификатора определяет количество цифр, выводимых после десятичной точки числа, или максимальную длину поля вывода для *строковых* и *символьных* значений. Как "*Длина*", так и "*Точность*" могут задаваться (*)-символом, идентифицирующим тот *факт*, что значения для них должны браться на основе соответствующих значений элементов **ПВ.** Параметр "*Код*" определяет тип форматируемой конструкции и его допустимые значения определяются сводной табл. **14.**

Таблица 14

Код	***Тип форматируемой Maple-конструкции:***
d	десятичное целое число со знаком
0	целое **8**-ричное число без знака
{**X** | **x**}	целое **16**-ричное число без знака; при {**X** | **x**}-*значении* для цифр используются соответственно буквенные обозначения {**A÷F** | **a÷f**}
{**E** | **e**}	десятичное *float*-число в научной нотации; соответственно {**E** | **e**}-основание
f	число *float*-типа с фиксированной десятичной точкой
{**G** | **g**}	соответствует значению ***кода*** согласно форматируемого **α**-результата, формула вычисления которого представлена в [12]
c	единственный символ, имеющий {*string* | *symbol* | *name*}-тип
s	строка {*string* | *symbol* | *name*}-типа длины "*Длина*" ≤ **L** ≤ "*Точность*"; "*Длина*" задает длину поля выводимой строки, а "*Точность*" - ее максимальную длину
{**a** | **A**}	произвольная ***Maple***-конструкция; если определены "*Длина/Точность*"-параметры, то производится усечение по длине аналогично случаю **s**-кода. Возврат производится в соответствии с ***Maple***-синтаксисом; для %**A**-кода конструкция выводится без ограничивающих ее кавычек, даже если в исходном виде они были
m	произвольный ***Maple***-объект выводится в формате ***m***-файла; сказанное относительно %{**a** | **A**}-кода имеет силу и для данного случая
%	производится вывод %-символа; для него не требуется выражения в **ПВ**

Так как по %**A-**коду производится опускание ограничивающих выражения апострофов и кавычек, даже если они и необходимы для их корректности, то использование данного *кода* форматирования требует внимательности. С другой стороны, %**A-***код* весьма полезен при необходимости, в частности, помещения ***Maple-***конструкций в файл с целью последующей их загрузки по **read-***предложению* и автоматическим вычислением с непосредственным доступом к результатам, как это иллюстрирует следующий фрагмент**:**

```
> fprintf("C:/ARM_Book/Academy/Artur.17", `%A`, "AGN:=evalf(Pi + Catalan);"): close(0):
> read  "C:/ARM_Book/Academy/Artur.17": AGN;   ⇒ 4.057558248
> Z:= "Art:= proc() local k, S;  S:= []: for k to nargs do S:= [op(S), args[k]] end do; if nops(S)
  <> nargs then RETURN(""Несоответствие между числом аргументов и длиной S-списка"", S, args) else S end if end proc;": F:= "C:/ARM_Book/Academy/Kristo.10":
> fprintf(F,`%A`,Z): close(F): read(F): Art(64, 59, 39, 10, 17);   ⇒ [64, 59, 39, 10, 17]
```

Описанным способом можно помещать в файл с возможностью последующей загрузки отдельные ***Maple-***процедуры или их наборы, а также полностью ***Maple-***программы, как это иллюстрирует последний пример фрагмента. В данном примере *определение* процедуры ***Art*** в *строчном* формате присваивается **Z-**переменной и по ***fprintf-***функции сохраняется в заданном файле, из которого по **read-**предложению загружается, автоматически вычисляя определение процедуры в текущем сеансе и делая ее доступной.

Если форматируемое значение имеет *hfarray*-тип, то выводятся все элементы массива. В этом случае форматирование его отдельных элементов определяется значениями параметров "*Длина*" и "*Точность*" %-спецификатора.

При этом, следует иметь в виду, что ***printf-***процедура не производит автоматического *перевода строки и возврата каретки***.** Поэтому для обеспечения данной операции следует в конце (*или в нужном ее месте*) форматирующей строки кодировать конструкцию следующего вида **"\n",** определяющую данную операцию. Более того, вывод по ***printf-***процедуре на экран не управляется *screenwidth*-параметром ***interface-***процедуры, о котором шла речь выше. Следует также иметь в виду, что результаты использования значений %{**o,x,X**} *код*-параметра определяются используемым типом процессора и на разных **ПК** могут достаточно заметно различаться на значениях *float*-типа.

При наличии для ***printf-***процедуры в качестве форматируемого выражения последовательности выражений (**ПВ**) каждому из них в форматирующей строке должен соответствовать свой %-спецификатор. В противном случае оставшиеся без %-спецификаторов выражения не выводятся. Исключение составляет *спаренный* %-символ, определяющий не спецификатор, а вывод %-символа. Между *код*-параметром %-спецификатора и следующим %-спецификатором (*либо концом форматирующей строки*) можно кодировать *любую* последовательность символов (*при этом, символы кавычки «*"*» и «******» *должны дублироваться*), выводимую после соответствующего результата *вычисления* выражения из **ПВ.** Подобная возможность может быть весьма полезным подспорьем при оформлении вывода форматируемых конструкций, как это хорошо иллюстрирует следующий достаточно простой фрагмент**:**

```
> printf(`%s<Пример возможного вывода строки>\n`, ``);  h:=`____`:  k:=`|`:
  printf(`% 22s \n`, `Таблица возрастов`);   h:=`____`:  k:=`|`:  printf(`% 22s \n`,
  `Таблица возрастов`);  printf(`%s%+4s%+4s_%+4s_%+4s_%+4s_%+4s_\n`, `___`, h, h, h,
  h, h, h);  printf(`%s% 3s|% 4s|% 4s|% 4s|% 4s|%4s|\n`, k, V, G, S, Art, Kr, Arn);
  printf(`%s%+4s%+4s_%+4s_%+4s_%+4s_%+4s_\n`, `___`, h, h, h, h, h, h, h);
  printf(`%s%-3a|%-4a|%-4a|%-4a|%-4a|%-4a|\n`, k, 64, 59, 39, 17, 10, 44);
  printf(`%s%+4s%+4s_%+4s_%+4s_%+4s_%+4s_\n`, `___`, h, h, h, h, h, h);
```

<Пример возможного вывода строки>
Таблица возрастов

| V | G | S | Art | Kr | Arn |

| 64 | 59 | 39 | 17 | 10 | 44 |

> printf(`%s<Комментарий>%s`, `Выражение_1`, `Выражение_2`);
Выражение_1<Комментарий>Выражение_2
> printf(`Произвольный комментарий`);
Произвольный комментарий
> print(`Произвольный комментарий`);
Произвольный комментарий
> G:= "SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS":
> printf(%); ⇒ *NULL*
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
> nprintf(%); ⇒ *NULL*
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
> sprintf("Academy of Noosphere"): %; ⇒ "Academy of Noosphere"

Строку можно выводить и по конструкции вида ***printf*("%s<*Строка*>", "").** В случае определения **a-***кода* в **%-**спецификаторе форматирующей строки не рекомендуется кодировать (*если нет такой необходимости*) значение параметра **"***Точность***",** т.к. может произойти *усечение* выводимого результата. По процедуре ***printf*("<*Строка*>")** производится вывод произвольной ***Maple-***строки, тогда как по вызову функции ***sprintf*("<*Строка*>") -** *возврат* произвольной ***Maple-***строки, т.е. оба средства в этом случае аналогичны рассмотренной выше ***print-***функции с очевидными *отличиями*. Для *символов* и *строк* по флажку {**+**|**-**} производится *выравнивание* выводимого результата по {*правой*|*левой*} границе поля. Ряд особенностей, присущих указанным средствам, рассматривается в [11,12]. Принцип действия ***printf-***процедуры распространяется и на остальные функции группы ***sprintf*** и ***fprintf*** с очевидными отличиями, рассматриваемыми ниже.

По ***sprintf-***функции результат форматирования возвращается в строчном представлении, а по ***fprintf-***функции **-** в заданный своим спецификатором *файл* или приписанный ему логический канал ввода/вывода с возвращением числа переданных символов. Вызов ***printf*(...)** функции эквивалентен вызову ***fprintf*(*default*,...).** Здесь представлены только форматирующие возможности функции, *полностью* переносимые на выводимые в файл результаты. Функция ***nprintf*** аналогична ***sprintf-***функции с тем лишь отличием, что возвращаемый ею результат может быть *symbol*-типа, если такого же типа является форматируемое значение, что представляется важным для возможности обеспечения *последующих* вычислений с данным результатом. Следующий комплексный пример иллюстрирует применение всех перечисленных форматирующих средств функций группы ***printf*** на примере простых ***Maple-***выражений**:**

> printf(`%s: %c|% 4.3s|% 4X|%+12.8E|%08.3f|%+6.2G|`, `Проверка функции`, `G`, `AVZ`, 39, evalf(Pi), 17, 350.65);
Проверка функции: G| AVZ| 27|+3.14159265E+00|0017.000|+3.51E+02|
> printf(`%s: %c|% 4.3s|% .7a|\nПеренос: %m|%A|%%350|`, `Проверка функции`, `G`, `AVZ1942`, sqrt(x), evalf(Pi), e^x);
Проверка функции**:** G| AVZ|x^(1/2)|
Перенос**:** $"+aEfTJ!"*|e^x|%350|

```
> `Результат форматирования`:= sprintf(`%s: %c|% 4.3s|%4X|%+12.8E|%08.6f|%+
    6.2G|`, `Проверка функции`, `G`, `AVZ`, 10, evalf(Pi*Catalan), 17, 350.65):
> `Результат форматирования`;
"Проверка функции: G| AVZ|  A|+2.87759078E+00|17.000000|+3.51E+02|"
> fprintf("C:/ARM_Book/Academy/Report..99", "%s: %c|% 4.3s|% .7a|\n Перенос: %m|
  %A|%%120|", `Проверка функции`, `G`, `AVZ64` ,sqrt(x), evalf(Pi), e^x);  ⇒ 65
```

При записи в файл (*указанный своим спецификатором*) по функции ***fprintf*** отформатированных выражений следует иметь в виду, что *новый* файл открывается на *запись* в режиме дописывания (***append-****режим записи*) как файл ***TEXT-***формата, а при записи в уже существующий его содержимое полностью обновляется. При этом, файл не закрывается и последующая попытка загрузить его по **read-**предложению инициирует ошибочную информацию, не отражающую суть особой ситуации *"конец файла"***.** Поэтому в данной ситуации требуется *предварительное* закрытие файла, как это иллюстрирует следующий фрагмент записи по ***fprintf-***функции в файл определения ***Kr-***процедуры с *последующей* загрузкой его в память и автоматическим вычислением.

```
> F:="C:/ARM/Academy/Books2006": fprintf(F, "%A", "Kr:= proc() [args] end proc;"); ⇒ 28
> read(F): Kr(64, 59, 39, 44, 10, 17, 2006, 65);  ⇒ Kr(64, 59, 39, 44, 10, 17, 2006, 65)
Error, "C:/ARM/Academy/Books2006" is an empty file
> close(F); read(F): Kr(64, 59, 39, 44, 10, 17, 2006, 65);  ⇒ [64, 59, 39, 44, 10, 17, 2006, 65]
```

Данное обстоятельство следует иметь в виду при использовании ***fprintf-***функции.

Ввиду наличия достаточно развитых средств по конвертации одного типа выражений в другой (***convert-****функция*) и функциональных средств ***printf-***группы пользователь имеет вполне приемлемые возможности (*при наличии соответствующего навыка*) для обеспечения *вполне* удовлетворительного форматирования результатов своих работ, прежде всего, математического характера, в среде ***Maple-***языка, включая подготовку их к публикации. В этом отношении может быть предложен один практически полезный прием, упрощающий процедуру подготовки ***Maple-***документа к публикации, *неоднократно* нами используемый ранее.

Для обеспечения возможности оформления в документе формульных конструкций, содержащих функции ***Maple-***языка, можно использовать следующий искусственный прием. Любую *английскую* букву идентификатора такой функции следует заменить на одинаковую по написанию букву русского алфавита, определяя такой идентификатор неизвестным для языка пакета. Например, заменив в именах процедуры ***solve*** и функции ***nops*** английскую букву "**o**" на *русскую*, получаем формульную конструкцию требуемого вида. Во втором примере в имени ***evalf-***функции английская "**a**" заменена на *русскую* "**а**". В качестве такого типа замен могут быть успешно использованы *русские* буквы {**а, е, к, о, р, с, у, х**}, например:

```
> R:= sum((x - solve(P, x))[k], k=1 .. nops([solve(P, x)]));
```

$$R := \sum_{k=1}^{\mathrm{nops}([\mathrm{solve}(P,x)])} (x - \mathrm{solve}(P,x))_k$$

```
> sqrt(evalf(Pi*Art^2 + gamma*Kr^2));  ⇒ √(evàlf(π Art^2 + γ Kr^2))
```

Между тем, следует отметить, что *искушенный* пользователь в рамках средств ***Maple-***языка может обнаружить массу подобных *искусственных* приемов, весьма полезных в оформительских целях. Ниже вопросы форматирования будут детализироваться в прикладных аспектах.

Глава 6. *Средства ввода/вывода Maple-языка*

Являясь встроенным языком программирования в среде пакета, ориентированного, в первую очередь, на символьные вычисления (*компьютерная алгебра*) и обработку, ***Maple***-язык располагает относительно ограниченными возможностями по работе с данными, находящимися во внешней памяти **ЭВМ**. И в этом отношении ***Maple***-язык существенно уступает традиционным языкам программирования ***C, Basic, Fortran, Cobol, PL/1, ADA, Pascal*** и др. Вместе с тем, ориентируясь, в первую очередь, на решение задач математического характера, ***Maple***-язык предоставляет набор средств для доступа к файлам данных, который вполне может удовлетворить достаточно широкий круг пользователей физико-математических приложений пакета. В предлагаемой главе средства доступа к файлам будут рассмотрены довольно детально, по полноте изложения в значительной степени перекрывая поставляемую с пакетом документацию [79-85], а также [54-62]. С целью развития *системы доступа* к файловой системе компьютера нами был разработан целый ряд эффективных средств, представленных в нашей *Библиотеке* [41,103,108,109]. Начиная с ***Maple 9,*** пользователю предоставляются дополнительные средства работы с файлами данных, поддерживаемые процедурами модуля **FileTools,** однако они существенно уступают средствам, предоставляемым нашей *Библиотекой* [41,103,108,109].

6.1. *Средства Maple-языка для работы с внутренними файлами пакета*

Средства ***Maple***-языка обеспечивают доступ пользователя к файлам нескольких типов, которые можно условно разделить на две большие группы: *внутренние* и *внешние*. При обычной 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 детализировать понятие *спецификатора файла* (**СФ**), кратко упоминавшегося выше. Как уже отмечалось **СФ** определяет *полный путь* к искомому файлу в файловой системе **ЭВМ.** Как правило, он кодируется в виде значения {*string*, *symbol*}-типа и для платформы {***DOS*** | ***Windows***} имеет следующий формат кодирования:

<*УВВ*>**:**{\\|/}<*Подкаталог*_1>{\\|/}<*Подкаталог*_2>…{\\|/}<*Подкаталог*_n>{\\|/}<*Файл*>

где: *УВВ* определяет логическое имя устройства (*например*, *жесткий диск*, CD и т.д.), *Подкаталог*_к – подкаталог файловой системы **ЭВМ** и *Файл* – имя искомого файла в формате <*Основное имя*>{**.**<*Расширение имени*>}. Символ {\\|/} служит в качестве разделителя в определении *пути* и по *dirsep*-параметру функции ***kernelopts,*** обеспечивающей механизм связи между пользователем и ядром ***Maple,*** можно получать его значение по *умолчанию*: **> kernelopts(dirsep);** ⇒ "\\". Данное значение не переопределяемо.

Под *текущей* понимается цепочка подкаталогов, с которой работают средства доступа пакета по *умолчанию*, т.е. в случае указания в качестве **СФ** только *имени* файла. По функции ***currentdir*()** возвращается значение текущей цепочки, тогда как по вызову функции ***currentdir*(**<*Цепочка*>**)** устанавливается указанная *цепочка* в качестве текущей:

> currentdir(), currentdir(`D:/Software/Academy`), currentdir();

"C:\Program Files\Maple 10", "C:\Program Files\Maple 10", "D:\Software\Academy"

Как иллюстрирует пример, при установке новой текущей цепочки подкаталогов функция ***currentdir*** возвращает предыдущее значение для текущей цепочки, что позволяет при необходимости 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 только для *внешних* файлов пакета, т.е. данная функция используется с рассматриваемыми ниже функциями доступа к внешним файлам данных пакета.

Между тем, при попытке определить текущим *несуществующий* каталог вызов функции ***currentdir*** вызывает ошибочную ситуацию, например:

```
> currentdir(`C:/Tallinn/Academy/Grodno`);
Error, (in currentdir) file or directory, C:/Tallinn/Academy/Grodno, does not exist
```

Поэтому, 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( clsd( CFF( b )[ 1 .. -2 ], "\" ) ), op( [ MkDir( b ), currentdir( b ), c ][ -1 ] ) ) ) )
end proc
> Currentdir(), Currentdir(`C:/Tallinn/Academy/Grodno`), Currentdir(), Currentdir();
                    "c:\program files\maple 8", "c:\program files\maple 8",
                  "c:\tallinn\academy\grodno", "c:\program files\maple 8"
```

Вызов ***Currentdir*(F)** делает *текущим* каталог, определенный *полным* путем к нему **F;** при этом, если путь **F** 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 пакета ***Maple.*** Во многих случаях работы с файловой системой компьютера данная процедура предпочтительнее стандартной. Процедура, в свою очередь, существенно использует нашу процедуре ***MkDir*** [41,103,108,109], которая существенно расширяет стандартную функцию ***mkdir,*** предназначенную для создания каталогов.

```
MkDir := proc( F::{ string, symbol } )
local cd, r, k, h, z, K, L, Λ, t, d, ω, u, f, s, g, v;
    s := "Path element <%1> has been found, it has been renamed on <%2>"
    if Empty( F ) then return NULL
    elif type( F, 'dir' ) and nargs = 1 then return CF( F )
    elif type( F, 'file' ) and nargs = 2 then return CF( F )
    else cd := currentdir( )
    end if;
    u, K := interface( warnlevel ), CF( F );
    assign( Λ = ( x → close( open( x, 'WRITE' ) ) ) ), assign( L = CFF( K ), ω = 0 );
```

```
    assign( r = cat( L[ 1 ], "\" )),
        `if`( nargs = 2 and args[ 2 ] = 1, assign( 'L' = L[ 1 .. -2 ], 'ω' = L[ -1 ], t = 1 ), 1 );
    if L = [ ] then
        assign( g = cat( r, r[ 1 ] ), v = cat( r, "_", r[ 1 ] ));
        if t ≠ 1 then return r
        elif type( g, 'dir' ) then return null( Λ( v )), v
        elif type( g, 'file' ) then return g
        else return null( Λ( g )), g
        end if
    end if;
    for k from 2 to nops( L ) do
        currentdir( r ), assign( 'g' = cat( r, L[ k ] ));
        if type( g, 'dir' ) then try mkdir( g ) catch : NULL end try
        elif type( g, 'file' ) then
            assign( 'L'[ k ] = cat( "_", L[ k ] ), 'd' = 9 );
            try mkdir( cat( r, L[ k ] ))
            catch "file I/O error": d := 9
            catch "directory exists and is not empty": d := 9
            end try;
            assign( 'd' = 9 ), WARNING( s, L[ k ][ 2 .. -1 ], L[ k ] )
        else mkdir( cat( r, L[ k ] ))
        end if;
        assign( 'r' = cat( r, L[ k ], "\" ))
    end do;
    if t = 1 then
        if type( cat( r, ω ), 'dir' ) then
            op( [ Λ( cat( r, "_", ω )), WARNING( s, ω, cat( "_", ω )) ] )
        else return `if`( d = 9, cat( r, ω ), CF( F )), Λ( cat( r, ω )), null( currentdir( cd ))
        end if;
        cat( r, "_", ω ), null( currentdir( cd ))
    else null( currentdir( cd )), `if`( d = 9, r[ 1 .. -2 ], CF( F ))
    end if
end proc
> MkDir(Tallinn), MkDir(Tallinn, 1);
Warning, Path element <tallinn> has been found, it has been renamed on <_tallinn>
        "c:\program files\maple 8\tallinn",  "c:\program files\maple 8\_tallinn"
> type(%[1], 'dir'), type(%[2], 'file');   ⇒  true, true
```

В случае кодирования несуществующего каталога он создается. Если вызов процедуры использует в качестве второго необязательного аргумента **1,** то создается по указанному пути 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, процедура ***MkDir*** расширяет не только стандартную функцию ***mkdir,*** но и одноименную команду ***MS DOS*** [41,103,108,109]. Нами в дополнение к базовой процедуре ***MkDir*** был создан еще ряд ее полезных *модификаций*, включая обобщение на случай, когда делается попытка создания каталога или файла

на устройстве или в каталоге, защищенном от записи. Вызов процедуры ***MkDir4*** обеспечивает создания нужного каталога или файла на дисковом устройстве с возвратом полного пути к нему. Если устройство либо какой-либо элемент создаваемого *пути* защищен от записи, то процедура выбирает первое доступное устройство (*по выбору может быть и устройство* «**A**»). При отсутствии доступных устройств инициируется ошибочная ситуация с диагностикой «*Error, in* (MkDir4) *accessible disk drives do not exist*» [108,109]. Ниже представлен исходный текст процедуры и примеры ее применения.

```
MkDir4 := proc(F::{string, symbol})
local a, b, c, f, g;
    assign(f = Path(F), g = ":/_avzagnvsv424767ArtKr1710_");
    try return MkDir(op(subs("A:" = NULL, [args])))
    catch "permission denied": a := sort([Adrive( )])
    catch "file or directory does not exist": a := sort([Adrive( )])
    catch "file I/O error": a := sort([Adrive( )])
    end try;
    assign(c = cat(f[2 .. -1]),
        'a' = `if`(member("A:", {args}), a, {op(a)} minus {"A"}));
    for b in a do
        if adftest(cat(b, g)) then return
            MkDir(cat(b, c), `if`(1 < nops({args} minus {"A:"}), 1, NULL))
        end if
    end do;
    error `if`(member("A:", {args}), "accessible disk drives do not exist",
        "accessible disk drives excluding <A> do not exist")
end proc
> MkDir4("D:/Work\\Grodno/Vilnius\\Tallinn.2006",10);
                        "f:\work\grodno\vilnius\tallinn.2006"
> MkDir4("C:/Grodno/Grsu/Maple.avz","A:",10);
Error, in (MkDir4) accessible disk drives do not exist
```

Наряду с представленными выше нами создан ряд других процедур *общего* назначения для работы с файловой системой компьютера. В частности, нами введены три *новых* типа выражений, а именно: *dir*, *file* и *path*, определяющие соответственно *каталог*, *файл* и *полный путь*. Так, представленная ниже процедура `***type/dir***` служит для тестирования выражения на предмет быть *каталогом*. Вызов ***type*(F, 'dir')** возвращает *true*-значение, если аргумент **F** определяет каталог, и *false*-значение в противном случае.

```
type/dir := proc(F::anything)
local a, b, c, d, k, u, ω, t, ν, h, p, n;
    `if`(type(eval(F), {name, string, symbol}), NULL, RETURN(false)),
        assign(h = currentdir( ));
    ω, u := ( ) → ERROR("<%1> is invalid path to a directory or a file", F),
        interface(warnlevel);
    `if`(Red_n(cat(" ", F), " ", 2) = " " or member(F, {``, ""}), ω( ), NULL),
        null(interface(warnlevel = 0));
    assign(a = Red_n(Subs_All("/" = "\", Case(cat("", F))), 1), "\", 2),
        ν = (( ) → null(interface(warnlevel = u))));
```

```
    if length(a) = 1 then assign('a' = cat(a, ":\"))
    elif length(a) = 2 and a[2] = ":" then assign('a' = cat(a, "\"))
    elif length(a) = 3 and a[2 .. 3] = ":\" or a[2 .. 3] = ":/" then NULL
    end if;
    `if`(a[2] = ":", `if`(member(a[1], map(Case, {Adrive( )})), NULL,
        RETURN(ν( ), false)), NULL);
    if length(a) < 4 then
        try p := currentdir(a)
        catch "directory exists and is not empty": NULL
        catch "file or directory does not exist": NULL
        catch "file or directory, %1, does not exist": NULL
        catch "permission denied": currentdir(h); RETURN(ν( ), true)
        catch "file I/O error": currentdir(h); RETURN(ν( ), true)
        end try;
        `if`(p = NULL or p = h, RETURN(ν( ), null(currentdir(h)), true), NULL)
    end if;
    `if`(a[-1] = "\", assign('a' = a[1 .. -2]), NULL);
    if a[2 .. 3] ≠ ":\" then
        assign(b = iostatus( ), d = CF(currentdir( )), n = cat("\", a),
            c = map(CF, [libname]));
        `if`(search(d, n) or {op(map(search, c, n))} ≠ {false},
            [ν( ), RETURN(true)], 158);
        `if`(nops(b) = 3, 1, seq(`if`(search(CF(b[k][2]), cat("\", a, "\")),
            [ν( ), RETURN(true)], NULL), k = 4 .. nops(b)))
    end if;
    `if`(type(holdof(hold), set(integer)), NULL, assign(t = 7));
    try close(fopen(a, READ))
    catch "file or directory does not exist":
        `if`(t = 7, holdof(restore, 7), NULL), ν( ), RETURN(ν( ), false)
    catch "file or directory, %1, does not exist":
        `if`(t = 7, holdof(restore, 7), NULL), ν( ), RETURN(ν( ), false)
    catch "file I/O error": `if`(t = 7, holdof(restore, 7), NULL), ν( ), RETURN(true)
    catch "permission denied":
        `if`(t = 7, holdof(restore, 7), NULL), ν( ), RETURN(true)
    end try;
    `if`(t = 7, holdof(restore, 7), NULL), ν( ), false
end proc
> type( "C:/Temp", 'dir'), type( "C:/", 'dir'), type( "C:/Program files/Maple 10/lib", 'dir');
                                    true, true, true
```

При этом, следует иметь в виду, что возврат *false*-значения говорит лишь о том, что тестируемый каталог **F** (*точнее последний элемент* **F**-*цепочки*) отсутствует в: (**1**) цепочке, определенной **F,** (**2**) цепочках, определенных предопределенной переменной ***libname,*** (**3**) цепочке, определенной вызовом ***currentdir*()** и (**4**) цепочках, определенных открытыми файлами данных текущего сеанса. Тогда как, вообще говоря, данный каталог **F** может быть элементом файловой системы компьютера. Детальнее с представленными средствами работы с файловой системой компьютера можно ознакомиться в [41,103,108,109].

Предложения ***Maple***-языка сохраняются во внутренних файлах пакета в одном из двух форматов (*входном **Maple**-формате и внутреннем **m**-формате*; *оба являются ASCII-файлами*) на основе **save**-предложения одного из следующих двух форматов:

save {*Id1*, *Id2*, ..., *Idn***,**} <СФ> или **save(**{*Id1*, *Id2*, ..., *Idn***,**}<СФ>**)**

По формату **save** <СФ> в указанный *спецификатором* (**СФ**) файл помещаются все определенные идентификаторы текущего *сеанса*, т.е. идентификаторы, которым производились присвоения по (**:=**)-оператору или ***assign***-процедуре. Однако следует иметь в виду, что корректное сохранение по данному формату **save** производится только для случая принимающего файла *внутреннего* формата, т.е. ***m***-файла (*имя файла завершается символами* «**.m**»). Иначе идентифицируется ошибочная ситуация. При этом, это относится ко всем ***Maple***-документам, открытым в текущем сеансе, т. е. в ***m***-файле сохраняется состояние текущего сеанса. Между тем, данный формат допустим только для ***Maple 6*** и для релизов **7** и выше он отсутствует.

Тогда как по формату **save** *Id1*, ..., *Idn*, <СФ> в указанный СФ-спецификатором *файл* выгружаются только те определения идентификаторов, чьи имена закодированы в последовательности перед СФ-аргументом. При этом, если в СФ определено ***m***-расширение имени, то файл сохраняется во *внутреннем **m***-формате, в противном случае используется *входной* формат ***Maple***-языка (*в этом формате производится ввод информации в **Maple**-документ с консоли*). В случае указания в **save** неопределенных *Id*-идентификаторов, они сохраняются в виде Id:='Id' с выводом соответствующих предупреждающих сообщений. *Успешное* завершение **save**-предложения возвращает *NULL*-значение. Следующий простой фрагмент иллюстрирует использование **save**-*предложения* для создания файлов обоих указанных форматов:

```
> A:= 64: B:= [64,59,39,44,10,17]: P:= () -> `+`(args)/nargs: M:= module () export sr; sr:= () ->
  `+`(args)/nargs end module:  S:= 2006:
> save(A, B, P, M, S, "C:/Temp/Tallinn");  save(A, B, P, M, S, "C:/Temp/Tallinn.m");
> restart;  read("C:/Temp/tallinn"): %;     ⇒ 2006
> A, B, 6*P(64,59,39,44,10,17), 6*M:- sr(1,2,3,4,5,6);   ⇒ 64, [64, 59, 39, 44, 10, 17], 233, 21
> restart;  read "C:/Temp/tallinn.m"; %;   ⇒ 10                                          (A)
> A, B, 6*P(64,59,39,44,10,17), 6*M:- sr(1,2,3,4,5,6);   ⇒ [64,59,39,44,10,17], 233, 6 sr(1,2,3,4,5,6)
```

В данном фрагменте определяются переменные и по **save**-предложению сохраняются в ***Tallinn***-файлах в форматах *входного **Maple***-языка и *внутреннем*. На небольшого объема файлах не ощущается преимущества от того или иного формата, однако при возрастании объема предпочтение *внутреннего **m***-формата становится все ощутимее, позволяя создавать более компактные файлы и быстрее их загружать в память.

Загрузка сохраненного по **save** файла производится по предложению **read** форматов:

read <СФ> или **read(**<СФ>**)**

но результат его выполнения зависит от формата загружаемого файла. Общим является тот факт, что после загрузки файла содержащиеся в нем определения становятся доступными в текущем сеансе, если впоследствии не определяется противного. Однако, если загружается файл *входного* формата ***Maple***-языка, то в случае завершения предложения **read** (**;**)-разделителем на дисплей выводится содержимое файла, а результат выполнения **read**-предложения возвращает значение *последнего* предложения файла. Тогда как по (**:**)-разделителю информации не выводится, но также возвращается значение последнего предложения загруженного файла. В случае же загрузки ***m***-файла информации не выводится и возвращается *NULL*-значение. Пример (**A**) предыдущего фрагмента иллюстрирует применение **read**-предложения для загрузки ***m***-файла. Однако, в

этом месте может возникнуть недопонимание в связи с возвратом %-вызовом значения 10. А дело здесь обстоит следующим образом. Как известно, такой *вызов* возвращает значение выполнения предыдущего ему предложения. Но так как перед **read** мы выполнили **restart,** а вызов **read** возвратил *NULL*-значение (*т.е. ничего*), то %-вызов после **restart** либо после загрузки пакета в зависимости от релиза и его клона возвращает следующие значения, а именно: результат вызова *последней* процедуры, определенной в инициализационном файле «***Maple.ini***» (***Maple 6/7*** *и* ***Maple 9-10*** *standard-клона*), **10** (***Maple 8*** *и* ***Maple 9/9.5*** *standard-клона*) и **-1** (***Maple 10*** *standard-клона*).

На первых порах работы с пакетом 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 сохраняются некорректно (*точнее, не сохраняются их тела*), как это хорошо иллюстрирует последний пример предыдущего фрагмента. Поэтому нами был создан ряд процедур как обобщающих, так и расширяющих стандартные предложения **save** и **read** относительно работы с файлами внутреннего ***Maple***-формата [41,103,108,109]. Так, в частности, пара нижеследующих процедур ***Save2*** и ***Read1*** обеспечивает *корректное* сохранение всех типов ***Maple***-объектов в файлах внутреннего ***m***-формата с последующим чтением в текущий сеанс. Их исходные тексты и примеры приводятся ниже.

```
Save2 := proc()
local a, b, c, j, k, G, S, ν, f, ω, h;
    `if`(nargs < 2, RETURN(NULL),
        assign(ν = interface(warnlevel), ω = (x → null(interface(warnlevel = x)))))
        ;
    if type(args[-1], 'file') then f := args[-1]; ω(0)
    elif not type(eval(args[-1]), {'symbol', 'string'}) then
        ERROR("<%1> can't specify a datafile", args[-1])
    else ω(0); f := MkDir(args[-1], 1)
    end if;
    assign(h = cat(currentdir( ), "/$Art16_Kr9$"));
    assign(G = [ ], S = ""), ω(2), assign(
        a = {op(SLD(convert('procname(args)', 'string')[7 .. -1], ",")[1 .. -2])});
    seq(`if`(type(eval(args[j]), {`module`, 'procedure'}),
        assign('G' = [op(G), args[j]]), NULL), j = 1 .. nargs − 1),
        assign(b = a minus {op(map(convert, G, 'string'))}),
        `if`(b = { } and G = [ ], RETURN(ω(ν)), NULL);
    if b ≠ { } then assign(c = cat("_Vars_", " := proc() global ",
        seq(cat(b[j], ","), j = 1 .. nops(b)), "; assign(", seq(
        cat(convert(cat("'", b[j], "'") = eval(convert(b[j], 'symbol')), 'symbol'), `,`),
        j = 1 .. nops(b)), ") end proc:")), seq(assign('S' =
        cat(S, `if`(member(k, {op(Search2(c, {"'", ",)", ",;"}))}), NULL, c[k]))),
        k = 1 .. length(c)), writebytes(h, S), fclose(h),
        (proc() read h; fremove(h) end proc)( )
    end if;
    Save1([[op(G), `if`(b = { }, NULL, _Vars_)]], [f]), ω(ν), f
end proc
```

```
> MM:=module () export x, m;  m:= module() export y; y:= () -> `+`(args) end module end
  module: MM1:= module()  local F,m; export a; option package; m:= module() export y;
  y:=() -> `+`(args) end module;  F:=() -> m:- y(args); a:=F end module:  SVEGAL:=module()
  export x; x:=() -> `+`(args)/nargs^6 end module:  PP:=proc()  `+`(args)/nargs^2  end proc:
 PP1:=proc()  `+`(args)/nargs  end proc: PP2:=proc() `+`(args)/nargs end proc: GS:=module()
  export x; x:=() -> `+`(args)/nargs^3  end module: avz:=61: agn:=56: Art:=14: Kr:=7:
> Save2(op({MM, 'avz', SVEGAL, MM1, 'agn', GS, 'Art', PP, 'Kr', PP1, 'Svet'}),
  "C:/RANS/IAN/RAC/REA/Academy/Art_Kr.txt.m");
                        "c:\\rans\\ian\\rac\\rea\\academy\\art_kr.txt.m"
 Read1 := proc(F::{list({string, symbol}), set({string, symbol})})
 local k, j, act_tab, rf, v, t, fg, nm, T, U, R;
      `if`(F = [ ] or F = { }, ERROR("files for reading are absent"), NULL);
      assign(fg = (a::string → `if`(search(a, "/"), a[2 .. -2], a)), U = {anames( )});
      assign(
           act_tab = table(['procs' = { }, 'mods1' = { }, 'mods2' = { }, 'vars' = { }]),
           T = table(['procs' = { }, 'mods1' = { }, 'mods2' = { }, 'vars' = { }]));
      v := table(["'" = 1, "#" = 2, "$" = 3, "%" = 4, "&" = 5, "'" = 6, "(" = 7, ")" = 8,
           "*" = 9, "+" = 10, "," = 11, "-" = 12, "." = 13, "/" = 14, "0" = 15, "1" = 16,
           "2" = 17, "3" = 18, "4" = 19, "5" = 20, "6" = 21, "7" = 22, "8" = 23, "9" = 24,
           ":" = 25, ";" = 26, "<" = 27, "=" = 28, ">" = 29, "?" = 30, "@" = 31, "A" = 32,
           "B" = 33, "C" = 34, "D" = 35, "E" = 36, "F" = 37, "G" = 38, "H" = 39,
           "I" = 40, "J" = 41, "K" = 42, "L" = 43, "M" = 44, "N" = 45, "O" = 46,
           "P" = 47, "Q" = 48, "R" = 49, "S" = 50, "T" = 51, "U" = 52, "V" = 53,
           "W" = 54, "X" = 55, "Y" = 56, "Z" = 57]);
      rf := proc(f)
           local _h, _p, _p1, k, _z, _g, _r, _q;
                close(f), assign(_p = { }, _p1 = { },
                     _z = {seq(lhs(op(2, eval(v))[k]), k = 1 .. nops([indices(v)]))});
                do
                     _h := readline(f);
                     if _h = 0 then break
                     else
                          if cat(" ",f)[-2 .. -1] = ".m" then
                               if _h[1] = "I" and member(_h[2], _z) then
                                    _p :=
                                         {cat(``, _h[3 .. 2 + eval(v[_h[2]])]), op(_p)}
                                         ;
                                    _p1 := _p
                               end if
                          else
                               if _h[1 .. 11] = "unprotect(" then
                                    _g := Search(_h, ` `);
                                    _p := {op(_p),
                                         cat(``,fg(_h[_g[1] + 1 .. _g[2] − 1]))}
```

```
                    elif evalb(
                    assign('_q' = Search2(_h, {" := mod", " := proc"})),
                    _q ≠ [ ]) then
                        _p := {op(_p), cat(``,fg(_h[1 .. _q[1] − 1]))}
                    elif search(_h, `:= `,'_r') then
                        _p1 := {op(_p1), cat(``,fg(_h[1 .. _r − 1]))}
                    end if
                end if
            end if
        end do;
        close(f), [_p, _p1]
    end proc;
for k to nops(F) do
    try
        if not type(F[k],'file') then
            WARNING("<%1> is not a file or does not exist",F[k]); next
        elif not TRm(F[k]) then
            WARNING("file <%1> is not readable by statement <read>",F[k])
                ;
            next
        else
            if nargs = 1 then read F[k]; next
            else
                nm := (proc(`_$$$_`) read `_$$$_`; rf(`_$$$_`) end proc)(
                    F[k]);
                for j to nops(nm[1]) do
                    if type(eval(nm[1][j]),'procedure') and
                    member(nm[1][j], {anames('procedure')}) then
                        act_tab['procs'] :=
                        {nm[1][j], op(act_tab['procs'])}
                    elif type(eval(nm[1][j]), `module`) and
                    member(nm[1][j], {anames(`module`)}) then
                        if type(nm[1][j], 'mod1') then act_tab['mods1'] :=
                            {nm[1][j], op(act_tab['mods1'])}
                        else act_tab['mods2'] :=
                            {nm[1][j], op(act_tab['mods2'])}
                        end if
                    end if
                end do;
                act_tab['vars'] := {
                    op(act_tab['vars']), op(nm[2] intersect {anames( )})}
                    minus map(op,
                    {act_tab['procs'], act_tab['mods1'], act_tab['mods2']})
            end if
        end if
```

```
        catch "local and global":
            NULL("Processing of especial situations with program modules")'
        end try
    end do;
    `if`(member(_Vars_, act_tab['procs']), _Vars_(), NULL),
        assign(R = act_tab['mods1'] union act_tab['mods2']),
        `if`(R ≠ {}, map(eval, R), NULL);
    `if`(act_tab['procs'] ≠ {},
        (proc() try map(act_tab['procs'], ``) catch "": NULL end try end proc)(),
        NULL);
    null(`if`(1 < nargs, [assign(args[2] = eval(act_tab))), `if`(nargs = 3, [seq(
        assign('T[k]' = U intersect act_tab[k]), k = ['procs', 'mods1', 'mods2', 'vars'])
        , assign(args[3] = eval(T))], NULL)], NULL))
end proc
> restart; Read1(["C:/RANS/IAN/RAC/REA/Academy/Art_Kr.txt.m"], 'h'), eval(h);
   table([procs = {_Vars_, SVEGAL, PP, PP1, MM, MM1, GS}, mods1 = {}, mods2 = {}, vars = {}])
> [avz, agn, Art, Kr], 36*PP(42, 47, 67, 62, 89, 96), SVEGAL:- x(42, 47, 67), GS:- x(42, 47, 67);
                                  [61, 56, 14, 7], 403, 52/243, 52/9
```

Процедура ***Save2*** весьма существенно расширяет предложение **save** с полным наследованием синтаксиса последнего. но в отличие от него позволяет корректно сохранять в ***m***-файлах переменные, процедуры и программные модули. Вызов процедуры возвращает полный путь к файлу с сохраненными ***Maple***-объектами. В свою очередь, процедура ***Read1*(F)** используется как для загрузки в память файлов, сохраненных по ***Save2***, так и для получения *имен* объектов, сохраненных в них, в разрезе их типов. В этом же контексте представляет интерес, например, реализованная однострочным экстракодом процедура ***savemu***, обеспечивающая корректное сохранение модулей в ***m***-файлах с возвратом *NULL*-значения. Последующее чтение таких файлов **read**-предложением делает доступными *экспорты* (*exp*) сохраненного в них *модуля* по конструкции **M[***exp***](***args***)**, при этом, сами сохраненные модули не распознаются пакетом в качестве таковых.

```
savemu := (M::{`module`, set(`module`), list(`module`)}, F) → (proc()
local a, b, c, f;
    assign(a = (x → `if`(type(x, `module`), [x], x)), c = "",
        f = cat([libname][1][1 .. 2], "\_$Art16Kr9$_"));
    map(mod21, a(M)), seq(
        assign('c' = cat(c, b, " := eval('parse'(convert(eval(", b, "), 'string'))):")),
        b = a(M));
    (proc() writeline(f, c), close(f); read f; fremove(f) end proc)(),
        (proc() save args, F end proc)(op(a(M)))
end proc)()
> module A () export B, G; B:= () -> `*`(args): G:= () -> sqrt(`+`(args)) end module: M:=
  module() export vs; vs:= () -> `+`(args)/nargs end module: savemu({A, M}, "C:/tmp/m.m");
  restart; read("C:/tmp/m.m");
> map(type, [A,M],`module`),A[B](1,2,3),A[G](9,10,17), M[vs](2,3,4,5,6); ⇒ [false, false], 6,6,4
```

С другими нашими средствами **Save**-группы можно ознакомиться в [41-43,103,108,109].

6.2. *Средства Maple-языка для работы с внешними файлами данных*

Средства доступа к *файлам* включают ряд функций, при этом под «*файлом*» подразумевается не только организованная совокупность данных во внешней памяти **ПК,** но и канал, а также непосредственный интерфейс с пользователем. В этом отношении все составляющие понятие *файл* компоненты являются эквивалентными и за редким исключением любой файл обрабатывается одним и тем же набором функций доступа.

Для обеспечения доступа к файлам данных ***Maple-***язык располагает рядом достаточно эффективных для **ПС** данного типа средств. Как и в случае большинства известных *систем* программирования, для инициации операций обмена с файлом данных требуется предварительное его открытие, а после завершения работы с файлом – его закрытие; при этом, если открытие файла должно быть явным (*кроме случая стандартных и специальных файлов*), то закрытие его, в общем случае, не обязательно должно быть явным, ибо в конце работы с прикладной программой все используемые ею ресурсы, включая файлы различных типов, закрываются автоматически. Однако, для большинства **ПС** число одновременно открытых файлов ограничено и их максимальное количество колеблется в пределах (8-24). Для случая нашего пакета количество открытых пользовательских файлов не должно превышать **7.** С целью устранения данного ограничения нами была создана процедура ***holdof,*** предлагающая следующий механизм расширения числа доступных каналов *ввода/вывода* (в/в), а именно.

Если число открытых каналов в/в меньше *семи*, 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, схема использования процедуры ***holdof*** имеет следующий принципиальный вид:

holdof(*hold*) => *обработка файла данных на выделенном канале в/в* **=> holdof(*restore*)**

```
holdof := proc(C::symbol)
local a, b, k, ψ, j, ω;
global _avzagnartkrarn63;
    ψ := 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(ψ(a)), NULL);
        for k from 4 to nops(a) do
            if not member(a[k][2], {"terminal", "default"}) and
            member(a[k][3], {'STREAM', 'RAW'}) then
                _avzagnartkrarn63 := filepos(a[k][2]), a[k];
                close(a[k][1]), RETURN(a[k][1])
            end if
        end do;
```

```
        if type( _avzagnartkrarn63 , 'assignable1' ) then
            ERROR( "too many files already open")
        end if
    else
        try
            `if`( not typeseq( _avzagnartkrarn63 , 'seqn' ), RETURN( ),
                assign67( ω = _avzagnartkrarn63 ) );
            unassign( '_avzagnartkrarn63' ), close( ω[ 2 ][ 1 ] )
        catch "file descriptor not in use": null( "processing of an open I/O channel")
        end try ;
        b := [ `if`( ω[ 2 ][ 3 ] = 'RAW', 'open', 'fopen' ), ω[ 2 ][ 2 ],
            op( ω[ 2 ][ 5 .. `if`( ω[ 2 ][ 3 ] = 'RAW', 5, 6 ) ] ) ];
        null( OpenLN( ω[ 2 ][ 1 ], op( b ) ), filepos( ω[ 2 ][ 1 ], ω[ 1 ] ) ), `if`( 1 < nargs,
            NULL, WARNING(
            "file <%1> had been restored on I/O channel <%2>", ω[ 2 ][ 2 ], ω[ 2 ][ 1 ] ) )
    end if
end proc
```

В ряде случаев, имеющих дело с доступом к файлам данных, требуется обеспечить открытие файла по *конкретному* каналу. Для этого может быть полезна процедура ***OpenLn.***

```
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, {'open', 'fopen'} ),
        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:/Books/bin.xls", 'READ'), iostatus();
        3, {0, 1, 2, 4, 5}, [2, 0, 7, [3, "D:/Books/bin.xls", RAW, FD = 14, READ, BINARY],
                                [6, "C:/Temp/Tallinn", RAW, FD = 17, WRITE, BINARY]]
```

Успешный вызов процедуры ***OpenLN*(N, O,F, *parms*)** возвращает 2-элементную последовательность, чей *первый* элемент определяет номер логического канала в/в, т.е. **N,** тогда как *второй* – множество остающихся номеров открытых каналов. *Четвертый* аргумент *parms* определяет остальные фактические аргументы, зависящие от функции открытия **O** {***open, fopen***}. Наконец, **F**-аргумент определяет имя либо полный путь открываемого файла данных. С рядом наших других полезных процедур, относящихся к *открытию* файлов данных, можно ознакомиться в [41,103,108,109].

В дальнейшем нам понадобится ряд понятий, хорошо известных имеющему опыт программирования читателю. Это относится к таким понятиям как типы файлов: *буферированные* (***STREAM***) и *небуферированные* (***RAW***). Какой-либо особой разницы между ними основные средства доступа ***Maple-***языка не делают, однако в их реализации первые во временном отношении более реактивны, т.к. используют механизм буферизации [12]. *Небуферированный* же тип файлов используется, как правило, при необходимости оперативного запроса состояния того или иного ресурса **ЭВМ.** Тогда как в общем случае следует использовать *буферированный* тип файлов, который по *умолчанию* используется большинством функций доступа ***Maple-***языка. В более простой терминологии работа (*чтение/запись*) с буферированными файлами производится не напрямую, а через специальный буфер, обеспечивающий накопление информации для возможности создания более крупных информационных блоков обмена, тогда как работа с небуферированным файлом производится напрямую, минуя какой-либо буфер. Второй подход повышает оперативность доступа к файлу, но существенно снижает его реактивность при интенсивном режиме обмена данными. Функции языка (*например*, ***iostatus***), возвращающие состояние системы в/в, для идентификации буферированных и небуферированных файлов используют соответственно идентификаторы *STREAM* и *RAW*.

Большинство операционных сред (***MS-DOS, Windows, MAC, VMS*** *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»*. Данное состояние исторически восходит к принципам работы пишущих машинок по созданию текстовых документов и в консоли **ЭВМ** реализуется ***Enter***-клавишей. В среде некоторых других операционных систем различие между файлами обоих типов могут носить несколько иной характер, который невидим на уровне рассматриваемых нами средств доступа ***Maple-***языка.

Реализация состояния *«перевод строки и возврат каретки»* зависит от используемой операционной среды **ЭВМ.** В среде ***Maple-***языка данное состояние идентифицируется одним символом, а в строчных конструкциях оно кодируется символами **“\n”.** Международный ASCII-стандарт для этих целей использует символ с десятичным кодом **<10>,** тогда как в операционных средах ***MS-DOS, Windows, VMS*** указанное состояние представляется парой символов с десятичными кодами **<13,10>**, а ***MAC***-система использует для этих целей только один символ с десятичным кодом **<13>**. При реализации доступа (*чтение/запись*) к текстовому файлу средства доступа при обнаружении *символов*, определяющих состояние *«перевод строки и возврат каретки»*, производят соответствующие действия, рассматриваемые ниже. Тогда как для *бинарных* файлов средства доступа рас-

сматривают все составляющие их символы как последовательность *равноценных* байтов. Средства доступа ***Maple***-языка поддерживают работу как с *текстовыми*, так и с *бинарными* файлами. Однако следует иметь в виду, что при использовании пакета в среде ***Unix***-подобных систем средства доступа ***Maple***-языка отождествляют оба типа файлов, полагая все файлы *бинарными*. Функции доступа ***Maple***-языка, различающие оба типа файлов, используют для них соответственно идентификаторы *TEXT* (*текстовый*) и *BINARY* (*бинарный*).

На *внутреннем уровне* ядро пакета средствами внутренней I/O-библиотеки (*iolib-библиотеки*) обеспечивает доступ к файлам данных следующих пяти видов:

STREAM – *буферированный* файл,соответствующий стандарту I/O-библиотеки ***C***-языка;

RAW – *небуферированный* файл,соответствующий стандарту ***Unix***- и ряда других сред;

PIPE – двухконцевой канал, соответствующий ***Unix***-стандарту и поддерживаемый только ***Unix***-платформой;

PROCESS – *канал*, который *одним* концом связан с *другим* процессом по ***Unix***-стандарту;

DIRECT – прямой доступ к текущему (*по умолчанию*) или высшего уровня (*терминал*) потоку ввода/вывода (*например, одновременный ввод запросов на обслуживание и получение возвращаемых результатов их обработки в интерактивном режиме работы с ядром* ***Maple****-пакета*).

Для обеспечения доступа к файлу *любого* из указанных пяти видов он должен быть предварительно открыт в одном из двух режимов доступа: ***READ*** (*на чтение*) или ***WRITE*** (*на запись*). При этом, *режим* открытия определяется или *явно* в момент вызова функций открытия: ***fopen, open, pipe*** или ***popen,*** либо *неявно* в результате вызова ряда функций в/в *внутренней* I/O-библиотеки ядра пакета. Если файл вида {**STREAM**|**RAW**|**DIRECT**} открыт в ***READ***-режиме доступа, то попытка записи в него информации автоматически закрывает его с последующим *переоткрытием* в ***WRITE***-режиме доступа к *последней* сканированной до этого позиции файла. Данная особенность позволяет перлюстрировать файл до нужной позиции, а затем переоткрывать его на запись с данной позиции. Другие виды файлов, открытые явно, не могут изменять режима доступа без их предварительного *явного* закрытия. Попытка записи в файл, отмеченный как «*только для чтения* (***readonly***)» вызывает ошибочную ситуацию с соответствующей диагностикой. Детальнее реакция ядра пакета на использование указанных режимов доступа будет рассмотрена ниже. *Функции* доступа ***Maple***-языка, использующие в своих форматах типы режима, применяют для них соответственно *специальные* идентификаторы ***READ*** (*чтение*) и ***WRITE*** (*запись*). При этом, следует констатировать, что для целого ряда пакетных имен, включая ***READ*** и ***WRITE,*** не предусмотрен *protected*-атрибут, что при их кодировании требует осмотрительности и использования *невычисленного* формата, например, ***'READ'.*** С целью устранения данной недоработки нами была создана процедура ***Lprot.***

```
Lprot := proc()
local a, b, c, d, k;
    assign( c = {  }, d = [  ] );
    if nargs ≠ 0 then for k in [ args ] do
            if member( CF( k ), { map( CF, { libname } ) } ) then d := [ op( d ), k ]
            elif type( k, 'symbol' ) then c := { op( c ), k }
            end if
        end do
    end if;
    if type( MkDir, 'libobj' ) then
```

```
        a := [ op( march('list', _libobj ) ), seq( op( march('list', k ) ), k = d ) ]
    elif d ≠ [ ] then a := [ seq( op( march('list', k ) ), k = d ) ]
    else error "user libraries are absent in the predefined variable <libname>"
    end if;
    protect( `if`( c = { }, NULL, op( c ) ), all, APPEND, Arguments, arity, assignable,
        assignable1, binary, BINARY, Break, boolproc, builtin, byte, bytes, chkpnt,
        coincidence, complex, complex1, false, γ, ∞, true, Catalan, FAIL, π, correct,
        create, Cycle, Decimal, decimal, default, del, delel, delel1, delete, digit, dir,
        dirax, direct, display, element, Empty, End, Exit, exit_type, exprseq, extract,
        file, file1, Fin, fpath, frame, globals, heap, hold, insensitive, insert, inside,
        invalid, left, letter, lex_tab, libobj, list, list1, listlist, listlist1, locals, lower,
        Lower, lowercase, LRM, mla, mlib, Mod, mod1, Next, nestlist, nonsingular,
        NRM, only, package, path, plotopt, plot3dopt, prn, procname, pvl, RAW,
        READ, realnum, rem_tab, Repeat, restore, right, rlb, Roman, Russian,
        sensitive, seqn, sequent, set, set1, setset, shape, Special, ssign, ssll,
        statement, sublist, Table, terminal, TEXT, toc2005, upper, Upper,
        uppercase, Variable, VGS, WRITE, variable,
        seq( `if`( b[ 1 ][ 1 .. 2 ] = ":-", NULL, cat( ``, b[ 1 ][ 1 .. -3 ] ) ), b = a ) )
end proc
```

Процедура своим вызовом ***Lprot*()** присваивает *protected*-атрибут *именам* пакета и нашей *Библиотеки* [109], не имеющих такого атрибута. Тогда как по вызову ***Lprot*(*args*)** *дополнительно* получают данный атрибут и *имена*, определенные необязательным аргументом.

В отличие от операционной ***Unix***-системы, рассматривающей *файлы* как последовательности байтов, большинство других операционных сред и оболочек типизируют файлы, классифицируя их на два *основных* типа ***TEXT*** (*текстовые*) и ***BINARY*** (*бинарные*). Оба типа оперируют с последовательностями соответственно *символов* и *байтов*. Для представления «*конца строки*» (*записи*) в ***TEXT***-файлах на платформах ***D0S*** и ***Windows*** служит **2-**х символьный идентификатор **hex(0D0A)** «*перевода строки и возврата каретки*». При *явном* открытии файла **STREAM**-*вида* имеется возможность определять его тип, тогда как при *неявном* открытии его тип определяется контекстуально. Файлы, отличные от **STREAM**-*вида* (*исключая* ***DIRECT***), автоматически открываются как ***BINARY***-*файлы*. Тогда как *файлы* **DIRECT**-вида автоматически открываются как ***TEXT***-файлы.

Для файлов **DIRECT**-вида дополнительно определяются два предопределенных типа: ***default*** и ***terminal.*** Подобно развитым программным средствам *функции* доступа ***Maple***-языка 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 *высшего* уровня ввод/вывод *текущего* сеанса работы с пакетом. В режиме интерактивного функционирования пакета *оба* типа пакетных файлов совпадают. *Точнее*, 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 предложений ***Maple***-языка посредством указанных функций. В случае ***Windows***-платформы ***default***-*файл* относится к потоку, из которого производится чтение, а файл ***terminal*** – к текущему сеан-

су работы с пакетом. Так как для ***Unix***-платформы входной поток поступает из файла либо канала, то ***terminal***-файл относится именно к ним.

В заключение напомним, что для идентификации в функциях доступа *собственно* самого файла используются два подхода: на основе его *спецификатора* (**СФ**), определяющего полный путь к нему и кодируемого в следующем виде:

`<*УВВ*>:{\\|/}<*Подкаталог*_1>{\\|/}...{\\|/}<*Подкаталог*_k>{\\|/}<*Имя*>.<*Расширение*>`

"<*УВВ*>:{\\|/}<*Подкаталог*_1>{\\|/}...{\\|/}<*Подкаталог*_k>{\\|/}<*Имя*>.<*Расширение*>"

например: `C:\\ARM_Book/Academy\\Tallinn/RANS.IAN`, и *номера* логического канала в/в, детально рассматриваемого ниже. В принципе, *оба способа* указания файла эквивалентны, однако в *отдельных* случаях предпочтение отдается одному из них. В частности, 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 в среде ***Maple*** пользователь практически начинает ощущать всю целесообразность использования того или иного способа в каждом конкретном случае. Большинство функций доступа ***Maple***-языка допускает использование как первого, так и второго способа идентификации файлов.

Между тем, ***Maple***-язык не отождествляет **СФ,** закодированные хоть и одинаковыми, но на различных регистрах буквами либо 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 в/в в различных режимах доступа [41,103,108,109].

Вызов процедуры ***open2*(F, L, P, N)** обеспечивает открытие файла со СФ **F** на двух логических каналах в/в, номера которых возвращаются через аргументы **L, P,** а через **N** размер файла в байтах соответственно. Файл **F** открывается посредством функции ***open*** на логическом канале **L** и посредством функции ***fopen*** на логическом канале **P.** Успешный вызов процедуры ***open2*(F, L, P, N)** 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. Детальнее с процедурой можно ознакомиться в [41-43,103,108,109].

```
open2 := proc(F::{string, symbol}, L::evaln, P::evaln, N::evaln)
local h, n, J;
    `if`(type(F, 'file'), assign(h = CF(F)), ERROR("<%1> is not a file", F));
    assign(n = op(convert(substring(h, 1), 'bytes'))), N = filepos(h, ∞));
    `if`(65 ≤ n and n ≤ 90,
        assign(J = cat(convert([n + 32], 'bytes'), substring(h, 2 .. length(h)))),
        assign(J = cat(convert([n − 32], 'bytes'), substring(h, 2 .. length(h)))));
    op([close(h), assign(L = open(h, `if`(4 < nargs, args[5], 'READ'))), P =
        fopen(J, `if`(5 < nargs, args[6], 'READ'), `if`(nargs = 7, args[7], 'BINARY')))
        ])
end proc
> open2("C:/Tmp/Demo\\Ans.rtf/", L, P, N, WRITE, APPEND, TEXT): iostatus(), L, P, N;
            [2, 0, 7, [0, "c:\tmp\demo\ans.rtf", RAW, FD = 11, WRITE, BINARY],
        [1, "C:\tmp\demo\ans.rtf", STREAM,FP = 2013506696, READ, TEXT]], 0, 1, 1
```

С другими нашими средствами, относящимися к открытию файлов данных, а также с рядом их особенностей, полезных для программирования, можно ознакомиться в [103].

6.2.1. *Открытие, закрытие и удаление внешних файлов любого типа*

В общем случае процедура доступа к файлу имеет *вложенную* структуру, в которой *ядро*, состоящее из функций собственно обработки файла (*запись*, *чтение*, *дописывание и др.*), обрамляется *оболочкой*, содержащей *функции* соответственно *открытия* и *закрытия* файла. При этом, следует иметь в виду, что процедура закрытия файлов в общем случае необязательна, ибо в результате завершения работы с пакетом (**Exit, quit, done, stop**) либо выполнения **restart**-предложения все открытые в текущем сеансе работы с ним файлы автоматически закрываются. Однако, во избежание потери информации (*в ряде случаев из пакетного буфера обмена в файл могут быть переданы не все данные*) рекомендуется производить *явное закрытие* всех открытых пользовательских файлов. Более того, ввиду возможности одновременно открывать относительно небольшое число файлов (*не более* 7) следует четко отслеживать режим *своевременного* закрытия не требующих доступа файлов, позволяя открывать для доступа другие файлы, не вызывая ошибочных ситуаций типа «… too many files already open». Для возможности расширения числа доступных логических каналов в/в можно использовать упомянутую выше процедуру ***holdof.***

Для обеспечения доступа к *буферированным* (**Б**) и *небуферированным* (**НБ**) файлам ***Maple***-язык располагает парами функций открытия/закрытия файлов соответственно:

(**Б**) **fopen(**<СФ>, {***READ*** | ***WRITE*** | ***APPEND***{, ***TEXT*** | , ***BINARY***}}**)** **fclose(**<СФ>**)**

(**НБ**) **open(**<СФ>, {***READ*** | ***WRITE***}**)** **close(**<СФ1>**, ...,** <СФn>**)**

Следует отметить, что ***Maple***-язык имеет соответствующее функциональное обеспечение для *оболочки* процедур доступа и к файлам видов **PIPE** и **PROCESS** (*модульные функции* ***popen*** *и* ***pclose*** *из* **process**-*модуля пакета*). Однако здесь данные средства не обсуждаются, ибо файлы **PIPE**-вида поддерживаются только для ***Unix***-платформы, а **PROCESS**-вида только для платформ, обеспечивающих многопроцессовый режим функционирования (*например*, ***Unix***-*систем*). Рассмотрим средства *оболочки* функций доступа несколько детальнее.

По ***fopen***-функции производится *открытие* файла, заданного своим спецификатором (*первый фактический* СФ-*аргумент*), для буферированного доступа, режим которого определяется вторым фактическим {***READ*** | ***WRITE*** | ***APPEND***}-аргументом. При этом, третий необязательный аргумент определяет {***TEXT*** (*по умолчанию*) | ***BINARY***}-тип создаваемого файла. При этом, следует иметь в виду, что корректные способы *кодирования* спецификатора приводятся в нижеследующем фрагменте, в иных же случаях возможны некорректные ситуации создания файлов. А именно, наиболее общее правило кодирования спецификатора состоит в соответствии его соглашениям среды ***MS DOS***, но с одним лишь исключением: если в качестве разделителя подкаталогов используется обратный слэш, то он должен удваиваться. Результатом успешного открытия файла является возврат целого числа (0-6), определяющего номер логического канала в/в.

Логический канал в/в идентифицируется уникальным для *всех* открытых файлов текущего сеанса **n**-номером, значение которого будет находится в диапазоне 0-6; при этом, нумерация каналов производится безотносительно их имен. Для стандартных и специальных каналов пакет располагает рядом зарезервированных имен, например имена: ***default*** и ***terminal*** определяют соответственно ввод/вывод пакета по умолчанию и высшего уровня в/в текущего сеанса. В функциях доступа к открытому файлу можно впос-

ледствии кодировать как непосредственно его спецификатор, так и номер *приписанного* ему логического канала в/в.

Вообще говоря, существующий файл не требует предварительного открытия по ***fopen***-функции для обеспечения к нему доступа, т.к. первая же функция доступа к нему открывает файл. Однако ***fopen***-функция имеет два весьма важных аспекта: (**1**) обеспечивает совместимость текстов программ, импортируемых из ряда популярных языков программирования, и (**2**) позволяет переопределять тип файла, присвоенный ему по *умолчанию* функциями доступа. При этом, если по ***fopen***-функции открывается *существующий* файл с указанием для него ***TEXT***-типа, то в нем выделяются управляющие **hex(0D0A)-**символы в качестве разделителей строк, т.е. файл построчно структурируется, позволяя организовывать к нему доступ на уровне составляющих его строк. Такого типа файлы являются наиболее широко используемым типом внешних данных для многих хорошо известных **ПС,** позволяя легко поддерживать мобильность информации. В противном случае файл будет представлять однородную последовательность байтов. При открытии существующего файла в режиме ***WRITE***-доступа он обновляется, тогда как по режиму ***APPEND*** производится дописывание информации в конец файла. Все возникающие особые и ошибочные ситуации при вызове ***fopen***-функции вполне адекватно отражаются выводимой диагностикой, доступной для обработки описанными ниже средствами ***Maple***-языка.

По ***open***-функции производится открытие *небуферированного* файла, спецификатор которого определен первым фактическим СФ-аргументом, в заданном вторым фактическим {***WRITE***|***READ***}-аргументом режиме. В случае *успешного* завершения операции открытия функция возвращает выделенный файлу номер логического канала. При этом, на количество открываемых по ***open***-функции файлов накладывается то же *ограничение*, что и для ***fopen***-функции, а *общее* одновременно открытое в текущем сеансе число пользовательских файлов не должно превышать **7.** Следует отметить, что для других платформ пакета данное число может быть большим. Файлы **RAW**-вида, открываемые по ***open***-функции, обрабатываются как файлы ***BINARY***-типа. В остальном сказанное относительно ***fopen***-функции в значительной степени относится и к ***open***-функции. Все возникающие особые и ошибочные ситуации при вызове ***open***-функции вполне адекватно отражаются выводимой диагностикой, *доступной* для обработки рассмотренными ниже средствами ***Maple***-языка.

Наконец, по ***iostatus*()**-функции предоставляется возможность 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 к файлам (*например*, ***readline***-*функцией*). Функция возвращает результат своего вызова в виде списочной структуры, содержащей 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.

Тогда как, начиная с четвертого (*если имеются открытые файлы*), элементы возвращаемого ***iostatus***-функцией списка представляют собой списки (*в количестве, соответствующем 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 (**STREAM, RAW, PIPE, PROCESS, DIRECT**);
4 – FP = адрес указателя для **STREAM**-файла и **FD =** номер для файлов другого вида;
5 – текущий режим доступа к файлу (***READ, WRITE***);
6 – тип файла (***TEXT, BINARY***).

При этом, второй элемент списка-*описателя* файла представляет *непосредственный* идентификатор файла, если файл относится к {**STREAM**|**RAW**}-виду либо идентификатор {***default***|***terminal***} для файлов **DIRECT**-вида. Четвертый элемент списка-описателя дает внутреннее значение либо для указателя данного файла (**FP**), или номера (**FD**) логического канала в/в. Данная информация представляет *особый* интерес при работе с файлами **PIPE**-вида, ибо значения номеров внутреннего логического канала и возвращаемого {***fopen, open, popen***}-функциями доступа в общем случае не совпадают. В случае отсутствия открытых файлов ***iostatus***-функция возвращает [0, 0, 7]-значение.

Закрытие всех открытых в текущем сеансе файлов производится как по предложению/функции {**done**|**stop**|**quit**}, так и по клавишам (**Alt+F4**) выхода из сеанса, а также и по **restart**-предложению, восстанавливающему состояние пакета на момент его начальной загрузки. Тогда как *выборочное* закрытие файлов производится по {***fclose***|***close***|***pclose***}-функции для *ранее* открытых соответственно по {***fopen***|***open***|***pipe***,***popen***}-функции файлов. Все эти функции имеют единый формат кодирования следующего вида:

{***fclose***|***close***|***pclose***}**(**<СФ1>, <СФ2>, ..., <СФn>**)**

где в качестве *единственного* аргумента выступает последовательность спецификаторов закрываемых файлов и/или логических каналов в/в. При этом, успешное завершение каждой из этих функций доступа закрывает соответствующие файлы и возвращает значение *NULL*. Явное закрытие файлов по рассмотренным функциям доступа не только обеспечивает *реальную* запись информации в файлы, но и *освобождает* занимаемые ими *логические* каналы в/в для других файлов, что в свете вышесказанного в целом ряде случаев представляется достаточно *актуальным*. Вообще говоря, все три функции закрытия файлов эквивалентны и взаимозаменяемы. Попытка закрыть *несуществующий* файл либо *незадействованный* логический канал в/в вызывает ошибочную ситуацию для ***Maple* 6/7,** тогда как для ***Maple 8-10*** возвращается *NULL*-значение, как и в случае существующих закрытых файлов во всех релизах. Это одна из досадных недоработок, способствующая *несовместимости* релизов. При этом, попытка закрыть файл {***default***|***terminal***}-вида вызывает ошибочную ситуацию. После закрытия логического канала в/в его номер вновь используется в текущем сеансе. Во избежание возникновения особых ситуаций рекомендуется производить закрытие файлов только после завершения работы с ними.

Наконец, следующие две функции доступа позволяют соответственно определять режим обязательной записи информации в файл и удалять как закрытые, так и открытые файлы из системы каталогов **DOS**. Прежде всего, по ***fflush*****(**<СФ1>, ..., <СФn>**)**- функции обеспечивается режим обязательной записи в файл при выполнении каждой такой операции. Это распространяется как на открываемые *неявно* файлы, так и на открываемые *явно* по {***fopen***|***popen***}-функции. При этом, данная процедура производится автоматически при *явном* закрытии файла. Вызов ***fflush***-функции ничего не возвращает. Использование ***fflush***-функции представляется целесообразным, например, при таком режиме работы с данными, когда в течение сеанса с пакетом производится длительное формирование результатов вычислений с их периодическим сохранением в файле в режиме дописывания (***APPEND***-режим), что позволяет существенно повышать надежность сохранности полученных результатов, перед реальной записью в файл накапливающихся в пакетном буфере обмена данных.

По функции ***fremove***(<СФ1>, ..., <СФn>) производится удаление из файловой системы **DOS** указанных своими *спецификаторами* и/или *номерами* логических каналов в/в 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 информации. Это не является *излишним*, ибо, например, по ***writeline***-функции производится запись информации 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. Применение ***fremove***-функции к *отсутствующему* файлу инициирует ошибочную ситуацию, которую можно обрабатывать программно. Успешное завершение вызова функции ***fremove*** возвращает *NULL*-значение. В следующем фрагменте иллюстрируется применение *всех* рассмотренных функциональных средств, образующих *оболочку* средств *доступа* к *внешним* файлам: открытие и закрытие файлов, тестирование состояния открытых файлов, а также удаление файлов из системы каталогов операционной среды **ЭВМ:**

```
> LK:= fopen(`C:/ARM_Book/Academy/Kristo`, WRITE, TEXT);   ⇒ LK := 0
> writeline(LK, `Russian`, `Academy of`, `Natural`, `Sciences`,` - 30.12.2006`);   ⇒ 50
> fclose(LK): LK:= fopen(`C:/ARM_Book/Academy/Kristo`, READ): M:= NULL:
  while LK <> P do P:= readline(LK): M:= M, P end do:   M:= M[k]$k=1..nops([M]) - 1;
                M := "Russian", "Academy of", "Natural", "Sciences", " - 30.12.2006"
> restart: readline("C:/ARM_Book/Academy/Kristo");   ⇒ "Russian"
> restart; Ln:= open("C:\\ARM_Book/Academy\\RANS.agn", WRITE):
  writeline(Ln, `Salcombe`, `Eesti`, `Ltd.`, `is terminated in`, `December 1999`);   ⇒ 51
> close(Ln): open("C:\\ARM_Book/Academy\\RANS.agn",READ): N:= NULL:
  while Ln <> P do P:= readline(Ln): N:= N, P end do: N:=N[k]$k=1..nops([N]) - 1;
                N := "Salcombe", "Eesti", "Ltd.", "is terminated in", "December 1999"
> K:= "C:/ARM_Book/Academy/": [readline(cat(K, Kristo)), readline(cat(K, `RANS.agn`))];
                                        ["Russian", "Salcombe"]
> iostatus();
    [3, 0, 7, [0, "C:\\ARM_Book/Academy\\RANS.agn", RAW, FD = 11, READ, BINARY],
       [1, "C:/ARM_Book/Academy/Kristo", STREAM, FP = 2009464032, READ, TEXT],
    [2, "C:/ARM_Book/Academy/RANS.agn", STREAM, FP = 2009464064, READ, TEXT]]
> Ofiles:= () -> `if`(iostatus() = [0, 0, 7], true, false): Ofiles();   ⇒ false
> CF:= cat(K, "RANS.agn"): fflush(CF): writeline(CF,"December 30th 2006");   ⇒ 19
> fclose(CF); fremove(CF, cat(K, `Kristo`));   ⇒ NULL
> map(writeline, [CF, cat(K, `RAC. REA`)], "Tallinn",`Moscow`,Grodno,`Vilnius`,Gomel):
> close(NULL): All_Close();   ⇒ NULL
```

С учетом сказанного особых вопросов фрагмент вызывать не должен. Следует обратить внимание на процедуру ***All_Close()***, вызов которой закрывает все открытые файлы.

```
All_Close := proc()
local a, k;
    assign(a = iostatus( )), close(`if`(a = [0, 0, 7], RETURN(NULL), seq(
    `if`(member(a[k][2], {'terminal', 'default'}), NULL, a[k][2]), k = 4 .. nops(a))))
end proc
```

Использованные выше для иллюстрации еще не рассмотренные процедуры ***writeline*** и ***readline*** следует пока рассматривать как некоторые *формальные* операции записи и чтения данных. Детализация их представляется несколько ниже.

6.2.2. *Средства обработки особой ситуации «конец файла» доступа к файлам данных*

При работе с файлами весьма существенную ее компоненту составляет обработка особой ситуации *«конец файла»*, особенно актуальная в условиях использования функциональных средств доступа, поддерживаемых ядром ***Maple***. Для этих целей ***Maple***-язык предлагает тестирующую ***feof*(**<СФ>**)**-функцию, возвращающую *true*-значение только тогда, когда в процессе применения функций доступа ***readline, fscanf*** или ***readbytes*** к файлу **STREAM**-вида, определенному фактическим аргументом (*спецификатор либо номер логического канала*) функции, была реально обнаружена ситуация *«конец файла»*, в противном случае возвращается *false*-значение. При этом, если в качестве фактического аргумента ***feof***-функции был указан спецификатор и файл ранее не открывался, то он открывается в ***READ***-режиме доступа как ***BINARY***-файл.

По функции ***filepos*(**<СФ> {**, p**}**)** возвращается *номер* текущей *сканируемой* позиции файла, указанного первым фактическим аргументом (*спецификатор или номер приписанного ему логического канала*). Под сканируемой понимается такая позиция файла, в которую установлен указатель, т.е. в которую будет писаться либо из которой будет читаться логическая запись при очередной операции доступа к файлу. При этом, под *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 же 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 положительные значения из диапазона [1 .. *infinity*]; нулевое значение ***filepos***-функция возвращает для пустого файла и по достижении *конца* файла. Тогда как по вызову функции ***filepos*(**<СФ>, ***infinity*)** производится установка указателя в конец файла с возвратом количества байтов, составляющих файл, указанный ее первым фактическим аргументом. При этом, вызов ***filepos*(**<СФ>**)** *эквивалентен* вызову ***feof*(**<СФ>**)**. С учетом сказанного и представленных в книгах [12,41] замечаний, ***feof***-функция не может служить для надежного тестирования особой ситуации *«конец файла»* и для этих целей в режиме ***READ***-доступа к ***TEXT***-файлам следует использовать *непосредственно* ***readline***-функцию или ***filepos***-функцию. Например, по конструкции следующего простого вида: **R:= *readline*(**<СФ>**): R:= `if`(R = 0, NULL, R) R-**переменная получает *фактические* значения записей файла, указанного СФ-аргументом, либо *NULL*-значение по достижении конца файла. Следующий фрагмент иллюстрирует использование ***filepos***-функции и некоторые варианты обработки ситуации *«конец файла»*, которые наряду с ранее представленными могут оказаться весьма полезными:

```
> F:= "C:/temp/grsu": for k to 5 do writeline(F, "string"||k) end do: n:= filepos(F, infinity):
  close(F);
> h:= 1: s:= NULL: while h <> 0 do h:= readline(F); s:=s, h end do: close(F), s, iostatus();
                "string1", "string2", "string3", "string4", "string5", 0, [0, 0, 7]
> h:= 1: s:= NULL: while filepos(F) <> n do s:= s, readline(F)  end do: close(F), s, iostatus();
                "string1□", "string2□", "string3□", "string4□", "string5□", [0, 0, 7]
```

Первый пример фрагмента представляет создание простого ***TEXT***-файла, содержащего **5** строк (*записей*), определяется его размер, затем файл закрывается. Во втором примере построчно считываются записи файла до получения ***readline*-**функцией *нулевого* значения, зпатем файл закрывается и производится проверка состояния логических каналов

в/в. Наконец, третий пример *подобен* второму с тем отличием, что ситуация «*конец файла*» отслеживается проверкой на исчерпание всего файла. Здесь мы можем обнаружить в конце выводимых строк символ «**\n**», определяющий перевод строки и возврат каретки. Это обусловлено тем, что *второй* способ контроля использует количество считанной информации, а не ее *построчную* обработку. Поэтому в выходных данных и появляются управляющие символы. В представленных примерах для иллюстрации были использованы не рассмотренные функции ***writeline*** и ***readline,*** которые следует рассматривать пока как некоторые формальные операции *записи* и *чтения* данных соответственно.

Существенное *различие* между файлами типов ***TEXT*** и ***BINARY*** имеет место относительно обработки ситуации «*the end of datafile*» («*конец файла*»). Для ***BINARY***-файлов у ***Maple*** нет средств для обработки этой особой ситуации и в каждом конкретном случае данная задача возлагается на пользователя, что во многих случаях нежелательно. Данный вопрос достаточно детально рассматривался в наших книгах [12,41,103]. С целью решения данной проблемы нами были предложены две процедуры ***Fend*** и ***Find.***

```
Fend := proc(F::{integer, string, symbol})
local k, a, b, c, p, h;
    assign(a = iostatus( )), seq(`if`(a[k][1] = F or a[k][2] = cat("", F),
        assign('h' = 9, 'c' = a[k][1]), NULL), k = 4 .. nops(a));
    if h ≠ 9 then error "<%1>: datafile is closed or datafile descriptor is not used",F
    end if;
    try
        assign(p = filepos(c));
        if filepos(c, ∞) ≤ p then return assign(b = filepos(c, p)), true
        else return false, `if`(nargs = 1, assign(b = filepos(c, p)), op(
            [assign([args][2] = [p, filepos(c, ∞) − p]), assign(b = filepos(c, p))]))
        end if
    catch : null(
        "Processing of an especial situation with datafiles {direct, process, pipe}")
    end try
end proc

Find := F::{0, 1, 2, 3, 4, 5, 6} → [`if`(type(eval(cat(__filesize, F)), 'symbol'),                    ]
    [assign(cat(__filesize, F) = filepos(F, ∞)), filepos(F, 0)], NULL),                              -1
    `if`(eval(cat(__filesize, F)) ≤ filepos(F), [true, unassign(cat(__filesize, F))]1, false)

> s:= NULL: p:= open(F, READ): while not Fend(p) do s:= s, readline(p) end do: close(F), s,
  iostatus();    ⇒ "string1□", "string2□", "string3□", "string4□", "string5□", [0, 0, 7]
> s:= NULL: p:= open(F, READ): while not Find(p) do s:= s, readline(p) end do: close(F), s,
  iostatus();    ⇒ "string1□", "string2□", "string3□", "string4□", "string5□", [0, 0, 7]
```

Вызов ***Fend*(F {, 'h'})** возвращает *true* в случае обнаружения у файла **F,** заданного СФ или номером логического канала, ситуации «*конец файла*» и *false* в противном случае. Через второй необязательный аргумент в случае *false*-значения возвращается список формата [*сканируемая позиция файла*, *длина остатка файла*]. Процедура ***Fend*** имеет смысл только для открытого файла. В отличие от ***Fend,*** процедура ***Find*** имеет только один аргумент, определяющий номер логического канала, по которому открыт тестируемый файл. Во фрагменте использован файл, полученный в предыдущем фрагменте.

6.2.3. *Обработка особых и ошибочных ситуаций процедур доступа к файлам данных*

В процессе выполнения процедур доступа к файлам данных возможны различного *рода* ошибочные и особые ситуации, *наиболее* часто встречающиеся из которых рассмотрены в наших книгах [12,32]. Обработку данных ситуаций можно производить, например, на основе функции ***traperror,*** переменных пакета ***lasterror*** и ***tracelast,*** функции ***ERROR*** и предложений **error** и **try,** из которых последнее является наиболее развитым средством обработки ошибочных ситуаций в *целом*. Нижеследующий фрагмент иллюстрирует использование **try**-предложения при реализации процедуры ***BootDrive,*** вызов ***BootDrive*()** которой возвращает 2-элементный список, чей первый элемент определяет логическое имя устройства начальной загрузки операционной системы, тогда как второй элемент определяет путь к главному системному каталогу. Предложение **try** рассмотрено выше.

```
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 Fend(n) do
                h := CF2( readline(n) );
                if Search2(h, {"multi(", "signature("}) ≠ [ ] 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], "\_$Art16_Kr9$_") ),
        system( cat("Ver > ", f) );
    assign( v = CF2( readbytes(f, 'TEXT', ∞) )), 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;
        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"]
```

При организации обработки особых и ошибочных ситуаций, возникающих при выполнении процедур доступа к файлам данных, следует обратить внимание на **tracelast**-команду, возвращающую как *последнюю* запись из стэка ошибок, так и выводящую дополнительную полезную информацию по *возникшей* ситуации. Данная информация может оказаться полезной при тестировании ***Maple***-процедур. Команда **tracelast** позволяет получать информацию *независимо* от ***trapperror***-функции, **try**-предложения и ***lasterror***-переменной. Однако, она не применима в теле процедур и ее следует использовать сразу же за местом возможного появления ошибочной ситуации, ибо обеспечиваемая командой информация может быть удалена средствами сборки *«муссора»* пакета. Детальнее с *основными* ошибочными и особыми ситуациями в/в пакета ***Maple*** можно ознакомиться по **?IO_errors-**конструкции.

6.3. *Базовые средства Maple-языка для обеспечения доступа к внешним файлам данных TEXT-типа*

Не отвлекаясь на детали, принципиальную схему организации средств доступа к внешним файлам можно с учетом вышесказанного представить в виде следующей простой схемы, позволяющей *систематизировать* изложение, хотя она и носит в значительной степени субъективный характер:

<table>
<tr><td colspan="3">Оболочка средств доступа: fopen, open, popen (функции открытия файлов)</td></tr>
<tr><td>TEXT-файлы</td><td>BINARY-файлы</td><td>FORMAT-файлы</td></tr>
<tr><td>writeline, readline</td><td rowspan="3">writebytes, readbytes</td><td>writedata, readdata</td></tr>
<tr><td>writestat, readstat</td><td rowspan="2">fprintf
fscanf</td></tr>
<tr><td>writeto, appendto</td></tr>
<tr><td colspan="3">Оболочка средств доступа: fclose, close, pclose (функции закрытия файлов)</td></tr>
<tr><td colspan="3">Общие функции доступа: feof, filepos, fremove, fflush, iostatus, system</td></tr>
</table>

В данной схеме выделяются три уровня: *оболочка*, *функции* непосредственного доступа к файлам и *общие обслуживающие* функции. *Оболочка* средств доступа обеспечивает 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 (***READ***, ***WRITE***, ***APPEND***). Хотя средства данного уровня в общем случае и не обязательны, т.к. функции непосредственного доступа (*второй уровень*) обеспечивают открытие файлов, а по завершении работы с пакетом все открытые в сеансе файлы автоматически закрываются, в целом ряде случаев использование средств оболочки не только желательно, но и обязательно, позволяя обеспечивать более *гибкое* управление общим механизмом доступа к внешним файлам пакета. Функциональные средства оболочки достаточно детально были рассмотрены в предыдущем разделе главы.

Третий уровень составляют *общие средства* доступа, обеспечивающие ряд *сервисных* процедур по обслуживанию файлов пользователя. Набор их невелик и поддерживаемые ими функции сводятся к *удалению* файлов (***fremove***) из файловой системы компьютера, тестированию состояния открытых файлов (***iostatus***), а также идентификации особой ситуации «*конец файла*» (***feof***) и текущей позиции сканирования в файле (***filepos***) наряду с обеспечением гарантированной записи данных в файл (***fflush***). Вместе с тем, 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 пакета собственными средствами обслуживания файлов.

Наконец, *второй уровень* представляет собственно функциональные средства непосредственного доступа к внешним файлам пакета, обеспечивая операции записи и чтения информации. Данные средства как в плане их *основной* функциональной нагрузки, так и в плане изложения целесообразно разделить на три группы, обеспечивающие работу с файлами соответственно типов ***TEXT***, ***BINARY*** и ***FORMAT***. Первые *две* группы файлов стандартно определяются ядром пакета 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 пакета. Более того, в целом ряде случаев классифицированные нами по трем группам средства могут функционально взаим-

но *перекрываться*, однако это вопрос отдельного рассмотрения. Суть каждого из указанных типов доступа к файлам будет рассматриваться ниже, сейчас же мы переходим к обсуждению средств непосредственного доступа к широко используемым в среде языка файлам ***TEXT***-типа.

По функции доступа ***writeline*(<СФ>, S1 ,S2, ..., Sn)** производится запись в указанный первым фактическим СФ-аргументом файл строчных **Sk**-выражений, разделяемых управляющим символом **hex(0D0A)** *«перевода строки и возврата каретки»*. В случае отсутствия второго фактического аргумента в файл помещается пустая строка, т.е. *один* управляющий **hex(0D0A)**-символ. Успешное завершение вызова ***writeline***-функции возвращает *общее число* записанных за одну операцию символов. При указании в качестве приемного файла стандартных каналов ***default*** либо ***terminal*** запись производится на приписанные им при загрузке ядра пакета системные устройства/каналы. При этом, если же файл *предварительно* был открыт в ***READ***-режиме и является файлом {***STREAM*|*RAW***}-вида, то по ***writeline***-функции он переоткрывается в ***WRITE***-режиме доступа как файл ***TEXT***-типа с сохранением последней позиции сканирования в качестве текущей. То же самое происходит и в случае закрытого файла. При этом, если ***writeline***-функция применяется к закрытому файлу, то он полностью обновляется, в противном случае производится *дозапись* в его текущую *сканируемую* позицию, т.е. реализуется ***APPEND***-режим.

По обратной к предыдущей ***readline*(<СФ>)**-функции производится чтение очередной записи-*строки* из указанного фактическим СФ-аргументом файла, где в качестве значения СФ-аргумента (*подобно другим функциям доступа*) может выступать как спецификатор файла, так и номер *приписанного* ему открытого логического канала в/в. Успешный вызов ***readline***-функции возвращает либо действительную строку без завершающего ее управляющего **hex(0D0A)**-символа, либо нулевое значение, идентифицирующее ситуацию *«конец файла* (*eof*)». При этом, во втором случае файл закрывается, если в качестве аргумента ***readline***-функции был указан спецификатор файла. Следующий фрагмент иллюстрирует использование рассмотренных функций доступа к внешним файлам:

```
> F:="C:/tmp/trg": n:=fopen(F, WRITE): for k to 5 do writeline(n, "Str"||k) end do: close(n);
> n:= fopen(F, READ): s:= NULL: while not Fend(n) do s:=s, readline(n)) end do: s, close(n);
                        "Str1", "Str2", "Str3", "Str4", "Str5"
> n:= fopen(F, APPEND):  for k to 5 do writeline(n, "Str_"||k) end do: close(n);
> n:= fopen(F, READ): s:= NULL: while not Fend(n) do s:= s, SUB_S(["\n"=""], readline(n))
  end do: s, close(n);
          "Str1", "Str2", "Str3", "Str4", "Str5", "Str_1", "Str_2", "Str_3", "Str_4", "Str_5"
> n:=fopen(F, READ): s:=NULL: while not feof(n) do s:=s,readline(n) end do: s, close(n);
          "Str1", "Str2", "Str3", "Str4", "Str5", "Str_1", "Str_2", "Str_3", "Str_4", "Str_5", 0
```

В данном фрагменте посредством ***writeline***-функции в цикле создается файл, содержащий 5 записей-строк. После закрытия посредством функции ***readline*** возвращается его содержимое. Затем после закрытия файл вновь открывается в ***APPEND***-режиме доступа для дозаписи информации в его конец. Дозапись производится в цикле, расширяя уже существующий файл еще на 5 записей. В завершении файл открывается на чтение для вывода его обновленного состояния. В данном фрагменте рекомендуется обратить внимание на использование нашей процедуры ***Fend*** для обработки особой ситуации *«конец файла»* вместо стандартной функции ***feof***, которая не обеспечивает в ряде конструкций приемлемого результата, как это иллюстрирует последний пример фрагмента.

При каждом новом обращении к ***readline***-функции считывается очередная строка файла и указатель (*сканирующий позицию в файле*) устанавливается на начало записи, следующей за считанной. По достижении указателем *конца файла* функция возвращает *нуль-*

значение; в случае невозможности выполнить очередную операцию чтения инициируется ошибочная ситуация. Если файл, к которому применяется ***readline***-функция, оказывается закрытым, то он открывается как ***TEXT***-файл в ***READ***-режиме доступа. Функция ***readline*** может использоваться в составе более *сложных* конструкций на правах *обычной* функции, возвращающей в качестве результата очередное считанное значение типа *string*. Следующий фрагмент представляет процедуру ***Nstring,*** возвращающую строку с указанным вторым фактическим аргументом номером из заданного первым аргументом файла ***TEXT***-типа. В качестве *второго* аргумента могут выступать множество либо список номеров искомых строк, возвращаемых в виде списка.

```
Nstring := proc(F::{string, symbol}, S::{set(posint), list(posint)})
local k, p, s, f, ω, ω1, ω2, ν, ν1;
    `if`(type(F, 'file') and type(F, 'rlb'),
        `if`(Empty(S), ERROR("the second argument is empty"), NULL),
        ERROR("<%1> is not a file or has a type different from TEXT",F));
    ω1, p, k :=
        () → WARNING("all required lines are absent in file"), [Close(F)], 0;
    ω2, ν1 := ν → WARNING("lines with numbers %1 are absent in file",ν),
        () → `if`(type(f, 'list'), true, false);
    `if`(nargs = 3 and type(eval(args[3]), {'string', 'symbol'}), [
        assign67(ω = interface(warnlevel)), null(interface(warnlevel = 0)),
        assign(f = MkDir(args[3], 1))], assign(f = []));
    assign('p' = fopen(F, 'READ', 'TEXT'), ν = {op(S)}),
        `if`(ν1(f), 1, null(interface(warnlevel = ω)));
    while not Fend(p) do
        [assign('s' = readline(p), 'k' = k + 1)];
        if member(k, ν) then
            `if`(ν1(), assign('f' = [op(f), s]), writeline(f, s)); ν := ν minus {k}
        end if
    end do;
    if ν = {op(S)} then RETURN(ω1(), `if`(ν1(),f, fremove(f)), Close(F))
    elif ν ≠ {} then ω2(ν), `if`(ν1(f), RETURN(f, Close(F)), NULL)
    else
    end if;
    Close(F), `if`(ν1(f),f,
        op([WARNING("the resultant file is in <%1>",f), Close(f),f]))
end proc
> Nstring("C:/RANS/TRG.txt", [7, 14, 17, 56, 61]);
Warning, lines with numbers {56, 61} are absent in file
                        ["7aaaaaaaaaa", "17aaaaaaaa", "14aaaaaaa"]
> Nstring("C:/RANS/TRG.txt", [7, 14, 17], "C:/Temp/Order/ABS.txt");
Warning, the resultant file is in <c:\temp\order\abs.txt>
                                    "c:\temp\order\abs.txt"
```

В случае отсутствия *искомой* записи возвращается соответствующее диагностическое сообщение. Тогда как третий необязательный аргумент позволяет определять приемный файл для указанных строк-записей исходного **F**-файла данных.

Следующий фрагмент представляет полезную процедуру ***FTabLine,*** обеспечивающую более эффективную обработку больших ***TEXT***-файлов данных различного назначения.

```
FTabLine := proc(F::file)
local d, k, h, m, n, p, T, x, y, t, avz, ω;
    assign( ω = cat( F, ".m"), ν = cat( ``, sstr( [ "." = "_" ], CFF( F )[ -1 ], avz ), "_fldt") ),
        `if`( type( ω, 'file'), ( proc()
            read ω;
            RETURN( WARNING( "the FLDT has been activated as <%1>", ν ) )
        end proc )( ), 7 );
    `if`( type( F, 'rlb' ), assign( k = 1, T[ 1 ] = 0, m = 0, n = ∞, Close( F ),
        t = fopen( F, 'READ', 'TEXT' ) ),
        ERROR( "<%1> has a type different from TEXT", F ) );
    while not Fend( t ) do readline( t ); k := k + 1; T[ k ] := filepos( t ) end do;
    for k to nops( op( 2, eval( T ) ) ) − 1 do T[ k ] := [ T[ k ], T[ k + 1 ] − 3 ] end do;
    Close( F ), ( proc(t) T[ t ] := 'T[ t ]'; NULL end proc )( k ), table( op( 2, eval( T ) ) ),
        null( `if`( 1 < nargs and args[ 2 ] ≠ 'FLDT', [
        assign( d = nops( op( 2, eval( T ) ) ) ), add( _1N( 0, [
        assign( 'p' = T[ k ][ 2 ] − T[ k ][ 1 ] + 1 ),
        `if`( m < p, assign( 'm' = p, 'x' = [ k, p ] ), 1 ),
        `if`( p < n, assign( 'n' = p, 'y' = [ k, p ] ), 1 ) ] ), k = 1 .. d ),
        assign( args[ 2 ] = [ d, y, x ] ) ], 1 ) );
    if member( 'FLDT', { args } ) then
        save1( cat( ``, sstr( [ "." = "_" ], CFF( F )[ -1 ], avz ), "_fldt"), T, ω );
        eval( T ), WARNING( "the FLDT for file <%1> has been saved in file <%2>",
            F, ω ), unassign( ν )
    else eval( T )
    end if
end proc
```

```
> GenFT("C:/Temp/Veeroj.SV", 61, 20);  T:= FTabLine("C:/Temp/Veeroj.sv", 'FLDT');
Warning, the resultant file is in <c:\temp\veeroj.sv>
                                        [1, 58, 666]
Warning, the saving result is in datafile <c:/temp/veeroj.sv.m>
Warning, the FLDT for file <c:/temp/veroj.sv> has been saved in file <c:/temp/veroj.sv.m>
T := table([1 = [0,0], 2 = [3, 8], 3 = [11, 64], 4 = [67, 77], 5 = [80,116], 6 = [119,167], 7 = [170, 201],
8 = [204, 240], 9 = [243, 300], 10 = [303, 351], 11 = [354, 382], 12 = [385, 386], 13 = [389, 415],
14 = [418, 467],15 = [470, 518], 16 = [521, 563], 17 = [566, 594], 18 = [597, 647], 19 = [650, 660],
20 = [663, 663]])
> FTabLine("C:/Temp/Veeroj.sv", 'h'):  h;
Warning, the FLDT has been activated as <veeroj_sv_fldt>
                                        [20, [1, 1], [9, 58]]
> eval(veeroj_sv_fldt);
T := table([1 = [0,0], 2 = [3, 8], 3 = [11, 64], 4 = [67, 77], 5 = [80,116], 6 = [119,167], 7 = [170, 201],
8 = [204, 240], 9 = [243, 300], 10 = [303, 351], 11 = [354, 382], 12 = [385, 386], 13 = [389, 415],
14 = [418, 467],15 = [470, 518], 16 = [521, 563], 17 = [566, 594], 18 = [597, 647], 19 = [650, 660],
20 = [663, 663]])
```

Вызов процедуры ***FTabLine*(F)** возвращает специальную *таблицу диспозиции строк файла* (***FLDT***), чьими *входами* являются *номера* строк ***TEXT***-файла, заланного **F**-аргументом, то-

гда как *выходами* являются 2-элементные списки, чьи элементы определяют *начальные* и *конечные* позиции соответствующей строки в файле **F.** При этом, при кодировании необязательного второго аргумента через него возвращается 3-элементный список вида **[d, [k, Min], [p, Max]],** где: **d** – число строк в файле **F** и **k, p** – номер строки с минимальной и максимальной *длиной* соответственно. Знание данной информации, в частности, позволяет организовывать быстрый доступ к строкам файла данных не последовательно, а по номерам строк, реализую своего рода *индексный* метод доступа для достаточно больших файлов данных ***TEXT***-типа.

В случае кодирования ключевого параметра ***FLDT*** процедура дополнительно позволяет сохранять ***FLDT*** для файла данных **F** в файле, путь к которому определяется как ***cat*(F, ".m").** Последующие вызовы ***FTabLine*(F)** в случае *доступности* такого ***m****-файла* активируют глобальную переменную с именем, указываемым в выводимом сообщении. В общем случае имя имеет следующий вид ***aaa_bbb_fldt,*** где ***aaa.bbb*** – имя **F** файла. Такая возможность оказывается довольно полезной, если файл не подвергается частым *обновлениям*.

Наряду с представленными, нами был создан ряд других полезных средств для работы с файлами данных ***TEXT***-типа. В частности, процедуры {***writedata1, readdata1***} существенно расширяют возможности стандартных процедур {***writedata, readdata***}, обеспечивая сохранение в файле данных любого типа {*list*, *listlist*, *vector*, *matrix*, *Matrix*}. При этом, сохраняемые данные формируются построчно с разделением их в строках пробелами.

Процедуры ***DAopen, DAread*** и ***DAclose*** обеспечивают поддержку *прямого доступа* к файлам ***TEXT***-типа (*и даже более общего* ***rlb****-типа*), точнее к его строкам-записям. Процедура ***DAopen*(F)** предназначена для открытия произвольного ***TEXT***-файла данных, заданного полным путем к нему в режиме *прямого* доступа к его записям (*строкам*) согласно их номерам в файле. В результате успешного вызова данной процедуры создается специальный файл данных (*доступный в том же каталоге, что и исходный файл* **F**). Данный файл содержит таблицу, чьи *входы* – номера строк файла **F,** тогда как *выходы* – соответствующие им *позиции* в файле. В результате вызова процедуры ***DAopen*(F)** производится активизация в текущем сеансе таблицы, соответствующей файлу данных F.

После этого, посредством вызовов процедуры ***DAread*(F, n)** в текущем ***Maple***-сеансе мы получаем доступ *непосредственно* к **n**-й записи файла данных **F,** возвращая ее значение. С таким файлом данных можно одновременно работать и стандартными средствами доступа, принимая во внимание влияние вызовов процедуры ***DAread*(F, n)** на позиционирование указателя текущей записи **F**-файла. Закрытие файла **F,** открытого по вызову ***DAopen*(F),** производится вызовом процедуры ***DAclose*(F),** который обеспечивает как естественное закрытие **F**-файла, так и удаление файла с сопутствующей ему таблицей, а также деактивацию в текущем сеансе соответствующей таблицы. Ниже приведен пример применения указанных средств с оценкой получаемого временного выигрыша.

```
> F:= "C:/Academy/TRG.txt": for k to 10000 do writeline(F,cat("", k)) end do: Close(F);
  t:= time(): for k to 10000 do readline(F) end do: time() - t; All_Close();   ⇒ 3.316
> DAopen(F); t:=time(): DAread(F, 3000), time() - t; t:=time(): DAread(F, 6000), time() - t;
                                "3000", 0.655
                                "6000", 0.622
```

Наряду с представленными, для файлов данных ***TEXT***-типа нами был создан ряд достаточно полезных средств различного назначения, с которыми можно ознакомиться в [41, 103,108,109]. Многие из них представляют не только чисто практический интерес, но и с *учебной* точки зрения полезны, используя эффективные приемы программирования алгоритмов, специфических для задач доступа к текстовым файлам данных.

6.3.1. *Базовые средства доступа к файлам данных на уровне Maple-выражений*

Выше нами были рассмотрены средства работы с текстовыми файлами на уровне логических записей, в качестве которых выступали выражения *string*-типа – текстовые строки, *разделяемые* управляющими ***hex*(0D0A)**-символами *«перевода строки и возврата каретки»*. Для обеспечения доступа к файлам на уровне произвольных *выражений* ***Maple*-я**зык располагает функциями ***writestat*** и ***readstat*,** обеспечивающими соответственно запись и чтение *логических* записей, в качестве которых выступают последовательности ***Maple***-выражений, разделенные запятой и завершающиеся управляющими ***hex*(0D0A)**-символами *«перевода строки и возврата каретки»*. В этом отношении указанные средства также имеют дело с внешними файлами ***TEXT***-типа.

По функции ***writestat*(**<СФ>**,V1,...,Vn)** производится операция *записи* в текстовый файл, определяемый первым СФ-аргументом (*спецификатор файла или номер приписанного ему логического канала*), последовательности **Vk**-выражений, завершающейся управляющим **hex(0D0A)**-символом *«перевода строки и возврата каретки»*. При этом сами элементы последовательности разделяются запятой. Если отсутствует второй фактический аргумент функции, то ее вызов производит запись в файл *пустой* строки, т.е. просто *управляющий* ***hex*(0D0A)**-символ. В случае успешного завершения *вызова* ***writestat***-функции возвращается записанное за операцию количество символов. Если до вызова ***writestat***-функции принимающий файл был закрыт, то он открывается как ***TEXT***-файл **STREAM**-*вида* в режиме ***WRITE*** доступа, полностью *обновляя* существующий файл. Относительно стандартных файлов ***default*** и ***terminal*** функция ведет себя *аналогично* случаю ***writeline***-функции доступа. При записи в файл ***Maple***-выражений они проверяются на синтаксическую корректность; при этом, вызов функции ***writestat*(*default*, V1, ..., Vn)** эквивалентен вызову **lprint(V1, ...,Vn)**-функции вывода на печать. Если до применения ***writestat***-функции файл был *создан* как файл {**STREAM**|**RAW**}-вида и открыт в ***READ***-режиме доступа, то по ***writestat***-функции он переоткрывается как ***TEXT***-файл в ***WRITE***-режиме доступа, не изменяя текущей позиции сканирования, т.е. допускается операция записи в файл, начиная с требуемой его позиции.

```
> F:= "C:/ARM_Book/Academy/REA.07": readline(F); iostatus(); close(F);
                "Mathematical Theory of Homogeneous Structures (Cellular Automata)"
     [1, 0, 7, [0, "C:/ARM_Book/Academy/REA.07", STREAM, FP=6913794, READ, TEXT]]
> writestat(F, [V = 64, G = 59, S = 39], Z = sqrt(V^2 + G^2 + S^2));      ⇒ 50
> writestat(F, `Package Maple 8`, `Professional Edition`, `Release 8`);   ⇒ 55
> close(F): readline(F); readline(F), readline(F);
                                "[V = 64, G = 59, S = 39], Z = (V^2+G^2+S^2)^(1/2)"
                              "`Package Maple 8`, `Professional Edition`, `Release 8`", 0
```

В приведенном фрагменте считывается первая строка текстового **F**-файла и по ***iostatus***-функции определяются характеристики открытого файла. После этого, *двумя* вызовами ***writestat***-функции в закрытый файл помещаются записи, содержащие последовательности ***Maple***-выражений, полностью обновляя содержимое существующего файла, что подтверждает *последующее* его считывание тремя вызовами ***readline***-функции. В случае открытого файла ***TEXT***-типа по ***writestat***-функции производится *дозапись* в него по месту установки указателя в результате последней операции с файлом.

Следует иметь ввиду, что файлы данных, созданные ***writestat***-функцией, не читаются предложением **read**. Для этих целей можно использовать несложную *процедуру* ***wsread,*** исходный текст которой и примеры применения иллюстрирует следующий фрагмент.

```
wsread := proc(F::file)
local a;
    if member(readbytes(F, TEXT, ∞)[-1], {":", ";"}) then
        close(F), WARNING("datafile is ready for the read call")
    else
        assign(a = cat("`_Res$$$`:=", readline(F), ";")), filepos(F, 0),
            writeline(F, a), close(F);
        WARNING("datafile <%1> had been converted in situ", F)
    end if;
    if 1 < nargs then
        read F;
        WARNING(
            "result is in the variable `_Res$$$` if datafile had been converted")
    end if
end proc
> writestat("D:\\work\\aladjev/datafile", (a+b)/(c+d), (x+y)^2, parse("assign(Proc=proc()
`+`(args) end proc)"));  ⇒ 64
> iostatus();
    [1, 0, 7, [0, "D:/work/aladjev/datafile", STREAM, FP = 2009398496, WRITE, TEXT]]
> wsread("D:/work/aladjev/datafile");
Warning, datafile <D:/work/aladjev/datafile> had been converted in situ
> writestat("D:/work/aladjev/datafile",(a+b)/(c+d),(x+y)^2,parse("assign(Proc=proc()
`+`(args) end proc)"));  ⇒ 64
> wsread("D:/work/aladjev/datafile", 18);
Warning, result is in the variable `_Res$$$`
> `_Res$$$`;
                                    a + b      2
                                    ─────, (x + y)
                                    c + d
> Proc(64, 59, 39, 44, 10, 18);  ⇒ 234
```

Первый аргумент процедуры определяет искомый файл, как правило, созданный процедурой ***writestat.*** Тогда как второй необязательный аргумент (*произвольное* ***Maple****-выражение*) определяет режим обработки исходного файла **F.** Если файл читаем предложением **read,** то при наличии второго аргумента он загружается в текущий сеанс. Иначе выводится соответствующее сообщение. Если же файл **F** был создан ***writestat*-**процедурой, то он конвертируется *«на месте»* в файл, пригодный для чтения **read**-предложением, с выводом соответствующего сообщения. Тогда как при наличии второго необязательного аргумента отконвертированный файл *дополнительно* загружается в текущий сеанс; результат загрузки становится доступным через **`_Res$$$`-**переменную.

По функции {***writeto*** | ***appendto***}**(**<СФ>**)** производится *переключение* режима вывода информации всех последующих ***Input***- и ***Output***-параграфов текущего сеанса в файл, указанный фактическим СФ-аргументом (*спецификатором, но не номером логического канала*), в режиме доступа соответственно {*обновления* | *дописывания*}. Таким образом, все последующие за вызовом функции результаты вычислений, не отображаясь на экране, непосредственно выгружаются в принимающий файл, определенный СФ-аргументом функции {***writeto*** | ***appendto***}. Выход из данного режима обеспечивает вызов ***writeto*****(*****terminal*****).**

Если в качестве принимающего СФ-файла определен уже существующий файл, то он в зависимости от функции {***writeto*** | ***appendto***} соответственно полностью обновляется либо дополняется новой информацией текущего сеанса.

Из анализа содержимого СФ-файла следует, что в нем оказывается, собственно говоря, протокол текущего сеанса. Таким образом, рассмотренное средство предполагает ряд интересных приложений (*протоколирование работы с пакетом*, *хранение результатов вычислений и т.д.*). Так, например, следующий простой фрагмент иллюстрирует результат использования сохраненного протокола части текущего сеанса работы с пакетом.

```
> F:= "C:/temp/program.txt": writeto(F);
> P:= proc() `+`(args)/nargs end proc: M:= module() export x; x:= () -> `*`(args) end module:
> writeto(terminal);
> n:= fopen(F, READ): while not Fend(n) do writeline(cat(F, 1), readline(n)[2..-1]) end do:
> restart; F:= "C:/temp/program.txt1":  read(F); P(42,47,67,89,95,62), M:- x(64,59,39,44,10,17);
                                67, 1101534720
```

Определяется режим записи текущего документа в заданный **F**-файл, в котором сохраняется вторая строка фрагмента, после чего производится выход из ***writeto-***режима. На следующем шаге файл **F** обрабатывается путем удаления из всех его строк первых «**>**»-символов, и результат обработки сохраняется в файле ***cat*(F,1).** Затем выполняется предложение **restart**, определяется путь к *полученному* файлу, который и загружается в текущий сеанс по предложению **read,** делая доступными находящиеся в нем процедуру **P** и модуль **M.** Представленный фрагмент дает некоторое представление о путях возможного использования ***writeto-***режима. Как правило, более серьезное его использование требует и более сложного алгоритма обработки полученного в таком режиме *файла*, содержащего протокол части текущего сеанса. Между тем, данный режим скрывает и все *сообщения* пакета, что требует определенной осмотрительности. В этом отношении наиболее эффективным применение данного режима предполагается в процедурах либо файлах с ***Maple***-программами, загружаемыми по **read**-предложению (*см. прилож.* 3 [12]).

Вызовы процедуры ***wrtfile*(F)** и ***wrtfile*()** обеспечивают *переключение* с режима ***writeto*(F)** на режим ***writeto*('terminal'),** где **F –** путь к принимающему файлу. Данная процедура может использоваться и в теле других процедур, как иллюстрирут следующий пример.

```
wrtfile := proc(F::{string, symbol})
    if nargs = 0 then parse("writeto('terminal');", 'statement')
    else MkDir(F, 1); parse("writeto(" || """ || F || """ || ");", 'statement')
    end if
end proc
> P:=proc() wrtfile("C:/Temp/test"); parse("read(""C:/Temp/aaa.txt""); A, G, S;", 'statement');
   wrtfile() end proc:
> P();  readbytes("C:/Temp/Test", 'TEXT', infinity); close("C:/Temp/Test");
"Warning, extra characters at end of parsed string"
```

Однако и здесь не все выглядит так радужно, в частности, в таком режиме не сохраняются в файле ряд диагностических сообщений, обусловленных ошибочными ситуациями при работе с графическими объектами пакета, а в более общем случае с теми диагностическими сообщениями, которые не отражаются в ***lasterror***-переменной. Именно это не позволяет широко использовать данный подход для их процедурной обработки.

Наконец, по ***readstat***-функции предоставляется возможность считывать ***Maple***-предложения из *входного* потока, определенного стандартным в момент загрузки ядра пакета.

По умолчанию *стандартным* полагается ввод с консоли, но это может быть как терминал, так и дисковый файл. После считывания предложения возвращается результат его выполнения. Не останавливаясь детально на данной функции, отметим ее *базовый* формат кодирования вида ***readstat***(**{|**<*Метка*>**}),** по которому вводится без предваряющей метки либо с оной считанное из входного потока ***Maple***-предложение и выполняется, возвращая результат вычисления. Следующий фрагмент иллюстрирует *ввод* предложений из стандартного входного потока в ответ на вызов ***readstat***-функции:

```
> V:= 64:  G:= 59:  S:= 39:  Art:= 17:  Kr:= 10:  readstat(`Common Result: `);
[ Common Result: R:= evalf(sqrt(V^2 + G^2 + S^2 + Art^2 + Kr^2)/5, 3);     ⇒ 19.5
> readstat(`Multi-line input of Maple-sentence: `);
[ Multi-line input of Maple-sentence:  evalf(
[ Multi-line input of Maple-sentence:  sqrt(
[ Multi-line input of Maple-sentence:  V^2+
[ Multi-line input of Maple-sentence:  G^2+
[ Multi-line input of Maple-sentence:  S^2+
[ Multi-line input of Maple-sentence:  Art^2+
[ Multi-line input of Maple-sentence:  Kr^2)
[ Multi-line input of Maple-sentence:  /5,8);     ⇒ 19.480246
> readstat(`Input of two sentences: `);
[ Input of two sentences: V:=42.0614: G:=47.07.29:      ⇒ 42.0614
  Warning, extra characters at end of parsed string
> readstat(`Erroneous Input: `);
[ Erroneous Input: R:= 10***x+x*Kr(x)+Art(Gamma);
syntax error, `*` unexpected:
R:= 10***x+x*Kr(x)+Art(Gamma);
          ^
[ Erroneous Input: R:=10**x+x*Kr(x)+Art(Gamma);    ⇒ 10^x + 10 x + 17
```

Примеры фрагмента иллюстрируют как использование ввода с предваряющей меткой, так и без нее. При этом, допускается разбивать ввод на *несколько* строк, завершая их сборку в одно ***Maple***-предложение по {**;|:**}-разделителю. Вместе с тем по ***readstat***-функции допустим ввод единственного ***Maple***-предложения и попытка ввода уже двух предложений инициирует ошибочную ситуацию, как это иллюстрирует фрагмент. При этом, не взирая на предупреждающее сообщение, возвращается результат выполнения первого из предложений, независимо от завершающего его {**;|:**}-разделителя.

В случае попытки ввести неполное ***Maple***-предложение (*т.е. не завершенное разделителем* {**;|:**}) ***readstat***-функция запрашивает дополняющий ввод. При обнаружении синтаксической ошибки выводится соответствующее *диагностическое* сообщение и ***readstat***-функция выводит метку (*если она была определена*), предлагая повторить ввод. Детальнее с возможностями данной функции можно ознакомиться по справочной системе пакета, что не вызывает каких-либо затруднений. На основе данной функции, определяя в качестве стандартного входного потока файл, можно выполнять записанную в него программу, вызывая ее в рабочую область пакета предложение за предложением.

6.4. *Средства Maple-языка для обеспечения доступа к внешним файлам данных BINARY-типа*

В отличие от функциональных средств предыдущего раздела, 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, представляемые здесь средства обеспечивают доступ к файлам данных на уровне отдельных символов (*байтов*). При этом, в отличие от предыдущих средств доступа, данные средства рассматривают все составляющие файл символы однородными, не выделяя в них *управляющих* типа **hex(0D0A)**-символов *«перевода строки и возврата каретки»*. Для обеспечения эффективной работы с файлом на уровне отдельных записей-байтов он должен быть определен при открытии как ***BINARY***-файл. Прежде всего, для работы с ***BINARY***-файлами предназначена ***filepos***-функция, рассмотренная в предыдущем разделе. Поддерживающая работу с файлами как ***TEXT***-типа, так и типа ***BINARY,*** наибольший смысл она имеет для файлов именно *второго* типа, т.к. позволяет адресовать указатель на любую позицию файла, содержащую *искомый* символ (*байт*). В нижеследующем фрагменте уже первый пример иллюстрирует применение данной функции для определения объемов файлов как *текстовых*, так и *бинарных*, в частности загрузочных модулей среды ***MS DOS*** и ***Windows.*** Эта же функция используется и для идентификации ситуации *«конец файла»*, возвращая на ней *нулевое* значение. Базовыми же функциями для файлов ***BINARY***-типа являются функции ***writebytes*** и ***readbytes*** записи и чтения байтов соответственно.

По функции ***writebytes*****(**<СФ>**,** <*Набор байтов*>**)** производится операция записи в файл, указанный первым фактическим СФ-аргументом (*спецификатор или номер приписанного ему логического канала*), ***BINARY***-типа *набора* байтов (*символов*), определяемого ее вторым фактическим аргументом. В качестве второго фактического аргумента функции могут выступать или ***Maple***-строка (*цепочка символов* {*string*, *symbol*, *name*}-типа), составляющие символы которой помещаются в принимающий файл в *порядке* их следования, или список целочисленных значений из [0 .. 255]-интервала, представляющих собой *десятичные* коды символов из кодовой таблицы **ПК,** также помещаемых в файл в *порядке* их перечисления в списке. В частности, для записи байтов с кодами [0 .. 31] можно использовать только *вторую* форму представления *второго* фактического аргумента ***writebytes***-функции. В результате успешного *завершения* операции *записи*, инициированной функцией ***writebytes,*** возвращается число символов (*байтов*), записанных за операцию в файл.

Если до вызова ***writebytes***-функции принимающий файл был закрыт, то он открывается как файл **STREAM**-вида в ***WRITE***-режиме доступа, полностью обновляя существующий файл. При этом, тип принимающего файла определяется формой кодирования второго фактического аргумента ***writebytes***-функции следующим образом:

- символьная строка $\Rightarrow$ ***TEXT***-тип
- целочисленный список $\Rightarrow$ ***BINARY***-тип

Следующий фрагмент иллюстрирует как данное обстоятельство, так и простой прием конвертации списка ***hex***-значений в список десятичных значений кодов символов, который может быть полезным при практическом программировании, учитывая более привычный способ указания байтов в 16-*ричном* виде. При этом, относительно стандартных файлов ***default*** и ***terminal*** функция ведет себя *аналогично* рассмотренному выше случаю ***writeline***-функции доступа (*см. прилож.* 3 [12]). Если до применения функции ***writebytes***

файл был создан как файл {**STREAM**|**RAW**}-вида и затем *открыт* в режиме ***READ*** доступа, то по ***writebytes***-функции он переоткрывается в ***WRITE***-режиме доступа, не изменяя текущей позиции сканирования, т.е. допускается *дозапись* в файл, начиная с требуемой его позиции. Тогда как его тип определяется согласно вышесказанного в зависимости от формы кодирования второго фактического аргумента функции. Дополнительно к сказанному в следующем фрагменте представлен ряд примеров применения *функции* ***writebytes*** для работы с файлами как ***TEXT-,*** так и ***BINARY***-типов, а именно:

```
> currentdir(`C:/ARM_Book/Academy`):  map(filepos, ["Kristo.m", "RANS.99", TRG,
  `RAC.REA`, "D:/ Grodno/Maple.doc"], infinity);  ⇒ [50896, 250399, 180, 564378, 5149696]
> restart;  writebytes(`C:/ARM_Book/Academy/Salcombe`,"String of symbols");   ⇒ 17
> writebytes(Salcombe, map(convert, [`80`,`AB`,`A0`,`A4`,`EC`,`A5`,`A2`], decimal, hex)):
> close(Salcombe);  readline(Salcombe);   ⇒ "Ђ« ¤мЃў"
> writebytes(default, `Набор символов, выводимых на экран монитора`);   ⇒ 43
Набор символов, выводимых на экран монитора
```

Наконец, по ***readbytes***-функции, имеющей следующий формат кодирования:

readbytes(<СФ>, { |**L**|**L**, ***TEXT***|***TEXT***})

возвращается заданное фактическим **L**-аргументом функции *количество* байтов/символов, считанных, начиная с текущей (сканируемой) позиции файла, указанного первым фактическим СФ-аргументом (*спецификатор или номер приписанного ему логического канала*) функции. В случае отсутствия **L**-аргумента (*целого положительного числа либо infinity-значения*) считывается *единственный* байт, расположенный в сканируемой позиции файла. В случае указания для **L**-аргумента *infinity*-значения считываются все байты, расположенные между сканируемой позицией и концом файла. При определении ***TEXT***-аргумента возвращаемые ***readbytes***-функцией байты представляются строчной структурой, в противном случае – в виде списка значений своих десятичных кодов.

При этом, если при вызове ***readbytes***-функции определен ***TEXT***-аргумент, то операция чтения завершается либо по достижении нулевого байта, либо исчерпания указанного **L**-аргументом количества байтов. Если же в файле содержится либо остается меньшее число байтов, чем определено **L**-аргументом при вызове ***readbytes***-функции, то все они считываются; при пустом остатке функция возвращает *нулевое* значение (*т. е. обнаружена ситуация «конец файла»*), а сам файл автоматически закрывается, если в качестве фактического СФ-аргумента функции был указан *спецификатор* файла. Однако, если определенный СФ-аргументом ***readbytes***-функции файл закрыт, он открывается в ***READ***-режиме как файл **STREAM**-вида и его тип определяется в зависимости от наличия ***TEXT***-аргумента *аналогично* случая ***writebytes***-функции, рассмотренной выше. Для работы с ***BINARY***-файлами нами разработан ряд полезных процедур [41,103], например, ***statf***-процедура поизводит анализ открытых файлов относительно типов ***TEXT*** и ***BINARY.***

```
statf := proc()
local a, t, k;
    assign67( a = iostatus( ), t['TEXT'] = NULL, t['BINARY'] = NULL), `if`(
    nops( a ) = 3, NULL, [ seq( assign67('t[ a[ k ][ -1 ] ]' = t[ a[ k ][ -1 ] ],
    [ op( a[ k ][ 1 .. 3 ] ), filepos( a[ k ][ 1 ] ), a[ k ][ -2 ] ] ), k = 4 .. nops( a ) ),
    RETURN( eval( t ) ) ] )
end proc
> fopen("C:/a", 'WRITE'): writebytes("C:/b", [64, 59,39]): statf();
  table([TEXT = [0, "C:/a", STREAM, 0, WRITE], BINARY = [1, "C:/b", STREAM, 3, WRITE]])
```

6.5. *Обеспечение форматированного доступа к внешним файлам данных*

Рассмотрев в предыдущих разделах главы средства работы с внешними файлами типов ***TEXT*** и ***BINARY,*** структурная организация которых основывается соответственно на *логических* записях *строчного* и *символьного* типов, здесь остановимся на средствах доступа ***Maple-***языка для обеспечения работы как с числовыми, так и с более сложной форматизации структурами логических записей. Для этих целей ***Maple-***язык располагает двумя парами процедур {***writedata, readdata***} и встроенных функций {***fprintf, fscanf***}, две последние из которых довольно детально рассматривались выше и здесь будут рассмотрены лишь в контекстуальном плане. В дальнейшем ради унификации будем и процедуры {***writedata, readdata***} именовать *функциями*, принимая во внимание, что с точки зрения концепции функциональных средств пакета эти понятия можно отождествлять.

По функции ***writedata,*** имеющей следующий формат кодирования:

writedata*(**<СФ>**,** <*Данные*> {**,** <*Формат*>} {**, *Proc})

производится запись в указанный первым фактическим СФ-аргументом (*спецификатор или номер приписанного файлу логического канала в/в*) файл ***TEXT***-типа числовых данных, представленных в виде списка, вложенного списка, множества или матрицы и определяемых вторым фактическим аргументом функции. При необходимости записи числового массива его предварительно по ***convert***-функции следует конвертировать в вектор, матрицу, список либо множество. Два последующих аргумента функции необязательны и при их наличии расширяют форматирующие возможности функции. При определении для СФ-аргумента ***terminal***-значения вывод данных производится на экран монитора. Данный прием рекомендуется использовать перед окончательным сохранением данных в файле в целях их визуального контроля, особенно в случае достаточно сложно структурированных (*форматированных*) данных.

Если данные представлены в виде списка, вектора либо множества, то каждое числовое значение его элементов представляется в файле (*или выводится на печать*) в виде отдельной логической *записи*-строки, завершающейся управляющим символом ***hex*(0D0A)** «*перевода строки и возврата каретки*». Поэтому, например, по ***readline***-функции их можно циклически считывать. Если же данные представлены в виде вложенного списка либо матрицы, то они выгружаются в файл (*или выводятся на печать*) построчно с символами табуляции {**hex(09)**} в качестве разделителей элементов строк. В этом случае их можно считывать построчно посредством, например, той же ***readline***-функции. Начало нижеследующего фрагмента иллюстрирует сказанное. Как правило, в качестве элементов структур данных используются вычисляемые выражения типа {*integer*|*float*}, однако допускаются как комплексные, так и символьные выражения.

Третий необязательный аргумент ***writedata***-функции определяет формат данных, в котором они сохраняются в дисковом файле либо выводятся на печать. Для данного аргумента допускается {*integer*|*float* (*по умолчанию*)|*string*}-значение, на основе которого *форматирование* ***writedata***-функцией производится следующим образом:

Формат	***выводится в файл либо на печать числовое N-значение:***
integer	**N,** если ***whattype*(N); ⇒** *integer* либо ***trunc*(N),** если ***whattype*(N); ⇒** {*float*|*fraction*}; числовые **N**-значения других типов инициируют ошибки
float	***evalf*(N, 7);** для **N**-значения допускаются типы {*integer*, *float*, *fraction*}
string	*string*-значение; допускается только *string*-тип

Так как по *float*-формату не допускается использования в качестве выводимых по функции ***writedata*** алгебраических чисел и пакетных констант {***Pi, gamma, Catalan, exp*(1)**}, то для них требуется предварительная конвертация в *float*-тип, например, по ***evalf***-функции. В случае структуры данных матричной или вложенного списка третий аргумент ***writedata***-функции должен кодироваться в виде *списка* {*integer*, *float*, *string*}-значений, по длине соответствующего числу столбцов выводимой структуры данных. В этом случае каждый элемент списка определяет формат выводимых значений для элементов соответствующего ему столбца структуры данных. Например, по третьему фактическому аргументу вида [*string*, *integer*, *float*, *string*] для столбцов выводимой по ***writedata***-функции структуры данных определяются форматы *string*, *integer*, *float* и *string* соответственно. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> V:= array(1..2, [ 10, 17]):  writedata(terminal, convert(V, 'set'));
10
17
> writedata("C:/Temp/grsu\\data", [[57, 52, 32], [3, 10, 37], [42, 47, 67]]);
  writedata(default, [[64, 59, 39], [10, 17, 44], [42, 47, 67]]);
64              59              39
10              17              44
42              47              67
> writedata("C:/Temp/grsu\\data");  ⇒ "64   \t59   \t39"
> writedata(terminal, [64/59, 2006], integer);
1
2006
> writedata(terminal, [52/10, -57, 95/99], float);
5.200000000
-57
0.9595959596
> writedata(terminal, [Tallinn, Moscow, Vilnius, Grodno], string);
Tallinn
Moscow
Vilnius
Grodno
> writedata(terminal, [[AVZ, 64, 2006/42, Gr_B], [AGN, 59, 2006/47, Gr_P], [VSV, 39,
  2006/67, Gr_P]], [string, integer, float, string]);
AVZ   64      47.76190476   Gr_B
AGN   59      42.68085106   Gr_P
VSV   39      29.94029851   Gr_P
> writedata(terminal, [64 + 42*I, 59 + 47*I, 39 - 67*I], float, proc(F, x) fprintf(F, `%G %a`,
  Re(x), Im(x)*I) end proc);
64      42 I
59      47 I
39      -67 I
> writedata(terminal, [[64 + 42*I, 59 - 47*I],[39 - 67*I, 10 + 17*I]], float, proc(F, z) local L;
  L:= cat(convert(Re(z), string), `if`(sign(Im(z)) = 1, `+`, NULL), convert(Im(z), string),
  `*`, `i`):  fprintf(F, `%s`, L) end proc);
64+42 i         59-47 i
39-67 i         10+17 i
```

Рекомендуется обратить внимание на принципы организации пользовательских форматов выводимых структур данных на основе четвертого фактического ***Proc***-аргумента

writedata-функции доступа. В качестве значения *первого* СФ-аргумента ***writedata***-функции можно указывать и ***default***-значение, определяющее, как правило, консоль **ПК.**

Наконец, четвертый необязательный ***Proc***-аргумент обеспечивает возможность вывода данных (*заданного третьим фактическим аргументом формата*) в файл, определяемый первым фактическим СФ-аргументом, в виде, определяемом ***Proc***-процедурой от двух аргументов. Механизм такого подхода в общих чертах можно проиллюстрировать на функциональной конструкции следующего простого вида:

***writedata*(**<СФ>**,** <*Данные*>**,** <*Формат*>**, proc(**F, x**)** y:= map(H, x); W(F, y) **end proc))**

В качестве четвертого фактического аргумента здесь выступает ***Proc*(F,x)**-процедура, получающая в качестве *первого* фактического **F**-аргумента переданное значение фактическому СФ-аргументу ***writedata***-функции, а в качестве значений фактического **x**-аргумента выступают значения структуры данных, определяемой вторым фактическим аргументом ***writedata***-функции. В общем случае тело процедуры включает определение **H**-функции преобразования значений, применяемой ко всем элементам **x**-структуры данных, и **W**-функции, обеспечивающей вывод структуры преобразованных данных в файл либо на печать. Последние два примера предыдущего фрагмента иллюстрируют использование ***Proc***-аргумента ***writedata***-функции для определения собственных форматов представления элементов структур данных, выводимых в файл или на печать. В частности, в последнем примере реализован формат вывода значений структур данных в общепринятой математической нотации. В качестве полезного упражнения читателю рекомендуется запрограммировать несколько процедур *форматирования* значений элементов структур данных, выводимых ***writedata***-функцией. Особенности использования четвертого аргумента ***writedata***-функции, также функции в целом довольно детально рассмотрены в наших книгах [12,32,41].

Обратной к рассмотренной ***writedata***-функции является ***readdata***-функция, имеющая следующий простой формат кодирования:

***readdata*(**<СФ>, {**n** | <*Формат*> | <*Формат*>, **n**})

и обеспечивающая чтение данных из файла, указанного ее *первым* фактическим СФ-аргументом (*спецификатор файла либо номер приписанного ему логического канала в/в*), в точку вызова функции (*переменную*). Если считываются *числовые* данные, то они должны быть {*integer*|*float*}-типа и *упорядочены* в форме матричной структуры, в которой столбцы разделяются символом *пробела* {***hex*(20)**}. *Текстовые* файлы такой структуры легко создаются как средствами системы ***MS DOS***, например, **copy**-командой непосредственно с *консоли*, так и в среде текстовых редакторов. В среде пакета для этих целей служит уже рассмотренная выше ***writedata***-функция и ряд других средств.

Если вызов ***readdata***-функции не определяет (**n**-*аргумент*) число считываемых столбцов данных, то по *умолчанию* считывается *первый* столбец и возвращается в виде списка значений. При считывании нескольких столбцов функция возвращает вложенный список, при этом элементы-*списки* представляют соответствующие строки возвращаемых столбцов данных. По умолчанию ***readdata***-функция считывает числовые данные *float*-типа. Использование необязательного *формат*-аргумента, допускающего {*integer*|*float*|*string*}-значение, позволяет определять формат считываемых функцией данных. При считывании нескольких столбцов данных для каждого из них можно определять свой формат из числа трех перечисленных. В этом случае *формат*-аргумент кодируется в виде списка, аналогично рассмотренному случаю для ***writedata***-функции, но здесь имеется ряд особенностей, требующих повышенного внимания [12,32,41]. Нижеследующий пример иллюстрирует применение ***readdata***-функции для считывания данных из файла.

```
> readdata("C:/Temp/data", 1), readdata("C:/Temp/data", 2), readdata("C:/Temp/data", 3);
      [59., 10., 42.], [[59., 59.], [10., 17.], [42., 47.]], [[59., 59., 39.], [10., 17., 44.], [42., 47., 67.]]
```

Детальнее с ***readdata***-функцией можно ознакомиться по справке пакета, тогда как с рядом важных особенностей данной функции можно ознакомиться в нашей книге [12]. В то же время функции ***writedata*** и ***readdata*** имеют существенные ограничения по типам данных, обрабатываемых ими, а именно допускаются только типы { *integer*, *float*, *string*}.

```
writedata1 := proc(F::{string, symbol}, data::{list, Matrix, matrix, vector, listlist})
local k, Λ, h, f, R;
    Λ := proc()
        local n, m, k, j, μ, μ1, st, L;
            assign(f = `if`(type(F, 'file'), F, pathtf(F))),
                assign(h = fopen(f, `if`(nargs = 2, 'APPEND', 'WRITE')));
            assign(μ = convert([255], 'bytes'), μ1 = convert([254], 'bytes'), st = "")
                ;
            if type(args[1], {'list', 'vector'}) then
                R := convert(args[1], 'list');
                for k to nops(R) do
                    if type(R[k], 'string') then R[k] := cat(R[k], μ)
                    elif type(R[k], 'symbol') then R[k] := cat(R[k], μ1)
                    end if;
                    null(writeline(h, convert(R[k], 'string')))
                end do
            else
                assign(R = convert(args[1], 'Matrix')), assign(n = Pind(R)[1],
                    m = Pind(R)[2], L = convert(R, 'listlist1'));
                for k to n do
                    for j to m do
                        if type(L[k][j], 'string') then L[k][j] := cat(L[k][j], μ)
                        elif type(L[k][j], 'symbol') then
                            L[k][j] := cat(L[k][j], μ1)
                        else L[k][j] := convert(L[k][j], 'string')
                        end if;
                        st := cat(st, L[k][j], " ")
                    end do;
                    null(writeline(h, st[1 .. -2]), assign('st' = ""))
                end do
            end if
        end proc;
    `if`(nargs = 3, Λ(data, 7), Λ(data)), close(f),
        WARNING("data have been saved in datafile <%1>",f)
end proc
> M:= Matrix(3, 4, [[7, "2", C, sin(7)], [7, `14`, "G", `ln(7)`], ["AVZ", Art, "Kr", 5+14*I]]):
> V:= vector([7, "C", `G`, "sin(14)", (a - b)/( c- d), a + b*I]):  M,  eval(V);
```

$$\begin{bmatrix} 7 & \text{"2"} & C & \sin(7) \\ 7 & 14 & \text{"G"} & ln(7) \\ \text{"AVZ"} & Art & \text{"Kr"} & 5+14\,I \end{bmatrix}, \left[7, \text{"C"}, G, \text{"sin(14)"}, \frac{a-b}{c-d}, a+b\,I \right]$$

> writedata1("C:/Temp/TestData1.dat", V); writedata1("C:/Temp/TestData2.dat", M, 7);

Warning, data have been saved in datafile <c:\temp\testdata1.dat>

Warning, data have been saved in datafile <c:\temp\testdata2.dat>

> readdata1("C:/Temp/TestData1.dat", 1), readdata1("C:/Temp/TestData2.dat", 1 .. 4);

$$\begin{bmatrix} 7 \\ \text{"C"} \\ G \\ \text{"sin(14)"} \\ \frac{a-b}{c-d} \\ a+b\,I \end{bmatrix}, \begin{bmatrix} 7 & \text{"2"} & C & \sin(7) \\ 7 & 14 & \text{"G"} & ln(7) \\ \text{"AVZ"} & Art & \text{"Kr"} & 5+14\,I \end{bmatrix}$$

Стандартная процедура ***writedata*(F, *data*)** пишет данные из выражений типов {*matrix*, *list*, *listlist*, *vector*} в текстовый файл **F.** При этом, если данные находятся в матрице либо во вложенном списке, то они выводятся построчно с *разделением* в строках символом табуляции «**\t**». В остальных случаях данные выводятся построчно с каждым значением в отдельной строке. Как правило, данные должны иметь тип {*integer*, *float*}; при этом, для возможности использования *нечисловых* и *комплексных* значений требуется программирование специальных процедур. Стандартная процедура ***readdata*** является обратной к ***writedata*,** обеспечивая чтение числовых данных из файла, подготовленного первой, в текущий сеанс. Между тем, во многих случаях возникает необходимость сохранения в ASCII-файлах произвольных данных. В частности, данные могут содержать значения *fraction*-типа и/или произвольные ***Maple***-выражения. Данная проблема легко решается вышепредставленной процедурой ***writedata1*** и сопутствующей ей ***readdata1*-**процедурой [41,103,109]. Процедура ***writedata1*(F, *data*)** пишет произвольные данные из выражений типов {*matrix*, *Matrix*, *list*, *listlist*, *vector*} в текстовый файл **F.** Если данные представлены вектором либо списком значений, то значения выводятся в отдельной строке, в противном случае значения выводятся построчно с резделением их *пробелом* в строке. В общем случае в качестве данных могут выступать произвольные ***Maple***-выражения, расширяя существенно сферу приложений данных средств при работе с файлами.

Наконец, для обеспечения *форматированного* доступа к файлам данных можно использовать рассмотренные выше пару функций {***fprintf*|*fscanf***}, обеспечивающих соответственно {*запись*|*чтение*} форматированных данных {*в*|*из*} файла{*а*}, а также функции вывода {***print*, *lprint***} форматированной информации на печать, которые можно использовать совместно с {***writeto*, *appendto***}-функциями для *сохранения* вывода в текстовых файлах. Детальнее на данных функциональных средствах останавливаться не будем, ибо они обсуждались в связи с *форматированным* выводом на печать, что легко переносится на случай вывода в файл с дополнительным первым СФ-аргументом [12].

При этом, следует иметь в виду, что по ***fprintf***-функции файл данных открывается как текстовый файл **STREAM**-вида в ***WRITE***-режиме доступа и остается открытым, обеспечивая возможность *обновления* файла в ***APPEND***-режиме. В последующем файл, созданный по ***fprintf***-функции, может читаться как *непосредственно* соответствующей ему функцией ***fscanf*,** так и рядом других рассмотренных в настоящей главе функций доступа к файлам. Механизм форматирования, поддерживаемый ***fprintf***-функцией, достаточно детально рассматривался в [12] и применение *пары* функций {***fprintf*, *fscanf***} для обеспе-

чения форматированного доступа к файлам не составляет труда, принимая также во внимание то обстоятельство, что перед выводов в файл отформатированную запись можно легко визуально верифицировать на экране монитора.

Еще на одном моменте следует акцентировать внимание. Выше уже отмечалось то обстоятельство, что при кодировании спецификатора файла в различных представлениях (*например*, *использование одинаковых букв на разных регистрах или разных разделителей каталогов*) появляется 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 и выгоды она имеет немалые. В качестве примера ниже приводится процедура ***utfile*(F, S),** обеспечивающая 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. Подстановки производятся во *все* строки текстового файла **F.** Успешный вызов процедуры возвращает *NULL*- значение с выводом соответствующего сообщения. Читателю рекомендуется рассмотреть организацию процедуры.

```
utfile := proc(F::file, S::list(equation))
local a, b, c, 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'), c = op(convert("" || F[1], 'bytes')));
    `if`(90 < c, assign(b = cat(convert([c - 32], 'bytes'), "" || F[2 .. -1])),
        assign(b = cat(convert([c + 32], 'bytes'), "" || F[2 .. -1])));
    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 insitu", F)
end proc
> utfile("c:/temp/test.txt", ["xyz"="aaa", mnp=350, "thu"="ccc"]);
Warning, datafile <c:/temp/test.txt> had been updated insitu
> utfile("c:/temp/test.txt", ["xyz"="aaa", "mnp"="bbb", "t"=5]);
Error, (in utfile) substitutions should have parts of identical length, but have received <t = 5>
```

В настоящем разделе и главе в целом рассматривались достаточно развернутые элементы программирования и использования функциональных средств доступа к внешним файлам ***Maple,*** составляющим *ядро* прикладного программирования в среде ***Maple***-языка задач, связанных с обработкой *данных*, находящихся на *внешних* носителях. При этом, следует отметить, что искушенный пользователь вполне может ограничиться рассмотренными средствами доступа, сочетая их с иными функциональными средствами языка для создания собственных *более* развитых средств доступа. Для целого ряда приложений данная задача является достаточно актуальной, ибо существующая система доступа к *внешним* файлам, на наш взгляд, **–** одно из наиболее слабых мест ***Maple,*** требующее существенной доработки и развития в ряде направлений. Не взирая на вполне достаточный уровень представления средств доступа к внешним файлам, за более детальной информацией следует обращаться к книгам и *Библиотеке* [9-14,29-33,39,41-43,46,103,109], посвященным детальной проработке широкого круга аспектов данной проблематики. Там же можно ознакомиться с целым рядом специальных вопросов системы в/в ***Maple.***

Глава 7. *Графические средства Maple-языка пакета*

Учитывая роль *визуальной* информации при решении различного рода задач как прикладного, так и теоретического характера, каждое **ПС** рассматривается с точки зрения наличия у него графических средств отображения информации. В этом отношении пакет ***Maple*** имеет целый ряд положительных аспектов, рассматриваемых ниже. Язык пакета располагает развитым *набором* функций *различных* уровней, обеспечивающих формирование графических структур данных и вывод соответствующих им графических объектов (**ГО**) как в двух, так и в трех измерениях, а также в широком диапазоне систем координат. Каждая графическая *функция* допускает использование большого числа опций, управляющих как режимом вывода **ГО,** так и его оформлением. В настоящей главе рассматриваются базовые графические средства ***Maple-***языка с акцентом на некоторых особенностях их применения и средствах, расширяющих стандартные функции. Ниже для удобства условимся графические средства называть *«функциями»*, хотя по природе своей они являются процедурами, в чем легко убедится посредством нашей процедуры ***ParProc*** [41,103]. Между тем, это не является столь принципиальным и вполне допустимо как аксиоматикой пакета ***Maple,*** так и сутью функционирования данных средств.

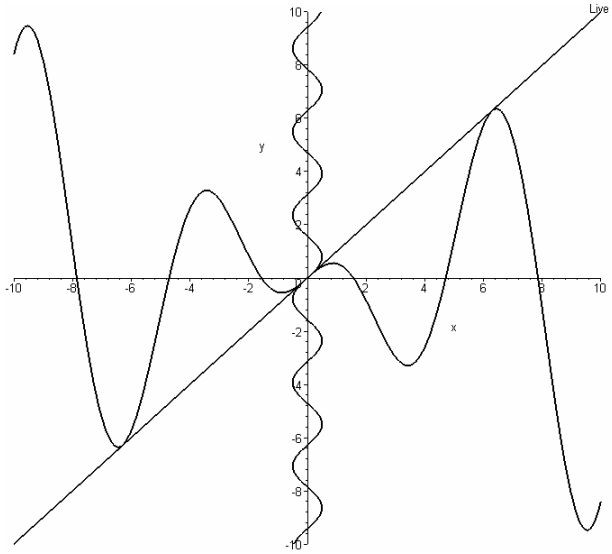
7.1. *Графическая интерпретация алгебраических выражений и уравнений в среде Maple-языка*

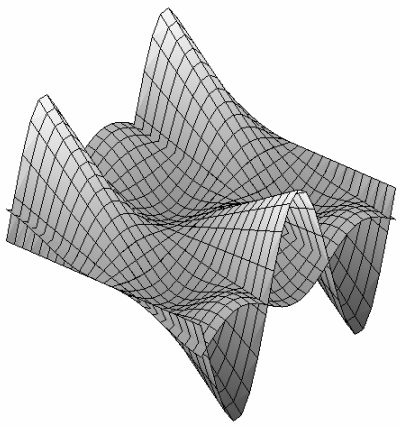
Прежде всего, графические средства предоставляют возможность получения предварительных (*набросков*) графиков, определяемых ***Maple***-выражениями. Такая возможность обеспечивается как на уровне оболочки пакета **GUI,** так и на уровне средств его языка в виде следующих двух процедур:

***smartplot*(V1,V2, …, Vn)** и ***smartplot3d*(V1,V2, …, Vn)**

возвращающих графики зависимостей, определяемых алгебраическими **Vj**-выражениями либо их множеством, в единой системе декартовых координат размерности **2** и **3** соответственно, как это иллюстрирует следующий простой фрагмент:

```
> smartplot(x*cos(x), sin(y)*cos(y), x);   smartplot3d({y*cos(x), sin(y)*cos(z)});
```





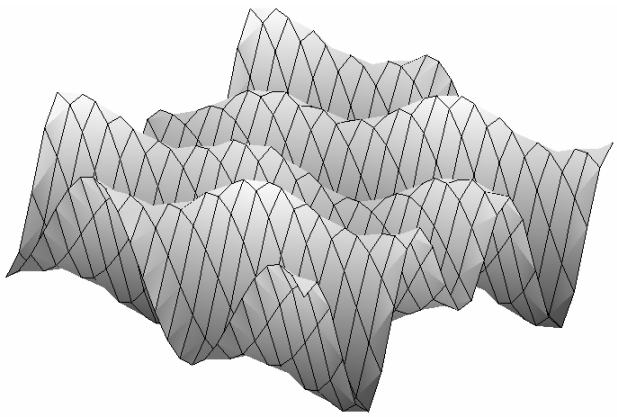
Первый пример фрагмента иллюстрирует использование ***smartplot***-процедуры для вывода 2D-графиков, определяемых тремя простыми функциями, тогда как второй иллюстрирует создание посредством ***smartplot3d***-процедуры 3D-объекта, составленного из **2** поверхностей, определяемых двумя функциями от двух переменных.

Не взирая на достаточно *развитые* графические средства ***Maple***-языка, позволяющие достаточно *гибко* управлять созданием графических объектов различных типов, представленные две процедуры имеют свою *важную* нишу приложений. Прежде всего, в отличие от рассматриваемых ниже базовых графических средств данные процедуры позволяют выводить *графики* функциональных зависимостей, определяемых алгебраическими выражениями/уравнениями, не требуя каких-либо *дополнительных* сведений по данному разделу языка. Это важно по двум основным причинам.

Во первых, для многих приложений языка в интерактивном режиме требуется получать быстрое общее графическое представление, пусть не совсем точное (*набросок*) и хорошо оформленное, для полученного результата в виде алгебраического выражения. Это хорошо отвечает задачам, в частности, исследовательского характера, когда *анализ* результата определяет путь дальнейшей работы и графическая составляющая играет определенную роль. Например, численное решение многих уравнений часто требует удовлетворительного определения интервала, на котором располагаются нулевые корни, многие физические уравнения предполагают графическую интерпретацию и т.д.

Следующий фрагмент иллюстрирует применение ***smartplot3d***-процедуры для получения предварительного графика результата дифференцирования выражения:

```
> smartplot3d(diff(cos(x)*sin(y) + y*abs(sqrt(y))/cos(y) - sin(x)*cos(x + y), x$2, y$4));
```



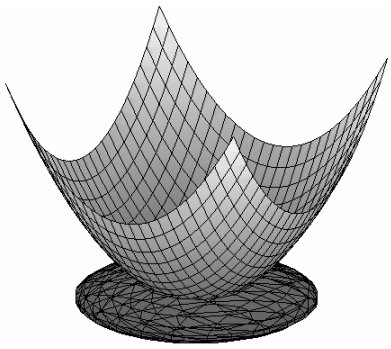
Отличительной чертой использования обоих графических процедур является то обстоятельство, что для получения вполне удовлетворительных графических представлений алгебраических выражений совершенно не требуется знания графических средств языка. Во многих случаях такое решение является вполне достаточным.

Во вторых, применение указанных процедур можно рассматривать как начальный этап подготовки графического объекта, когда его набросок позволяет выбирать *более* продуманные установки опций для базовых графических **plot**-функций, рассматриваемых на протяжении остальных разделов главы.

Между тем, использование данных процедур для выражений/уравнений от количества независимых переменных, больше чем размерность без одного, может вызвать ошибочные ситуации. Поэтому для более эффективного использования *данных* процедур рекомендуется использовать ***Smart***-процедуру, которая обрабатывает основные как особые, так и ошибочные ситуации, возникающие при вызове стандартных процедур ***smartplot*** и ***smartplot3d*.** Вызов процедуры ***Smart*(*args*)** выводит график одного или более алгебра-

ических выражений или уравнений, определенных фактическими аргументами ***args***. Синтаксис фактических аргументов ***args*** полностью соответствует синтаксису аргументов стандартных процедур ***smartplot*** и ***smartplot3d.*** Нижеследующий фрагмент представляет исходный текст процедуры ***Smart*** и пример ее применения.

```
Smart := proc()
local k, L, Z;
    if nargs = 0 then RETURN( ) else L, Z := { }, { } end if;
    for k to nargs do
        if type( args[ k ], 'algebraic' ) then
            L := { op( L ), nops( indets( args[ k ], 'symbol' ) ) }
        elif type( args[ k ], `=` ) and type( rhs( args[ k ] ), 'algebraic' ) and
        type( lhs( args[ k ] ), 'algebraic' ) then
            Z := { op( Z ), nops( indets( args[ k ], 'symbol' ) ) }
        else ERROR( "Among arguments are invalid types <%1>",args )
        end if
    end do;
    if L minus { 0, 1 } = { } and Z minus { 1, 2 } = { } then smartplot( args )
    elif L minus { 1, 2 } = { } and Z minus { 2, 3 } = { } then smartplot3d( args )
    else ERROR( "Among arguments are invalid expressions <%1>",args )
    end if
end proc
> Smart(x^2 + y^2 + z^2 = 20, y^2 + x^2);
```



Детальнее со ***smart***-механизмом создания графических объектов можно ознакомиться в [12] или в *справке* по пакету. В частности, реализованный для ***smart***-графиков механизм *«перетаскивания»* позволяет *удалять* из многофункционального графика любую кривую / поверхность, не затрагивая остальных кривых/поверхностей, либо дополнять график новыми кривыми/поверхностями.

7.2. *Двухмерное представление функциональных зависимостей и данных в среде Maple-языка*

Графическое представление функциональных зависимостей составляет важную компоненту изучения различного рода процессов и явлений. Поэтому развитые современные **ПС,** ориентированные на такие задачи, предоставляют пользователю того либо иного уровня выразительности графические средства. В этом отношении ***Maple***-язык отличается рядом положительных черт.

Графическое представление функциональных зависимостей. Для представления собственно 2D-функциональных зависимостей ***Maple***-язык располагает *базовой* графической ***plot***-функцией, имеющей три основных формата кодирования, а именно:

(**1**) **plot(F(x), x=a..b** {**,y = c..d**} {**,** <*Опции*>**)**

(**2**) **plot([F1(x), ..., Fn(x)], x = a..b** {**, y = c..d**}{**,** <*Опции*>**)**

(**3**) **plot([X(t), Y(t), t = a..b]** {**,** <*Опции*>**)**

и возвращающей 2D-**ГО** в виде графика соответственно *единственной* функции, определенной функциональным **F(x)**-выражением по ведущей **x**-переменной, *нескольких* функций, определенных списком **[F1(x), ..., Fn(x)],** и *параметрически* заданной функции, определенных на диапазоне **a..b** значений **x**-переменной (**t**-*параметра*) в единой системе декартовых (**X, Y**)-координат. Так как ***plot***-функция возвращает 2D-**ГО** только с вычисляемыми (**X,Y**)-точками, то для обеспечения этого требования предполагается, что функциональные зависимости являются действительными. В дальнейшем будем предполагать, что все графические функции, возвращающие графики функциональных зависимостей, требуют действительных значений над областью их определения, в противном случае не выводя отличные от действительных значения **F**-функций, т.е. выводятся части графика, содержащие только действительные координаты {**x,F(x)**}. При определении диапазонов изменения значений как ординат, так и абсцисс графика в виде **<*Id*>=a..b, <*Id*>** появляется в качестве названия соответствующей оси координат.

Первые два аргумента каждого формата ***plot***-функции, в общем случае, обязательны, оп-ределяя соответственно функциональную зависимость и область значений ее ведущей переменной, тогда как третий аргумент первых двух форматов необязателен и определяет *диапазон* выводимых значений собственно функциональной зависимости. Как правило, данный параметр не кодируется во *избежание* возможной потери частей выводимого графика. Наконец, в качестве необязательного четвертого аргумента всех форматов ***plot***-функции допускается использование специальных опций, управляющих режимом создания, вывода и оформления графиков. Речь о них будет идти в главе ниже.

В качестве первого фактического аргумента ***plot***-функция допускает использование: (**1**) действительных функции или выражения от *одной* ведущей переменной, (**2**) ***Maple***-процедуры, (**3**) параметрически определенной функции и (**4**) списка координат точек. Для случая ***Maple***-процедуры используется в общем случае конструкция следующего вида: ***plot*(*Proc*, a..b**) (***Proc*** – *имя процедуры*), а для списка координат точек – конструкция вида **plot([[x1, y1],...,[xn, yn]]),** в которой каждая [**xj, yj**]-пара определяет отдельную точку, выводимую отдельно либо соединенной с другими точками прямой линией.

В случае *определения* многофункционального графика, содержащего графики нескольких функциональных зависимостей, определяемых первым аргументом-списком ***plot***-функции, они выводятся в единой координатной сетке с различным цветовым оформлением. Наряду со списком в качестве определителя многофункционального графика может выступать и множество, однако по ряду причин списочная структура предпочтительнее. Для удобства *визуальной* идентификации *отдельных* кривых **ГО** производится их расцветка согласно значения глобальной переменной ***_COLORRGB*,** являющейся последовательностью, состоящей из **6** элементов-списков, а именно:

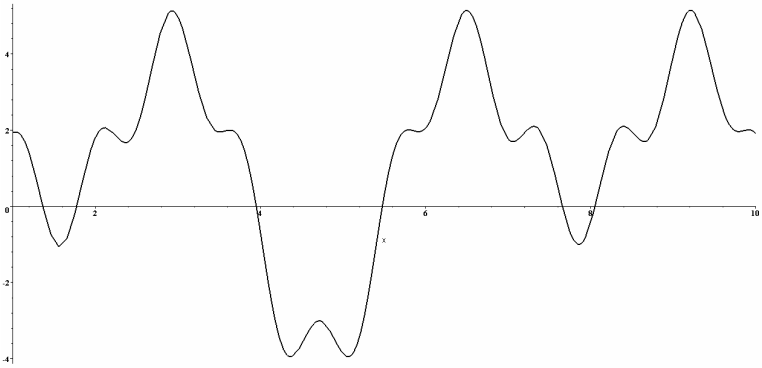
> _COLORRGB; ⇒ [1.0, 0., 0.], [0., 1.0, 0.], [1.0, 1.0, 0.], [0., 0., 1.0], [1.0, 0., 1.0], [0., 1.0, 1.0]

Следовательно, если по ***plot***-функции одновременно выводится **n** кривых в одном **ГО,** то цвета (**n-6**) из них будут повторяться.

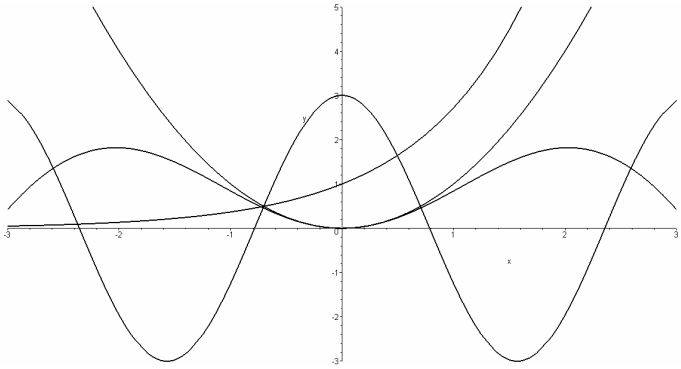
Функциональная зависимость, заданная параметрически (*формат* **3**), в качестве первого аргумента ***plot***-функции предполагает также списочную структуру, однако в отличие

от списка функций *второго* формата, данный список имеет отличный набор элементов. Его элементы определяют {**X(t),Y(t)**}-функции координат точки графика соответственно по {**X,Y**}-осям и **t**-параметр, чей диапазон изменения значений кодируется третьим элементом упомянутого списка-аргумента 3-го формата функции. При необходимости одновременного вывода в одном **ГО** нескольких параметрически определенных функций должен использоваться единый для них параметр, тогда как диапазоны его значений для функций могут различаться. Нижеследующий фрагмент представляет примеры создания ряда графиков на основе ***plot***-функции рассмотренных выше форматов.

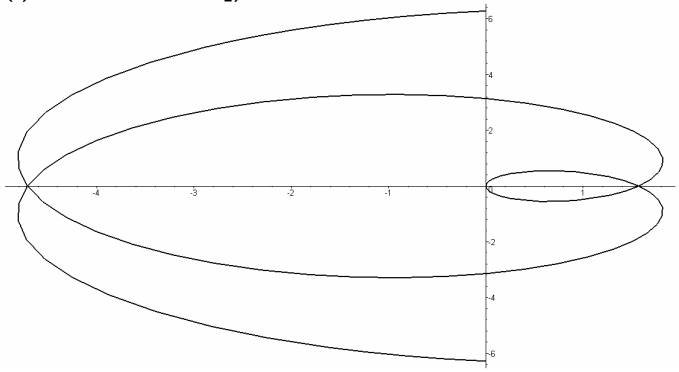
> plot(x*sin(1/x) + 3*cos(2*x) + 2*sin(3*x)*cos(4*x) + 3*sin(x), x = 1..10);



> plot([3*cos(2*x), x^2, exp(x), x*sin(x)], x = -3..3, y = -3..5);



> plot([t*sin(t), t*cos(t), t = -2*Pi..2*Pi]);



В виду своей прозрачности особых пояснений данный фрагмент не требует. Читатель и сам может привести немало других более интересных примеров применения графической ***plot***-функции ***Maple***-языка.

Резюмируя сказанное, можно констатировать, что мы уже располагаем *вполне* достаточной информацией для создания 2D-графиков функциональных зависимостей, определяемых ведущими переменными ***Maple***-выражений, функций и процедур, включая и пользовательские. В наиболее часто используемом *исполнении* для вывода 2D-**ГО** функ-

циональных зависимостей используются конструкции вида **plot(F(x), x= a..b).** При этом, следует иметь в виду, что по конструкциям **plot(F(x),*x*)** и **plot(F)** выводится график функции **F** в диапазоне **x= -10..10.** В объеме рассмотренных графических средств можно как создавать 2D-**ГО,** так и производить ряд процедур по режиму их визуального представления в рамках декартовой системы координат, используя значения по умолчанию для остальных *опций*, управляющих 2D-графическим *режимом*. Рассмотрим вкратце данные средства, поддерживаемые набором опций, кодируемых в качестве четвертого необязательного аргумента ***plot***-функции и называемых в дальнейшем графическими *опциями*.

Управления созданием и выводом 2D-ГО. В общем случае *опция* имеет вид *уравнения*, левая часть которого представляет *ключевое* слово, определяющее *опцию*, а правая – ее значение. Опции располагаются в произвольном порядке в качестве последнего фактического аргумента ***plot***-функции, т.е. являются *ключевыми*. Функция ***plot*** имеет опции согласно следующей табл. **15,** в которой каждая опция определяет соответственно:

Опция	***Опция plot-функции определяет:***
adaptive	разрешение на использование режима адаптивного вывода (***true***)
axes	используемый тип осей координат (***normal***)
axesfont	шрифт для меток маркируемых точек осей координат (***Default***)
color	расцветку кривых графиков функциональных зависимостей
coords	используемую систему координат (***cartesian***)
disconts	режим вывода зависимостей, имеющих точки разрыва (***Default***)
filled	режим закраски области между кривой и **X**-осью
font	шрифт для текстовой информации **ГО** в виде **[***шрифт***,** *стиль***,** *размер***]**
labels	метки для осей системы координат (***Default***)
labelfont	шрифт для меток осей координат в виде **[***шрифт***,** *стиль***,** *размер***]**
linestyle	стиль определения линий функциональных зависимостей (***solid***)
numpoints	минимальное число генерируемых точек для кривой (**49**)
resolution	разрешающую способность монитора по горизонтали в пикселях (**200**)
sample	список значений параметров для инициализации ***plot-***функции
scaling	режим шкалирования **ГО** (***unconstrained***)
style	стиль определения линий функциональных зависимостей (***Line***)
symbol	тип символов для точек графика в ***point-***режиме (***Point***)
thickness	толщину линий функциональных зависимостей (**0**)
tickmarks	минимальное число маркируемых точек по **X,Y-**осям координат
title	заголовок для графического объекта (***None***)
titlefont	шрифт для заголовка графического объекта (***Default***)
view	минимальные и максимальные координаты выводимых точек **ГО**
xtickmarks	минимальное число маркируемых точек по **X-**оси координат (***Default***)
ytickmarks	минимальное число маркируемых точек по **Y-**оси координат (***Default***)

Смысл большинства *опций* достаточно прозрачен, однако более детально можно с ними ознакомиться в справке по пакету. Тогда как целый ряд особенностей их применения с соответствующими рекомендациями достаточно детально рассмотрены в [12].

Наряду с базовыми, ***Maple-***язык располагает целым рядом *модульных* средств, обеспечиваемых *пакетными* модулями **plots** (**56**) и **plottools** (**36**), где в скобках указано количество поддерживаемых ими графических функций для ***Maple 10*** (*как правило, с ростом номера*

релиза пакета растет и набор его функциональных средств). Ряд средств данных модулей будет рассмотрен ниже, здесь же пока ограничимся ***polarplot***-функцией из **plots**-модуля (*см. прилож.* 1 [12]). Как известно, вызов функции, определенной в пакетном модуле с ***Id***-идентификатором, производится тремя основными способами, а именно:

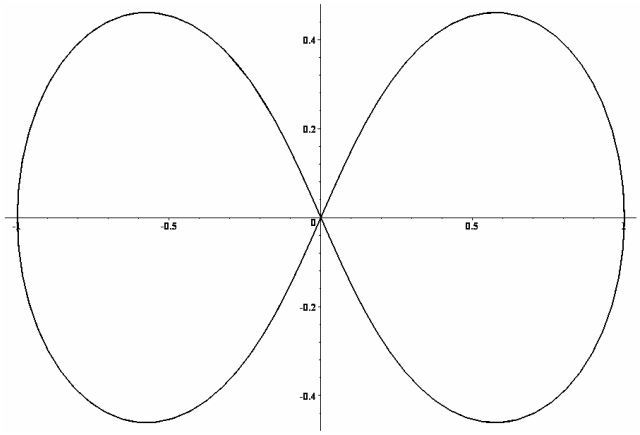
(**1**) **with(*Id*)** (**2**) **Id[**<*Функция*>**](**<*Аргументы*>**)** (**3**) **with(Id,** <*Функция*>**)**

Первый способ обеспечивает непосредственный доступ сразу ко всем средствам ***Id***-модуля (*на весь текущий сеанс*), вызовы которых становятся аналогичными вызовам встроенных функций ***Maple***-языка. Поэтому, по **with(plots)**-предложению получаем доступ ко всем функциям **plots**-модуля, включая ***polarplot***-функцию следующего формата:

polarplot([r(t), alpha(t) {**, t = a..b**}**]** {**,** <*Опции*>})

где: **r(t)** и ***alpha*(t)** параметрически определяют *радиус* и *полярный угол* соответственно, а третий элемент списка - *диапазон* изменения параметра (*полярного угла*). При отсутствии третьего элемента списка в качестве диапазона изменения угла по умолчанию полагается значение **-Pi..Pi**. Последующие формальные аргументы функции определяют **plot**-опции, рассмотренные выше. В качестве первого аргумента ***polarplot***-функции может выступать список/множество функций, выводимых в единой системе координат. Следующий пример иллюстрирует использование ***polarplot***-функции для создания графика функции в полярной системе координат.

```
> plots[polarplot]([cos(t), sin(t), t = 0..4*Pi], color = black, thickness = 2, axesfont = [TIMES,
  BOLD, 11]);
```



Графические 2D-ГО-структуры данных. Для представления графических объектов пакет использует *специальную* структуру данных, называемую в дальнейшем **2*D_ГО*-**структурой**.** Данная структура генерируется графической функцией ***plot*** и ей подобными**.** На основе данной структуры строятся графические ***примитивы*,** которые затем передаются драйверам заданных устройств с целью вывода результирующего **ГО.** Принципиальная схема визуализации графического **2D-ГО** представлена на следующем рис. **1.**

Рис. **1.** Принципиальная схема визуализации графического **2D-ГО**

Графические структуры данных можно присваивать в качестве значений переменным, конвертировать их в другие структуры**,** сохранять их или выводить на печать и**,** нако-

нец, выводить соответствующие им графические *объекты* на заданные устройства. Линейную внутреннюю организацию графической структуры легко получать для каждого конкретного **ГО,** для чего достаточно присвоить вызов графической функции некоторой переменной, как это иллюстрирует следующий простой пример**:**

> PS:= plot(x, x = 0..1, numpoints = 3, adaptive = false, axesfont = [TIMES, BOLD, 10]);
PS := PLOT(CURVES([[0., 0.], [0.523131699999999977, 0.523131699999999977], [1., 1.]], COLOUR(*RGB*, 1.0, 0., 0.)), AXESLABELS("x", ""), AXESTICKS(*DEFAULT*, *DEFAULT*, FONT(*TIMES*, *BOLD*, 10)), VIEW(0. .. 1., *DEFAULT*))

В данном примере переменной ***PS*** присваивается вызов ***plot***-функции, определяющей график функции **y = x** на **[0, 1]**-интервале с тремя опорными точками (***numpoints***), с отменой адаптивного механизма (***adaptive***) и заданным шрифтом (**axesfont**) для разметки осей координат. Результатом *вычисления* данной переменной является возврат соответствующей **plot**-функции ***2D_ГО***-структуры, из рассмотрения которой следует, что она содержит тип **ГО** (**CURVES** - кривая), список координат точек кривой, а также установки для пяти графических опций ***COLOUR, AXESLABELS, AXESTICKS, FONT*** и ***VIEW.***

В общем случае ***2D_ГО-***структура имеет следующую простую организацию**:**

<*тип ГО*>(<*Объектная информация*> {**, <*Локальная информация*>}**)**

где *тип* **ГО** является одним из**: POINTS** (*точки*), **CURVES** (*кривые*), **POLYGONS** (*многоугольники*) и **TEXT** (*текст*). *Объектная* информация содержит координаты *опорных* точек для соответствующего типа **ГО,** тогда как *локальная* информация *необязательна* и содержит установки графических *опций* для данного конкретного **ГО.** Формат объектной информации определяется типом **ГО** и в зависимости от него принимает вид**:**

POINTS: POINTS([x1,y1],[x2,y2],...[xn,yn]) **-** множество *опорных точек* в плоскости**;**

CURVES: CURVES([[x11,y11],...[x1n,y1n]], [[x21,y21],...[x2n,y2n]],...[[xm1,ym1],...[xmn,ymn]]) **-** набор из **m** *кривых*, определяемых *опорными* точками на плоскости**;**

POLYGONS: POLYGONS([[x11, y11],...[x1n, y1n]], [[x21, y21],...[x2n, y2n]],...[[xm1, ym1],... [xmn,ymn]]) **-** набор из ***m*** *многоугольников*, определяемых *опорными* точками**;**

TEXT: TEXT([x,y]**, "<***строка***>"**) **-** текстовая *строка* с *опорной* (**x,y**)-точкой**;** в общем случае в качестве *строки* могут выступать выражения {*string, symbol, name*}-типа.

Для двухмерного случая ***2D_ГО-***структура наряду с обязательной *объектной* информаци-й может содержать следующие опции: **AXESSTYLE, AXESLABELS, FONT, SYMBOL, STYLE, *VIEW*, COLOUR, AXESTICKS, LINESTYLE, THICKNESS, SCALING, TITLE,** при этом, с появлением новых релизов список опций может расширяться. Пользователь через необязательную *локальную* информацию может заменять глобальные установки указанных опций на период воспроизведения **ГО.** В частности, **COLOUR-***опция* в структуре может определять отдельный цвет для каждого из составляющих общий **ГО** подобъектов**;** например, для *каждой кривой* многофункционального графика может быть определен свой цвет. При этом, набор допустимых для ***2D_ГО-***структуры опций определяется типом представляемого ею графического объекта.

По вызову функции ***convert*(O, {*PLOToptions*|*PLOT3Doptions*})** предоставляется возможность конвертации графических опций, определяемых первым фактическим **O-**аргументом в терминах синтаксиса ***Maple-***языка**,** во *внутренний* формат графических структур данных соответственно {***PLOT*|*PLOT3D***}-типа. При этом фактический **O-**аргумент функции кодируется либо в виде единственного уравнения <*Опция*> **=** <*Значение*>**,** либо *списка* таких уравнений. При этом, следует иметь в виду, что ряд *графических* опций, определенных в терминах ***Maple-***языка, не конвертируется во внутренний формат, игнорируясь ***convert-***функцией. Так как результат конвертации графических опций при-

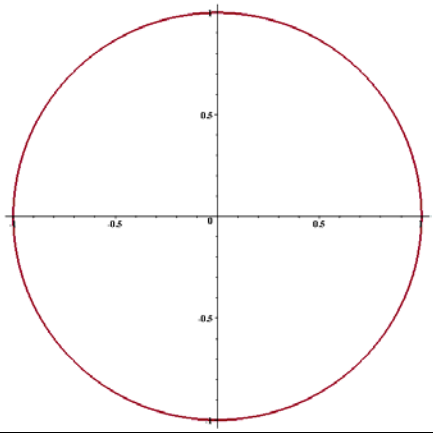
годен для непосредственного погружения во внутреннюю {*2D*|*3D*}*_ГО*-структуру данных, то рассмотренное средство представляется нам достаточно важным при программировании целого ряда сложных графических задач в среде ***Maple***-языка [12]. Следующий простой фрагмент иллюстрирует средство такой конвертации:

```
> convert([title=`Activity Graphic of the TRG`, titlefont=[TIMES,BOLD,14], axes=boxed,
  thickness=2, color=green, labels=["Year", "Books"]], 'PLOToptions');
```

[AXESLABELS("Year", "Books"), TITLE(*Activity Graphic of the TRG*, FONT(*TIMES*, *BOLD*, 14)),
AXESSTYLE(*BOX*), COLOUR(*RGB*,0., 1.00000000,0.), THICKNESS(2)]

Так как ***plot***-структура имеет текстовый формат и прозрачную организацию, то в принципе, ее можно готовить в среде текстового процессора или непосредственно в среде текущего ***Maple***-документа. Продемонстрируем принцип данной процедуры на весьма простом примере. Непосредственно внутри структуры создается последовательность **2**-элементных числовых списков, представляющих координаты точек будущей *кривой* (*окружности*). Затем строится ***plot***-структура ***Svetla***, описанной выше организации; после чего структура дополняется установками для необходимых ***plot***-опций. Наконец, сформированная ***Svetla***-структура вычисляется, выводя искомый **2D-ГО**, а именно:

```
> Svetla:= PLOT(CURVES([evalf([cos(2*'k'*Pi/200), sin(2*'k'*Pi/200)])$'k' = 1..650],
  THICKNESS(3), COLOUR(HSV, 0.96, 1, 0.57)), AXESTICKS(DEFAULT, DEFAULT,
  FONT(TIMES, BOLD,10))):  Svetla;
```



Целый ряд довольно интересных примеров прямого использования графических ***plot***-структур для создания **2D-ГО** можно найти в [12]. Отметим, работа с **2D-ГО** на уровне определяющих их ***2D_ГО***-структур при определенном навыке не должна вызывать особых сложностей, позволяя создавать достаточно сложные графические объекты.

Таким образом, в общем случае **2D**-графические функции имеют дело со структурами данных четырех базовых типов, а именно: **POLYGONS** (*многоугольники*), **TEXT** (*текст*), **CURVES** (*кривые*), **POINTS** (*точки*), каждая из которых структурно достаточно проста и все они имеют единую принципиальную организацию следующего простого вида:

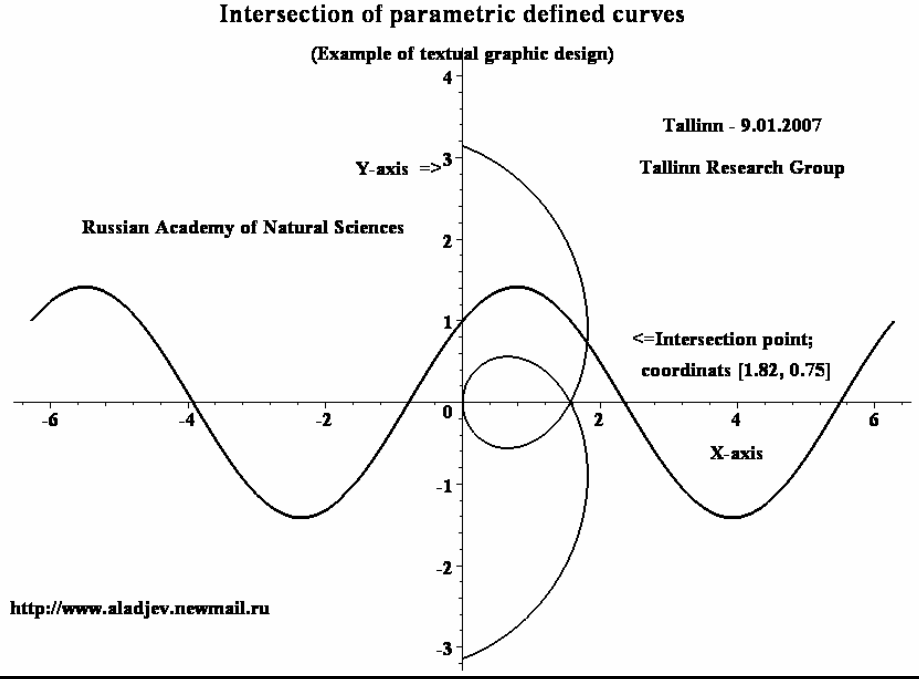
<*Тип* **ГО>(**<*Описание объекта*> {**,** <*Опции*>}**)**

где *описание объекта* является определяющим для его *типа*, а *опции* необязательны и характеризуют присущие ему *локальные* свойства. По вызову графической функции {***plot***|***polarplot***}**(**<***2D_ГО-структура*>**)** выводится собственно сам **2D-ГО.** Детальнее с графическими структурами, поддерживаемыми пакетом, можно ознакомиться в [12] и в определенной мере в справочной системе пакета.

Текстовое оформление 2D-графических объектов. Из рассмотренной выше *графической* функции ***plot*** следует, что посредством поддерживаемых ею опций можно в определен-

ной мере производить *текстовое* оформление **2D-ГО** (*заголовки и осевые метки*), однако их выразительные возможности весьма ограничены, к тому же относительно последних имеется ряд ограничений и особенностей, в значительной степени отмеченных нами в прилож. **1** [12] и делающих эти средства в ряде случаев *не совсем* приемлемыми для указанных целей. В качестве более гибкого средства текстового оформления **2D-ГО** выступает модульная функция ***textplot*(*Txt* {, <*Опции*>}),** где в качестве второго необязательного аргумента выступают уже рассмотренные ***plot*-**опции и дополнительно одна специфическая для текста ***align*-**опция, определяющая режим *расположения* текстовой строки относительно *опорной* точки. В качестве значений для ***align*-**опции допускаются следующие**: *BELOW* (*ниже*), *ABOVE* (*выше*), *RIGHT* (*вправо*) и *LEFT* (*влево*)**;** по *умолчанию* полагается центровка текста по обоим осям координат относительно *опорной* точки**.** Допускается одновременное применение пар значений, например ***align*={*ABOVE*, *LEFT*}.** По ***color*-**опции производится расцветка текста, что позволяет улучшить оформление **ГО.**

```
> with(plots): F:= [TIMES, BOLD]:  T1:= textplot([0, 4.2, "(Example of textual graphic
  design)"], align=ABOVE, font=[TIMES, BOLD, 14]): T2:= textplot([-0.3, 2.8, `Y-axis  =>`],
  align={ABOVE, LEFT}, font=[op(F), 14]): T3:=textplot([4, -0.6, `X-axis`], font= [op(F), 14]):
  T4:= textplot({[3.8, 0.8, `<=Intersection point;`], [4, 0.42, `coordinats [1.82, 0.75]`]},
  font=[op(F), 14]): T5:= textplot({[4.074, 3.413, "Tallinn - 9.01.2007"], [4.074, 2.9 ,`Tallinn
  Research Group`]}, font= [op(F), 14]): T6:= textplot([-4.7, -2.5, `www.aladjev.newmail.ru`],
  font=[op(F), 14]):  G:= plot({[t, sin(t) + cos(t), t = -2*Pi..2*Pi], [t*sin(t), t*cos(t), t = -Pi..Pi]},
  thickness= [2, 3], color= [red, green], title= `Intersection of parametric defined curves`,
  labels= [``, "Russian Academy of Natural Sciences"], labelfont= [op(F), 14],
  titlefont= [TIMES, BOLD, 18]): display([G, T1, T2, T3, T4, T5, T6], axesfont= [op(F), 14]);
```



В случае вызова функции ***textplot*([a,b], "<*Текст*>")** генерируемая ею *text*-структура имеет весьма простую организацию**:**

PLOT(TEXT([a, b], <*Текст*> {, G}{, V}) {, <*Опции*>})

вычисление которой выводит заданный текст с **[a, b]-**координатами *опорной* точки в декартовой системе координат. Если не указаны явно ключевые параметры **G** и **V,** определяющие соответственно расположение выводимого *текста* по *горизонтали* и *вертикали* относительно *опорной* точки, то для них принимаются значения по *умолчанию* (*цент-*

ровка по обоим измерениям). Для *обоих* параметров структуры *допустимыми* являются значения **G∈{*ALIGNLEFT*, *ALIGNRIGHT*}** и **V∈{ALIGNABOVE, *ALIGNBELOW*}.** Дополняя *text*-структуру определениями ***plot*-**опций, можно соответствующим образом дооформлять текстовую информацию, выводимую в качестве *составной* **2D-ГО.**

В качестве *первого* обязательного ***Txt*-**аргумента ***textplot*-**функции выступает *текстовый элемент* (**ТЭ**) либо их список/множество. Отдельный **ТЭ** представляет собой 3-элементную списочную структуру, в качестве первых двух элементов которой выступают координаты *опорной* точки, а в качестве третьего элемента **-** собственно сама текстовая строка, например, **[5.7, -4.2, "*Russian Ecology Academy*"].** Прежде, чем переходить к вопросу совмещения сугубо графических объектов, подготовленных ***plot*-**функцией, с текстовыми объектами, подготовленными ***textplot*-**функцией, рассмотрим модульную ***display*-**функцию, обеспечивающую такие интеграционные средства. Определения этой и предыдущей функции ***textplot*** находятся в пакетном модуле **plots.**

Модульная ***display*-**функция, имеющая следующий формат кодирования**:**

***display*(<*2D_ГО-структура*> {, *insequence* = {*true*|*false*}} {, <*Опции*>})**

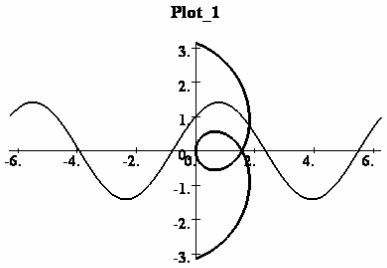
обеспечивает вывод ***2D_ГО*-**структур**,** определяемых ее первым фактическим аргументом**,** в виде единого **ГО,** получающегося в результате объединения исходных графических компонент. В качестве фактических значений первого аргумента могут выступать список, множество или массив ***2D_ГО*-**структур, которые требуется вывести в виде единого объединенного **ГО.** Если *первый* аргумент представляет собой *список* или *множество* и определен необязательный ключевой аргумент ***insequence=false*** (*либо опущен, т.к. принимается по умолчанию*), то ***display*-**функция выводит *единый* **ГО,** созданный объединением представленных им *исходных* ***2D_ГО*-**структур. Если же при сделанном предположении определен аргумент ***insequence = true*,** то производится анимация **ГО,** определенного графическими структурами из первого аргумента функции.

Анимация в этом случае состоит в циклической смене на экране в едином фрейме **ГО** в том порядке, как они определены в первом аргументе ***display*-**функции. При этом, если списочная структура первого аргумента точно определяет порядок смены *анимируемых* **ГО,** то структура множественного типа в общем случае определяет иной порядок, обсуждение которого проводилось выше. В случае**,** если первый аргумент функции ***display*** имеет *array*-тип**,** то результирующий **ГО** наследует его структуру с исходными ***2D_ГО*-**структурами в качестве элементов массива. Наконец, необязательный третий аргумент ***display*-**функции определяет глобальные установки для ***plot*-**опций, управляющих выводом и оформлением результирующего единого **ГО** и перекрывающих локальные установки одноименных им опций из отдельных исходных ***2D-ГО*-**структур, определяемых первым аргументом функции. В этом отношении ***display*-**функция выступает в роли своего рода *компановщика* графических объектов на основе как сугубо графических**,** так и текстовых структур *данных*, включая и поддерживаемые пакетным **plottools-**модулем пакета графические примитивы, рассматриваемые ниже. Особенности взаимодействия локально и глобально определенных графических опций в условиях применения ***display*-**функции достаточно детально рассмотрены в [12].

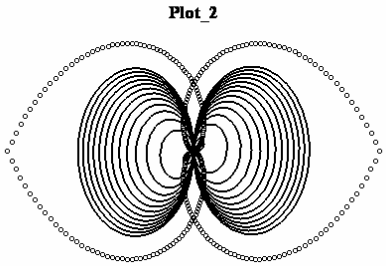
В качестве простого примера создания *единого* **ГО** из семи **2D-ГО**-структур **G, T1..T6** посредством ***display*-**функции можно рассмотреть предыдущий фрагмент. Тогда как два примера следующего фрагмента иллюстрируют *создание* посредством ***display*-**функции единого **ГО** *табличной* структуры, элементами которой являются *заранее* определенные **2D_ГО**-структуры **S1..S3** и ***TxT*,** созданные функциями ***plot*, *polarplot*** и ***textplot*** соответственно. Во втором примере фрагмента иллюстрируется определение режима анимации трех структур типов **CURVES** и одной **TEXT-**типа (*будучи динамическим, данный ре-*

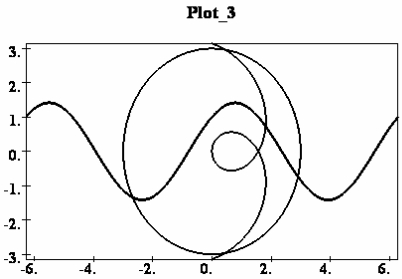
жим реализуем лишь на компьютере). С учетом вышесказанного каких-либо дополнительных пояснений данный фрагмент не требует.

```
> S1:= plot({[t, sin(t) + cos(t), t=-2*Pi..2*Pi], [t*sin(t), t*cos(t), t= -Pi..Pi]}, thickness= [2, 3],
  color= [ red, pink], title= `Plot_1`, axesfont= [op(F), 12]):  S2:=polarplot({[t*cos(t^2), t^2,
  t=-2*Pi..2*Pi], [t^2, t, t=-Pi..Pi], [-t^2, t, t=-Pi..Pi]}, thickness= [2, 4, 4], color= [ gold, red,
  tan], axes=none, style= [line, point, point], title= "Plot_2", symbol=circle, numpoints=
  180): S3:= plot({[t, sin(t)+cos(t), t=-2*Pi..2*Pi], [t*sin(t), t*cos(t), t=-Pi..Pi], [3*sin(t), 3*cos(t),
  t=-2*Pi..2*Pi]}, thickness= [2, 3], color= [ blue, brown, yellow], title= "Plot_3", axesfont=
  [op(F), 12], axes=boxed):  TxT:= textplot([0, 2, `Example of tabular graphic structure`],
  axes=none):   display(array(1..2, 1..2, [[S1, TxT], [S2, S3]]), font= [op(F), 14]);
```



Example of tabular graphic structure






```
> display([TxT, S1, S2, S3], insequence = true);      # Режим анимации из 4-х фреймов
```

В нашей книге [12] представлен целый ряд практического характера рекомендаций по созданию *составных* и *анимируемых* графических объектов на основе ***display***-функции.

Следует отметить, что и *статические*, и *анимируемые* **ГО** можно сохранять в файлах, для чего следует определить соответствующую установку ***plotsetup***-функции вида:

***plotsetup*(*DT*, *plotoutput* = "*Путь к принимающему файлу*")**

гду ***DT*** – тип устройства (*например*, ***bmp, char, cps, gif, hpgl, jpeg, wmf*** *и др.*). Между тем, в опреративном режиме посредством щелчка *правой* клавишей мыши по выбранному **ГО** открывается меню, в котором выбором опции «***Export As***» получаем возможность сохранять **ГО** в файле в одном из следующих форматов:

Encapsulated PostScript (**EPS**), *Graphics Interchange Format* (**GIF**), *Windows Bitmap* (**BMP**), *JPEG File Interchange Format* (**JPG**), *Windows Metafile* (**WMF**)

Это позволяет проводить дальнейшее редактирование **ГО,** например, *графическими* редакторами и целым рядом других подобных средств. Подобный прием может быть использован и для сохранения **ГО** в других форматах для последующей обработки.

Перед окончательным формированием **ГО,** содержащим *графики* функциональных зависимостей и *текстовое* оформление, рекомендуется предварительно произвести полную отладку собственно графической части объекта (*выбор диапазонов по осям координат, обработка особых точек, идентификация отдельных кривых, изменение размера рамки* **ГО** *и т. д.*). После ее завершения описанным выше *способом* устанавливаются *опорные* точки **ГО,**

относительно которых будет помещаться в объект *текстовая* информация**,** формируемая ***textplot-***функцией**.** На заключительном этапе по ***display-***функции выводится единый **ГО,** требующий, как правило, окончательной доводки на соответствие всех составляющих его компонент. При этом, *локальные* ***plot-***опции, используемые в ***2D_ГО-***структурах, составляющих *объединенный* **ГО,** не следует перекрывать одноименными опциями в ***display-***функции, определяя в ней лишь опции глобального характера**.** При конкретной работе с графическими средствами пакета необходимый навык у пользователя появляется достаточно быстро, чему способствует и высокая наглядность задачи.

Специальные типы графиков функциональных зависимостей. Наряду с рассмотренными *базовыми* средствами **2D-**графики язык располагает целым рядом модульных функций, поддерживаемых упоминаемым выше *модулем* **plots,** которые существенно расширяют графические возможности ***Maple-я***зыка**.** Все эти средства становятся доступными после выполнения ***with(plots)-***вызова или иным рассмотренным выше способом**.** Выше уже рассмотрены функции ***display, textplot, polarplot*** этого модуля. Кратко охарактеризуем ряд других, используемых для более *специальных* типов **2D-ГО.** Основные средства в данном направлении представляет следующая табл. **16.**

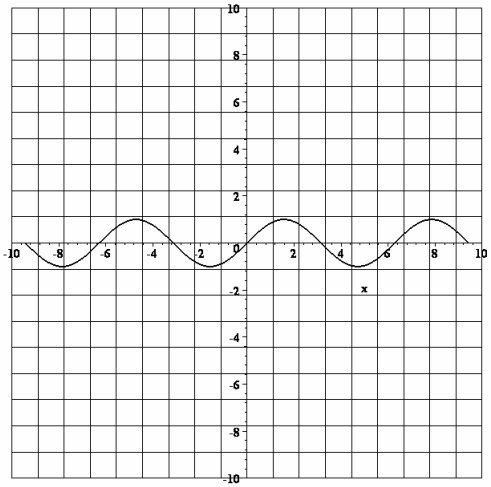
Таблица 16

Функция	***Функция обеспечивает вывод:***
changecoords	конвертирует ***2D_ГО-***структуру в *новую* систему координат
complexplot	***2D-***графика *комплексной* функции
conformal	*конформального* ***2D-***графика комплексной функции
contourplot	*контурного* графика функции
coordplot	координатных сеток поддерживаемых систем координат
densityplot	***2D-***графика *плотности*
fieldplot	графика ***2D-***векторного поля
gradplot	графика *градиента* ***2D-***векторного поля
implicitplot	графика ***2D-***функции, заданной *неявно*
inequal	*области*, ограниченной линейными *неравенствами*
listcontplot	*контурного* графика числового ***2D-***массива
listplot	***2D-***графика *списка* числовых значений
listdensityplot	***2D-***графика *плотности* вложенного числового списка
{***semi***}***logplot***	*полулогарифмического* графика функций
loglogplot	*дважды логарифмического* графика функций
pareto	частотной *диаграммы* Паретто
pointplot	графика {*множества*|*списка*} ***2D-***числовых точек
polygonplot	графика плоских многоугольников
replot	старая версия ***display-***функции**;** рекомендуется *вторая*
rootlocus	графика *комплексных* корней рациональной функции
sparsematrixplot	графика *ненулевых* значений матрицы или **2D-**массива

Некоторые из представленных табл. **16** графических функций допускают использование дополнительно к рассмотренным ***plot-***опциям и специфические опции**,** управляющие режимом создания и вывода **2D-ГО.** Следует иметь в виду**,** что *глобальные* установки ряда ***plot-***опций, сделанные посредством ***setoptions-***функции, не распространяются на некоторые графические функции, поддерживаемые ***plots-***модулем, поэтому их следует определять *локально* непосредственно в момент вызова функции. Детальнее с фор-

матами кодирования и назначением данных модульных графических функций можно ознакомиться в книгах [11-14,88,110,112-120] либо по справочной системе пакета. Мы ограничимся с целью иллюстрации лишь примером применения ***coordplot-***функции**:**

```
> display([plot(sin(x), x=-3*Pi..3*Pi, color=red, labelfont=[TIMES, BOLD, 12], axesfont=
  [TIMES, BOLD,12], thickness=2), coordplot(cartesian, grid=[19,19], color=[black, black])],
  axes=normal);
```

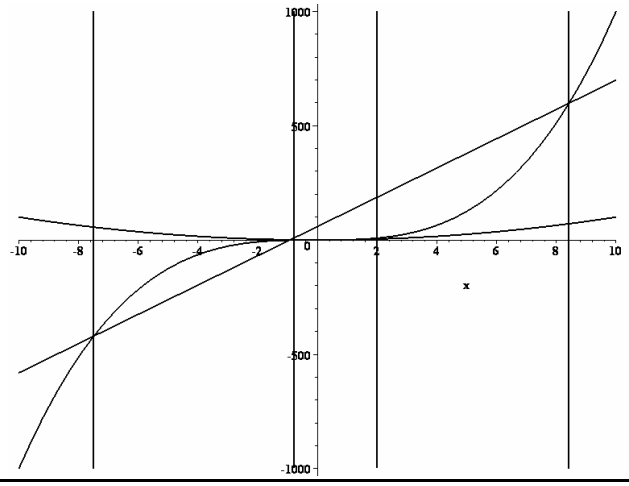


На основе ***coordplot-***функции предоставляется возможность накладывать на функциональный график *координатную* сетку, позволяющую не только лучше представлять поведение кривых, но и решать графически целый ряд численных задач. Один из таких механизмов представлен в [12]. Вместе с тем, данное средство уступает подобным средствам пакета ***Mathematica*** [12,41,111]. Для расширения возможностей пакета в этом направлении нами была создана процедура ***plotvl,*** представленная ниже своим исходным текстом и примерами применения.

```
plotvl := proc(F, h)
local a, b, c, d, g, vl, k, z, t, u, Xmin, Xmax, Ymin, Ymax;
    assign67( b = [ ], t = [ ], u = NULL );
    unassign('black', 'color', 'linestyle', 'thickness');
    vl := proc(L::list( realnum ), n::realnum, m::realnum)
        local a, k;
            a := [ color = 'black', linestyle = 1, thickness = 1 ];
            plots[ display ]( seq(
                plottools[ 'line' ]( [ args[ 1 ][ k ], n ], [ args[ 1 ][ k ], –abs( m ) ] ),
                k = 1 .. nops( args[ 1 ] ) ), `if`( nargs = 3, op( a ), args[ 4 .. nargs ] ) )
        end proc ;
    seq( `if`( type( args[ k ], 'equation' ) and lhs( args[ k ] ) = 'pvl',
        assign( 'c' = rhs( args[ k ] ) ), assign( 'b' = [ op( b ), args[ k ] ] ) ), k = 1 .. nargs);
    if nops( b ) = nargs then plot( op( b ) )
    else
        if type( c, 'list'( 'realnum' ) ) and c ≠ [ ] or
        type( c, 'nestlist' ) and type( c[ 1 ], 'list'( 'realnum' ) ) and c[ 1 ] ≠ [ ] then
            if type( c, 'list'( 'realnum' ) ) then z := map( evalf, c )
            else z := map( evalf, c[ 1 ] ); u := op( c[ 2 .. -1 ] )
            end if ;
```

```
            try
                g := plot( op( b ) );
                a := convert( g, 'string' );
                d := Search2( a, { "]]", "[[" } );
                for k by 2 to ∞ do
                    try t := [ op( t ), op( came( a[ d[ k ] .. d[ k + 1 ] + 1 ] ) ) ]
                    catch : break
                    end try
                end do;
                assign( Xmin = SLj( t, 1 )[ 1 ][ 1 ], Xmax = SLj( t, 1 )[ -1 ][ 1 ],
                    Ymin = SLj( t, 2 )[ 1 ][ 2 ], Ymax = SLj( t, 2 )[ -1 ][ 2 ] );
                z := [ seq( `if`( belong( z[ k ], Xmin .. Xmax ), z[ k ], NULL ),
                    k = 1 .. nops( z ) ) ]
            catch :
                WARNING( "main plot has been defined incorrectly - %1", b );
                RETURN( vl( z, 10, -10, u ) )
            end try;
            if z = [ ] then plot( op( b ) )
            else t := vl( z, Ymax, Ymin, u ); plots[ display ]( g, t )
            end if
        else ERROR( "option <pvl> is invalid - %1", c )
        end if
    end if
end proc
```

```
> f:=[TIMES, BOLD, 12]: plotvl([x^2, x^3, 64*x+59], x=-10..10, thickness=[2, 2, 2], pvl=[[-7.5,
  -0.8, 2, 8.4, 62, 57], color= blue, thickness= 2], color=[red, brown, magenta], labelfont= f,
  axesfont=f);
```



Вызов процедуры ***plotvl*(F, h,...)** без ***pvl***-опции эквивалентен вызову процедуры ***plot*(F,h, ...).** Опция ***pvl*** допускает один из следующих двух форматов кодирования, а именно:

pvl = [a, b, c, ...] или **pvl = [[a, b, c, ...],** *графические опции***]**

гле: **a, b, c, ... -** выражения of *realnum*-типа, тогда как *графические* опции те же, что и для стандартной процедуры ***plot,*** но в предположении, что рассматривается только *одна* выводимая функция. Точки **[a, b, c, ...]** установки вертикальных линий должны принадле-

жать **h**-интервалу, в противном случае такие линии не выводятся. Детальнее с процедурой ***plotvl*** можно ознакомиться в [108,109].

Средства анимации графических зависимостей. Рассмотренные до настоящего момента графические средства позволяют отражать только *статическую* суть представляемой графически функциональной закономерности, что не дает возможности оценивать определенные аспекты поведения данной закономерности со временем, т.е. ее *динамические* свойства. Такую возможность предоставляют основные средства анимации, поддерживаемые модульными функциями ***animate*** и ***animate3d*****,** определяемые **plots-**модулем. Принцип *анимации***,** поддерживаемый данными средствами, состоит в быстрой смене последовательности *фреймов* (*моментных снимков*) **ГО** один за другим, создавая в человеческом восприятии эффект *движения*, подобно тому, как это делается в современных (*да и делалось уже на заре их зарождения*) видео-средствах.

Модульная ***animate*****-**функция имеет следующий простой формат кодирования**:**

animate(<*Функция*>**, <X-***диапазон*>**, <A-***диапазон*>** {, <***Опции***>}**)**

где график *функции* или нескольких *функций* представляет собой непосредственно *анимируемый* объект. Функция **F(X, A)** должна быть действительной от двух аргументов **X** и **A,** где **X-**аргумент определяет собственно *ведущую переменную*, а **A-**аргумент **–** переменную *анимации***.** Обязательные второй и третий фактические аргументы должны принимать действительные значения**.** Если **X-**диапазон определяет отображаемый *участок* выводимого графика функциональной зависимости**,** то **A-**диапазон **–** режим изменения координат при смене фреймов в процессе *анимации*. Функция допускает также определение вертикального **Y-**диапазона**,** кодируемого непосредственно за **A-**диапазоном. В качестве первого фактического аргумента ***animate*****-**функции допускаются**:** одна или более *функций* (*включая заданные параметрически в виде множества функций*), *процедуры* либо списки значений координат *опорных* точек.

В качестве фактического необязательного аргумента ***animate*****-**функция допускает использование ***plot*****-**опций, рассмотренных выше, а также *специальной* ***frames*****-**опции, определяющей число участвующих в процессе анимации *фреймов* (*по умолчанию полагается* **16**). Вместе с тем, при использовании с ***animate*****-**функцией ***plot*****-**опций имеется ряд особенностей**,** которые необходимо учитывать**.** Фреймы для анимируемой функции **F(x, t)** на диапазоне **t = a..b** анимации создаются по следующему простому принципу**: t-**диапазон анимации разбивается на $\mathbf{t_k}$**-**точки и в них вычисляются *"моментные снимки"* (***фреймы***) анимируемой функции, т.е. ее **F(x, $\mathbf{t_k}$)-**образы на **x-**интервале по ведущей переменной (**k=1..*frames***). Последовательная *смена* таких фреймов собственно и составляет суть процесса *анимации*. По ***animate*****-**функции создается графическая структура данных вида**:**

PLOT(ANIMATE([2*D-ГО*_1], [2*D-ГО*_2], ..., [2*D-ГО*_n]), *графические опции***)**

где ***2D-ГО*_k** представляет собой базовую графическую ***plot*****-**структуру **CURVES-**типа для **к-**го фрейма анимируемого **ГО,** организация которой рассматривалась нами выше.

Выбор анимируемого **2D-ГО** сенситивно переводит ядро в режим анимации с одновременным выходом в графическое подокно **2D-**анимации**.** Средства *анимационного* подокна обеспечивают оперативный доступ к ряду функций, позволяющих управлять режимом анимации выделенного **2D-ГО,** а именно: **9** кнопок **4-**й строки **GUI** имеют следующее функциональное назначение**:**

(**1**) **-** немедленное *прекращение* процесса анимации (■); **(***кнопка* **1)**
(**2**) **-** *инициация* начала процесса анимации (▸); **(***кнопка* **2)**
(**3**) **-** переход к *следующему* фрейму анимации **ГО** (➔▌); **(***кнопка* **3)**
(**4**) **-** определение {*обратного* | *прямого*} *направления* процесса анимации ({🡐 | ➔}); **(4, 5)**

(**5**) **-** {*уменьшение*|*увеличение*} *скорости* анимации (*частоты смены фреймов*) ({«|»}); **(6, 7)**

(**6**) **-** определение режима анимации в рамках одного цикла *фреймов* (⇌); **(***кнопка* **8)**

(**7**) **-** определение *периодического* режима анимации цикла фреймов (↻). **(***кнопка* **9)**

При этом следует отметить, что переключатели **Animation-**группы **GUI** полностью дублируются более *оперативными* кнопками **4-**й строки окна, что делает **Animation-**группу с ее нынешним функциональным наполнением в значительной мере *избыточным* средством, хотя в ряде случаев они оказываются единственно доступными.

Вызов ***animate-***функции возвращает график анимируемой функции, точнее *первый* его *фрейм*. Последующее его *выделение* (*визуализируется рамкой* **ГО**) переводит ядро пакета в состояние графического ***animation-****подокна*, структура которого рассмотрена выше. Для дальнейшего управления процессом *анимации* выделенного **ГО** *вполне* достаточно средств **4-**й строки **GUI**. Активировав кнопку {**8**|**9**} определяем соответственно режим {*одноцикловый*|*периодический*} смены набора (*пакета*) фреймов**;** в первом случае после выполнения *цикла* (*последовательности всех фреймов*) процесс анимации прекращается**,** во *втором* **-** состоит из бесконечного чередования цикла фреймов. По кнопке **2** производится *запуск* собственно *процесса анимации*, а по кнопке **1 -** его *прекращение***.** Процесс анимации продолжается в фоновом видеорежиме до его явного прекращения, не влияя на выполнение других вычислений в среде пакета. В состоянии анимации одновременно может находиться несколько **2D-ГО.** Более того, созданный по ***animate-***функции графический объект может в последующем участвовать в процессе анимации без предварительного вызова создавшей его функции. При этом, *анимация* производится не в состоянии вычислений и не может прерываться по ***stop-***кнопке **3-**й строки **GUI.** Кнопки {**4,5**} позволяют изменять с точностью "*до наоборот*" порядок выполнения фреймов в цикле. Наконец, **3-**я кнопка действует только после активации одной из кнопок {**4, 5**} и позволяет производить переход в рамках одного цикла от фрейма к фрейму в определенном указанными кнопками порядке.

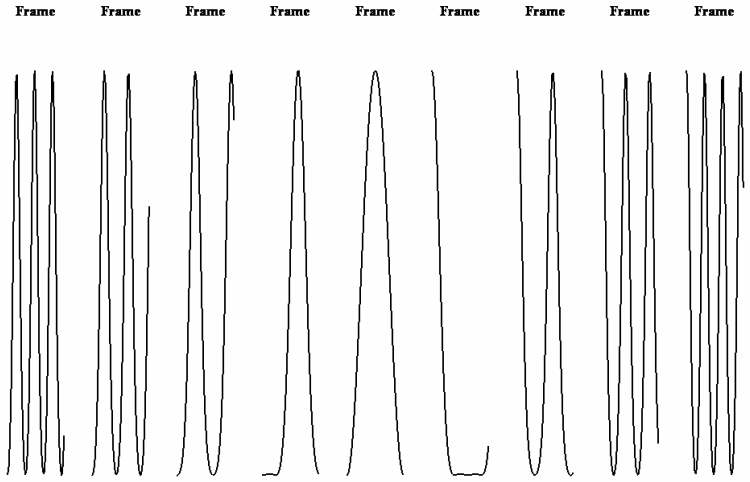
С учетом сказанного, управление процессом анимации достаточно прозрачно и легко усваивается на практической основе. Имея возможность визуально отслеживать *процесс* анимации, пользователю уже относительно несложно выбирать необходимые значения для рассмотренных выше параметров *процесса анимации*, а то и вовсе изменять характер самой функциональной зависимости с целью получения желаемой динамики. Именно таким способом могут быть обнаружены функциональные зависимости с весьма интересной динамикой [11-14,41,103].

По вызову ***with*(*plots*)** обеспечивается доступ ко всем средствам **plots-**модуля**,** включая и ***animate-***функцию**.** Затем на основе ***animate-***функции создается графический **2D-ГО** согласно представленной выше процедуре. С учетом ожидаемой динамики поведения **ГО** в процессе его анимации уделяется повышенное внимание выбору диапазона для ведущей **x-**переменной *анимируемой* функции и **a-**диапазона, управляющего собственно самой анимацией**.** Весьма существенную роль играет и выбор количества участвующих в процессе анимации *фреймов* (***frames-****опция*). Вызов ***animate-***функции допускает использование и установок для ***plot-***опций, например, ***color***, но здесь имеются проблемы.

Следующий фрагмент на основе графической ***animate-***структуры позволяет выделять из нее (*оставляем разбор деталей искушенному читателю*) и вывести *все* фреймы цикла некоторого анимируемого **ГО.** Именно последовательность смены этих фреймов и вызывает эффект *анимации* данного конкретного **2D-ГО.** Ради простоты и в целях компактности выбран цикл только из **9** фреймов. В частности**,** данный прием можно успешно использовать и для разложения динамики функциональных зависимостей на элементарные составляющие. Читателю в порядке и полезного, и захватывающего по зрелищ-

ности упражнения рекомендуется практическое освоение рассмотренных средств анимации ***Maple-***языка.

```
> Fr:= 9:  A:= animate(R*cos(R*x + sin(R*x)/R), x=0..2*Pi, R=-Pi..Pi, frames=Fr, color=red):
  Ar:=array(1..Fr):  for k to Fr do Ar[k]:= op(op(op(A)[1])[k]) end do: display(Ar, axes=none,
  title=`Frame`, thickness=2, axes=none, titlefont=[TIMES, BOLD, 14]);
```



В случае определения по ***color-***опции расцветки для нескольких кривых**,** участвующих в анимации по ***animate-***функции, выбирается общий для них цвет**;** при этом, каждый новый вызов ***animate-***функции циклически изменяет цвет кривых**.**Так, по вызову вида**:** ***animate*****({*cos*(t*x), *sin*(t*x)}, x=-Pi..Pi, t=-Pi..Pi, *color=green*)** производится *анимация* графиков двух кривых, окрашенных *общим* цветом**;** совершенно аналогичная картина имеет место и при отсутствия ***color-***опции. Следовательно, попытки определения *цветности* для различных кривых имеют идентичный эффект, искажая смысл ***color-***опции. Данный момент существенно ограничивает эффект от анимации и ее наглядность в случае многофункционального **2D-ГО.** О других особенностях анимации можно найти в [12].

В целях снятия ограничений на возможность окрашивания анимируемой по функции ***animate*** кривой можно воспользоваться методом *динамической* модификации соответствующей ей **ANIMATE-**структуры, как это иллюстрирует следующий фрагмент**:**

```
A_Color := proc(Art::function)
local k, n, m, p, z, h, t, d, v, Kr, LC, LC1, L, L1, Ln, F;
    [ interface( labelling = false ), assign( F = cat( currentdir( ), "/$6$3$" ) ) ];
    Kr, LC1, n, L, p, h, t, m, L1 := convert( Art, 'symbol' ), [ ],
        nops( [ op( op( 1, Art ) ) ] ), [ ], 1, 0, 0, length( convert( Art, 'symbol' ) ), [ ];
    LC := [ seq( op( op( op( op( 1, Art ) )[ k ] ) )[ -1 ], k = 1 .. n ) ];
    `if`( nargs < 2, ERROR( "Number of real arguments is incorrect: %1", args ), NULL )
        ;
    `if`( nops( LC ) ≠ nargs – 1, ERROR(
        "Number of Color-arguments is not equal to number of frames: %1", args ),
        NULL );
    for k from 2 to nargs do
        LC1 := [ op( LC1 ), op( convert( args[ k ], 'PLOToptions' ) ) ]
    end do;
    LC1 := map( convert, LC1, 'symbol' );
```

```
    do
        z := searchtext('COLOUR', Kr, p .. m);
        if z = 0 then break else L := [op(L), z + h]; h := h + z + 3; p := h + 1 end if
    end do;
    for k to nops(L) do
        h := searchtext(")", Kr, L[k] .. m); L1 := [op(L1), L[k] + h - 1]
    end do;
    assign('t' = open(F, 'WRITE')),
        writebytes(t, cat(`D_R_Col:=`, substring(Kr, 1 .. L[1] - 1)));
    for k to nops(L) - 1 do
        writebytes(t, LC1[k]);
        writebytes(t, substring(Kr, L1[k] + 1 .. L[k + 1] - 1))
    end do;
    writebytes(t, LC1[nargs - 1]);
    [writebytes(t, cat(substring(Kr, L1[nargs - 1] + 1 .. m), `;`)), close(t)];
    (proc(x) read x end proc)(F),
        [fremove(F), RETURN(D_R_Col, unassign('D_R_Col'))]
end proc
> Art:=animate(sin(t*x^2), x=-Pi..Pi, t=1..3, frames=7, thickness=2): A_Color(Art, color=
  red, color=green, color=tan, color=pink, color=gold, color=navy, color=blue);
```

Данный фрагмент представляет процедуру ***A_Color*(*Art*)** [41] от неопределенного числа формальных аргументов, предназначенную для модификации расцветки *фреймов* анимируемой **2D-**функциональной зависимости. *Первый* фактический ***Art*-**аргумент процедуры определяет **ANIMATE-**структуру, тогда как все последующие ее аргументы кодируются в виде последовательности уравнений вида ***color* =** *цвет*. При этом, их количество должно строго соответствовать количеству фреймов (***frames-****опция*) для анимируемого объекта. Суть алгоритма, реализуемого данной процедурой, в общих чертах достаточно прозрачна и оставляется читателю в качестве весьма полезного упражнения тем более, что использованные в процедуре приемы можно успешно применять при программировании модификаций **ANIMATE-**структур и в более широком аспекте.

```
Porshen := proc(N::posint)
local k, z, tor, C, Lf;
    [With(plots, 'display'), With(plottools, 'torus', 'cylinder'),
        assign(Lf = [ ], C = [XYZ, XY, Z, ZGREYSCALE, ZHUE]),
        assign(tor = torus([0, 0, 0], 5, 12))];
    for k from 0 to N do
        z := 12×sin(evalf(2×k×π/N));
        Lf := [op(Lf), display(tor,
            cylinder([0, 0, z - 16], 3, 32, 'shading' = C[(k mod 5) + 1]))]
    end do;
    display(Lf, 'insequence' = true, 'orientation' = [45, 64], 'scaling' = 'constrained')
end proc
```

Как уже упоминалось выше, по ***display-***функции установка ***insequence = true*** опции также определяет режим *анимации* для элементов списка **2D_ГО-**структур (*первого аргумента*), рассматриваемых в качестве последовательности фреймов. Созданный таким обра-

зом анимируемый объект аналогичен созданному по функции ***animate*** за некоторыми исключениями (*см. прилож.* **1** [12]). При этом, следует иметь в виду, что определение режима *анимации* на основе ***display-***функции в целом ряде случаев более предпочтительно, позволяя несколько шире использовать ***plot-***опции и совмещать в анимационном процессе как сугубо графические, так и текстовые фреймы, а также объекты, которые не допускают определения для них параметров анимации, включая графические примитивы, определяеиые как пакетным модулем **plottools,** так и самим пользователем. В предыдущем фрагменте представлен например анимации тора и цилиндра [12,103].

```
Animate2D := proc(F::algebraic, x::symbol, xa::range)
local a, b, c, d, n, k;
global __0;
    `if`( map( type, { rhs( xa ), lhs( xa ) }, 'numeric' ) = { true }, NULL,
        ERROR( "3rd argument is invalid" ) );
    assign( a = Indets( F ), n = nargs, '__0' = [ ], b = "", d = "" );
    if not member( x, a ) then error "quantity of unknowns is more than 1"
    elif not ( nops( a ) = 1 ) and nargs = 3 then
        error "<%1> contains non-numeric values", F
    elif n = 3 then return plot( F, x = xa )
    elif not ( nops( a ) - 2 ≤ nargs - 3 ) then
        error "quantity of parameters for animation is invalid"
    else
        a := a minus { x };
        if nargs - 3 < nops( a minus { x } ) then error "quantity of parameters for a
            nimation is more than quantity of variation ranges for them"
        elif { seq(
        type( args[ k ], { 'set'( 'numeric' ), 'list'( 'numeric' ), 'range'( 'numeric' ) } ),
        k = 4 .. nargs ) } ≠ { true } then
            error "values of parameters for animation are invalid"
        else
            for k to nops( a ) do b := cat( b, "seq(" ) end do;
            for k to nops( a ) do
                d := cat( d, a[ k ], "=", convert( args[ 3 + k ], 'string' ), ")," )
            end do;
            c := cat( "assign(`__0`'=[op(`__0`)," convert( F, 'string' ), "])," );
            parse( cat( b, c, cat( d[ 1 .. -2 ], ":" ) ), 'statement' );
            assign( '__0' = map( evalf, __0 ) )
        end if
    end if;
    plots[ display ]( map( plot, __0, x = xa ), insequence = true ), unassign( '__0' )
end proc
> p:=evalf(2*Pi): Animate2D(cos(d*x)*(c*x + a)*sin(b*x), x, -p..p, [1, 2, 3], {5, 6, 7}, 7..15,
  {7, 15}, [1, 5, 7]);
```

Функция ***animate*** из пакетного модуля **plots** обеспечивает анимацию **2D**-функциональных зависимостей, зависящих только от одного параметра анимации, что в целом ряде случаев недостаточно. Поэтому нами была определена процедура ***Animate2D,*** представленная предыдушим фрагментом. Процедура допускает *любое* конечное число аними-

рующих параметоров для алгебраического выражения **F(x, a, b, c, …)** от одной ведущей переменной **x** и конечного числа параметров {**a, b, c, …**} анимации. Детальнее с данной процедурой можно ознакомиться в [41,103,109,112,113].

Дополнительно к отмеченным, нами был создан еще ряд средств, полезных для работы с графическими объектами. Так, следующий фрагмент предоставляет две процедуры, позволяющие тестировать корректность графических {***plot*** | ***plot3d***}-опций.

```
type/plotopt := proc(po::equation)
local a, b;
    assign( b = interface( warnlevel ) ), interface( warnlevel = 0 );
    try a := plot( I, po ) catch : return false, null( interface( warnlevel = b ) ) end try ;
    true, null( interface( warnlevel = b ) )
end proc
type/plot3dopt := proc(po::equation)
local a;
    try a := plot3d( 1, 0 .. 1, 0 .. 1, po ) catch : return false end try ; true
end proc
> map(type, [thickness=2, color=aaa, font=[TIMES, BOLD, 16]], 'plotopt');
                                    [true, false, true]
> map(type, [filled=FAIL, scaling=CONSTRAINED, style=WIREFRAME], 'plot3dopt');
                                    [false, true, true]
```

Вызов процедуры **type(x, *plotopt*)** возвращает *true*-значение, если **x** - корректная **plot**-опция, и *false*-значение иначе. Очевидно переносится это и на вызов **type(x, *plot3dopt*).**

Вызов процедура ***p3listlist*** возвращает список **[[x1,y1, F(x1,y1)], ..., [xn, yn, F(xn, yn)]]** типа *listlist*, обеспечивая конвертацию структуры **3D-GRID** *Array*-типа в список. Процедура полезна при обработке ***PLOT3D***-структур, например, при *числовой* обработке **3D-ГО.**

```
p3listlist := proc(P::function)
local a, b, c, d;
    if convert( P, 'string')[ 1 .. 6 ] ≠ "PLOT3D" then error
        "factual argument should be PLOT3D data structure, but has received %1,"P
    else
        assign( a = [ op( 1, op( P )[ 1 ] ), op( 2, op( P )[ 1 ] ) ], c = op( 3, op( P )[ 1 ] ) );
        b := [ rhs( op( 2, c )[ 1 ] ), rhs( op( 2, c )[ 2 ] ) ];
        if nargs = 2 and type( args[ 2 ], 'symbol' ) then assign( args[ 2 ] = c ) end if;
        d := [ ( rhs( a[ 1 ] ) – lhs( a[ 1 ] ) )/( b[ 1 ] – 1 ),
            ( rhs( a[ 2 ] ) – lhs( a[ 2 ] ) )/( b[ 2 ] – 1 ) ];
        assign( 'a' = map( lhs, [ a[ 1 ], a[ 2 ] ] ) ), [ seq(
            seq( [ a[ 2 ] + d[ 2 ]×k, a[ 1 ] + d[ 1 ]×j, c[ k + 1, j + 1 ] ], j = 0 .. b[ 2 ] – 1 ),
            k = 0 .. b[ 1 ] – 1 ) ]
    end if
end proc
```

Следует помнить, что не все определенные явно опции можно переопределять в **GUI.** Переходим теперь к рассмотрению *функциональных* средств ***Maple***-языка, поддерживающих работу с **3D-**графикой. Изложение в целях удобства и приемственности проводится аналогично контексту **2D-**графики в виду того, что различия **2D-** и **3D-**графических средств в значительной мере определяются размерностью при большой их общности.

7.3. *Трехмерное представление функциональных зависимостей и данных в среде Maple-языка*

Графические средства ***Maple-***языка по представлению **3-**мерных функциональных *зависимостей* во многом аналогичны средствам для **2-**мерного случая**,** между тем, здесь имеется и ряд *принципиальных* отличий, которые будут рассмотрены в этом разделе. В первую очередь, это обусловлено появлением дополнительной размерности, что определяет появление и новых свойств таких, как цветовая иллюминация поверхностей, выбор точки обзора **3D-ГО,** более сложные функции расцветки поверхностей и др.

Графическое представление 3D-функциональных зависимостей. Для представления **3d-**функциональных зависимостей ***Maple-***язык располагает *базовой* ***plot3d-***функцией, имеющей три основных формата кодирования, а именно**:**

(1) ***plot3d*(F(x, y), x=a..b, y=c..d** {**, z=e..f**} {**,** <*Опции*>**})**

(2) ***plot3d*({F_1(x, y) ,..., F_n(x, y)}, x=a..b, y=c..d** {**, z=e..f**} {**,** <*Опции*>**})**

(3) ***plot3d*([X(t, p), Y(t, p), Z(t, p)], t=a..b, p=c..d** {**,** <*Опции*>**})**

и возвращающей **3D-ГО** в виде графика соответственно единственной функции, определенной функциональным **F(x, y)-**выражением по ведущим **(x, y)-**переменным, *нескольких* функций, определенных списком **[F_1(x, y) ,...., F_n(x, y)],** и параметрически заданной функции**,** определенных на диапазонах **a..b** и **c..d** значений **(x, y)**-переменных (**t, p-***параметров*) в единой системе декартовых **(X, Y, Z)-**координат. Так как ***plot3d-***функция возвращает **3D-ГО** только с вычисляемыми **(X, Y, Z)-**точками, то для обеспечения этого требования предполагается**,** что функциональные зависимости являются действительными. В дальнейшем предполагается, что все графические функции, возвращающие графики (*точнее графические структуры данных*) функциональных **3D**-зависимостей, требуют действительных значений над областью их определения, аналогично **2D**-случая.

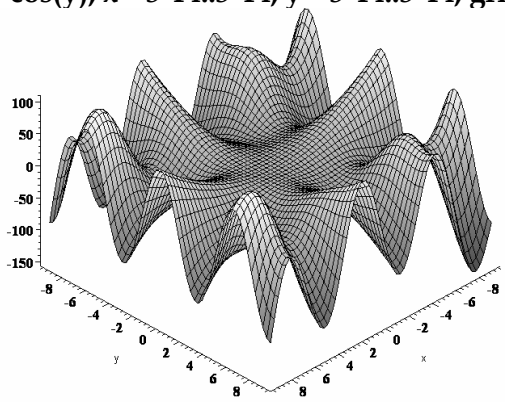
Первые три аргумента каждого формата ***plot3d-***функции, в общем случае, обязательны (*о них будет уточнено несколько ниже*), определяя соответственно функциональную зависимость и области значений ее *ведущих* переменных, тогда как четвертый аргумент первых двух форматов необязателен и определяет *выводимый* диапазон значений функциональной зависимости. Как правило, данный аргумент не кодируется во избежание возможной потери частей выводимого графика. Наконец, в качестве *необязательного* пятого аргумента всех форматов ***plot3d-***функции допускается использование специальных *графических опций***,** управляющих режимами создания, вывода и оформления **3D-ГО.**

В качестве первого фактического аргумента ***plot3d-***функция допускает использование**:** (**1**) действительной *функции* или *выражения* от двух ведущих переменных, (**2**) ***Maple-***процедуры и (**3**) *параметрически* определенной функции. В случае процедуры используется в общем случае конструкция вида ***plot3d*(*Proc*, a..b,c..d).** При определении *многофункционального* графика, содержащего графики нескольких функциональных зависимостей, определяемых первым аргументом-*множеством* ***plot3d-***функции, они выводятся в единой координатной сетке с возможностью различного цветового оформления. При этом, следует иметь в виду, что в отличие от рассмотренной ***plot-***функции, в качестве определителя *многофункционального* графика может выступать только *множество*, так как *список* идентифицирует *параметрически* заданную *поверхность*. Расцветка многофункционального **3D-ГО** также существенно отличается от случая **2D-ГО.**

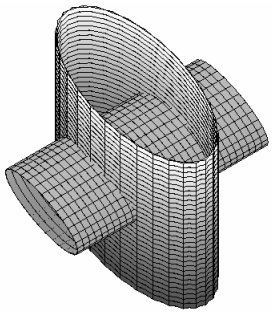
Функциональная зависимость, заданная параметрически (*формат* **3**), в качестве первого аргумента ***plot3d-***функции предполагает *списочную* структуру. Ее *элементы* определяют

{**X(t,p), Y(t,p),Z(t,p)**}**-**функции координат точки графика соответственно по осям **{X,Y,Z};** диапазоны изменения значений (**t, p**)**-**параметров которых кодируются вторыи и третьим фактическими аргументами **3-**го формата функции. Следующий простой фрагмент иллюстрирует применение ***plot3d-***функции для получения графиков некоторых функциональных зависимостей**:**

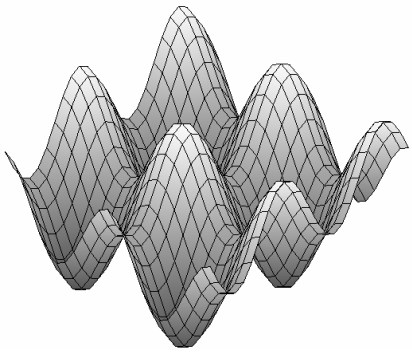
```
> plot3d(y^2*sin(x) + x^2*cos(y), x=-3*Pi..3*Pi, y=-3*Pi..3*Pi, grid=[59, 59], axes=framed);
```




```
> a:= 2*Pi: plot3d({[5.7*sin(u), 5.7*cos(u), v], [v, 3*sin(u), 3*cos(u)]}, u=-a..a, v=-2*a..2*a,
  grid= [32, 32]);
```



```
> Kr:= proc(x, y)  sin(x) + cos(y)  end proc:   plot3d(Kr, -a..a, -a..a);
```



Ввиду своей прозрачности каких-либо пояснений данный фрагмент не требует. Ниже представлены и другие интересные примеры применения ***plot3d-***функции.

Подобно случая **2D** вывод графика производится в той же секции **ТД,** что и определяющая его функция ***plot3d*** (*в дальнейшем такую секцию* **ТД** *будем называть графической*). При этом, установка курсора на **ГО** производится следующее. Графический объект ограничивается прямоугольной рамкой с *управляющими* точками по ее углам и серединам сторон. Будем в дальнейшем **ГО** с визуализированной вокруг него рамкой указанного вида называть *выделенным***.** Путем фиксации *мыши* на нужной управляющей точке рамки можно соответственным образом производить сжатие/растяжение **ГО** в плоскости по осям координат как независимо (*срединные точки*)**,** так и одновременно (*угловые точки*).

Выделенный **ГО** можно копировать в **СБО** для последующего импортирования как в другие места текущего **ТД,** другие **ТД,** так и в другие ***Windows*-**приложения в ***bitmap*-**формате. Именно такой подход использовался нами при подготовке оригинал-макета настоящей книги к публикации.

Между тем, в опреративном режиме щелчком правой клавиши мыши по выделенному **ГО** открывается меню, в котором выбором опции «***Export As***» получаем возможность сохранять **ГО** в файле в следующих форматах (*дополнительно к случаю* **2D**):

Drawing Exchange Format (**DXF**) и *Persistence of Vision* (**POV**)

Это позволяет проводить дальнейшее редактирование **ГО,** например, *графическими* редакторами и целым рядом других подобных средств. Подобный прием может быть использован и для сохранения **ГО** в других форматах для последующей обработки.

Управления созданием и выводом 3D-ГО*.** Для *управления* режимом и оформлением выводимых **3D-ГО** служат (*подобно случая* **2D**) специальные опции. Опции располагаются в произвольном порядке в качестве *последнего* фактического аргумента ***plot3d-функции, т.е. являются ключевыми. Функция ***plot3d*** имеет опции согласно следующей табл. **17,** в которой каждая опция определяет соответственно:

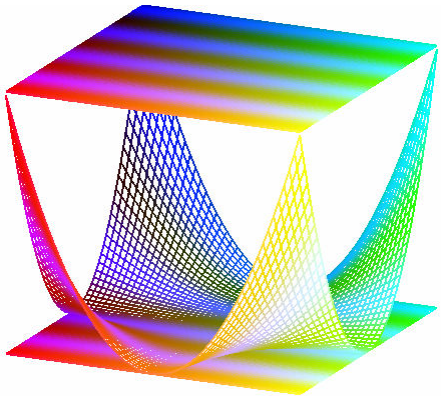
Таблица 17

Опция	***Опция графической plot3d-функции определяет:***
ambientlight **=[к,з,с]**	интенсивность *красного*, *зеленого* и *синего* цветов для пользовательской схемы освещенности**;** {**к, з, с**} принимают значения из интервала [0,1]
axes	используемый *тип* осей координат {***normal*** | ***boxed*** | ***frame*** | ***none***} (***none***)
axesfont	*шрифт* для меток маркируемых точек осей координат (***Default***)
color	функцию *окрашивания* точек выводимой поверхности **3D-ГО** (**XYZ**)
coords	используемую для вывода *систему* координат (***cartesian***)
contours	число *контурных* линий для выводимой поверхности (**10**)
font	шрифт для текстовой информации **3D-Г0** в виде [*шрифт*, *стиль*, *размер*]
***grid*=[n,m]**	размерность я*чеек* (**X, Y**)-сетки, в узлах которой вычисляется функция
gridstyle	используемый тип (**X, Y**)-сетки**;** допускается {***rectangular*** | ***triangular***}
labels	идентификаторы *меток* для осей координат (***No labels***)
labelfont	шрифт для идентификаторов меток осей координат (***Default***)
light	направление источника освещенности и его цветовые характеристики
lightmodel	выбор схемы освещенности выводимой поверхности (***none***)
linestyle	стиль оформления линий, образующих выводимую поверхность (**0**)
numpoints	минимальное число генерируемых *опорных* точек поверхности (**25x25**)
orientation	углы, под которыми обозревается выводимая поверхность [**45º, 45º**]
projection	перспективу обзора выводимой поверхности (***orthogonal***)
scaling	режим шкалирования по осям координат **3D-ГО** (***unconstrained***)
style	*стиль* оформления выводимых поверхностей (***patch***)
shading	режим раскрашивания выводимой поверхности (***Default***)
symbol	*тип* символов для точек поверхности в ***point*-**режиме (***Default***)
thickness	*толщину* линий, образующих выводимую поверхность (**0**)
title	*титульный* заголовок для выводимой поверхности (***none***)
titlefont	шрифт для *заголовка* выводимой поверхности (***Default***)
tickmarks	число маркируемых точек по осям координат (***Default***)
view	минимальные и максимальные координаты выводимых точек **3D-ГО**

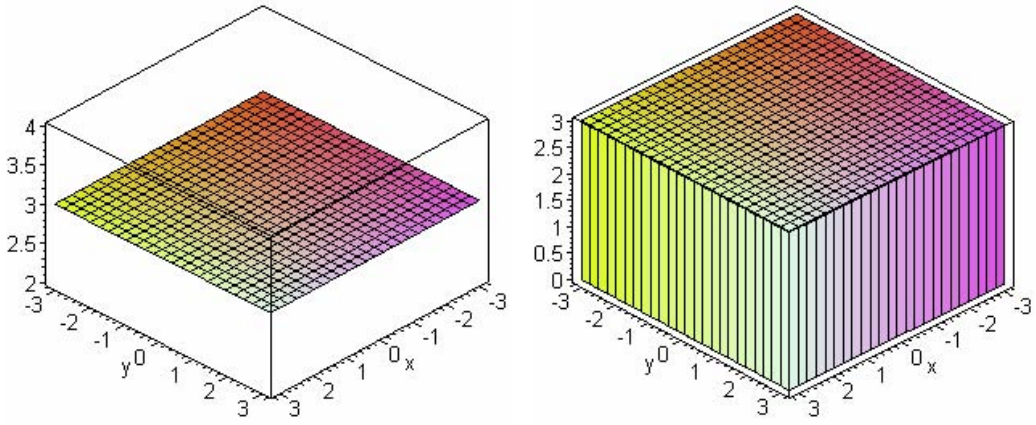
При этом, в зависимости от релиза состав и назначение *графических* опций уточняются.

Смысл большинства опций достаточно прозрачен и полностью соответствует (*с очевидными поправками на размерность*) одноименным ***plot-***опциям (табл. **15**), однако требуются некоторые пояснения для новых опций, используемых ***plot3d-***функцией, с которыми можно детально ознакомиться в [12] и в справке по пакету. Как отмечалось выше, пакет для обеспечения *расширенных* графических функций располагает модулем **plots.** В частности, он содержит ***setoptions3d-***функцию, чей вызов позволяет переопределять значения по *умолчанию* для всех графических опций ***plot3d-***функции на весь текущий сеанс. В [12] представлены рекомендации по организации эффективной работы с опциями. Следующий фрагмент илюстрирует использование *опций* для управления созданием и выводом **3D-**поверхностей, определяемых рядом функциональных зависимостей**:**

```
> with(plots): A:= [TIMES, BOLD, 10]: setoptions3d(axes=frame, axesfont=A, coords=
  cartesian, contours= 14, grid= [57, 57], font=[A[1], cat(A[2], ITALIC), 10], labelfont=A,
  lightmodel= light1, numpoints= 3237, thickness=2, tickmarks=[5, 5, 5], titlefont=[A[1],
  cat(A[2], ITALIC), 14]): plot3d({x^2*y^2, sin(x)^2+cos(y)^2, 21000}, x=-12..12, y=-12..12,
  axes=none, color=[0.39*x, 0.64*y, sin(x)], orientation=[25, 70], light=[45,45,0.64,0.59,0.39],
  style=hidden);
```



```
> plot3d(3, x= -Pi..Pi, y= -Pi..Pi);   plot3d(3, x= -Pi..Pi, y= -Pi..Pi, filled = true);
```



Примеры фрагмента дают определенное представление о возможностях ***plot3d-****функции* по графическому представлению **3D-**зависимостей, определяемых функциями (*включая заданных параметрически*), *процедурами* и *многофункциональных* **3D-ГО.** В порядке сравнения следует отметить, что упоминаемый пакет ***Mathematica*** располагает *более* развитым механизмом *опций* по созданию и управлению выводом функциональных **3D-***графиков*.

Текстовое оформление 3D-графических объектов. *И*з рассмотренной выше *графической* функции ***plot3d*** следует, что посредством поддерживаемых ею опций возможно в определенной мере производить текстовое оформление **3D-ГО** (*заголовки* и *осевые метки*), однако их *выразительные* возможности весьма ограничены**,** к тому же относительно *послед-*

них имеется ряд ограничений и особенностей [12], в значительной мере делающих эти средства не совсем приемлемыми для целого ряда важных приложений.

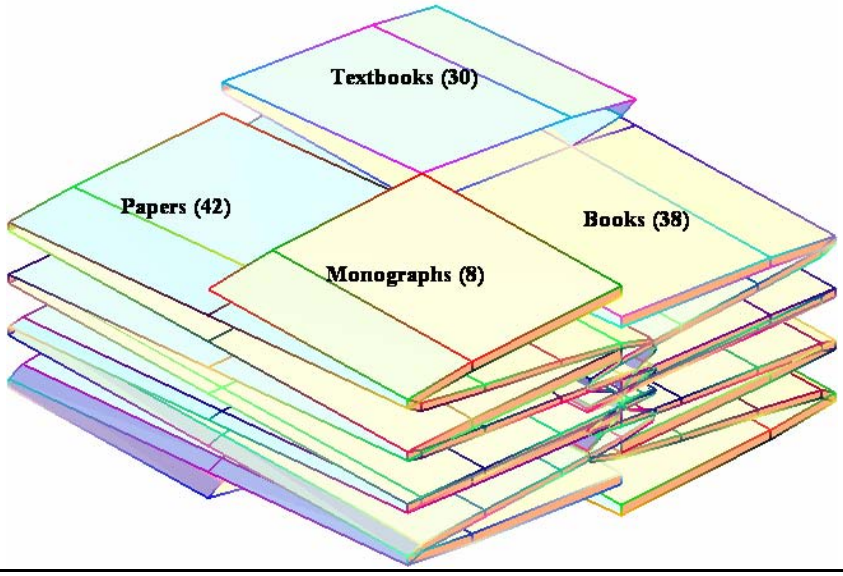
В качестве более гибкого средства текстового оформления **3D-ГО** выступает модульная функция ***textplot3d*(T {, <*Опции*>}),** где в качестве второго *необязательного* аргумента выступают уже рассмотренные ***plot-***опции и дополнительно *одна* специфическая для текста ***align-***опция, определяющая режим *расположения* текстовой строки *относительно опорной* точки. В качестве значений для ***align-****опции* допускаются следующие**:** ***ABOVE*** (*выше*), ***BELOW*** (*ниже*), ***RIGHT*** (*вправо*) и ***LEFT*** (*влево*)**;** по *умолчанию* полагается центровка текста по всем осям координат относительно *опорной* точки**.** Допускается одновременное применение триплетов значений ***align*****=** {***ABOVE***, ***LEFT***, ***RIGHT***}**.**

В качестве первого обязательного аргумента ***textplot3d-***функции выступает *текстовый элемент* (**ТЭ**) либо их *список* или *множество*. Отдельный **ТЭ** представляет собой *четырех*-элементную списочную структуру, в качестве первых трех элементов которой выступают координаты *опорной* точки**,** а в качестве *четвертого* **-** собственно сама *текстовая* строка, например, **[6.4, 5.9, 10, `*Russian Ecology Academy*`].** Недостатком ***textplot3d-***функции является невозможность изменения *ориентации* текстовой строки в зависимости от ориентации собственно самого **3D-ГО.** Поэтому изменение ориентации **ГО** влечет за собой, как правило, искажение взаимного расположения текстовой и графической компонент объекта, требуя редактирования координат *опорных* точек первых. *Текстовая* компонента **ТЭ** должна быть {*string*|*symbol*}-типа. По ***color-***опции производится расцветка текста, что позволяет улучшать оформление **ГО.**

В качестве простого примера представим создание *единого* **3D-ГО** посредством функции ***display3d*** из пяти **3D_ГО-**структур **P, T1..T4.**

```
> with(plots): a:=2*Pi: F:= [TIMES, BOLD]: T1:= textplot3d([12, 0.2, 37, `Monographs (8)`]):
  T2:= textplot3d([18, -0.18, 38, `Papers (42)`]): T3:= textplot3d([-11, -0.25, 37, `Textbooks
  (30)`]): T4:=textplot3d([-8.5, 0.3, 36, `Books (38)`]): P:=plot3d([t^2*sin(t), sin(h)*cos(h), t*h],
  t= -a..a, h= -a..a, thickness= 2, color= [sin(t*h), cos(t^2*h), h*t^2], orientation= [47, 42],
  ambientlight=[0.42, 0.47, 0.6], light=[57, 52, 0.2, 0.3, 0.4], lightmodel=light2): display3d(P,
  T1, T2, T3, T4, titlefont=[op(F), 18], style=hidden, color=black, title="Transactions of the
  Tallinn Research Group (1995 - 2006)",  font=[op(F), 14]);
```

Transactions of the Tallinn Research Group (1995 - 2006)

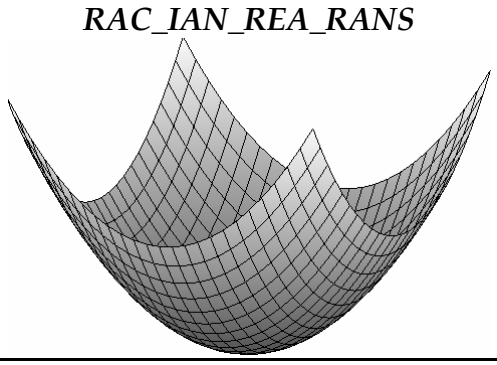


С учетом сказанного каких-либо особых пояснений данный фрагмент не требует. Пример фрагмента иллюстрирует создание посредством ***display3d-***функции *единого* **3D-ГО** из заранее заготовленных **3D_ГО-**структуры ***MESH-***типа, созданной на основе функции ***plot3d,*** и четырех текстовых структур, созданных на основе ***textplot3d-***функции**.** Следует иметь в виду, что идентификатор ***display3d*** является *дополнительным* именем ***display-***функции, рассмотренной выше. Однако *первый* идентификатор отмечает тот факт, что с ***display3d-***функцией допустимо использование именно графических ***plot3d-***опций. В [12] достаточно детально рассмотрены вопросы оформления составных **3D-ГО** и представлен целый ряд рекомендаций.

Так, в ряде случаев в результате *переопределения* шрифтов для заголовка **3D-ГО** возможна потеря текста (*см. прилож.* 1 [12]). Простая процедура устраняет данный недостаток**:**

```
T_Font := proc(ST::function, U::`=`)
local n, m, h, p, Kr, Ln, F;
    assign( Kr = convert( ST, 'symbol' ), F = cat( currentdir( ), "/$Art16_Kr9$" ) );
    `if`( nargs ≠ 2, ERROR( "Number of arguments is wrong: %1", nargs ), `if`(
        searchtext( 'PLOT', Kr ) = 0 and searchtext( 'PLOT3D', Kr ) = 0, RETURN( ),
        `if`( lhs( args[ 2 ] ) ≠ 'titlefont', RETURN( ), assign( h = length( Kr ) ) ) ) );
    assign( n = searchtext( 'TITLE', Kr ) ),
        assign( m = n + searchtext( ")", Kr, n .. h ) − 1 );
    p := n + searchtext( 'FONT', Kr, n .. m ) − 1;
    assign( Ln = open( F, 'WRITE' ) ),
        writebytes( Ln, cat( `_Art:=`, substring( Kr, 1 .. n − 1 ) ) );
    if p = n − 1 then writebytes( Ln, cat( substring( Kr, n .. m − 1 ), ",", "FONT(",
        substring( convert( rhs( args[ 2 ] ), 'symbol' ), 2 .. -2 ), ")", substring( Kr, m .. h ),
        ";" ) )
    else writebytes( Ln, cat( substring( Kr, n .. p + 4 ),
        substring( convert( rhs( args[ 2 ] ), 'symbol' ), 2 .. -2 ), substring( Kr, m .. h ), ";" ) )
        , close( Ln )
    end if;
    ( proc(x) read x; fremove( x ) end proc )( F )
end proc
```

```
> S:= plot3d(x^2+y^2, x=-3..3, y=-3..3, title=`RAC_IAN_REA_RANS`, orientation=[30, 83]):
  T_Font(S, titlefont=[HELVETICA, BOLDOBLIQUE, 18]);  _Art;
```



Первый фактический **ST-**аргумент процедуры ***T_Font*(ST, U)** определяет **3D-ГО-**структуру, а второй **-** *опцию* ***titlefont* = <***Значение*>**.** Процедура определяет глобальную переменную ***_Art,*** значением которой является **3D-ГО-**структура **–** результат соответствующей модификации *исходной* графической **ST-**структуры. Предыдущий фрагмент представ-

ляет исходный текст процедуры и пример ее применения. В качестве полезного упражнения читателю рекомендуется разобраться в организации процедуры ***T_Font***.

Специальные типы 3D-графиков функциональных зависимостей. *Наряду* с рассмотренными базовыми средствами **3D-**графики ***Maple-***язык располагает целым рядом *модульных* функций, поддерживаемых вышеупомянутым модулем **plots,** которые существенно расширяют графические возможности пакета. Все эти средства становятся доступными после выполнения вызова ***with*(*plots*)** или иным рассмотренным выше способом. Выше из средств **plots-**модуля мы рассмотрели целый ряд функций **2D-**графики, а для **3D-***случая* такие функции как**: *setoptions3d, display3d, textplot3d.*** Здесь мы кратко охарактеризуем ряд других, используемых для более специальных целей при работе с **3D-**графикой. Не останавливаясь детальнее, упомянем соответствующие одноименным *средствам* для **2D-**случая такие функции как: ***complexplot3d, listcontplot3d, implicitplot3d, contourplot3d, polygonplot3d, fieldplot3d, gradplot3d, listplot3d, pointplot3d, listcontplot3d*** и ***coordplot3d,*** обеспечивающие работу со специальными типами графиков. Их описания достаточно прозрачны и особых пояснений не требуют. Остальные интересные средства в данном направлении представляет следующая сводная табл. **18.**

Таблица 18

Функция	***Функция обеспечивает создание и вывод:***
cylinderplot	поверхности в *цилиндрической* системе координат
matrixplot	поверхности, основанной на значениях входов *матрицы*
polyhedralplot	многогранника на основе заданных *точек* либо *типов граней*
polyhedra_supported*() -** множества имен фигур, выводимых по ***polyhedralplot	
spacecurve	*пространственных* кривых
sphereplot	поверхности в *сферической* системе координат
surfdata	поверхности, основанной на значениях *координат точек*
replot	графика для модифицированной {**2D**|**3D**}_**ГО-**структуры
tubeplot	*трубчатых* поверхностей вогруг пространственных кривых

Некоторые из представленных в табл. **18** графических функций допускают использование дополнительно к рассмотренным ***plot3d-***опциям и специальных опций**,** управляющих режимом создания и вывода **3D-ГО.** Следует иметь в виду, что глобальные установки ряда ***plot3d-***опций, сделанные посредством ***setoptions3d-***функции, не распространяются на некоторые графические функции, поддерживаемые **plots-**модулем, поэтому их следует определять локально непосредственно в момент вызова функции. Детальнее с форматами кодирования и назначением данных функций можно ознакомиться в книгах [79-85] либо в справке по пакету. Мы ограничимся лишь некоторыми примерами их применения, учитывая относительно несложное освоение этих средств при практической работе в среде языка пакета, а также краткими пояснениями.

```
> restart; with(plots): Fmt:= matrix(2, 3, [[64, 59, 39], [44, 17, 10]]): F:= [TIMES, BOLD, 14]:
> Sp:= sphereplot([sin(v)*cos(u), sin(v), cos(u)], u= 0..Pi, v= -Pi..Pi, orientation= [6, 72],
  color=u+v, title=``, shading=ZHUE):  T:= textplot3d([0, 0.4, 0, `RAC_IAN_REA_RANS`],
  font= [TIMES, BOLDITALIC, 18]): matrixplot(Fmt, heights= histogram, axes= frame,
  gap=0.3, style=patch, labels=[``, ``, `Age  `], labelfont=F, axesfont=F, title="Diagram on
  the basis of matrixplot-function", shading=ZHUE, titlefont= subs(14=20, F)):
  display3d(Sp, T, color=red):  %; %%%;  polyhedraplot([64, 59, 39], polytype=icosahedron,
  orientation=[65, 44], thickness=2);
Warning, the name changecoords has been redefined
```

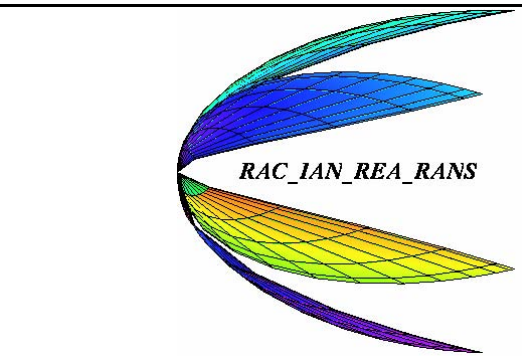
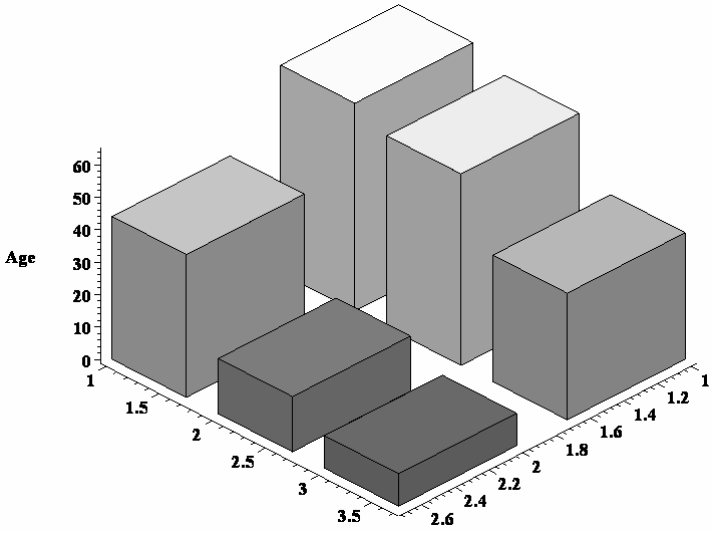
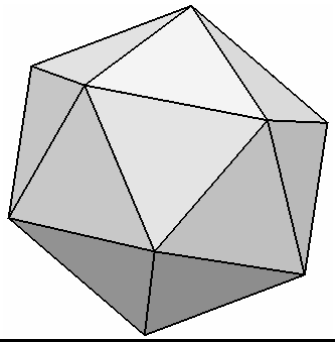


Diagram on the basis of matrixplot-function





Читатель и сам может создать интересные **3D-ГО,** использующие как ***plot3d***-функцию, так и средства из пакетного модуля **plots.** Это весьма *полезное* и *интересное* упражнение.

Средства анимации графических 3D-объектов. Выше механизм *анимации*, поддерживаемый ***Maple,*** был на содержательном уровне рассмотрен для случая **2D-**графики**.** С очевидными изменениями он переносится и на случай **3D-**размерности, поэтому более детально данный вопрос анимации нами здесь не рассматривается. Модульная функция ***animate3d*** имеет следующий формат кодирования**:**

animate3d*(**<*Функция*>**, <X-диапазон*>**, <Y-***диапазон*>**, <A-***диапазон*> {**, <***Опции*>**})**

где график функции либо нескольких функций представляет собой непосредственно *анимируемый* объект. Функция **F(X, Y, A)** должна быть действительной от трех аргументов **X, Y** и **A,** где **X,Y-**аргументы определяют собственно ведущие переменные, а **A-**аргумент **–** переменную анимации. Обязательные второй**,** третий и четвертый фактические аргументы ***animate3d*-**функции должны принимать действительные значения. Если **X-**

диапазон и **Y-**диапазон определяют *отображаемую* область *выводимого* **3D-**графика функциональной зависимости, то **A-**диапазон **–** определяет режим изменения координат при смене фреймов в процессе анимации. В качестве первого фактического аргумента функции ***animate3d*** допускаются: одна либо более функций (*включая заданные параметрически***;** *кодируются в виде множества функций*) и ***Maple-***процедуры**.**

В качестве фактического *необязательного* аргумента ***animate3d-***функция допускает использование ***plot3d-***опций, рассмотренных выше, а также *специальной* ***frames-***опции, определяющей число участвующих в процессе анимации фреймов (*по умолчанию полагается* ***frames*=8**). Механизм создания набора (*цикла*) фреймов и организация их смены, собственно составляющих суть процесса анимации, полностью соответствуют (*с очевидными изменениями*) **2D-**случаю, рассмотренному выше, и здесь детально не обсуждаются.

По ***animate3d-***функции создается графическая структура следующего вида**:**

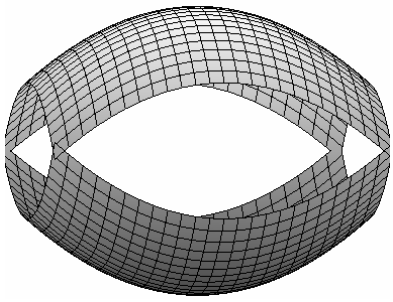
PLOT3D(ANIMATE([3D-ГО_1, 3D-ГО_2, ..., 3D-ГО_n)))

где **3D-ГО_k** представляет собой *базовую* графическую ***plot3d-***структуру **GRID-**типа для **к-**го фрейма анимируемого **ГО,** организация которой рассматривалась нами выше. На основе знания *организации* ***animate-***структуры относительно несложно можно осуществлять ее редактирование, расширяя тем самым возможности анимации. Однако в случае **3D-ГО** это представляется более трудоемкой процедурой, чем в **2D-**случае и ее проведение может потребоваться лишь в случае достаточно продвинутого программирования обработки графической информации в среде ***Maple-я***зыка [11-14,88,110,112-120].

Выбор анимируемого **3D-ГО** переводит ядро в режим *анимации* с одновременным выходом в графическое *подокно* **3D-**анимации, имеющее практически тот же вид, что и окно анимации для **2D-**случая за исключением *одного* естественного отличия. Если для *случая* **2D-**анимации **4-я** строка окна содержала поле с *координатами* выбранной точки области **2D-ГО**, то для случая **3D-**анимации на этом месте располагаются поля-регуляторы значений *углов* поворота в горизонтальной (**v-***окно*) и вертикальной (**φ-***окно*) плоскостях. Через эти же поля-*регуляторы* можно изменять значения указанных углов поворота**,** определяющих ориентацию **3D-ГО** в пространстве. В остальном же оба окна анимации *идентичны* как по структуре, так и по функциональному наполнению, а также по *принципу* предоставляемых ими средств для управления процессом анимации.

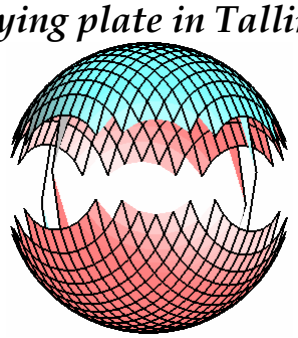
С учетом сказанного проиллюстрируем *принцип* работы с анимируемым **3D-ГО** на следующем простом фрагменте с учетом статичности книжной иллюстрации**:**

```
> with(plots): Fnt:= [TIMES, BOLDITALIC, 18]:  setoptions3d(font=Fnt, titlefont=Fnt):  T:=
  textplot3d([2, 0, 2, "RAC_IAN_REA_RANS"], color=red, font=Fnt): S:=plot3d({-x^2-y^2 +
  17, x^2 + y^2 - 17}, x=-3..3, y=-3..3):  display3d(S, T, insequence=true, orientation=[-9, 90]);
```



```
> A:= -4*Pi..4*Pi:  animate3d(sin(p)*sin(y*cos(p))*cos(x*sin(p)), x=A, y=A, p=A, frames=16,
  thickness=1, shading=ZHUE);
```

```
> R:=9:  A:= -R..R:  Z:=sqrt(R^2 - x^2 - y^2):  animate3d({[x, y, p*Z], [-x, -y, -p*Z]}, x=A, y=A,
  p=A, frames=16, thickness=2, shading=ZHUE, title="Flying plate in Tallinn",
lightmodel= light3, orientation= [42, 99]);
```

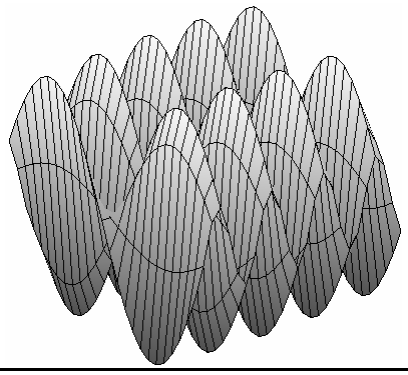


По вызову ***with*(*plots*)** обеспечивается доступ ко всем средствам **plots-**модуля, включая и ***animate3d*-**функцию**,** и посредством ***setoptions3d*-**функции делаются *глобальные* установки для ***plot3d*-**функций**,** определяющих объекты для *анимации*. В первом примере фрагмента определяются две **3D-**графические структуры типов **GRID** (**S**) и **TEXT** (**T**), для которых определяется *общий* анимируемый **3D-ГО** на основе ***display3d*-**функции с опцией ***insequence=true*.** Процесс *анимации* состоит в чередовании двух фреймов**,** определенных указанными выше **3D-**структурами. В частности, ***display*-**функции особенно полезны в случае необходимости анимации **ГО,** определяемых *списочными* структурами значений координат их точек, а также графическими примитивами**.** *Последние* два примера фрагмента иллюстрируют *анимацию* как *единственного*, так и *двух* **3D-ГО,** заданных *параметрически*, на основе ***animate3d*-**функции. В примерах фрагмента используется ряд опций ***plot3d*,** управляющих как созданием фреймов для обеспечения процесса анимации, так и визуализацией *собственно* самого процесса анимации. Вместе с тем, иллюстрация процесса анимации в отрыве от **ПК** носит достаточно условный характер.

Вызов ***animate3d*-**функции возвращает график анимируемого **3D-ГО,** точнее его *первый* фрейм. Последующее его выделение (*визуализируется рамкой* **ГО**) переводит ядро в состояние графического подокна ***animation*,** структура которого, практически, одинакова для **2D-** и **3D-**случаев. Для дальнейшего управления процессом анимации выделенного **ГО** вполне достаточно средств **4-**й строки подокна. Однако, в целом ряде случаев возникает необходимость выделения из анимируемой цепочки *фреймов* выделить отдельный. Данная задача решается несложной процедурой ***frame_n*.** Вызов процедуры ***frame_n*(d, F, x,** {**, y**} **,** t {**, gopt**} {**, *frame*=n**}**)** возвращает **n**-й *фрейм* анимируемого объекта, определяемого теми же фактическими аргументами, что и для процедуры ***animate*{*3d*}**. Детальнее с данной процедурой и примерами ее применения можно ознакомиться в [41,103,109].

```
frame_n := proc(d::{2, 3}, F)
local as;
    if type(args[-1], 'equation') and lhs(args[-1]) = 'frame' then
        if belong(rhs(args[-1]), rhs(args[d + 2])) then
            as := plots[`if`(d = 2, 'animate', 'animate3d')](args[2 .. -2]);
            return plots[display](op(op(as)[1])[rhs(args[-1])])
        else error "<%1> is inadmissible value for the frame number",rhs(args[-1])
        end if
    end if;
    plots[`if`(d = 2, 'animate', 'animate3d')](args[2 .. -1])
end proc
```

```
> A:= evalf(-4*Pi .. 4*Pi):  frame_n(3, sin(p)*sin(y*cos(p))*cos(x*sin(p)), x = A, y = A, p = A,
  shading = ZHUE, frame = 5);
```



Читателю в порядке как полезного, так и захватывающего по зрелищности упражнения рекомендуется практическое *освоение* средств *анимации* пакета на различного рода примерах. В подавляющей мере сказанное относительно процесса *анимации* **2D-ГО** переносится и на случай **3D-ГО.** В частности, копируется в **СБО** (*Clipboard*) также только *первый* фрейм анимируемого **3D-ГО.** Данное обстоятельство следует иметь ввиду.

7.4. *Создание графических объектов на основе базовых примитивов*

Рассмотренные выше графические средства языка позволяют как создавать *графическую* структуру данных на основе *требований* пользователя (*функциональная зависимость, массив или матрица данных, списочная структура и др.*), так и выводить созданный на их основе графический объект, определяя для него целый ряд характеристик, обеспечиваемых как базовыми {***plot, plot3d***}**-**опциями, так и специальными опциями, связанными со спецификой той или иной графической функции. При этом, *базовые* функциональные средства обеспечиваются ***plot-***функцией и ***plot-***опциями, тогда как *расширенные* **–** средствами **plots-**модуля пакета, рассмотренными в предыдущих разделах главы.

Наряду с рассмотренными язык располагает рядом графических средств, поддерживаемых **plottools-**модулем, обеспечивающих операции по созданию и манипулированию с графическими объектами на основе базовых графических *примитивов*, в качестве которых выступают наиболее часто используемые компоненты графических объектов**:** ***arc, arrow, circle, cone, cuboid, curve, cutin, cutout, cylinder, disk, ellipticArc, hyperbola, sphere, dodecahedron, ellipse, hemisphere, hexahedron, icosahedron, line, octahedron, pieslice, point, polygon, rectangle, semitorus, tetrahedron, torus.*** В виду, практически, интернациональной терминологии для подавляющего большинства из них *терминологический* перевод не требуется, а их *смысл* полностью определяется их названием. Средства, обеспечивающие данные *примитивы*, служат только для определения соответствующих им графических структур. Поэтому реальный *вывод* графических *примитивов* возможен только посредством ***display-***функции как в монопольном режиме, так и в составе списка/множества/массива графических структур, определяющих *составной* графический объект, пригодный, в частности, для участия в анимационном процессе. Например, по ***circle-***функции создается *графическая* структура **CURVES-**типа, определяющая *окружность* с заданными *центром* и *радиусом*, которая может участвовать в композиционном процессе по созданию составных **ГО** либо выводиться отдельным графическим объектом с заданными посредством опций характеристиками.

Наряду со средствами определения графических *примитивов*, **plottools-**модуль поддерживает восемь специальных функций, обеспечивающих основные процедуры преобразования графических объектов и манипулирования с ними**,** а именно**:** ***transform*** (*функция преобразования* **Г0**)**,** ***stellate*** (*изменение направленности углов много*{*гранника*|*угольника*}), ***translate*** (*стандартные преобразования* **ГО**), ***rotate*** (*вращение*), ***homotety*** (*шкалирование* {**2D**|**3D**}***_ГО-****структур*), ***reflect*** (отражение {**2D**|**3D**}***_ГО-****структур*), ***project*** (проектирование {**2D**|**3D**}**_ГО-***структур*) и ***scale*** (*шкалирование*). Ниже каждая из указанных функций рассматривается несколько детальнее.

В свете рассмотренных графических средств и обеспечивающих их функций *более* детального рассмотрения вопросов кодирования функций, определяющих *графические примитивы***,** не требуется и они легко усматриваются из их описания. Поэтому, представим только *определяющую* характеристику для *каждого* из поддерживаемых **plottools-**модулем *примитивов*. В табл. **19** представлены *графические примитивы***,** поддерживаемые модулем **plottools,** каждый из которых обеспечивается соответствующей графической функцией и представляет следующий *базовый* графический объект**:**

Таблица 19

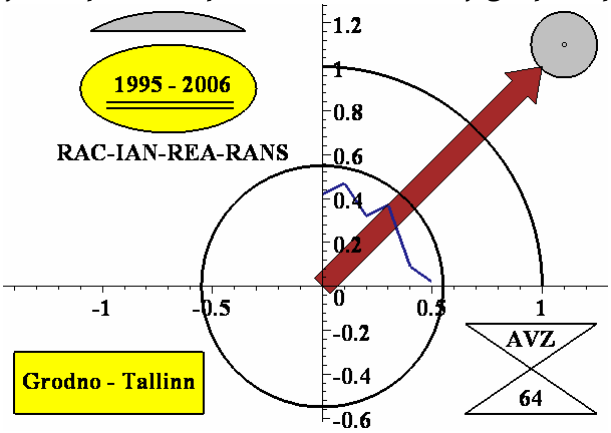
Функция	***Создает и выводит ГО-примитив соответственно для:***
arc	**2D-*дуги окружности*** с заданными определяющими параметрами
arrow	{**2D**|**3D**}***-стрелки*** с заданными определяющими параметрами
circle	***окружности*** с заданными центром и радиусом
cone	***конуса*** с заданным центром и *растром* (*углом дуги*)
cuboid	***куба*** с заданной *диагональю*
curve	{**2D**|**3D**}***-кривой,*** соединяющей заданные точки пространства
cutin	***объекта,*** являющегося результатом замены *многогранников* в ***3D_ГО-***структурах **POLYGONS-**типа на *подобные* им многогранники
cutout	***объекта,*** являющегося результатом *вырезки* из многогранников в ***3D_ГО-***структурах **POLYGONS-**типа *подобных* им окон
cylinder	***цилиндра*** с заданными определяющими параметрами
disk	***круга*** с заданными центром и радиусом
dodecahedron	***додэкаэдра*** с заданными определяющими параметрами
ellipse	***эллипса*** с заданными определяющими параметрами**;** ***filled-***опция определяет режим закраски области эллипса
ellipticArc	***дуги эллипса*** с заданными определяющими параметрами**;** ***filled-***опция определяет режим закраски области дуги эллипса
hemisphere	***полусферы*** с заданными центром и радиусом**;** ***capped-***опция определяет режим закраски внутренней поверхности полусферы
hexahedron	***гексаэдра*** с заданными определяющими параметрами
hyperbola	***гиперболу*** с заданными определяющими параметрами
icosahedron	***икосаэдра*** с заданными определяющими параметрами
line	{**2D**|**3D**}***-отрезка*** прямой между двумя заданными точками
octahedron	***октаэдра*** с заданными определяющими параметрами
pieslice	***сектора*** с заданными центром, радиусом и растром
point	{**2D**|**3D**}***-точки*** с заданными координатами
polygon	***ломаной линии,*** соединяющей заданные точки пространства
rectangle	***прямоугольника*** с заданными определяющими параметрами

semitorus	***полутора*** с заданными определяющими параметрами**;** ***capped*****-**опция определяет режим закраски внутренней поверхности *полутора*
sphere	***сферы*** с заданными центром и радиусом
tetrahedron	***тетраэдра*** с заданными определяющими параметрами
torus	***тора*** с заданными определяющими параметрами

В таблице **19** под выражением «*с заданными определяющими параметрами*» понимаются те *числовые* параметры, которые определяют тип выводимого {**2D**|**3D**}**-ГО,** его расположение**,** ориентацию в пространстве и др. Ряд особенностей использования приведенных в табл. **19** функций можно найти в книге [12]**.** Описание представленных выше функций, определяющих графические структуры для примитивов, достаточно прозрачно и применение их не вызывает особых затруднений. Примитивные структуры можно использовать с ***display*****-**функциями как для вывода отдельных (*определяемых ими*) *примитивных* **ГО,** так и для создания на их основе *составных* графических *объектов*. В качестве иллюстрации последнего целый ряд из представленных функций используется для создания и вывода достаточно простых составных графических **2D-** и **3D-**объектов, представленных следующим достаточно простым фрагментом**:**

```
> with(plots): with(plottools): F:=[TIMES, BOLD, 18]: setoptions(axesfont=F, font=F):
  setoptions3d(axesfont=F): Arc:= arc([0, 0, 0], 1, 0..Pi/2, color=blue, thickness=3): Arrow:=
  arrow([0, 0], [1, 1], 0.1, 0.2, 0.1, color=red): Circle:= circle([0, 0], 0.55, color=green,
  thickness=3): Curve:=curve([[0, 0.42], [0.1, 0.47], [0.2, 0.32], [0.3, 0.37], [0.4, 0.09], [0.5, 0.02]],
  thickness=3, color=navy): Disk:= disk([1.1, 1.1], 0.15, color=green, thickness=2): Ellipse:=
  ellipse([-0.7, 0.9], 0.4, 0.2, filled=true, color=yellow, thickness=2): T1:=textplot([-0.68, 0.93,
  `1995-2006`], scaling=constrained): EllipticArc:=ellipticArc([-0.7, 0.95], 0.5, 0.3, Pi/4..3*Pi/4,
  thickness=2, filled=true, color=gold): Line:= line([-0.98, 0.84], [-0.4, 0.84], thickness=2):
  Line1:= line([-0.98, 0.81], [-0.4, 0.81], thickness=2): Point:= point([1.1, 1.1], thickness=2,
  symbol=circle): T2:=textplot([0.95, -0.23, "AVZ"], scaling=constrained, color=navy): T3:=
  textplot([0.95, -0.5, `64`], scaling= constrained, color=red): T4:= textplot([-0.97, -0.42,
  "Grodno - Tallinn"], font=F): Polygon:= polygon([[0.65, -0.17], [1.25, -0.17], [0.65, -0.58],
  [1.25, -0.58]], thickness=2): Rectangle:= rectangle([-1.4, -0.3], [-0.54, -0.58], thickness= 2,
  color=wheat): T5:=textplot([-0.68, 0.62, "RAC-IAN-REA-RANS"]): Sv:=display(T1, Circle,
  Arc, Arrow, Curve, Disk, Ellipse, EllipticArc, Line, Polygon, T2, T3, Rectangle, T4, title=
  "Creation of complex 2D-plot on the basis of graphic primitives", Point, T5, Line1):
  T_Font(Sv, titlefont=[HELVETICA, BOLDOBLIQUE, 18]); _Art;
```

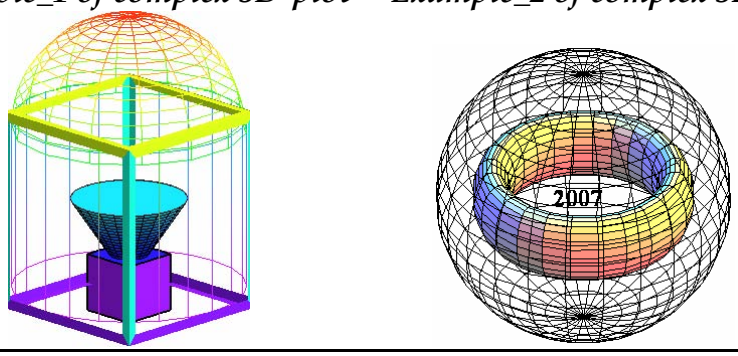
Creation of complex 2D-plot on the basis of graphic primitives




```
> Cone:= cone([0.5, 0.5, -0.5], 1, 1.5, shading = ZHUE, orientation = [42, 99]): Cuboid:=
  cuboid([1, 1, -1], [0, 0, 0], thickness=2): Cutin:= cutin(cone([0.5, 0.5, -0.5], 0.5, 1.5), 2/3):
```

```
Cutout:= cutout(cuboid([3, 3, 3], [0, 0, -0.5]), 0.9): Cylinder:= cylinder([0, 0, -0.8], 2, 3,
style=line, grid=[15, 15]): H:= hemisphere([0, 0, -2], 2, capped=false, style=line, grid= [15,
15]): H:= rotate(H, 0, Pi, 0):  ST2:= display3d(Cone, Cuboid, Cutout, Cylinder, H,
orientation= [45, 72], shading=ZHUE): T6:= textplot3d([2, 0.5, 1, "2007"], font= F, color=
black): Sphere:= sphere([0, 0, 0], 8, style = line):  D1:= display3d(scale(ST2, 3., 3., 2.),
title= "Example_1 of complex 3D-plot", lightmodel=light1): D2:= display3d(T6, Sphere,
scale(torus([0, 0, 0], 1, 5), 1, 1, 2), title= "Example_2 of complex 3D-plot", lightmodel=
light2, orientation=[48, 36]):  T_Font(D1, titlefont=[HELVETICA, BOLDOBLIQUE, 14]);
_Art;  T_Font(D2, titlefont = [HELVETICA, BOLDOBLIQUE, 14]);  _Art;
```

Example_1 of complex 3D-plot *Example_2 of complex 3D-plot*



В данном фрагменте, условно состоящем из двух частей, после создания набора графических объектов-*примитивов* на основе рассмотренных выше функций **plottools-**модуля соответственно для **2D-** и **3D-**случаев посредством ***display-****функций создаются* и *выводятся составные* графические объекты соответствующей размерности. Для определения ряда характеристик графических примитивов и составных объектов в *целом* используются соответствующие ***plot/plots-***опции, а также специальные опции, определяющие режим вывода отдельных примитивов. Из модульных функций, определяющих структуры для графических примитивов (табл. **19**), только отдельные допускают использование такого рода *специальных* опций, а именно**: *filled*=**{***false*|***true***} для функций ***ellipse, ellipticArc*** определяют режим *закраски* областей и ***capped*=**{***true*| ***false***} определяет для трех функций ***cylinder, hemisphere, semitorus –*** режим закраски *внутренних* поверхностей (*первыми указаны значения опций по умолчанию*).

Наряду с перечисленными средствами ***Maple-***языка для определения *шрифтов* заголовков графиков используется представленная выше процедура ***T_Font,*** обеспечивающая (*в отличие от стандартных средств языка*) надежное переопределение шрифтов. Лежащий в основе создания данной процедуры алгоритм может быть распространен на целый ряд интересных прикладных задач программирования *графики*, связанных с модификациями {**2D**|**3D**}**_ГО-**структур.

Наряду со средствами создания структур данных для графических *примитивов*, модуль **plottools** поддерживает средства, обеспечивающие функции *редактирования* этих структур с целью изменения ряда характеристик описываемых ими **ГО** или *изменения* характеристик непосредственно для указанных **ГО.** К таким характеристикам можно отнести вращения **ГО** в пространстве, изменение его масштабов относительно выбранных измерений и др. На некоторых из них остановимся несколько детальнее.

По функции ***scale*(ST, {Kx, Ky**|**Kx, Ky, Kz})** возвращается отредактированная графическая **ST-**структуры {**2D**|**3D**}**-ГО,** соответствующий объект которой относительно объекта исходной **ST-**структуры оказывается шкалирован (*сжат/растянут*) по осям координат согласно **Kh-**коэффициентам (**h**∈{**x, y, z**}) *шкалирования*. Наряду с графической структурой в качестве *первого* фактического аргумента ***scale-****функции* может выступать и непос-

редственно сам шкалируемый **ГО.** Использование ***scale-***функции имеет смысл только с установкой опции ***scaling=constrained.***

Специальным случаем ***scale-***функции является модульная ***homothety-***функция**:**

homothety*(**{**2D** | **3D**}**_ГО-структура*, <*Коэффициент*> {, <*Точка подобия*}**)**

определяющая *коэффициент* шкалирования заданной {**2D** | **3D**}**_ГО-**структуры одинаковым относительно осей координат. Третьим необязательным аргументом может выступать *точка подобия*. Функция возвращает указанным образом *шкалированную* графическую структуру данных, в последующем выводимую по ***display-***функции.

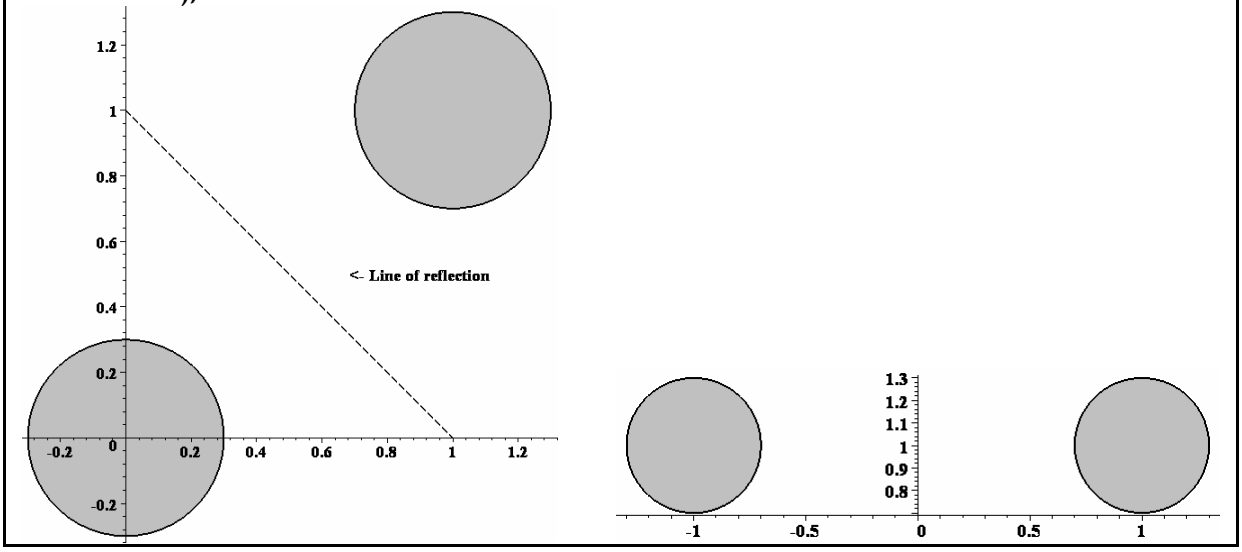
Функция ***rotate*(S, {α | α, β, γ})** по принципу действия подобна предыдущей, но с тем отличием, что возвращает отредактированную графическую **S-**структуру или непосредственно сам **ГО,** подвергнутые вращению относительно начала осей координат на указанные в радианах углы по соответствующим измерениям. В **2D-**случае вращение объекта производится против часовой стрелки. В предыдущем фрагменте обе указанные функции применялись для преобразований создаваемых составных **3D-ГО.**

Функция ***translate*(ST, {a, b | a, d, c})** по принципу действия аналогична ***rotate-***функции, но в отличие от второй производит параллельный перенос **ГО** (*или соответствующее редактирование отвечающей ему* **ST-***структуры*) на указанные действительными значениями отрезки вдоль осей координат**.** По функции ***stellate*(ST, h)** возвращается отредактированная графическая **ST-**структура **POLYGONS-**типа, определяющая *стеллированный* (*заострение граней*) многогранник**.** При этом **h-**коэффициент стеллирования определяет как высоту граней-*стелл*, так и их направленность**:** при **h < 1** *стеллы* направлены к центру многогранника, в противном случае от него. По функции формата вида**:**

reflect*(**{{**2D** | **3D**}**-ГО** | {**2D** | **3D**}**_ГО-структура*}**, <***Список опорных точек***>)**

производится генерация соответственно *новых* **ГО** или графической структуры**,** определяющих результат отражения {**2D** | **3D**}**-ГО** относительно заданного объекта (*точка или прямая для* **2D-***случая либо точка*, *прямая или плоскость для* **3D-***случая*), заданного списком определяющих его точек**.** Следующий фрагмент иллюстрирует примеры применения рассмотренных функций ***reflect*** и ***rotate:***

```
> map(with, [plots, plottools]): F:= [TIMES, BOLD, 14]: G:= disk([1, 1], 0.3, thickness=2,
  color=gray): Kr:= [0, 1], [1, 0]: L:= line(Kr, linestyle=3, color=black, thickness=2): T:=
  textplot([0.9, 0.5,"<- Line of reflection"], color=black, font=F): display([G, reflect(G, [Kr]),
  L, T], scaling=constrained, axesfont=F); display([G, rotate(G, Pi/2)], scaling=constrained,
  axesfont=F);
```

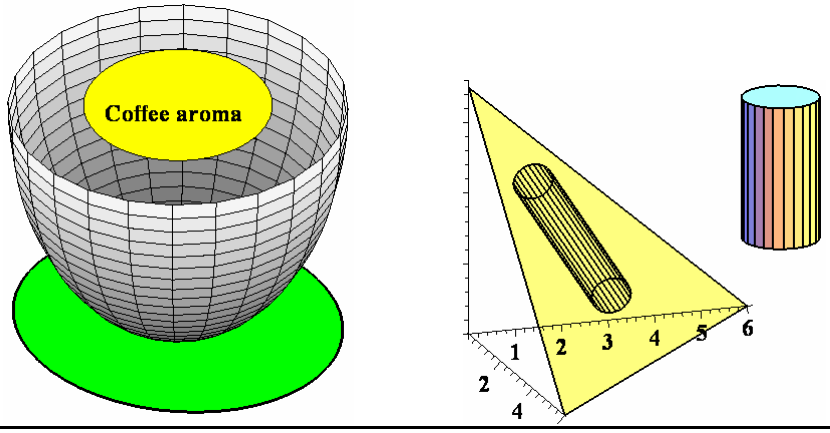


По модульной (**plottools-***модуль*) функции следующего формата кодирования**:**

project*(**{{**2D**|**3D**}-**ГО**|{**2D**|**3D**}_ГО-структура*}**, <***Список опорных точек***>)**

производится генерация соответственно *новых* **ГО** или графической структуры**,** определяющих результат проектирования {**2D**|**3D**}-**ГО** относительно заданного объекта (*прямая для* **2D-***случая либо прямая или плоскость для* **3D-***случая*), заданного списком определяющих его точек. Наконец, по ***transform*(*Proc*)-**функции генерируется пользовательская процедура/функция, позволяющая *редактировать* графические структуры данных посредством применения ее ко всем точкам исходной графической структуры. В качестве полезного практического примера продемонстрируем принцип создания таких процедур на фрагменте погружения **2D-ГО** в **3D-ГО:**

```
> map(with, [plots, plottools]): F:= [TIMES, BOLD, 18]: Tr:= transform((x, y) -> [x, y, 0]):
  T:=textplot([1, 0.4, "Coffee aroma"], color=black, font=F): Tr1:=transform((x,y) -> [x,y, -9]):
  Ch:=hemisphere([0, 0, 0], 9, capped=false): BL:=disk([0, 0], 5, color=yellow): H:=ellipse([0,
  0], 8, 9, color=green, filled=true, thickness=3):  Art:= display(Ch, map(Tr, [BL, T]), Tr1(H),
  shading = ZHUE):  Art;  Cyl:= cylinder([5, 5, 4], 0.8, 4):  Op:= [[6, 0, 0], [0, 6, 0], [0, 0, 7]]:
  display(Cyl, polygon(Op), project(Cyl, Op), orientation = [-12, 71], axes = normal,
  axesfont = F, lightmodel = light2, thickness = 2, shading = none);
```



Принцип организации такого механизма достаточно прозрачен и особых пояснений с учетом вышесказанного не требует. Читателю в качестве весьма полезного упражнения рекомендуется *практически* рассмотреть подобные графические примеры. Второй пример фрагмента иллюстрирует применение ***project*-**функции для *вывода* результата *проектирования* цилиндра на плоскость. Таким образом, функции ***changecoords, project, scale, reflect, rotate, stellate, homothety, transform*** и ***translate*** определяют полезные процедуры преобразования уже готового **ГО** либо его структуры.

Дополнительные замечания*.** Как и любое программное средство пакет ***Maple *постоянно* развивается, *пополняясь* как новыми средствами, так и *модифицируя* уже существующие. В этом отношении можно отметить и графические *средства* пакета. Прежде всего, начиная с *релиза* **9,** введен дополнительный формат модульной ***animate*-**функции, а именно:

animate*(<Функция***>, [<***Аргументы***>], t={a..b|L}, ...)**

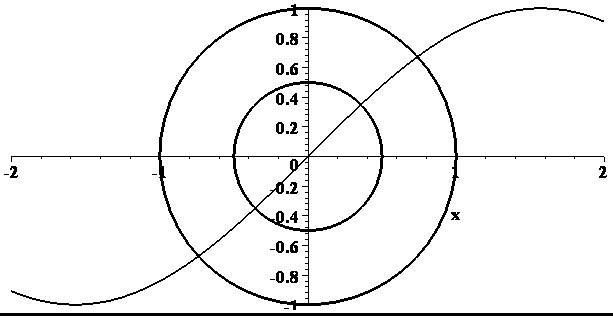
где *функция* определяет ***Maple*-**процедуру, генерирующую *графический* {2D|3D}-объект, второй аргумент определяет передаваемые ей фактические *аргументы*, параметр *анимации* **t** может определяться *диапазоном* действительных констант либо *списком* действительных или комплексных констант. При этом, ***animate*-**функция обеспечивает *анимацию* для любой ***plot*-**функции пакета, однако только по одному параметру. Тогда как наши процедуры ***Animate2D*** и ***Animate3D*** [41,103,109] позволяют анимировать функциональные зависимости по любому конечному числу параметров. Первая из указанных проце-

дур была представлена выше. Следующий фрагмент представляет примеры организации анимации одной и той же функциональной зависимости тремя способами, а именно: на основе соответствующей установки ***insequence***-опции ***display***-функции, ***animate***-функции первого формата и ***animate***-функции расширенного формата:

```
> plots[display]([seq(plot(sin(x), x = 0..k, thickness = 2, color = green, axesfont = [TIMES,
  BOLD, 12]), k = [0.01*p$p = 1..628])], insequence = true);
> plots[animate](sin(0.01*k*x), x = 0..2*Pi, k = 1..628, thickness = 2, color = green, axesfont =
  [TIMES, BOLD, 12]);
> plots[animate](plot, [sin(k*x), x = 0..2*Pi, thickness = 2, color = green, axesfont = [TIMES,
  BOLD, 12]], k = [0.01*p$p = 1..628]);
> plots[animate](plot, [sin(k*x), x = 0..2*Pi, thickness = 2, color = green, axesfont = [TIMES,
  BOLD, 12]], k = [0.01*p$p=1..628], background = plot({1, -1}, x = 0..2*Pi, thickness = 3,
  color = [red, red]));
```

При использовании *третьего* способа анимации в процессе его отображается динамика изменения значений **t**-параметра анимации, позволяя более четко отслеживать *динамику* фреймов в процессе анимирования объекта. Более того, третий способ анимации допускает использование опции ***background*=P,** где **P –** {***PLOT*** | ***PLOT3D***}-объект либо генерирующая его графическая процедура. В этом случае такой объект становится фоном, на котором происходит анимация основного объекта, определяемого вторым фактическим аргументом. *Четвертый* пример предыдущего фрагмента иллюстрирует указанную возможность. Естественно, режим анимации на фоне других объектов можно производить и другими способами, как это мы и делали ранее и как это иллюстрирует следующая достаточно простая процедура ***animbkgd:***

```
animbkgd := proc(X::list, Y::{set, list})
local k;
    plots[display]([seq(plots[display](k, op(Y)), k = X)], insequence = true,
    `if`(nargs = 2, NULL, args[3 .. -1]))
end proc
> P1:= plot(sin(x), x = -2..2, color= red, thickness= 2): P2:= plot({1, -1}, x = -2..2, color = blue,
  thickness=3): P3:=plot(cos(x), x=-2..2, color=gold, thickness=2): F1:=plottools[circle]([0,0],
  1, color=magenta, thickness=3): F2:= plottools[circle]([0, 0], 0.5, color= tan, thickness= 3):
  X:=[P1, P2, P3]: Y:={F1, F2}:  animbkgd(X, Y, axesfont=[TIMES, BOLD, 14], scaling=
  CONSTRAINED);
```



Первый аргумент **X** процедуры определяет *список* {***PLOT*** | ***PLOT3D***}-объектов, подлежащих анимации на фоне {***PLOT*** | ***PLOT3D***}-объектов, определяемых *списком* либо *множеством* **Y.** При этом, вызов процедуры ***animbkgd*(X, Y** {**,** *опции*}**)** может содержать дополнительные графические *опции*, обеспечивающие соответствующее дооформление результирующего анимируемого объекта. Пример фрагмента иллюстрирует сказанное. Наш подход *более* универсален, распростаняясь на *любые* графические объекты, не обязатель-

но отвечающие функциональным зависимостям. Между тем, третий способ анимации более удобен для работы с {2D|3D}-мерными функциональными зависимостями.

Начиная с ***Maple 9,*** пакет дополнен возможностью *интерактивного построения* графиков на основе алгебраических выражений, а именно. Щелчком *правой* клавиши по *алгебраическому* выражению ***Output***-параграфа текущего документа открываем окно, в котором по цепочке команд **Plots –> Plot Builder** открываем *подокно* «***Interactive Plot Builder***», через поля которого можно создавать требуемого оформления графический объект на основе исходного выражения с последующим его выводом. На данном аспекте внимания не концентрировалось, ибо на наш взгляд, он не относится к вопросам программирования в ***Maple*** и, во-вторых, он не дает *особых* преимуществ. Проще скопированное в **СБО** из ***Output***-параграфа выражение поместить в конструкцию следующего вида

{***plot***|***plot3d***}**(**<*Выражение*>**, ...)**

Input-параграфа и вычислить ее. Сразу же получаем искомый графический объект, желаемое оформление которого производится оперативно и весьма просто.

Наряду с рассмотренными графическими средствами как стандартными, так и определяемыми пакетными модулями **plots** и **plottools,** ряд других модулей также предоставляет средства для работы с более специфическими графическими объектами, среди таких модулей (*обладающих графическими средствами*) можно отметить следующие:

algcurves – исследование 1-мерных алгебраических кривых, определяемых полиномами от нескольких переменных

DEtools – содержит средства для графического представления решений дифференциальных уравнений

geom3d, geometry – средства работы с объектами 3- и 2-мерной *Эвклидовой* геометрии соответственно

ImageTools – средства для работы в среде ***Maple*** с файлами общих графических форматов

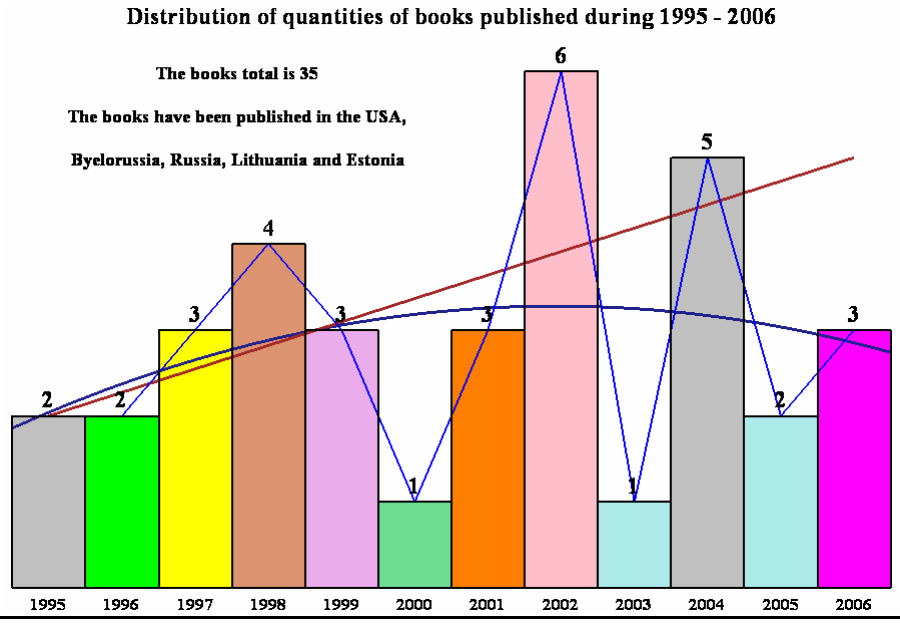
LREtools – содержит средства для графического представления линейных рекуррентных уравнений

Statistics (**stats**) – средства для графического *представления* статистических данных

Детально с графическими средствами данных модулей можно ознакомиться в справке по пакету. ***Maple*** имеет достаточно *ограниченный* набор средств для графического представления статистических данных. Поэтому, в целом ряду случаев пользователь вынужден для этого использовать иное программное обеспечение или программировать собственные процедуры в среде ***Maple***. *Встроенный* язык ***Maple*** – достаточно развитая программная среда для обеспечения необходимых для этого инструментов.

```
sHisto := proc(d, L::nestlist)
local k, var, t1, t2, n;
    n := nops(L);
    var := [seq(cat(`0`, k), k = 1 .. n)];
    t1 := [seq(cat(`1`, k), k = 1 .. n)];
    t2 := [seq(cat(`2`, k), k = 1 .. n)];
    seq(assign(var[k] = plottools['rectangle']([d×(k − 1), 0], [k×d, L[k][1]],
        'color' = L[k][2], 'thickness' = 2), seq(t1[k] = plots['textplot'](
        [1/2×(2×k − 1)×d, -0.1, convert(L[k][3], 'symbol')], 'align' = 'BELOW',
        'color' = 'black'), k = 1 .. n), seq(t2[k] = plots['textplot'](
```

```
        [1/2×(2×k − 1)×d, L[k][1] + 0.1, convert(L[k][1], 'symbol')],
        'align' = 'ABOVE', 'color' = 'black', 'font' = ['TIMES', 'BOLD', 18]), k = 1 .. n))
        , k = 1 .. n);
    RETURN(plots['display']({op(eval(var)), op(eval(t1)), op(eval(t2))},
        'tickmarks' = [0, 0], 'font' = ['TIMES', 'BOLD', 13], 'axes' = 'none', `if`(
        nargs = 3 and type(args[3], 'string'),
        op({'title' = args[3], 'titlefont' = ['TIMES', 'BOLD', 18]}), NULL)),
        unassign(op(var), op(t1), op(t2)))
end proc
> map(with, [plots, plottools, stats]): H:= sHisto(6, [[2,grey,1995], [2,green,1996], [3,yellow,
  1997], [4,tan, 1998], [3,plum, 1999], [1,aquamarine, 2000], [3,coral, 2001], [6,pink, 2002],
  [1,turquoise,2003], [5,gray,2004], [2,turquoise,2005], [3,magenta,2006]], "Distribution of
  quantities of books published during 1995 - 2006"): L:= [2, 2, 3, 4, 3, 1, 3, 6, 1, 5, 2, 3]:
> Lz:= listplot([seq([3+(k-1)*6, L[k]], k = 1..12)], thickness=2, color=blue):
> fit[leastsquare[[x, y], y=a*x^2+b*x+c, {a, b, c}]]([[seq(3+(k-1)*6, k = 1..12)], L]):
> V:= plot(A(x, [seq(3+(k-1)*6, k = 1..12)], L), x = 3..69, thickness=3, colour=orange):
> P:= plot(rhs(%%), x = 0..72, thickness=3, color=navy):
> Z:= plottools[line]([3, 2], [69, 5], thickness=3, color=brown):
> g:=textplot({[18.5, 6, `The books total is 35`], [18.5, 5.5, `The books have been published in
  the USA,`], [18.5, 5, `Byelorussia, Russia, Lithuania and Estonia`]}):
> display({H, Lz, P, Z, V, g}, font = [TIMES, BOLD, 14]);
```



В качестве достаточно полезного и поучительного примера выше представлена процедура ***sHisto,*** позволяющая выводить специальный тип столбиковых диаграмм (*гистограмм*). Детально с процедурой ***sHisto,*** а также с другими нашими средствами в данном направлении можно ознакомиться в [41,103,109].

Используя стандартные средства пакета для работы с *графическими* {2D|3D}-объектами совместно со знанием графических {***PLOT***|***PLOT3D***}-структур, уже достаточно несложно создавать эффективные средства работы с такого типа объектами, базируясь только на стандартных средствах программной среды пакета. Пример тому не только созданные нами средства [103], но и многочисленные средства других пользователей пакета, с которыми можно ознакомиться в цитируемой нами литературе и не только.

Глава 8. *Создание и работа с библиотеками пользователя в Maple*

Пакет ***Maple*** релизов **6 - 10** располагает рядом средств для создания достаточно эффективных механизмов работы с пользовательскими библиотеками, структурно аналогичными главной ***Maple-***библиотеке; эти библиотеки позволяют использовать в среде пакета содержащиеся в них средства на *уровне* доступа, аналогичного стандартным средствам пакета. В настоящей главе мы представим три достаточно эффективных *уровня* организации *пользовательских* библиотек *процедур*, *модулей* и *функций*. Между тем, средства, представленные в [103], позволяют существенно упрощать и расширять *набор* функций по работе с библиотеками пользователя. Как показывает наш опыт и опыт наших коллег, данные средства расширяют возможности пользователя по созданию и организации библиотек собственного 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 мы должны быть знакомы со средствами доступа к *файловой* системе компьютера и с основными типами файлов, с которыми работает ***Maple***. Являясь *встроенным* языком программирования в среде пакета**,** ориентированного, в первую очередь, на *символьные* вычисления (*компьютерная алгебра*) и обработку, ***Maple-***язык располагает относительно ограниченными возможностями по работе с данными, находящимися во внешней памяти **ПК**. И в этом отношении ***Maple-***язык существенно уступает таким традиционным языкам программирования как ***ADA, C++, Fortran, Cobol, PL/1, Pascal, Basic*** и др. Вместе с тем, ориентируясь, в первую очередь, на решение задач математического характера, ***Maple-***язык предоставляет *набор* средств для доступа к *файлам* данных, который *вполне* может удовлетворить достаточно широкий круг пользователей физико-математических приложений пакета**.** В наших книгах [7-14,41-43,103] средства ***Maple*** для доступа к файлам *различных* типов рассмотрены достаточно детально, по *полноте* изложения перекрывая как поставляемую с пакетом документацию, так и известную нам литературу по пакету [54-62,78-89]. С целью *расширения* пакетных средств доступа к файлам данных нами был создан целый ряд средств, с которыми можно ознакомиться в вышеупомянутых наших книгах и *Библиотеке* [41,103], ориентированной на ***Maple*** релизов **6 - 10**. Можно ознакомиться с данными средствами и по *демоверсии* этой *Библиотеки* [108]. Начиная с релиза **9,** пакет включает пакетный модуль ***FileTools*,** содержащий набор средств для работы с файлами двух основных типов, с которыми имеет дело пакет и его приложения – *бинарными* (***BINARY***) и *текстовыми* (***TEXT***). Наши средства, в массе своей, не пересекаются со средствами данного *модуля* и существенно расширяют возможности пакета по работе с файлами данных. Поэтому настоящая книга содержит описание базовых средств пакета для доступа к файлам данных, не претендующее на полноту. Вместо этого рекомендуется обратиться либо к нашей книге [12], либо бесплатно скачать исходные тексты наших книг по ***Maple***-тематике с *web*-адреса (*локальная копия сайта первого автора*)

http://www.grsu.by/cgi-bin/lib/lib.cgi?menu=links&path=sites

Данные материалы относятся, в основном, к релизам **5 - 7** пакета, однако ввиду вполне достаточной пролонгированности представленных в них средств вполне приемлемы и для последующих релизов пакета, прежде всего это относится к базовым средствам языка ***Maple*,** включая систему ввода/вывода пакета.

8.1. *Классический способ создания Maple-библиотек*

Главная библиотека пакета содержит наиболее часто используемые процедуры и модули (*которые не включены в ядро пакета*). Эта библиотека расположена в справочнике **LIB** пакета и содержит набор файлов, представленный на рис. **2;** библиотека содержит три *главных* файла ***"Maple.hdb", "Maple.ind"*** и ***"Maple.lib",*** тогда как *наличие* некоторых других *файлов* зависит от текущего *релиза* пакета (*так* ***Maple* 6, 7** *содержат файл* ***"Maple.rep"***).

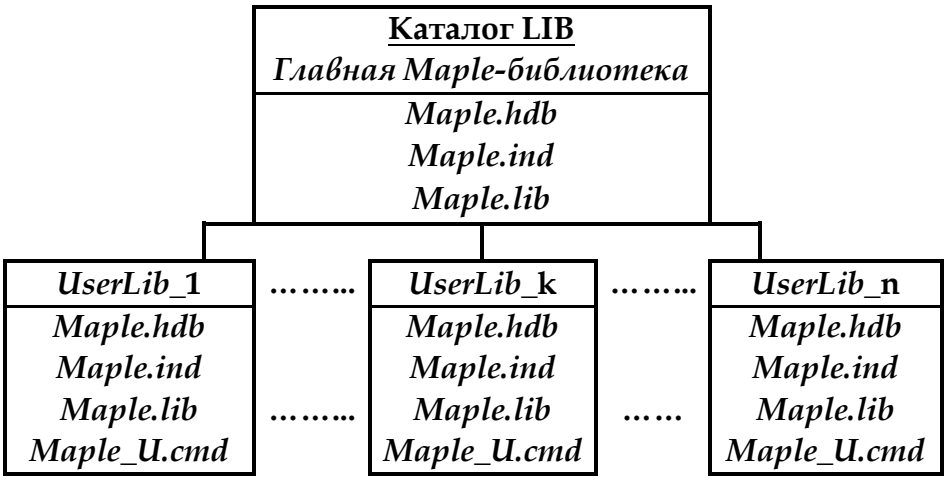


Рис. **2.** Принципиальная файловая организация главной ***Maple-***библиотеки и пользовательских библиотек, аналогичных главной библиотеке

В отличие от предыдущих релизов в ***Maple 10*** *главная* и другие библиотеки пакета организационно устроены несколько иначе: вместо трех файлов "***Name.hdb***", "***Name.ind***" и "***Name.lib***" (*библиотека mlib-типа*) они состоят из двух файлов "***Name.hdb***" и "***Name.mla***" (*библиотека mla-типа*), где *первый* (*в общем случае необязательный*) *файл* "***Name.hdb***" структурно остался неизменным, тогда как файл "***Name.mla***" представляет собой, по сути дела, слияние двух файлов "***Name.ind***" и "***Name.lib***" прежней организации с соответствующей коррекцией входов в начальной индексной части файла "***Name.mla***". На данном аспекте (*как не принципиальном*) внимания не акцентируется, принимая во внимание то обстоятельство, что библиотека *mlib*-типа легко конвертируется в эквивалентную библиотеку *mla*-типа, и наоборот.

На *втором уровне* библиотечной организации обеспечивается создание *пользовательских* библиотек в подкаталогах каталога **LIB,** содержащего главную ***Maple-***библиотеку, стандартно поставляемую с пакетом. В этом случае файловая организация *пользовательских* библиотек принимает следующий простой вид, наследуя структурную организацию *главной* ***Maple-***библиотеки (*рис*. **2**). При этом, каждая библиотека пользователя располагается в отдельном подкаталоге каталога **LIB** под именем **UserLib_k** (**k = 1 .. n**). 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 три *дополнительных* файла ***"Maple.rep", "elpam.ind"*** *и* ***"elpam.lib",*** чье описание может быть найдено в наших предыдущих книгах [29-33,39,41]. Ниже под термином "***Maple-****библиотека*" будет пониматься любая библиотека, структурно и организационно подобная главной ***Maple-***библиотеке пакета.

Организация библиотек *пользователя*, представленная выше, является достаточно удобной и четко локализует их расположение, обеспечивая их весьма *удобную* программную обработку. Наряду с этим, данная организация позволяет использовать для поддержки пользовательских библиотек средства пакета, а именно его *встроенную* функцию ***march***. Процедуры, которые представлены в [103], обеспечивают *создание* и *обновление* библиотек пользователя согласно вышеупомянутой файловой организации, а также их *логическое* соединение с главной ***Maple***-библиотекой, что обеспечивает возможность доступа к средствам, находящимся в них, на уровне стандартных средств. Использование библиотечной организации, поддерживаемой пакетом, существенно упрощает работу с *пользовательскими* библиотеками.

```
User_pflM := proc(R::{name, string, symbol}, U::{list(symbol), set(symbol)})
local
a, b, k, n, G, Art, W, U_cmd, M, Q, Z, P, V, h, BackSlash, F, MR, L, T, TU, TM, `0`;
    RegMW( ),
        assign(F = cat("", R), MR = Release('h'), 'h' = h, L = [op(U)], n = [ ]);
    try `if`(F[2] = ":", assign('G' = CF(F, 'string')),
        assign('G' = CF(cat(h, "/Lib/", F), 'string'))), [MkDir(G),
        march('create', G, `if`(3 ≤ nargs and type(args[3], 'posint'), args[3], 650))]
    catch "directory exists and is not empty": NULL("Especial situation 1")
    catch "there is already an archive in": NULL("Especial situation 2")
    end try;
    BackSlash := proc(S::{string, symbol})
        local a;
            assign(a = "");
            seq(assign('a' = cat(a,
                `if`(convert(S, 'string')[k] = "\", "/", convert(S, 'string')[k]))),
                k = 1 .. length(S)), a
        end proc;
    U_cmd := proc(S::string, L::{list, set})
        local k, Kr;
            Kr := fopen(cat(S, "/maple_U.cmd"), 'APPEND', 'TEXT');
            writeline(Kr, cat("$ New user procedures and modules = ",
                ssystem("date")[2][17 .. 30])),
                seq(writeline(Kr, L[k]), k = 1 .. nops(L)), fclose(Kr)
        end proc;
    writeline("$Help$", "Help database of the Aladjev's Library * version 2.169 *\
        nternational Academy
         of Noosphere, The Baltic Branch = Tallinn = May 3, 2003"),
        fclose("$Help$");
    makehelp(Empty, `$Help$`, convert(BackSlash(G), 'symbol')),
        fremove("$Help$");
    [assign('savelibname' = G, Art = (proc(x, k)
        local a;
            if search(x, `/`) or search(x, `*`) or type(x, `module`) then savelib(x)
            else a := cat("/$", x, cat(``, k), `.m`); save x, cat(h, a); a
```

```
            end if
        end proc )), seq( assign( 'n' = [ op( n ), Art( L[ k ], k ) ] ), k = 1 .. nops( L ) ) ];
    V := subsop( seq( `if`(
        search( L[ k ], `/` ) or search( L[ k ], `*` ) or type( L[ k ], `module` ), k = NULL,
        NULL ), k = 1 .. nops( L ) ), L );
    [ assign( 'W' = march( 'list', G ) ),
        assign( 'M' = [ seq( convert( W[ k ][ 1 ], 'symbol' ), k = 1 .. nops( W ) ) ] ) ];
    seq( march( `if`( member( cat( V[ k ], `.m` ), M ), 'update', 'add' ), G, cat( h, n[ k ] ),
        cat( V[ k ], `.m` ) ), k = 1 .. nops( V ) ), U_cmd( G, L ),
        delf( seq( cat( h, n[ k ] ), k = 1 .. nops( n ) ) ), map( march, [ 'pack', 'reindex' ], G ),
        assign( Q = cat( CCM( ), "\Maplesys.ini" ) );
    assign( 'Z' = readbytes( Q, 'TEXT', ∞ ) ), fclose( Q );
    assign( 'Z' = cat( Z[ searchtext( "UserD", Z ) + 14 .. searchtext( "
        ", Z, searchtext( "UserD", Z ) .. length( Z ) ) + searchtext( "UserD", Z ) − 2 ],
        "/Maple.ini" ) );
    assign( TM = cat( "libname:=libname,"", BackSlash( G ), "":" ),
        TU = cat( "libname:="", BackSlash( G ), "",libname:" ) );
    try
        assign( P = readbytes( Z, 'TEXT', ∞ ) ), fclose( Z );
        `if`( search( P, BackSlash( G ) ), NULL, [ fopen( Z, 'APPEND', 'TEXT' ),
            writeline( Z, `if`( 3 ≤ nargs and member( args[ 3 ], { "U", "M" } ),
            `if`( args[ 3 ] = "U", TU, TM ), `if`(
            nargs = 4 and member( args[ 4 ], { "U", "M" } ),
            `if`( args[ 4 ] = "U", TU, TM ), TU ) ) ] )
    catch "file or directory does not exist": writeline( Z, `if`(
        3 ≤ nargs and member( args[ 3 ], { "U", "M" } ), `if`( args[ 3 ] = "U", TU, TM ),
        `if`( nargs = 4 and member( args[ 4 ], { "U", "M" } ),
        `if`( args[ 4 ] = "U", TU, TM ), TU ) ) )
    catch "file or directory, %1, does not exist": writeline( Z, `if`(
        3 ≤ nargs and member( args[ 3 ], { "U", "M" } ), `if`( args[ 3 ] = "U", TU, TM ),
        `if`( nargs = 4 and member( args[ 4 ], { "U", "M" } ),
        `if`( args[ 4 ] = "U", TU, TM ), TU ) ) )
    end try ;
    ( proc() global libname; libname := G, libname; libname end proc )( ),
        WARNING( "the User library <%1> has been created/updated", G ),
        fclose( Z )
end proc
> Art:= () -> `+`(args)/nargs: Kr:= module () option package; export mean; mean:= () ->
  sqrt(`+`(args)/nargs) end module: Sv:= () -> [nargs, {min(args),max(args)}]: module Vict ()
  export gr; gr:= () -> WARNING("factual arguments %1", [args]) end module: Gal:= proc()
  Close() end proc: User_pfl("C:/RANS/Tallinn\\Grodno\\SVEGAL", [Art, Kr, Sv, Vict,
Gal],7, "M");
Warning, the user library <c:/rans/tallinn/grodno/svegal> has been created/updated
        "c:/rans/tallinn/grodno/svegal", "c:/program files/maple 9/lib/userlib",
                                "C:\Program Files\Maple 9/lib"
```

Для обеспечения работы с библиотеками пользователя, подобными ***Maple***-библиотеке, нами был создан целый ряд средств, описание которых можно найти в книгах [41,103] и в прилагаемой к ним *Библиотеке*. Данные средства были созданы еще для ***Maple 6*****,** тогда как некоторый 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 –*** **16** процедур. Однако целый ряд *функций* по работе с библиотеками так и не был реализован. Наш же набор средств наряду с *наиболее* массовыми представляет средства для восстановления поврежденных библиотек и средства их оптимизации. Некоторые из них будут представлены и в настоящей главе.

В частности, процедура ***User_pflM*****(F,U** {**, S** {**, R**}**)** обеспечивает создание или обновление пользовательских библиотек, структурно аналогичных главной ***Maple***-библиотеке, с их логической связью с главной библиотекой, что обеспечивает доступ к содержащимся в них средствам на уровне стандартных библиотечных средств пакета. Исходный текст процедуры с примерами ее применения представляет предыдущий фрагмент.

Первый аргумент **F** процедуры определяет местоположение пользовательской библиотеки. При этом, если указывается в качестве значения для **F,** каталог, то процедура рассматривает его как подкаталог каталога **LIB.** Если же указан полный путь, то по нему и создается/обновляется библиотека пользователя. Во втором случае достигается полная свобода по размещению пользовательских библиотек в файловой системе компьютера.

Второй аргумент **U** определяет список либо множество имен процедур, модулей и др., чьи определения вычислены в текущем сеансе и которые подлежат *сохранению* в существующей либо создаваемой библиотеке **F.** Для сохранения в библиотеке программных модулей в их определениях следует предусмотреть ***package*****–**опцию.

Третий необязательный аргумент **S** *posint*-типа определяет размер создаваемой библиотеки. Значение **S** рассчитывается из того условия, что для планируемого размещения в библиотеке **2p** средств выбирается **p.** Данное значение не лимитирует число сохраняемых в библиотеке объектов, однако его превышение вызывает создание нового индексного файла «***Maple.ind***», что не всегда целесообразно. По умолчанию полагается **650**.

Если в качестве третьего аргумента закодировано значение {**"U", "M"**}, то процедура устанавливает приоритет для создаваемой библиотеки: *максимальный* {*путем определения в предопределенной переменной* ***libname*** *цепочки путей к библиотекам* **-** <*user library***,** *libname*> (**"U"**)} или *минимальный* {*путем определения в предопределенной переменной* ***libname*** *цепочки путей к библиотекам* **-** <*libname***,** *user library*> (**"M"**)}. По умолчанию полагается **"U".**

Четвертый необязательный аргумент **R** имеет тот же смысл, что и третий **S,** допуская те же значения {**"U", "M"**}. По умолчанию, для данного аргумента полагается значение **"U"** для вновь создаваемой библиотеки и **"M"** для обновляемой библиотеки. Успешный вызов процедуры возвращает текущее состояние предопределенной ***libname***-переменной и выводит сообщение по созданной/обновленной библиотеке. После этого *пользователь* имеет возможность работать с ее средствами *аналогично* стандантным средствам пакета. Более детально с работой и возможностями процедуры ***User_pflM*** и ее модификациями можно ознакомиться в [41,103]. Здесь же мы не ставим целью рассмотрение всех созданных нами средств по работе как с ***Maple***-библиотеками, так и с библиотеками иных организаций, отсылая заинтересованного читателя к [12-14,41,103,108,109], а рассмотрим лишь стандартный подход к созданию ***Maple***-библиотек пользователя.

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 создает-

ся пустая библиотека, используя встроенную ***iolib*-функцию** (*ранее она была на уровне утилиты*) ***march*, имеющую** следующий формат кодирования для этой цели:

march('create', <*Полный путь к библиотеке*>**,** <*Размер*>**)**

Однако перед дальнейшим рассмотрением *целесообразно* детализировать понятие «*полный путь*», уже использованное выше**.** *Полный путь* к искомому файлу/каталогу в файловой системе компьютера**,** как правило**,** кодируется в виде значения {*string*, *symbol*}-типа и для среды {***DOS*|*Windows***}** имеет следующий формат кодирования**:**

<**УВВ**>**:**{\\|/}<*подкаталог*_1>{\\|/}<*подкаталог*_2>**...**{\\|/}<*подкаталог*_n>{\\|/}{<*файл*>}

где**: УВВ** определяет *логическое имя* устройства (*например*, *жесткий диск*, CD-ROM, USB *и т.д.*), *подкаталог*_к **-** подкаталог файловой системы и *файл* - имя файла в виде <*Основное имя*> {**.**<*Расширение имени*>}. Символ {\\|/} служит в качестве *разделителя* в определении пути и по ***dirsep*-**параметру функции ***kernelopt*** (*обеспечивающей механизм связи между ядром **Maple** и пользователем*) можно получать *его* значение *по умолчанию* ***kernelopts*(*dirsep*);** ⇒ "\\"**.** Однако его нельзя переопределять. Между тем, ***Maple*** допускает в качестве разделителей в определении пути к файлу/каталогу *любой* из символов {"\\"|"/"}, однако их использование в общем случае неэквивалентно, хотя и предоставляет целый ряд дополнительных возможностей, детально рассматриваемых в [103]. Здесь же и ниже будет использоваться любой из указанных разделителей. Если указанная выше цепочка каталогов (*полный путь*) завершается каталогом, то она определяет путь к последнему каталогу, в противном случае к файлу данных.

Перед созданием библиотеки по ***march*-**функции предварительно мы должны создать для нее каталог (*если он не был создан ранее*), в котором она будет размещаться. Создавать каталог можно либо средствами ***DOS*, *Windows*,** либо в среде самого ***Maple*,** для чего существует *iolib*-функция ***mkdir*,** имеющая следующий простой формат кодирования:

mkdir(<*Путь к каталогу*>**)**

Если используется *полный* путь к каталогу, то он начинается с **УВВ,** в противном случае создаваемый каталог будет расположен в текущем каталоге пакета, который определяется по вызову *iolib*-функции ***currentdir*(),** например:

```
> currentdir();   ⇒ "C:\Program Files\Maple 10"
> mkdir("UserLib"); currentdir("UserLib"); currentdir();
                        "C:\Program Files\Maple 10\UserLib1"
```

Первый пример фрагмента определяет текущий каталог пакета (*как правило*, *это основной каталог пакета*), тогда как второй пример иллюстрирует создание в нем нового каталога **UserLib** под создаваемую библиотеку, определение его текущим и проверку нового текущего каталога. Под *текущей* понимается *цепочка подкаталогов*, с которой работают средства доступа пакета по умолчанию, т.е. в случае указания только *имени* файла. По вызову ***currentdir*()** возвращается значение текущей цепочки**,** тогда как по вызову вида ***currentdir*(**<*цепочка*>**)** устанавливается указанная *цепочка* в качестве текущей.

При установке *новой* текущей *цепочки* подкаталогов функция ***currentdir*** возвращает предыдущее значение для текущей цепочки**,** что позволяет при необходимости легко возвращаться к предыдущему значению. При этом, следует иметь в виду, что установка текущей цепочки отменяется только перезагрузкой пакета или ее переопределением. Более того, установленный по ***currentdir*-**функции каталог становится текущим *только* для *внешних* файлов пакета, т.е. эта функция используется со средствами доступа к *внешним* файлам данных пакета.

Между тем, по ***mkdir***-функции мы имеем возможность создавать *цепочки каталогов* только одного уровня вложенности либо только один каталог, находящийся в конце уже существующей цепочки каталогов, иначе возникает ошибочная ситуация, например:

```
> mkdir("C:\\Program Files\\Maple 8\\UserLib1/Dir1/Dir2");
Error, (in mkdir) file or directory does not exist
```

Это является весьма существенным ограничением в задачах, имеющих дело с доступом к элементам ***файловой*** системы компьютера. Для устранения данного недостатка нами была создана процедура ***MkDir*** [103], рассмотренная выше Процедура ***MkDir*(*dirName*)** создает *каталог*, *цепочку каталогов* и/ или *файл данных* в файловой системе базовой операционной среды. Аргумент ***dirName*** типа {*string*, *symbol*} определяет *цепочку* каталогов, которая должна быть создана. Процедура ***MkDir*** допускает кодирование фактического аргумента ***dirName*** *строкой* или символом следующего вида:

Устройство:*Каталог*/*Подкаталог*_1\\.../*Подкаталог*_n

Набор символов, допускаемых в названиях каталогов, является системо-зависимым, также как и символ, используемый для разделения компонент цепочки каталогов. Так, если в качестве разделителя используется обратный слэш (*backslash*), то он должен удваиваться, т. к. строки ***Maple*** используют этот символ в качестве управляющего (*escape*).

Создав по встроенной функции ***mkdir*** или по нашей процедуре ***MkDir*** каталог под создаваемую библиотеку (*в нашем примере каталог* "c:/program files/maple 10/UserLib"), теперь мы можем в нем создать *«пустую»* библиотеку с именем **UserLib**. Делается это по вызову следующего формата кодирования:

march('create', "C:\\Program Files\\Maple 10\\UserLib", <*Размер*>**)**

где аргумент <*Размер*> определяет размер создаваемой библиотеки. Размер библиотеки определяется количеством содержащихся в ней ***m***-файлов с ***Maple***-объектами (*процедуры*, *модули*, *таблицы и т.д.*). При этом, практически, имеется возможность сохранения в такой библиотеке числа ***m***-файлов, примерно равного удвоенному числу, заданному во втором фактическом аргументе. О ***m***-файлах речь шла выше. Таким образом, по вызову следующего формата кодирования:

march('create', "C:\\Program Files\\Maple 10\\UserLib", 350)

мы создаем пустую библиотеку **UserLib,** предназначенную для сохранения порядка **700** различных ***Maple***-объектов (*процедуры*, *модули*, *таблицы и т.д.*).

В библиотеке создаются два макетных файла ***"Maple.ind"*** (*индексный*) и ***"Maple.lib"*** (*с* ***m****-файлами* ***Maple****-объектов*), впоследствии обновляемые при каждом помещении в библиотеку новых ***Maple***-объектов либо при ее реорганизации средствами той же ***march***-функции. В данной ситуации библиотека готова к своему *наполнению* программными средствами – процедурами, модулями и другими ***Maple***-объектами, например, таблицами.

<u>***Этап 2***</u>. На данном этапе необходимо в текущем сеансе вычислить определения ***Maple***-объектов, помещаемых в библиотеку, например:

```
> Sr:= () -> `+`(args)/nargs;  Ds:= () -> sqrt(sum((args[k] - Sr(args))^2, k = 1..nargs)/nargs);
```

$$Sr := (\,) \to \frac{\text{`+`(args)}}{\text{nargs}}$$

$$Ds := (\,) \to \sqrt{\frac{\sum_{k=1}^{\text{nargs}} (\text{args}_k - \text{Sr(args)})^2}{\text{nargs}}}$$

```
> 6*Sr(64, 59, 39, 10, 17, 44),  6*Ds(64, 59, 39, 10, 17, 44);
```

$\Rightarrow \quad 233, \sqrt{14249}$

Теперь две процедуры ***Sr*** и ***Ds*** готовы для включения в созданную библиотеку **UserLib**.

Этап 3. Для определения *пути* к библиотеке, принимающей ***Maple***-объекты, служит глобальная переменная ***savelibname,*** первоначально имеющая *неопределенное* значение. Для указания пути к библиотеке данной 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 имеем:

```
> savelibname;  savelibname:= "C:\\Program Files\\Maple 10\\UserLib":  savelibname;
                                    savelibname
                          "C:\Program Files\Maple 10\UserLib"
```

Этап 4. На этом этапе производится непосредственно сохранение в библиотеке подготовленных выше процедур. Делается это вызовом процедуры ***savelib,*** имеющей формат вызова следующего вида:

***savelib*(N1, N2, ...)**

где **Nj –** *имена* сохраняемых в библиотеке ***Maple***-объектов. Для нашего примера будет:

```
> savelib(Sr, Ds);
```

Процедура возвращает *NULL*-значение, т.е. ничего, и для проверки результата сохранения рекомендуется использовать еще один формат вызова ***march***-функции, а именно:

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 10\\UserLib");
    [["Ds.m", [2006, 10, 2, 10, 24, 18], 1090, 115], ["Sr.m", [2006, 10, 2, 10, 24, 18], 1024, 66]]
```

Возвращаемый *вложенный* список содержит по одному подсписку для каждого содержащегося в библиотеке ***m***-файла (*первый элемент в виде* ***"FileName.m"***), второй содержит список с датой и временем создания файла, тогда как третий и четвертый указывает *начальную позицию* данного файла (*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* подключение созданной библиотеки к основной библиотеке пакета, что позволит впредь обращаться к находящимся в ней объектам подобно стандартным средствам пакета ***Maple***. Для этих 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. Для нашего случая текущее состояние ***libname-***переменной есть:

```
> libname;   ⇒  "c:/program files/maple 10/lib/userlib", "C:\Program Files\Maple 10/lib"
```

определяющее, что пакет располагает логически сцепленной с *главной* ***Maple***-библиотекой "c:\program files/maple 10/lib" библиотекой "c:/program files/maple 10/lib/userlib", имеющей высший приоритет. Приоритет определяется *местоположением* библиотеки в ***libname***-цепочке – чем ближе она к началу, тем выше ее приоритет, определяющий порядок поиска библиотечных средств при их вызове: *поиск* производится, начиная в библиотеке с *самым* высоким приоритетом. Для подключения созданной нами библиотеки требуется переопределить значение ***libname***-переменной, а именно:

```
> restart:  libname:= libname, "C:\\Program Files\\Maple 10\\UserLib":  libname;
        "C:\Program Files\Maple 10/lib", "C:\Program Files\Maple 10\Lib/UserLib"
```

В результате переопределения наша библиотека **UserLib** получила наименьший приоритет. Именно так и следует поступать при создании новых библиотек пока не проведена их детальная апробация в совокупности с другими сцепленными библиотеками пакета. Теперь мы можем реально проверить доступность нашей библиотеки и находящихся в ней средств, а именно:

> 6*Sr(64, 59, 39, 10, 17, 44), 6*Ds(64, 59, 39, 10, 17, 44); $\Rightarrow$ $233, \sqrt{14249}$

Из примера следует, что *созданная* библиотека (*в рамках ее процедур* ***Sr*** *и* ***Ds***) функционирует корректно. Однако, здесь есть одно но. При переопределении ***libname****-переменной* мы предварительно выполнили **restart**-предложение, чтобы деактивировать в текущем сеансе ранее вычисленные определения процедур ***Sr*** и ***Ds*****,** сохраняемых в библиотеке. Именно такой прием позволяет нам убедиться, что после переопределения ***libname***-переменной мы будем иметь дело именно с библиотечными процедурами ***Sr*** и ***Ds***. Между тем, при такой организации мы должны будем каждый раз перед вызовом средств из **UserLib-**библиотеки выполнять вышеуказанное переопределение ***libname***-переменной, что, естественно, неудобно.

Во избежание этого рекомендуется поступить следующим *образом*. ***Maple*** допускает ряд инициализационных файлов (***ini****-файлов*), из которых файл ***"Maplen.ini"*** создается уже при инсталляции пакета в его **USERS**-подкаталоге, где **n –** *номер релиза* пакета. В этот же подкаталог рекомендуется *любым* доступным средством (*например*, *по* ***Notepad***) записать файл ***"Maple.ini"*** с единственной строкой следующего содержания:

libname := libname, "C:/Program Files/Maple 10/UserLib":

или *дополнить* файл этой строкой, если он уже существовал. Данный подход обеспечит вам автоматическое подключение вашей библиотеки к *главной* ***Maple***-библиотеке после каждой загрузки пакета. В результате реализации описанных этапов *создана* библиотека пользователя **UserLib,** средства которой становятся *доступными* наравне со стандартными при каждой загрузке пакета ***Maple***.

При этом, подкаталог **USERS** предполагается по *умолчанию*, но данная установка может быть пересмотрена при инсталляции. Следующая таблица определяет целесообразное расположение файла ***"Maple.ini"*** в каталогах пакета, где: **B – BIN, L – LIB** и **U – USERS,** {**+**|**+1**|**+2**} обозначает, что ***Maple*** использует этот файл с {*высшим*|*первым*|*вторым*} *уровнем* приоритета соответственно, тогда как при "**–**" ***Maple*** игнорирует файл ***"Maple.ini"***.

Релиз	B	L	U	B, L	B, U	L, U	B, L, U
6	**+**	**–**	**+**	**B**	**+1, +2**	**U**	**B, U**
7	**–**	**+**	**+**	**L**	**U**	**+1, +2**	**L, U**
8	**–**	**+**	**+**	**L**	**U**	**+1, +2**	**L, U**
9	**+**	**+**	**+**	**+2, +1**	**+, –**	**+1, +2**	**+2, +1, –**
10	**–**	**+**	**+**	**L**	**U**	**U**	**U**

Таким образом, *заштрихованный* **U-**столбец определяет каталог **USERS,** *файл* ***"maple.ini"*** которого используется пакетом всех релизов **6 - 10,** однако для **9**-го релиза данный файл игнорируется, если каталог **BIN.WIN** также содержит *подобный* файл. В столбцах **<B, L>, <B, U>** и **<L, U>** затенены клетки, которые определяют вышеупомянутые каталоги, целесообразные для того, чтобы определить местоположение файлов ***"Maple.ini"***для их приоритетного использования пакетом. Данные файлы могут содержать любую полезную информацию инициализации как определенного, так и общего характера, *включая* определения процедур, модулей либо их вызовов. В конкретном случае нашей *Библиотеки* [103,109] данный файл расположен в **U-**каталоге и используется для организации

логической связи с *главной* ***Maple***-библиотекой пакета. На основе файла ***"Maple.ini"*** имеется возможность поддерживать достаточно эффективные и простые механизмы связей пользовательских библиотек. Рекомендуется располагать данный инициализационный файл именно в каталоге **USERS** пакета.

Этап 6. Пополнение созданной библиотеки новыми средствами можно выполнять согласно *этапам* **3 – 4**, представленным выше. Однако для этих целей вполне подойдет и весьма простая процедура ***uplib,*** обеспечивающая *пополнение* библиотеки, заданной *полным* путем **L,** ***Maple***-объектами, имена **P** которых могут быть представлены как в единственном числе, так и их списком либо множеством. Перед вызовом процедуры ***uplib*****(L, P)** определения сохраняемых в **L**-библиотеке ***Maple***-объектов **P** должны быть предварительно вычислены. В следующем фрагменте приведен исходный текст процедуры ***uplib*** и пример ее применения для обновлений **UserLib-**библиотеки программным модулем **M.** Там же представлен пример и более общей процедуры ***UpLib*** [41,42,103,109]:

```
uplib := (L::{string, symbol}, P::{symbol, list(symbol), set(symbol)}) → op([
    assign('savelibname' = L), savelib(`if`(type(P, {'list', 'set'}), op(P), P)),
    unassign('savelibname')])
> M:= module() export x; x:= () -> `*`(args)/nargs end module:
> uplib("C:\\Program Files\\Maple 10\\UserLib", M);
> restart;  M:- x(64, 59, 39, 10, 17, 44), with(M);  x(64, 59, 39, 10, 17, 44),  eval(x);
                                  183589120, [x]
                                  183589120, ( ) → `*`(args)/nargs
> march('list', "C:\\Program Files\\Maple 10\\UserLib");
    [["Ds.m", [2006, 10, 2, 10, 24, 18], 1090, 115], ["Sr.m", [2006, 10, 2, 10, 24, 18], 1024, 66],
     ["M.m", [2006, 10, 2, 16, 15, 15], 1205, 70], [":-1.m", [2006, 10, 2, 16, 15, 15], 1275, 85]]
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, 'mlib') then h := cat("", L)
    elif type(t, '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`, 'procedure', 'table'}), 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("C:/Program Files/Maple 10/LIB/UserLib", [BootDrive]);
Warning, <BootDrive> does exist and will be updated
Warning, Library update has been done!
```

В данном фрагменте вычисляется определение процедуры ***uplib*** и программного модуля **M,** предназначенного для сохранения в существующей (*созданной на предыдущих этапах работы*) библиотеке **"c:\\program files\\Maple 10\\UserLib".** По вызову процедуры **uplib("C:\\Program Files\\Maple 10\\UserLib", M)** производится *сохранение* модуля **M** в указанной первым аргументом библиотеке. Успешный вызов процедуры ***uplib*** возвращает *NULL*-значение, т.е. *ничего*. Последующий *вызов* экспорта **x** модуля **M** после **restart**-предложения подтверждает корректность выполненной операции обновления ***UserLib***-библиотеки *модулем* **M**. При этом, предполагалось, что на этапе **5** созданная ***UserLib***-библиотека была логически сцеплена с *главной* ***Maple***-библиотекой (*через* ***libname***-*переменную*) посредством файла ***"Maple.ini"*** в **USERS**-каталоге пакета. Затем пример фрагмента представляет вывод *нового* состояния ***UserLib***-библиотеки, в котором кроме ***m***-файла с модулем **M** представлен и *сопутствующий* ему ***m***-файл "**:-1.m**"; с такого типа файлами библиотеки можно детально ознакомиться в наших книгах [41-43,103].

Тогда как вторая часть фрагмента представляет более развитую и универсальную процедуру ***UpLib.*** Успешный вызов процедуры ***UpLib*(F, N)** обновляет ***Maple***-библиотеку, заданную *первым* фактическим аргументом **F** процедурами, таблицами и/или модулями, чьи имена задаются *вторым* фактическим аргументом **N** *list*-типа). Вызов процедуры возвращает *NULL*-значение с выводом соответствующих сообщений. Процедура обрабатывает основные особые и ошибочные ситуации. Именно она используется нами при практическом обновлении ***Maple***-библиотек [41,42,103,109].

Сохранение ***Maple***-объектов можно производить и по вызову ***march***-функции формата:

march('add', <*Путь к библиотеке*>**,** <***m***-*файл*>**,** <*Имя объекта*>**)**

где третий аргумент определяет ***m***-файл с *сохраненным* в нем по **save**-предложению объектом. Например:

```
> Sr2:= () -> `+`(args)/nargs:   save(Sr2, "C:\\Program Files\\Maple 10/Sr2.m");
> march('add', "C:\\Program Files\\Maple 10\\UserLib", "Sr2.m", Sr2);
> march('list', "C:\\Program Files\\Maple 10\\UserLib");
> restart;  3*Sr2(64, 59, 39, 43, 10, 17);   ⇒ 116
```

Последующий вызов процедуры ***Sr2*** после **restart**-предложения подтверждает корректность выполненной операции обновления **UserLib-**библиотеки процедурой ***Sr2***. При этом, предполагается, что на этапе **5** созданная **UserLib-**библиотека была *логически* сцеплена с главной ***Maple***-библиотекой пакета.

По вызову функции **march('delete',** <*Путь к библиотеке*>**,** <*Имя объекта*>**)** производится удаление из библиотеки, определенной вторым аргументом, *объекта*, указанного третьим аргументом, как это иллюстрирует следующий фрагмент:

```
> march('delete', "C:\\Program Files\\Maple 10\\UserLib", Ds);
> march('list', "C:\\Program Files\\Maple 10\\UserLib");
      [["Sr.m", [2006, 10, 2, 10, 24, 18], 1024, 66], ["M.m", [2006, 10, 2, 16, 15, 15], 1205, 70],
                                  [":-1.m", [2006, 10, 2, 16, 15, 15], 1275, 85]]
```

Что и подтверждает последующая проверка содержимого библиотеки, отличного от состояния предыдущего фрагмента *именно* на удаленную процедуру ***Ds***. Объекты из библиотеки удаляются сразу же (*точнее информация о них соответствующим образом помечается в индексном файле* ***"Maple.ind"***), однако занимаемое ими место не освобождается в файле ***"Maple.lib"*** библиотеки. Для освобождения этого места (*уплотнения библиотеки*) можно использовать вызов функции **march('pack', <***Путь к библиотеке***>).** С другими же *операциями* с библиотеками, поддерживаемыми ***march***-функцией, можно ознакомиться в справке по пакету по **?march.** С учетом сказанного это не должно вызвать каких-либо затруднений. В книге [103] и приложенной к ней *Библиотеке* можно найти много полезных средств по поддержанию работы с библиотеками пользователя, включая процедуры восстановления поврежденных библиотек и работы с библиотеками, организационно отличными от ***Maple***-библиотек.

```
delres := proc(L::{mla, mlib}, N::symbol, t::{0, 1})
local a, b, c, d, f, W;
    `if`(Release( ) < 10, NULL, conmlib(L)), assign(d = interface(warnlevel)),
        interface(warnlevel = 3);
    W := (x, y) → ERROR("%1 deleted means with name of length %2 had been\
        ound; correct restoring is difficult enough",nops(x), y);
    if t = 0 then
        march('delete', L, N);
        WARNING("means <%1> has been deleted from library <%2>",N, L),
            interface(warnlevel = d)
    else
        if length(N) = 1 then assign(a = [1, 0, 109], f = cat(L, "\",
            `if`(3 < nargs and type(args[4], 'symbol'), args[4], 'maple'), ".ind"))
        elif length(N) = 2 then assign(a = [1, 0, 46, 109], f = cat(L, "\",
            `if`(3 < nargs and type(args[4], 'symbol'), args[4], 'maple'), ".ind"))
        else assign(a = [1, 0, op(convert("" || N[3 .. -1], 'bytes')), 46, 109], f = cat(
            L, "\", `if`(3 < nargs and type(args[4], 'symbol'), args[4], 'maple'),
            ".ind"))
        end if;
        assign(b = readbytes(f, ∞)), close(f), assign(c = sblist(a, b));
        if c = [ ] then
            interface(warnlevel = d);
            error "means with name <%1> does not exist or had been not delete\
                from library <%2>", N, L
        elif length(N) = 1 then
            if 1 < nops(c) then interface(warnlevel = d), W(c, 1)
            else writebytes(f, [op(b[1 .. c[1] − 1]), op(convert("" || N, 'bytes')), 46,
                op(b[c[1] + 2 .. -1])]), close(f)
            end if;
            WARNING(
```

```
                "access to means <%1> located in library <%2> had been restored",
                N, L), interface(warnlevel = d)
        elif length(N) = 2 then
            if 1 < nops(c) then interface(warnlevel = d), W(c, 2)
            else writebytes(f, [op(b[1 .. c[1] - 1]), op(convert("" || N, 'bytes')),
                op(b[c[1] + 2 .. -1])]), close(f)
            end if;
            WARNING(
                "access to means <%1> located in library <%2> had been restored",
                N, L), interface(warnlevel = d)
        else
            writebytes(f, [op(b[1 .. c[1] - 1]), op(convert("" || N[1 .. 2], 'bytes')),
                op(b[c[1] + 2 .. -1])]), close(f);
            WARNING(
                "access to means <%1> located in library <%2> had been restored",
                N, L), interface(warnlevel = d)
        end if
    end if
end proc
> delres("C:/temp/Userlib", agn, 0);
Warning, means <agn> has been deleted from library <C:/temp/Userlib>
> delres("C:/temp/Userlib", agn, 1);
Warning, access to means <agn> located in library <C:/temp/Userlib> had been restored
verdel := proc(L::{mlib, mla}, Tab::evaln)
local a, b, c, f, k, h, t, r, i, j;
    `if`(Release() < 10, NULL, conmlib(L)), assign(f = cat(L, "\",
        `if`(3 < nargs and type(args[4], 'symbol'), args[4], 'maple'), ".ind"));
    assign(h = readbytes(f, ∞), b = convert([7], 'bytes'), t = []), close(f);
    for k in [[1, 0, 109], [1, 0, 46, 109]] do
        assign('c' = sblist(k, h));
        if c ≠ [] then
            Tab[nops(k) - 2] := nops(c);
            t := [op(t), seq(cat(b $ (i = 1 .. nops(k) - 2)), j = 1 .. nops(c))];
            h := parse(Sub_st([cat(convert(k, 'string')[2 .. -2], ",") = ""],
                convert(h, 'string'), r)[1])
        end if
    end do;
    assign('c' = sblist([1, 0], h));
    if c ≠ [] then for k in c do
            a := searchL(h, k, [46, 109]);
            t := [op(t), cat(b, b, convert(h[k + 2 .. a - 1], 'bytes'))];
            Tcounter(Tab, a - k)
        end do
```

```
    end if;
    map(convert, t, 'symbol')
end proc
> map2(delres, "C:/Temp/UserLib", [MkDir, UpLib, Iddn1, helpman], 0);
Warning, means <MkDir> has been deleted from library <C:/Temp/UserLib>
Warning, means <UpLib> has been deleted from library <C:/Temp/UserLib>
Warning, means <Iddn1> has been deleted from library <C:/Temp/UserLib>
Warning, means <helpman> has been deleted from library <C:/Temp/UserLib>
> verdel("C:/Temp/UserLib", T);
                              [••Dir, ••Lib, ••dn1, ••lpman]
```

При работе с пользовательскими ***Maple***-библиотеками в ряде случаев возникает *необходимость* восстановления библиотечных средств в случае, когда для их удаления использовалась ***march***-функция. Естественно, восстановление можно выполнять и повторным сохранением удаленных средств. Между тем, в некоторых случаях требуется восстановить удаленное средство, но еще находящееся в файле **"maple.lib".** В этом случае может оказаться достаточно полезной процедура ***delres.***

Вызов процедуры ***delres*(L, N, t)** обеспечивает *удаление/восстановление* средства с именем **N,** расположенного в ***Maple***-библиотеке, заданной полным путем **L** к ней, в зависимости от значения третьего фактического **t**-аргумента, а именно: **0 –** *удаление*, **1 –** *восстановление*). Главное имя библиотечных файлов произвольно и четвертый необязательный аргумент определяет его, однако по умолчанию полагается главное имя ***"Maple".***

Успешное восстановление предполагает выполнение следующего условия: в промежутке между удалением средства и его последующим восстановлением посредством процедуры ***delres*** к обрабатываемой библиотеке не должна применяться функция ***march*** с опциями ***'reindex'*** и/или ***'pack'.*** Процедура ***delres*** имеет *два* ограничения, а именно она: (**1**) применима к пользовательским библиотекам только типов {*mlib*, *mlab*}, (**2**) обеспечивает корректное восстановление средств с именами длины 1 и 2 при наличие однократных удалений и любого числа удалений для имен длины большей двух. Успешный вызов ***delres*(L, N, 1)** восстановливает доступность к удаленному средству **N** библиотеки **L.**

Тогда как процедура ***verdel*(L, R)** возвращает список имен процедур, удаленных из библиотеки **L,** в рамках их последних **p - 2** символов, где **p –** длина имени. При этом, через второй **R**-аргумент возвращается таблица, чьи *входы* определяют длины имен *удаленных* процедур, а ее *выходы* – количества, соответствующие им. Успешный вызов процедуры предполагает выполнение того же условия, что и для предыдущей ***delres***-процедуры.

Этап 7. Создав ***Maple****-библиотеку* описанным выше способом и имея средства ее обновления, вы уже вполне можете использовать ее средства наравне с пакетными для программирования своих приложений и дальнейшего развития этой и других ей подобных библиотек. Однако для придания вашей библиотеке статуса *законченного* программного продукта весьма желательно снабдить 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 значительных огрехов*) *вполне* прилично организованной.

Прежде всего, нам требуется создать *саму* справочную базу библиотеки (*для нашего конкретного случая – библиотеки* **"C:\\Program Files\\Maple 10\\UserLib"**). Однако здесь ситуация несколько отлична от традиционной, а именно. *Справочные страницы* по средствам ***Maple***-библиотеки находятся в справочной *базе* в виде файла ***"Maple.hdb",*** располо-

женного, как правило, в том же каталоге, что и сама библиотека. Каждый такой файл базы данных содержит одну либо несколько страниц справки, а также служебную информацию, необходимую для обеспечения работы броузера пакета при работе со справочной базой как пакета, так и пользователя.

Справочная система ***Maple*** расположена в **GUI,** поэтому она не может непосредственно обрабатываться программными средствами пакета. Вместо этого обращение к справочной системе обеспечивается через функциональные *запросы* **INTERFACE_HELP(..)-**формата. Например, запрос **INTERFACE_HELP('display', topic=helpman)** выводит справочную страницу по процедуре ***helpman***. В общем случае запрос **INTERFACE_HELP** имеет следующий формат кодирования:

INTERFACE_HELP(<*Операция*>**, topic =** *Имя* {**, text = TEXT("***Строка*_1**", "***Строка*_2**", ...)**}
{**, library =** <*Библиотека*>}**);**

В качестве первого аргумента допускаются такие операции как ***display, insert*** и ***delete,*** смысл котрых особого пояснения не требует. Каждая справочная страница имеет следующие атрибуты:

topic – *имя*, под которым сохраняется справочная страница; имена разделов могут быть простыми, например, ***MkDir*** либо сложными (*многоуровневыми*). Для многоуровневого имени уровни разделяются запятыми, например, **`type,dir`**. Каждая справочная страница должна иметь *уникальное* имя раздела, которое для броузера является регистро-зависимым;

aliases – список альтернативных имен для раздела (*справочной страницы*);

text – *содержание* справочной страницы, сохраненной в формате ***Maple***-документа, должно помещаться в форме **TEXT("Строка_1", "Строка_2", ...)**. В качестве содержимого **TEXT** может выступать произвольный ***ASCII***-текст. Для помещения в *файл* ***"Maple.hdb"*** в качестве справочной страницы подготовленного ***mws***-файла следует использовать процедуру ***makehelp;***

parent – имя справочной страницы, которая будет загружена, когда запрошена *порождающая* ее страница. Как правило, это делается по умолчанию на основе имени раздела;

active – при определении ***true***-значения справочная страница загружается в качестве *активной* вместо стандартной справочной страницы.

Для операций *обновления* справочной базы должна использоваться ***library***-опция, определяющая путь к библиотеке с модифицируемой справочной базой ***"Maple.hdb"***; при этом, база главной ***Maple***-библиотеки защищена от модификации. Если же библиотека с модифицируемой справочной базой отражена в предопределенной ***libname***-переменной, то опцию достаточно закодировать в виде **library=libname[k],** где **к –** порядковый номер библиотеки в цепочке библиотек, отраженных в ***libname***-переменной. Детальнее с принципами работы с функцией **INTERFACE_HELP** можно ознакомиться по вызову **?help,update**.

С учетом сказанного, для создания *начальной* справочной базы для нашей **UserLib-**библиотеки используем следующий **INTERFACE_HELP**-вызов:

```
> INTERFACE_HELP('insert', topic="UserLib", text=TEXT("Help on my means in UserLib.
  Created 7.01.2007"), library="C:\\Program Files\\Maple 10\\UserLib");
```

В результате этого вызова в каталоге **"C:\\Program Files\\Maple 10\\UserLib"** библиотеки создается справочная *база* ***"Maple.hdb"*** с единственной справочной страницей (*разделом*) под именем ***UserLib*** с возвратом ***NULL***-значения. В качестве содержимого такой страницы можно, например, помещать общие сведения по библиотеке. При этом, следует иметь в виду, что допускается использование и кириллицы, однако здесь имеется

ряд недостатков, здесь не рассматриваемых. Однако, если ставится цель создания приложения, ориентированного на широкий рынок, то следует использовать латиницу.

Последующее наполнение справочной базы производится по мере создания, отладки и помещения в библиотеку новых объектов (*процедуры*, *модули*, *таблицы и др.*). Создав, отладив и апробировав то или иное программное средство, и посчитав целесообразным поместить его в библиотеку, весьма важно его документировать и в качестве такого документа (*инструкции по использованию*) и выступает подготовленный по нему ***Maple***-документ и помещаемый в справочную базу. Сделать это можно средствами **GUI** – создать в качестве *текущего* документа справку по требуемому средству (*рекомендуем взять за основу оформление страниц* (*разделов*) *справочной базы самого пакета; именно таким образом оформлена справочная база нашей библиотеки* [109]) и выполнить цепочку из двух команд **GUI <Help ⇒ Save to Database....>**. В результате открывается диалоговое окно **`Save Current Worksheet As Help`,** в котором достаточно выполнить следующие операции: (**1**) в **Topic**-поле поместить имя средства (*страницы/раздела*) и (**2**) в поле **`Writable Databases in libname`** щелчком мыши выбрать путь к искомой справочной базе (*результат выбора отражается в поле* **Database,** *расположенном выше*) и щелкнуть клавишей мыши по кнопке "**Save Current**". Результатом будет помещение текущего документа в справочную базу библиотеки под заданным в **Topic**-поле именем. Так как наша библиотека отражена в *предопределенной* ***libname***-переменной пакета, то и созданная для нее справочная база логически сцепляется с аналогичной базой пакета, обеспечивая принятую в пакете технологию работы со справочной информацией. На наш взгляд, стравочная система пакета организована достаточно эффективно и удобна для практического использования при работе с ***Maple*** и его приложениями.

Между тем, способ *обновления* (*дополнение/удаление*) справочной базы через **GUI** недостаточно надежен и в ряде случаев не дает результата, прежде всего при попытке обновления существующих справочных страниц. Поэтому, более надежным средством пополнения/обновления справочной *базы* библиотек является процедура ***makehelp,*** имеющая формат кодирования следующего вида:

makehelp(<*Раздел*>**,** <***mws****-файл*> {**,** <*Библиотека*>}**)**

где *раздел* определяет *имя* создаваемой *справочной* страницы, второй аргумент указывает *имя* или *полный путь* к ***mws****-файлу*, содержащему содержимое сохраняемой страницы, и *библиотека* определяет имя или полный путь к библиотеке, чья справочная база обновляется. Все фактические аргументы процедуры должны иметь *symbol*-тип. Имя раздела может быть простым, например, **`*имя*`** или сложным, например, **`*имя1/имя2*`**. При этом, третий аргумент необязателен, в этом случае указанный вторым аргументом файл выводится на экран в формате стандартной справочной страницы. Как правило, это делается для проверки созданной страницы перед сохранением ее в справочную базу. В любом случае, если вызов ***makehelp***-процедуры завершается точкой с запятой (**;**), то файл выводится на экран в формате стандартной справочной страницы. В качестве второго аргумента может указываться как ***mws***-файл (*наиболее удобно в плане оформления*), так и текстовый файл. В нашем конкретном случае вызов ***makehelp***-процедуры может иметь следующий вид:

```
> makehelp(savem1, `D:/Academy/UserLib6789/Common/HelpBase/savem.mws`,
  `C:\\Program Files\\Maple 10\\UserLib`);
```

по которому в справочную *базу* нашей **UserLib-**библиотеки помещается *справочная* страница под *именем* ***savem,*** создаваемая на основе одноименного ***mws***-файла, расположенного по *адресу*, указанному вторым фактическим аргументом. При этом, в случае завер-

шения вызова точкой с запятой (**;**), данный файл выводится на экран в формате *справочной* страницы.

Для автоматизации функций поддержки ведения *справочных баз* пользовательских библиотек, аналогичных главной ***Maple***-библиотеке пакета, нами была создана процедура ***helpman***. Процедура ***helpman*(R, L, U** {**, Z**}**)** имеет **3** обязательных и **1** необязательный аргументы. Первый аргумент **R** определяет режим работы со справочной базой библиотеки, полный путь к которой (*но не имя*) определен вторым фактическим **L** аргументом, а именно:

R = *insert* – *вставка* в справочную базу библиотеки **L** новых справочных *разделов* (*topics*), *имена* которых определены третьим фактическим **U** аргументом типа {*list*, *dir*}; если тип аргумента **U** – список, то четвертый аргумент **Z** определяет *список* путей к ***mws***-файлам со справочными разделами; при этом, между списками **U** и **Z** предполагается наличие взаимно-однозначного соответствия; если тип **U** аргумента – каталог, то вызов процедуры должен иметь только три фактических аргумента, где **U** аргумент определяет каталог с ***mws***-файлами со справочными разделами. Во втором случае процедура выбирает все ***mws***-файлы из указанного **U** каталога (*если они действительно существуют*) и на их основе формирует *разделы* в справочной базе библиотеки **L**. При этом, справочные разделы базы получают имена**,** определенные главными именами соответствующих ***mws***-файлов **U** каталога. Например, справочный раздел базы для средства '**XyZ**' содержится в файле ***"XyZ.mws"***. Процедура предполагает, что имена файлов со справочными разделами для имен объектов вида '**a/b**' должны кодироваться как **"a,b.mws"** при их создании. При этом, процедура поддерживает регистро-зависимый режим для *имен* справочных разделов и эти имена не должны содержать пробелов.

R = *delete* – *удаление* справочного *раздела* базы с *именем*, заданным третьим **U** аргументом типа {*symbol*, *string*}, из базы данных, определенной вторым аргументом **L.**

R = *display* – *вывод* на экран справочного раздела, чье имя определено третьим аргументом **U.**

При этом, следует иметь в виду следующие обстоятельства: (**1**) доступ к *главной* библиотеке пакета (*кроме режима* ***'display'***) запрещен, и (**2**) режимы ***'display'*** и ***'delete'*** предполагают только *один* справочный раздел; для первого режима такой подход является естественным, тогда как для второго он обеспечивает больший уровень безопасности. Процедура ***helpman*** обрабатывает *основные* ошибочные и особые ситуации, инициируя ошибки либо выводя соответствующие информационные сообщения.

Таким образом, процедура ***helpman*** представляет достаточно удобное средство для обновления справочной *базы* данных библиотек пользователя на основе заранее подготовленных ***mws***-файлов, обеспечивая возможность создания *одним вызовом* несколько справочных разделов, тогда как их *просмотр* и *удаление* производится *по одному* для каждого вызова процедуры. Кроме того, необходимо обратить внимание на одно важное обстоятельство. В некоторых случаях цепочкой функций ***"Help -> Save to Database"*** **GUI** пакета помещение в пользовательскую справочную базу данных раздела не гарантируется (*при регистрации его в соответствующем индексном файле* ***'Maple.ind'***), тогда как процедура ***helpman*** свободна от данного недостатка. Именно данное обстоятельство и послужило причиной создания ***helpman***-процедуры.

Ниже представлен исходный текст процедуры и примеры ее применения. Именно данной процедуре, зарекомендовавшей свои эксплуатационные качества, мы отдаем предпочтение при работу со справочными базами своих ***Maple***-библиотек. Во многих случаях она помогала решать задачу *обновления* справочных баз наших ***Maple***-библиотек там, где стандартные стрдства **GUI** пакета оказывались бессильными.

```
helpman := proc(
R::symbol, L::{string, symbol}, N::{string, symbol, dir, list({string, symbol})})
local k, a, b, c, d, f, g, h, p, n, m;
global libname;
    `if`( not member(R, {'display', 'delete', 'insert'}), ERROR(
        "the 1st argument should be {insert, display, delete}, but has received <%1>",
        R), `if`(Path(L) = Path(cat(CDM( ), "\LIB")) and R ≠ 'display',
        ERROR("access prohibition to the main Maple library"), `if`(
         not type(L, 'dir'), ERROR("path to library <%1> does not exist", L),
        assign(c = cat(L, "\maple.txt"), d = cat(L, "\maple.hdb")))));
    if not type(d, 'file') then
        WARNING(
            "Help database for library <%1> does not exist; it has been created", L);
        if N = [ ] then writeline(c, "Empty Help database for your library"), close(c),
            makehelp(Empty, ``||c, ``||L), fremove(c), RETURN(WARNING(
            "Empty Help database for library <%1> has been created", L))
        else writeline(c, ""), close(c), makehelp(Empty, ``||c, ``||L), fremove(c),
            procname(args)
        end if
    end if;
    if R = 'display' then
        if type(N, 'symbol') then return eval((proc()
                libname := libname, L;
                parse(cat("INTERFACE_HELP('display','topic'=",
                    convert(N, 'string1'), ")"))
            end proc )( ))
        else error
            "third argument should has type 'symbol', but has received type <%1>",
            whattype(N)
        end if
    end if;
    if R = 'delete' then
        if type(N, 'symbol') then return (proc(N, L)
                eval(parse(cat("INTERFACE_HELP('delete', 'topic'=",
                convert(N, 'string1'), ",'library'=", convert(L, 'string'), "")")))
            end proc )(N, L)
        else error "third argument %1 is invalid", [N]
        end if
    end if;
    `if`(R = 'insert' and nargs = 4 and type(N, 'list'({'symbol', 'string'})) and
        type(args[4], 'list'({'symbol', 'string'})), `if`(nops(N) = nops(args[4]),
        assign('b' = 14), ERROR(
        "mismatching of quantities of names and mws-files: %1<>%2", nops(N),
        nops(args[4]))), NULL);
    if b = 14 then
```

```
        for k to nops(N) do
            try makehelp(convert(N[k], 'symbol'), convert(args[4][k], 'symbol'),
                convert(L, 'symbol'))
            catch "file or directory does not exist":
                WARNING("file <%1> does not exist, the operation with it has\
                    been ignored", args[4][k]);
                next
            catch "file or directory, %1, does not exist":
                WARNING("file <%1> does not exist, the operation with it has\
                    been ignored", args[4][k]);
                next
            end try
        end do;
        RETURN(WARNING("Work has been done!"))
    elif R = 'insert' and type(N, 'dir') then
        [assign(n = [ ], m = [ ]), writeto(f), Dir(N), writeto('terminal')];
        do
            h := readline(f);
            if h = 0 then fremove(f); break
            else
                h := SLD(h, " ");
                `if`(cat("aaa", h[-1])[-4 .. -1] = ".mws",
                    assign('p' = [op(p), h[-1]]), NULL)
            end if
        end do
    end if;
    if p ≠ [ ] then
        for k to nops(p) do
            n := [op(n), p[k][1 .. -5]]; m := [op(m), cat(Path(N), "\", p[k])]
        end do;
        RETURN(procname(R, L, n, m))
    else RETURN(WARNING("mws-files have not been saved in Help database"))
    end if;
    WARNING("Work has not been done, mistakes at the call encoding: %1",
        'procname(args)')
end proc
```

> helpman(insert, "C:/program files/maple 10/LIB/userlib", ["DF1"], ["C:/temp/df1.mws"]);

вставка справочного раздела по процедуре ***DF1*** в справочную базу библиотеки ***UserLib***

Warning, Work has been done!

> helpman(insert, "C:/Program Files/maple 10/LIB/UserLib", ["AFdes", `a,b`],
["C:\\Temp\\AFdes.mws", "C:\\Temp/RANS\\IAN\\a,b.mws"]);

Warning, file <C:/Temp/RANS\IAN/a,b.mws> does not exist, the operation with it has been ignored

Warning, Work has been done!

> helpman(insert, "C:/program files/maple 10/LIB/userlib", "C:/temp"); **#** *вставка* справочного раздела в справочную базу библиотеки ***UserLib*** на основе ***mws***-файлов каталога **"C:\\Temp"**

Warning, Work has been done!

> helpman(display, "C:/program files/maple 10/LIB/userlib", "AFdes"); **#** *вывод* справочного раздела по процедуре ***AFdes***

> helpman(delete, libname[1], ArtKr); **#** *удаление* справочного раздела ***ArtKr*** из справочной базы библиотеки, путь к которой находится в начале цепочки библиотек ***libname***-переменной

Таким образом, нами представлены довольно простые пути создания и основных функций ведения пользовательских библиотек, аналогичных *главной* ***Maple***-библиотеке. Для резюмирования рассмотренного материала создадим на его *основе* простую процедуру, обеспечивающую базовые функции ведения пользовательских ***Maple***-библиотек.

Процедура ***ulibrary*** имеет два формальных аргумента, но допускает *любое* число дополнительных в зависимости от значения первого **O**-аргумента, получающего значения из диапазона **1..10** и определяющего тип запроса на обработку библиотеки, определенной *именем* или полным путем к ней **L**. Аргумент **O** определяет следующие типы обработки библиотеки **L**:

1 – *создание новой* библиотеки, определенной **L**-аргументом; может определяться *именем* или полным путем, во втором случае допускается *любой уровень* вложенности каталогов

2 – *сохранение* в библиотеке **L** объектов с именами, определяемыми фактическими аргументами, начиная с **3**-го; неопределенное имя в библиотеке не сохраняется

3 – *удаление* из библиотеки **L** объектов с именами, определяемыми фактическими аргументами, начиная с **3**-го

4 – *упаковка* библиотеки **L** после удаления из нее ненужных объектов

5 – *возврат* списка объектов, содержащихся в библиотеке **L**

6 – *создание* пустой справочной базы для библиотеки **L**

7 – *обновление* справочной базы библиотеки **L** *новой* страницей (*разделом*); все аргументы при вызове процедуры должны иметь *symbol*-тип; обновление производится на основе ***mws***-файла, находящегося по адресу, указанному четвертым фактическим аргументом, тогда как третий фактический аргумент определяет *имя* сохраняемой страницы

8 – *удаление* из справочной *базы* библиотеки **L** страницы (*раздела*), указанной третьим аргументом; этот аргумент при вызове процедуры должен иметь *symbol*-тип

9 – *вывод* на экран *справки* по странице (*разделу*) справочной базы библиотеки **L** с *именем*, определяемым *третьим* фактическим аргументом; при этом, производится логическое *сцепление* библиотеки **L** с *главной* ***Maple***-библиотекой с *наименьшим* приоритетом *первой*

10 – *создание* логического сцепления библиотеки **L** с *главной* ***Maple***-библиотекой; данное сцепление действует до переопределения ***libname***-переменной, до **restart**-предложения либо до перезагрузки пакета. Вызов процедуры возвращает текущее состояние ***libname***-переменной пакета.

Ниже представлен *исходный* текст процедуры и примеры ее применения на все случаи. Процедура поддерживает *десять* вышеперечисленных операций с библиотеками пользователя, подобными главной ***Maple***-библиотеке пакета. Кроме операций **8** и **9** (*удаление страницы из базы и вывод страницы на экран*) вызов процедуры на *других* допустимых значениях первого аргумента наряду с выполнением соответствующих *операций* с библио-

текой выводит соответствующие предупреждения. Процедура ***ulibrary*** обрабатывает основные особые и ошибочные ситуации.

```
ulibrary := proc(O::{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, L::{string, symbol})
local a, b, c, mkdir1;
global libname;
    mkdir1 := proc(L::{string, symbol})
        local a, b, c;
            assign(a = "", b = ""), [seq(`if`(c = "\" or c = "/",
                assign('a' = cat(a, "","")), assign('a' = cat(a, c))), c = cat("", L))];
            for c in parse(cat("[", a, "]")) do
                b := cat(b, c, "\");
                try mkdir(b)
                catch "directory exists and is not empty": next
                catch "file I/O error": next
                catch "permission denied": next
                end try
            end do
        end proc;
    if O = 1 then
        mkdir1(L);
        march('create', L, `if`(nargs = 3 and type(args[3], 'posint'), args[3], 65));
        WARNING("library <%1> has been created", L)
    elif O = 2 then
        assign('savelibname' = L), savelib(
            `if`(nargs = 2, ERROR("no names specified to save"), args[3 .. -1]));
        WARNING("names %1 have been saved in library <%2>", [args[3 .. -1]], L)
    elif O = 3 then
        march('delete', L,
            `if`(nargs = 2, ERROR("no names specified to delete"), args[3 .. -1]));
        WARNING("names %1 have been deleted out of library <%2>",
            [args[3 .. -1]], L)
    elif O = 4 then
        march('pack', L); WARNING("library <%1> has been packed", L)
    elif O = 5 then WARNING("content of library <%1> is:", L); march('list', L)
    elif O = 6 then
        WARNING("Help database for library <%1> has been created", L);
        INTERFACE_HELP('insert', topic = "UserLib", text = TEXT("The help\
            on my means located in UserLib. The library has been created 5.10.2\
            006"), library = L)
    elif O = 7 then
        makehelp(`if`(nargs = 4 and map(type, {args[3 .. 4]}, 'symbol') = {true},
            args[3 .. 4], ERROR("3rd and 4th arguments should have symbol-ty
```

```
            pe and represent topic name and mws-file accordingly"), L);
        WARNING(
            "topic <%1> had been added to the help database of library <%2>",
            args[3], L)
    elif O = 8 then INTERFACE_HELP('delete', topic = `if`(
        nargs = 3 and type(args[3], 'symbol'), args[3],
        ERROR("topic specified to delete is missing")), library = L)
    elif O = 9 then
        libname := libname, L;
        INTERFACE_HELP('display', topic = `if`(
            nargs = 3 and type(args[3], 'symbol'), args[3],
            ERROR("topic specified to display is missing")))
    elif O = 10 then
        libname := libname, L;
        WARNING(
            "library <%1> has been logically connected with main Maple-library",L)
            ;
        libname
    end if
end proc
```

```
> P:= () -> `+`(args)/nargs: P1:= () -> `*`(args): M:= module() export x; x:= () -> `+`(args) end
  module:
> ulibrary(1, `C:/AVZ/AGN\\VSV/Art\\Kr`);
Warning, library <C:/AVZ/AGN\VSV/Art\Kr> has been created
> ulibrary(2, `C:/AVZ/AGN\\VSV/Art\\Kr`);
Error, (in ulibrary) no names specified to save
> ulibrary(2, `C:/AVZ/AGN\\VSV/Art\\Kr`, P, P1, M, Sv);
Warning, names [P, P1, M, Sv] have been saved in library <C:/AVZ/AGN\VSV/Art\Kr>
> ulibrary(5, `C:/AVZ/AGN\\VSV/Art\\Kr`);
Warning, content of library <C:/AVZ/AGN\VSV/Art\Kr> is:
  [[":-1.m", %1, 1215, 75], ["P.m", %1, 1024, 65], ["P1.m", %1, 1089, 56], ["M.m", %1, 1145, 70]]
                                    %1 := [2006, 10, 5, 10, 0, 44]
> ulibrary(3, `C:/AVZ/AGN\\VSV/Art\\Kr`, P, AVZ);
Warning, member "AVZ" not found in archive, skipping
Warning, names [P, AVZ] have been deleted out of library <C:/AVZ/AGN\VSV/Art\Kr>
> ulibrary(4, `C:/AVZ/AGN\\VSV/Art\\Kr`);
Warning, library <C:/AVZ/AGN\VSV/Art\Kr> has been packed
> ulibrary(5, `C:/AVZ/AGN\\VSV/Art\\Kr`);
Warning, content of library <C:/AVZ/AGN\VSV/Art\Kr> is:
      [[":-1.m", [2006, 10, 5, 10, 0, 44], 1024, 75], ["P1.m", [2006, 10, 5, 10, 0, 44], 1099, 56],
                          ["M.m", [2006, 10, 5, 10, 0, 44], 1155, 70]]
> ulibrary(6, "C:/AVZ/AGN\\VSV/Art\\Kr");
Warning, Help database for library <C:/AVZ/AGN\VSV/Art\Kr> has been created
> ulibrary(7, `C:/AVZ/AGN\\VSV/Art\\Kr`, MKDIR,
`D:/Academy/UserLib6789/Common/HelpBase/MkDir.mws`);
```

```
Warning, topic <MKDIR> had been added to the help database of library
<C:/AVZ/AGN\VSV/Art\Kr>
> ulibrary(9, `C:/AVZ/AGN\\VSV/Art\\Kr`, P1);
Error, Could not find any help on "P1"
> ulibrary(9, `C:/AVZ/AGN\\VSV/Art\\Kr`, MKDIR);  # Вывод справки по MKDIR на
  экран
> ulibrary(8, `C:/AVZ/AGN\\VSV/Art\\Kr`, MKDIR);
> ulibrary(9, `C:/AVZ/AGN\\VSV/Art\\Kr`, MKDIR);
Error, Could not find any help on "MKDIR"
> ulibrary(10, `C:/AVZ/AGN\\VSV/Art\\Kr`);
Warning, library <C:/AVZ/AGN\VSV/Art\Kr> has been logically connected with main
Maple-library
        "c:/program files/maple 8/lib/userlib", "C:\Program Files\Maple 8/lib",
                                "C:/AVZ/AGN\VSV/Art\Kr"
```

Между тем, процедура не обеспечена развитой системой обработки ошибочных ситуаций, что требует от пользователя внимательности при кодировании фактических аргументов при ее вызове. Сделано это было в целях *упрощения* алгоритма процедуры и его большей прозрачности в учебных целях. В то же время данная процедура представляет достаточно простое и эффективное средство при выполнении *базовых операций* с ***Maple***-библиотеками пакета. Рекомендуется обратить *внимание* на подпроцедуру ***mkdir1***, обеспечивающую создание цепочки каталогов любого уровня вложенности. Она намного проще нашей стандартной (*используемой процедурами нашей библиотеки* [103]) процедуры ***MkDir***, однако не снабжена развитой системой обработки ошибочных ситуаций.

Достаточно полезной оказывается и процедура, чей вызов ***Plib*(L)** возвращает полный путь к ***Maple***-библиотеке, заданной своим именем либо путем **L** к ней, и которая логически связана с главной библиотекой пакета. При отсутствии такой библиотеки либо логической *связи* для нее возвращается *false*-значение. Нижеследующий фрагмент представляет исходный текст процедуры и пример ее применения.

```
Plib := proc(L::{string, symbol})
local a, b, c, k, h, p, ω;
    assign(a = [libname], b = cat("/", Case(L, 'lower')),
    ω = (( ) → Case(sub_1("\" = "/", args))), c = cat("\", Case(L, 'lower'))), seq(`if`(
    ω(a[k]) = ω(L) or Search1(Case(a[k], 'lower'), b, 'h') and h = ['right'] or
    Search1(Case(a[k], 'lower'), c, 'p') and p = ['right'], RETURN(a[k]), NULL),
    k = 1 .. nops(a)), false
end proc
> Plib(userlib), Plib(UserLib), Plib("C:\\RANS/Academy/ModProcLib"), Plib(Svetlana);
      "c:/program files/maple 9/lib/userlib", "c:/program files/maple 9/lib/userlib",
                          "c:/rans/academy/modproclib", false
```

Для поддержки разнообразных процедур работы с ***Maple***-библиотеками нами создан целый ряд полезных средств, представленных в книгах [41,42,43,103] и в прилагаемых к ним библиотекам программных средств для пакета ***Maple*** *релизов* **6 – 10**. Рассмотрим теперь еще несколько *способов* организации пользовательских библиотек, которые в ряде случаев оказываются даже эффективнее стандартного *подхода*, поддерживаемого ***Maple.***

8.2. *Специальные способы создания пользовательских библиотек в среде Maple*

Прежде всего, созданные и отлаженные процедуры и программные модули можно сохранять в текстовых файлах во *входном* формате ***Maple*-языка**. В данном случае они впоследствии читаются **read**-преложением, корректно загружая в текущий сеанс *определения* как процедур, так и программных модулей (*стандартные средства пакета для модулей не могут обеспечить их корректного сохранения в* ***m****-файлах внутреннего* ***Maple****-формата*). Сохранение процедур и модулей производится в результате *выполнения* **save-**предложения следующего формата кодирования**:**

save N1, N2, …, Nk, <СФ> либо **save(N1, N2, …, Nk,** <СФ>**)**

где фактические аргументы **Nj** определяют идентификаторы сохраняемых процедур и/или модулей, а СФ **-** спецификатор принимающего файла (*имя либо полный путь к файлу*). Если имя файла завершается символами "**.m**", то принимающий файл СФ будет во внутреннем ***Maple***-формате, который не позволяет корректно сохранять программные модули [12-14,41-43,103,109]. Поэтому, совместное сохранение процедур и модулей следует делать в файле входного ***Maple***-формата, для чего имя принимающего файла достаточно кодировать без завершающих его символов "**.m**". Загрузка сохраненных процедур и модулей производится по предложению **read** <СФ> {**read(**<СФ>**)**}**,** в результате чего определения процедур и модулей, находящихся в файле СФ, вычисляются и они становятся доступными текущему сеансу пакета. Следующий весьма простой фрагмент иллюстрирует вышесказанное**:**

```
> G:= () -> `+`(args)/nargs:  S:= module() export Sr; Sr:= () -> `+`(args)/nargs end module:
> save(G, S, "C:/Temp/File.lib");
> restart;  read "C:/Temp/File.lib":  5*G(42, 47, 67, 85, 96), with(S), 5*S:- Sr(42, 47, 67, 85, 96);
                                  337, [Sr], 337
```

Суть фрагмента сводится к следующему. Определяются простая **G-**процедура, возвращающая сумму значений передаваемых ей фактических аргументов, и программный модуль **S** с единственным экспортом ***Sr*,** и производится их вычисление с последующим сохранением по **save**-предложению в файле *входного* ***Maple***-формата. По **restart-**предложению восстанавливается исходное состояние текущего сеанса. После этого по предложению **read** производится загрузка из файла **G-**процедуры и **S**-модуля с последующим вызовом процедуры и ***Sr***-экспорта модуля **S,** завершившиеся вполне корректно.

Таким образом, по **save**-предложению можно создавать файлы *входного* ***Maple*-языка**, содержащие корректные *определения* сохраненных в них объектов (*процедуры*, *модули и др*.). В случае же внутреннего ***Maple***-формата (***m****-файлы*) программные модули сохраняются некорректно, без их *тела*. Данная проблематика детально рассмотрена в наших книгах [13,14,41-43,103] и там же представлен целый ряд средств по устранению данной ситуации. В частности, в целях *устранения* данного недостатка нами была создана процедура ***SaveMP*,** обеспечивающая для пакета релизов **6-10** корректное выполнение данной операции наряду с другими. Процедура и примеры ее применения представлены ниже.

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 средств производится в режиме *дописывания* (***APPEND***) с выводом соответствующего сообщения. Тогда как вызов ***SaveMP*(F, "list")** обеспечивает возврат имен содержащихся в файле **F** средств с индикацией их типов, как показывает пример ниже.

```
SaveMP := proc(F::string, M::{set, list, string, set(`module`), list(`module`)})
local d, k, n, m, h, p, t, v, G, z, u, S, Kr, Art, q, r, f, ω, g;
    `if`( not type(F,'file') and args[2] = "list" or
          not type(F,'file') and nargs = 3 and args[3] = "load",
        ERROR("file <%1> does not exist",F),
        `if`(type(F,'file'), assign(f = F), assign(f = pathtf(F))));
    if args[2] = "list" or nargs = 3 and args[3] = "load" then
        assign(u = [ ], z = [ ], S = "");
        do
            assign('h' = iolib(2,f));
            if h = 0 then break
            else `if`(h = "#AVZ_AGN_2003;", [assign('h' = readline(f)),
                assign('n' = searchtext(" := ",h)), `if`(h[n+4 .. n+9] = "proc (",
                assign('u' = [op(u), convert(h[1 .. n-1],'symbol')]), `if`(
                h[n+4 .. n+12] = "module ()",
                assign('z' = [op(z), convert(h[1 .. n-1],'symbol')]), NULL))], 1)
            end if
        end do;
        g := cat([libname][1][1 .. 2], "\_$Art16_Kr9$_");
        if args[2] = "list" then RETURN(fclose(f), `if`(z = [ ] and u = [ ],
            WARNING("file <%1> does not contain a software",f), `if`(u = [ ],
            op(['modules', z]),
            `if`(z = [ ], op(['procedures', u]), op(['procedures', u, 'modules', z])))))
        else
            assign('q' = {op(M)} minus ({op(z), op(u)} intersect {op(M)})), `if`(
                q = { }, NULL,
                WARNING("tools %1 do not exist in file <%2>",q,f));
            assign('z' = {op(z), op(u)} minus {op(M)});
            seq(assign('S' = cat(S, z[k], ":=", z[k], ":")), k = 1 .. nops(z));
            (proc(a, b)
                    read f; writebytes(a, b); fclose(a); read a; delf(a)
                end proc)(g, S), assign('r' = {op(M)} minus q),
                RETURN(WARNING("tools %1 have been activated",r))
        end if
    else Art := proc(S)
        local a, b, c, p;
            [assign('a' = S, 'b' = Search(S, ":= module")),
            `if`(nops(b) = 1, RETURN(S), assign('b' = b[2 .. -1])), seq([
            assign('c' = searchtext("()", a, p .. length(a)) + p), `if`(
            a[p+8 .. c-1] = " ", ``,
            assign('a' = cat(a[1 .. p+8], a[c-1 .. length(a)])))], p = b), a][2]
        end proc
    end if;
    `if`(f[-2 .. -1] = ".m", assign67(ω = f[1 .. -3], delf(f)), assign(ω = f)),
        fopen(ω, 'APPEND');
```

```
    for v to nops(M) do
        assign('n' = length( cat( "unprotect('", M[v], "'); ") ), 't' = length( M[v] ),
            'm' = length( cat( "'", M[v], ":-") ),
            'h' = length( cat( " protect('", M[v], "');") ) + 2 );
        ( proc(x) save x, g end proc )( M[v] ), assign('G' = readbytes( g, 'TEXT', ∞ ) );
        `if`( type( M[v], `module` ), [ `if`( search( G, cat( "unprotect('", M[v], "'); ") ),
            assign('G' = G[ n + 1 .. –h ] ), G ),
            assign('z' = Search( G, cat( "'", M[v], ":-") ) ), `if`( z ≠ [ ], [
            assign('Kr' = G[ 1 .. z[ 1 ] – 1 ] ), seq(
            assign('Kr' = cat( Kr, G[ z[ k ] + m .. z[ k + 1 ] – 1 ] ) ), k = 1 .. nops( z ) – 1 )
            , assign('Kr' = cat( Kr, G[ z[ -1 ] + m .. length( G ) ] ) ) ], assign('Kr' = G ) ),
            assign('d' = Search( Kr, cat( " ", M[v], " ") ) ),
            `if`( d = [ ], assign('d' = "a" ), assign('d' = op( d ) ) ) ], assign('Kr' = G ) ),
            delf( g ), writeline( ω, "#AVZ_AGN_2003;", `if`(
            whattype( eval( M[v] ) ) ≠ `module`, Kr,
            `if`( d ≠ "a", Art( cat( Kr[ 1 .. d – 1 ], Kr[ d + t + 1 .. -1 ] ) ), Kr ) ) ), `if`(
            not ( v = nops( M ) ), 0, RETURN(
            WARNING( "tools %1 have been saved in file <%2>", M, ω ), close( ω ) ) )
    end do
end proc

> f1:=() -> `+`(args)/nargs: f2:=()->add(args[k]^2,k=1..nargs)/nargs: f3:=()->[`+`(args), nargs]:
  f4:=()->`*`(args)/nargs^2: M1:=module() export Sr; option package;  Sr:=()->`+`(args)/nargs
  end module: module M2() export Kr; option package; Kr:=()-> `*`(args)/nargs end module:
> SaveMP("C:\\Archive/Tallinn\\Book/RANS_IAN.m", [f1, f2, M1, f3, M2, f4]);
Warning, tools [f1, f2, M1, f3, M2, f4] have been saved in file
<c:\archive\tallinn\book\rans_ian>
> SaveMP("C:\\Archive/Tallinn\\Book/RANS_IAN", "list");
                        procedures, [f1, f2, f3, f4], modules, [M1, M2]
> restart;  SaveMP("C:\\Archive/Tallinn\\Book/RANS_IAN", [f1, Art_Kr, M1], "load");
Warning, tools {Art_Kr} do not exist in file <C:\Archive/Tallinn\Book/RANS_IAN>
Warning, tools {f1, M1} have been activated
```

Наконец, вызов процедуры ***SaveMP*(F, M, "load")** обеспечивает загрузку в текущий сеанс средств файла **F,** чьи имена определены списком/множеством **M.** Детальнее с процедурой можно ознакомиться в [41,103]. Данная процедура успешно использовалась для создания простых и эффективных библиотек (*архивного уровня*) пользователя.

Сохраняя процедуры и программные модули по **save**-предложению в файлах входного ***Maple***-формата, мы, тем самым, создаем своего рода их простейшие библиотеки, средства которых загружаются в текущий сеанс по **read**-предложению и сразу же становятся доступными *подобно* стандартным средствам пакета. Недостатком такой библиотечной организации (*наряду с некоторыми другими*) является то, что каждое *новое обновление* входящего в такую библиотеку средства требует обновления всей библиотеки. Поэтому в рамках подобного подхода можно предложить процедуру ***simplel,*** полезую в целом ряде приложений. Вызов процедуры ***simplel*(L {, N1, N2, …, Nk})** допускает один или более фактических аргументов, где *первый* **L**-аргумент определяет полный путь к файлу *входного* ***Maple***-формата с сохраняемыми или сохраненными средствам (*процедурами и/или*

программными модулями). Вызов процедуры ***simplel*(L)** возвращает список (*определяющий содержимое файла, созданного ранее* ***simplel****-процедурой*) следующего формата:

[<*Д1*>, [*P1*, *Proc*], [*M1*, *Mod*], [*P2*, *Proc*], [*M2*, *Mod*], …, [<*Д2*>, [*PP1*, *Proc*], [*MM1*, *Mod*], [*PP2*, *Proc*], [*MM2*, *Mod*], …]

где **Дj** – определяют дату *сохранения* следующих за ней программных средств, тогда как следующие за ней (*до следующей даты, если таковая имеется*) **2**-элементные *подсписки* определяют *имена* и *типы* сохраненных в файле **L** программных средств: *Proc – процедура* и *Mod – программный модуль*. При попытке загрузить *файл*, не созданный ***simplel***-процедурой, инициируется ошибочная ситуация с возвратом соответствующей диагностики.

Успешный вызов процедуры ***simplel*(L,N1,N2,…,Nk)** возвращает ***NULL***-значение, сохраняя в файле **L** объекты (*процедуры и/или программные модули*), *имена* которых определены фактическими аргументами, начиная со *второго*. При этом, предварительно *определения* сохраняемых объектов должны быть *вычислены*, в противном случае они *не сохраняются*.

```
simplel := proc(L::{symbol, string})
local a, b, c, f, mkf;
    mkf := proc(f::{symbol, string})
        local a, b, c, d, cc;
            cc := proc(d::{symbol, string})
                local a, b, c, k;
                    assign(a = [ ], b = "" || d, c = 1), `if`(
                        member(b[-1], {"\", "/"}), assign('b' = b[1 .. -2]), NULL)
                        ;
                    for k to length(b) do
                        if member(b[k], {"\", "/"}) then
                            a := [op(a), b[c .. k - 1]]; c := k + 1
                        end if
                    end do;
                    [op(a), b[c .. -1]]
                end proc;
            assign(b = "" ||f), `if`(length(b) ≤ 3 or b[2] ≠ ":", ERROR("argume\
                nt should be by full path to datafile, but had received <%1>",f),
                [assign('b' = cc(f)[1 .. -2]), assign(c = b[1])]);
            try assign('d', fopen(f, 'READ'), close(f)), "" ||f
            catch "file or directory does not exist":
                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;
                assign('d', fopen(f, 'WRITE'), close(f)), "" ||f
            end try
        end proc;
```

```
    if nargs = 1 and "" || L[-2 .. -1] = ".m" then error "1st argument should define\
        a datafile of the input Maple-format, but had received <%1>",L
    elif nargs ≠ 1 and "" || L[-2 .. -1] = ".m" then
        f := "" || L[1 .. -3];
        WARNING("target datafile had been redefined as file <%1> of the input\
            Maple-format", f)
    else f := L
    end if;
    if nargs = 1 then
        try open(f, 'READ'), close(f), assign(a = [ ], c = 17)
        catch "file or directory does not exist": error "library <%1> does not exist", f
        end try;
        if readline(f) ≠ "`Software database simplel`:" then
            close(f);
            error "datafile <%1> had been not created by procedure simplel", f
        end if;
        do
            c := readline(f);
            if c = 0 then close(f); break
            elif c[1] = """ then a := [op(a), c[2 .. -3]]
            else
                search(c, ":=", 'b');
                if type(b, 'symbol') then next
                else
                    a := [op(a), [ `` || c[1 .. b - 1],
                        `if`(c[b + 3 .. b + 6] = "proc", 'Proc', 'Mod')]];
                    unassign('b')
                end if
            end if
        end do;
        a
    else
        fopen(mkf(f), 'APPEND'), writeline(f, "`Software database simplel`:"),
            writeline(f, """ || [ssystem("date /T")][1][2][1 .. -3] || """ || ":");
        seq(`if`(type(args[k], {`module`, 'procedure'}),
            writeline(f, "" || args[k] || ":= " || (convert(eval(args[k]), 'string')) || ":"),
            NULL), k = 2 .. nargs);
        close(f)
    end if
end proc
```

```
> M:=module() export Sr; Sr:= () -> `+`(args)/nargs end module: P:= () -> `*`(args)/`+`(args):
> simplel("D:/AVZ\\AGN/VSV\\Art/Kr/library", P, M);
> simplel("D:/AVZ\\AGN/VSV\\Art/Kr/library", MkDir, came);
```

```
> simplel("D:/AVZ\\AGN/VSV\\Art/Kr/library");
                ["07.10.2006", [P, Proc], [M, Mod], "07.10.2006", [MkDir, Proc], [came, Proc]]
> restart;  read("D:/AVZ\\AGN/VSV\\Art/Kr/library");
> eval(P), eval(M), P(42, 47, 67, 89, 96), M:- Sr(64, 59, 39, 10, 17, 44);
                ( ) → `*`(args)/`+`(args), module() export Sr;  end module, 1130012352/341, 233/6
> simplel("D:/AVZ\\AGN/VSV\\Art/Kr/library.m");
Error, (in simplel) 1st argument should  define a datafile of the input Maple-format, but had
received <D:/AVZ\AGN/VSV\Art/Kr/library.m>
> M1:=module() export Sr; Sr:=() -> `+`(args)/nargs end module: P1:=() -> `*`(args)/`+`(args):
> simplel("D:/AVZ\\AGN/VSV\\Art/Kr/library.m", M1, P1);
Warning, target datafile had been redefined as file <D:/AVZ\AGN/VSV\Art/Kr/library>
of the input Maple-format
> simplel("D:/AVZ\\AGN/VSV\\Art/Kr/library");
        ["07.10.2006", [P, Proc], [M, Mod], "07.10.2006", [MkDir, Proc], [came, Proc],
                                "07.10.2006", [M1, Mod], [P1, Proc]]
> simplel("C:/Temp/avz.cmd");
Error, (in simplel) datafile <C:/Temp/avz.cmd> had been not created by procedure simplel
```

Сохранение объектов производится в режиме дописывания (***APPEND***), позволяя сохранять (*архивировать*) все версии средств с индикацией дат их сохранения. При этом, при последующей загрузке файла **L** активируются в текущем сеансе только *последние* сохраненные версии объектов. Процедура позволяет создавать библиотечный файл **L** в цепочке каталогов *любого* уровня вложенности, что обеспечивает ее подпроцедура ***mkf***. Процедура ***simplel*** обрабатывает основные особые и ошибочные ситуации.

Вышеприведенный фрагмент содержит *исходный* текст и примеры применения процедуры ***simplel***. Представленная процедура может оказаться полезным средством при организации простых библиотек пользователя, несущих и архивные черты программных средств. Данная процедура может быть расширена *новыми* функциональными возможностями, которые оставляем читателю в качестве достаточно полезного упражнения. С более общими средствами поддержки ведения простых библиотек пользователя, отличных от ***Maple***-библиотек, можно познакомиться в наших книгах [13,14,41-43,103,108] и в прилагаемых к ним библиотеках.

Для организации простых библиотек пользователя можно 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 формате:

'parse("module <*Имя***> () export ... end module")'**

где *Имя* определяет имя программного модуля. После этого созданная таким образом таблица сохраняется посредством **save**-предложения в файле любого допустимого формата (*внутреннем либо входном*). Последующие загрузки данного файла по **read**-предложению активируют в текущем сеансе сохраненные в нем программные средства, *доступ* к которым аналогичен доступу к средствам пакетного модуля. При этом, если обращение к *процедуре* обеспечивается стандартным для таблиц способом, т.е. по конструкции следующего формата кодирования

<*Имя таблицы***>[<***Имя процедуры***>](…)**

то к *программному модулю* по конструкции формата

<*Имя модуля*> **:-** <*Имя экспорта*>**(…)**

только после вызова процедуры **with(**<*Имя таблицы*>**)**. Нижеследующий *весьма* простой фрагмент хорошо иллюстрирует вышесказанное.

```
> T:= table([P = (() -> `+`(args)/nargs), M = 'parse("module M () export Sr;
  Sr:= () -> `+`(args)/nargs end module")', P1 = proc() `*`(args)/`+`(args) end proc,
  M1 = 'parse("module M1 () export X; X:= () -> `*`(args)/nargs end module")']);   #  Maple 8
```

$$T := \text{table}\left([P1 = (\textbf{proc}()\ \text{`*`}(\text{args})/\text{`+`}(\text{args})\ \textbf{end proc}),\ P = \left(( ) \to \frac{\text{`+`}(\text{args})}{\text{nargs}}\right),\right.$$

```
    M1 = parse( "module M1 () export X; X:= () -> `*`(args)/nargs end module"),
    M = parse( "module M () export Sr;
      Sr:= () -> `+`(args)/nargs end module")
    ])
> save(T, "C:/Temp/library.m");  save(T, "C:/Temp/library");
> restart;  read "C:/Temp/library.m";  with(T);    ⇒ [M, M1, P, P1]
> 3*T[P](64, 59, 39, 10, 17, 43),  29*T[P1](64, 59, 39, 10, 17, 43),  3*M:- Sr(64, 59, 39, 10, 17, 43),
  M1:- X(64, 59, 39, 10, 17, 43);    ⇒ 116, 134562480, 116, 179416640
> restart;  read "C:/Temp/library";  with(T);    ⇒ [M, M1, P, P1]
> 3*T[P](64, 59, 39, 10, 17, 43),  29*T[P1](64, 59, 39, 10, 17, 43),  3*M:- Sr(64, 59, 39, 10, 17, 43),
  M1:- X(64, 59, 39, 10, 17, 43);    ⇒ 116, 134562480, 116, 179416640
```

При этом, в релизах ***Maple***, начиная с **9**-го, *вызов* процедуры **with** на таблицах вышеописанной организации вызывает ошибочную диагностику (*приведенную ниже*) на *первом* же модуле, погруженном в таблицу, однако *все последующие* вычисления выполняются корректно. Поэтому, при необходимости, такие *ошибочные* ситуации *легко* обрабатываются программно посредством **try**-предложения.

```
> restart;  read "C:/Temp/library.m";
> with(T);
Error, (in M) attempting to assign to `M` which is protected
> 3*T[P](64, 59, 39, 10, 17, 43),  29*T[P1](64, 59, 39, 10, 17, 43),  3*M:- Sr(64, 59, 39, 10, 17, 43),
  M1:- X(64, 59, 39, 10, 17, 43);    ⇒ 116, 134562480, 116, 179416640
```

Как следует из приведенного фрагмента, обращение к средствам сохраненной табличной **T**-структуры аналогично принятому в ***Maple*** для пакетных модулей, имеющих табличную организацию и представляет еще один способ сохранения программных модулей в файлах внутреннего ***Maple***-формата. В ряде случает представленный прием оказывается довольно полезным при программировании приложений в среде пакета.

```
SoftTab := proc(T::symbol, F::{symbol, string}, R::symbol)
local a, b, s, k;
    b := ( ) → ERROR(
        "procedure call does not contain procedures and/or modules for saving - %1",
        [args[3 .. -1]]);
    if 2 < nargs then
        for k from 3 to nargs do
            if type(args[k], 'procedure') then assign('T'[args[k]] = eval(args[k]))
            elif type(args[k], `module`) then
                s := convert(eval(args[k]), 'string');
```

```
                assign('T'[args[k]] = parse(
                    cat("'parse(", """, cat(s[1 .. 7], args[k], s[7 .. -1]), """, ")'")))
            else a := 17
            end if
        end do;
        if a ≠ 17 then save T, F else b(args) end if
    else b(args)
    end if
end proc
```

```
> Proc:= () -> `*`(args)/nargs:  Mod:= module () export Sr; Sr:= () -> `+`(args)/nargs end
  module: Proc1:= () -> `+`(args):  Mod1:= module () export avz;  avz:= () -> `+`(args)^2 end
  module:  SoftTab(Tab, "C:/Temp/library", Proc, Mod, Proc1, Mod1);
> SoftTab(Tab, "C:/Temp/library.m", Proc, Mod, Proc1, Mod1);
> restart;  read "C:/Temp/library":  with(Tab);   ⇒ [Mod, Mod1, Proc, Proc1]
> Tab[Proc](64,59,39,10,17,43), 3*Mod:- Sr(64,59,39,10,17,43), Tab[Proc1](64,59,39,10,17,43),
  Mod1:- avz(64, 59, 39, 10, 17, 43);   ⇒ 179416640, 116, 232, 53824
> restart;  read "C:/Temp/library.m";  with(Tab);   ⇒ [Mod, Mod1, Proc, Proc1]
> Tab[Proc](64,59,39,10,17,43), 3*Mod:- Sr(64,59,39,10,17,43), Tab[Proc1](64,59,39,10,17,43),
  Mod1:- avz(64, 59, 39, 10, 17, 43);   ⇒ 179416640, 116, 232, 53824
> read "C:/Temp/library.m";  with(Tab);   ⇒ 179416640, 116, 232, 53824
Warning, the protected names Mod and Mod1 have been redefined and unprotected
> Tab[Proc](64,59,39,10,17,43), 3*Mod:- Sr(64,59,39,10,17,43), Tab[Proc1](64,59,39,10,17,43),
  Mod1:- avz(64, 59, 39, 10, 17, 43);   ⇒ 179416640, 116, 232, 53824
> SoftTab(Tab, "C:/Temp/library", A, V, Z);
Error, (in b) procedure call does not contain procedures and/or modules for saving - [A,V,Z]
> SoftTab(Tab, "C:/Temp/library");
Error, (in b) procedure call does not contain procedures and/or modules for saving - []
```

В свете представленного выше подхода к организации пользовательских библиотек на основе табличной организации может оказаться довольно полезной процедура ***SoftTab,*** исходный текст которой и примеры применения представлены *предыдущим* примером.

Процедура ***SoftTab*(T,F,R)** имеет не менее трех аргументов, из которых *первый* аргумент имеет *symbol*-тип и определяет *имя* сохраняемой таблицы **T,** тогда как второй определяет принимающий файл **F,** имеющий *входной* ***Maple***-формат или *внутренний* ***Maple***-формат. Вызов процедуры ***SoftTab*(T, F, X, Y, Z, …)** возвращает *NULL*-значение, обеспечивая сохранение в файле **F** таблицы **T,** содержащей определения процедур и/или модулей, имена которых определены фактическими аргументами, *начиная* с третьего. Таблица **T** в качестве *входов* имеет имена сохраненных процедур и модулей, тогда как *выходы* – их определения. При этом, если определения *процедур* представляются в **T**-таблице непосредственно, то определения *модулей* представляются в специальном формате, который обеспечивает последующее корректное их использование. Вызов процедуры с *двумя* аргументами либо с аргументами **X, Y, Z, …,** чей тип отличен от {*procedure*, `*module*`} вызывает ошибочную ситуацию с возвратом соответствующей диагностики.

Файл **F,** созданный вызовом процедуры ***SoftTab*(T,F, X, Y, Z,…),** читается **read**-предложением, активируя в текущем сеансе сохраненные в нем *объекты* **X,Y,Z,…**. При этом, обращения к данным объектам производятся принятыми в ***Maple*** способами, а именно:

1) к *процедурам* по конструкции **T[**<*Имя*>**](...)**
2) к программным *модулям* по конструкции <*Имя*>**:-** <*Экспорт*>**(...)** после вызова **with(T)**

Еще на *одном* моменте следует акцентировать внимание. Сохранение **T**-таблицы следует выполнять отдельно для каждого *нового* **F**-файла, разделяя их **restart**-предложениями во избежание возникновения ошибочных ситуаций с диагностикой следующего вида "Error, (in assign/internal) invalid left hand side in assignment". Представленные в предыдущем фрагменте примеры весьма наглядно иллюстрируют сказанное. В качестве весьма полезного упражнения читателю рекомендуется рассмотреть организацию ***SoftTab***-процедуры и расширить ее возможностью сохранять файл с таблицей по произвольному адресу подобно тому, как это реализовано в предыдущей процедуре ***simplel***. Такая возможность *весьма* существенна при программировании задач, имеющих дело с доступом к файлам данных различных типа и организации.

С рядом других средств ведения простых библиотек пользователя (*т.н. первый уровень*), организационно отличных от ***Maple***-библиотек пакета, можно познакомиться в книгах [8-14,41,42,45, 46,103]. В них достаточно детально описана сущность алгоритмов, реализованных средствами, которые оказываются достаточно полезными при работе с библиотеками пользователя нестандартной организации, а также с *файлами*, содержащими определения ***Maple***-объектов, прежде всего процедур и программных модулей.

Ко *второму* уровню средств работы с библиотеками пользователя можно отнести набор наших процедур, обеспечивающих создание, обновление, просмотр библиотек, подобных ***Maple***-библиотеке пакета, *основной* из которых является процедура ***User_pflMH,*** наряду с перечисленными обеспечивающая три режима создания *справочной базы* библиотек. Данную процедуру можно рассматривать в качестве наиболее общего и универсального инструмента создания пользовательских библиотек, структурно подобных главной библиотеке пакета, которые логически связаны с ней. В общем, процедура выполняет следующие основные операции [41,103,109]:

***** *регистрация* в системном *файле* ***"Win.ini"*** текущего релиза пакета, если ранее этого не было сделано;

***** *создание* либо *обновление* инициализационного *файла* ***"Maple.ini"*** с целью обеспечения логической связи создаваемой библиотеки пользователя с главной ***Maple***-библиотекой (*при этом, последняя операция обеспечивает возможность создания справочной базы библиотеки согласно пользовательскому предложению*).

Процедура выводит соответствующие *сообщения* о проделываемой работе. Таким образом, при создании или обновлении библиотеки пользователь должен определить только ее имя (*если библиотека будет расположена 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 для вновь создаваемой библиотеки, ее справочную базу данных и режим логической связи библиотеки с главной библиотекой пакета. Ряд других процедур [41,103,109] обеспечивают наиболее массовые операции с ***Maple***-подобными библиотеками, включая средства восстановления *поврежденных* библиотек, обновления библиотек и их справочных баз, отмены/восстановления логической связи с главной библиотекой пакета, эффективной упаковки, конвертации библиотек первого уровня организации во второй. В этом смысле они в определенном отношении поддерживают продвинутые функциональные возможности автоматизации работы с библиотеками подобно случаю известной утилиты ***sed*** операционной системы ***UNIX*** (*Linux*).

Наконец, *третий* уровень поддерживается стандартной процедурой ***savelib,*** обеспечивающей помещение таблиц, программных модулей и процедур, а в *более* общем случае определений многих других вычисленных объектов, в разделяемую *главную* библиотеку пакета. Это дает возможность впоследствии использовать сохраненные объекты на

логическом уровне доступа также, как встроенные средства пакета. Реализация данного механизма сохранения обеспечена выполнением в текущем сеансе нескольких шагов, которые достаточно подробно рассмотрены в наших книгах [14,29-33,39,42-46,103]. Для обеспечения обновления главной ***Maple***-библиотеки таблицами, процедурами и программными модулями предназначены две довольно полезные процедуры ***MapleLib*** и ***UpLib.*** Данные процедуры позволяют также обновлять библиотеки пользователя, аналогичные главной библиотеке пакета. В частности, именно последняя библиотека используется нами наиболее активно для обновления ***Maple***-библиотек пакета. Она рассмотрена выше, тогда как процедура ***MapleLib*** представлена ниже.

```
MapleLib := proc(P::{set(symbol), list(symbol)})
local a, b, c, h, r, p, k, l, i;
    assign(r = Release('h'), p = {  }, b = {  }), `if`(nargs = 1,
        assign(l = cat(h, "\lib\maple.lib"), i = cat(h, "\lib\maple.ind")), `if`(
        type(args[2], 'mlib'),
        assign(l = cat(args[2], "\maple.lib"), i = cat(args[2], "\maple.ind")),
        ERROR("library <%1> does not exist", args[2]))),
        assign(a = `if`(nargs = 1, cat(h, "\lib"), args[2]));
    `if`(map(type, {l, i}, 'file') = {true}, NULL,
        ERROR("library <%1> does not exist or is damaged", a));
    seq(assign('p' = {op(p), `if`(type(P[k], {`module`, 'procedure', 'table'}), P[k],
        assign('b' = {op(b), k}))}), k = 1 .. nops(P)), `if`(p = {  },
        ERROR("1st argument does not contain procedures, tables or modules"), `if`(
        nops(p) < nops(P),
        WARNING("arguments with numbers %1 are invalid", b), NULL));
    map(F_atr1, [l, i], [  ]), assign(c = interface(warnlevel), 'savelibname' = a),
        null(interface(warnlevel = 0));
    try savelib(op(p))
    catch "unable to save %1 in %2": AtrRW(savelibname); savelib(op(p))
    finally null(interface(warnlevel = c)), AtrRW(savelibname)
    end try;
    WARNING("tools <%1> have been saved/updated in <%2>", p, savelibname),
        unassign('savelibname')
end proc
```

```
> Art:= proc() `+`(args)/nargs end proc:  Kr:= module ()  export mean;  option package;
  mean:= () -> `+`(args)/nargs  end module:  MapleLib([Art, Kr]);
Warning, library <C:\Program Files\Maple 9\lib> has received the `READONLY`-attribute
Warning, tools <{Art, Kr}> have been saved/updated in <C:\Program Files\Maple 9\lib>
```

Успешный вызов процедуры ***MapleLib*(P)** обновляет главную ***Maple***-библиотеку процедурами, таблицами и/или модулями, *имена* которых определены первым фактическим **P** аргументом (*множество или список имен*). Если был закодирован второй дополнительный **L** аргумент, то он определяет полный путь к пользовательской библиотеке, аналогичной главной библиотеке. Процедура всегда выводит соответствующее сообщение о проделанной работе. Процедура обрабатывает основные ошибочные ситуации, связанные с отсутствием либо повреждением обрабатываемой библиотеки, или с отсутствием сохраняемых объектов. При возникновении подобных ситуаций возвращается соответствующая диагностика. Процедура обеспечивает вышеупомянутые функции для биб-

лиотек указанного типа для пакета релизов **6 - 10**. Более того, библиотеки в ***Maple*** релизов **6** и **7** получают ***readonly***-атрибут (*в концепции* ***MS DOS***) после обновления, тогда как библиотеки в ***Maple*** релизов **8 – 10** после обновления получают ***READONLY-***атрибут (*в концепции* ***Maple;*** процедура ***AtrRW***). Процедура ***AtrRW*** представлена ниже.

```
AtrRW := proc(L::path)
local a, r, h, p, l, i, ω;
    assign( r = Release( ), l = cat( L, "\maple.lib"), i = cat( L, "\maple.ind") );
    `if`( map( type, { l, i }, 'file') = { true }, NULL,
        ERROR( "library <%1> does not exist or is damaged", L ) );
    ω := x → WARNING( "library <%1> has received the %2-attribute", L, x );
    if member( r, { 6, 7 } ) then
        assign( a = map( F_atr1, [ l, i ] ) ), map( F_atr1, [ l, i ], [ ] );
        if not member( map2( member, "R", { op( a ) } ), { { true }, { true, false } } )
        then null( map( F_atr1, [ l, i ], [ "+R" ] ) ), ω( "`readonly`" )
        else ω( "`writable`" )
        end if
    else
        try filepos( l, 265 ), assign( p = ( l → filepos( l, 265 ) ) ), p( l )
        catch "file I/O error": map( F_atr1, [ l, i ], [ ] ), filepos( l, 265 ),
            assign( p = ( l → filepos( l, 265 ) ) ), p( l )
        catch "permission denied":
            map( F_atr1, [ l, i ], [ ] );
            filepos( l, 265 ), assign( p = ( l → filepos( l, 265 ) ) ), p( l )
        end try;
        if readbytes( l ) = [ 2 ] then
            null( p( l ), writebytes( l, [ 1 ] ) ), close( l ), ω( "`READONLY`" )
        else null( p( l ), writebytes( l, [ 2 ] ) ), close( l ), ω( "`WRITABLE`" )
        end if
    end if
end proc
> AtrRW("C:/program files/maple 7/lib");
Warning, library <C:/program files/maple 7/lib> has received the `readonly`-attribute
> AtrRW("C:/program files/maple 9/lib");
Warning, library <C:/program files/maple 9/lib> has received the `READONLY`-attribute
```

Для пакета ***Maple*** релизов **8, 9** и **10** вызов процедуры ***AtrRW*(L)** по принципу *«переключателя»* изменяет атрибут ***READONLY*** или ***WRITABLE*** (*в* ***Maple****-концепции*) на *противоположный* для ***Maple***-библиотеки**,** *полный* путь к которой определяется аргументом **L.** Тогда как для ***Maple*** релизов **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.** Успешный вызов процедуры возвращает *NULL*-значение с выводом сообщения о проделанной работе. С целым рядом других полезных средств для работы с библиотеками пользователя как аналогичными главной ***Maple***-библиотеке, так и нестандартными можно ознакомиться в [109].

8.3. *Создание пакетных модулей пользователя*

С каждым новым релизом пакета в нем появляются новые *пакетные модули*, средства которых ориентированы на тот или иной круг приложений. Получать перечень *пакетных* модулей, имеющихся в *текущем* релизе пакета ***Maple,*** оперативно по запросу вида ***?index, package***. Во многих изданиях и технической документации по пакету ***Maple*** такие 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 [12,41-43,103]. В этом смысле *программные* и *пакетные* модули в общем случае являются *разными* объектами. *Модульная* организация обеспечивает целый ряд преимуществ, детально не обсуждаемых здесь. В частности, она позволяет вести *независимую* разработку с ориентацией на конкретный круг приложений, допускает использование *одноименных* со стандартными средств. Имеются и другие немаловажные преимущества.

```
M_Type := proc(M::symbol)
local a, T, ω, ν, t, k;
    assign( a = { anames( procedure ) }, ω = interface( warnlevel ) ), `if`( M = liesymm,
        RETURN('Tab'), assign( ν = ( 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":
        try ν( 0 ); t := with( M )
        catch "%1 is not a package":
            ν( ω ); RETURN( WARNING( "<%1> is not a module", M ) )
        catch "invalid input: %1":
            ν( ω );
            RETURN(
                WARNING( "<%1> is not a module or can't be initialised", M ) )
        catch "system level initialisation for":
            ν( ω );
            RETURN( WARNING( "module <%1> cannot 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;
    ν( ω ), 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 ) ) )
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, GaussInt,Groebner, LREtools, PDEtools, Ore_algebra, algcurves, orthopoly, combinat, combstruct, difforms, geometry, inttrans, networks, numapprox, numtheory, plottools, powseries, sumtools, ListTools, RandomTools, RealDomain, SolveTools, StringTools, XMLTools, SumTools, TypeTools, Worksheet, OrthogonalSeries, FileTools, RationalNormalForms, ScientificConstants, 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*]

> Mulel(%); ⇒ [[*Proc*, 35, 1], [*Tab*, 13, 16], [*Mod*, 1, 50]]

Организационно пакетные модули в настоящее время бывают трех типов – *процедуры*, *программные модули* (*как правило, второго типа*) и *таблицы*. С помощью нашей процедуры ***M_Type*** [103,109] можно проверять тип пакетного модуля; ее применение в среде пакета ***Maple 10*** дает результат, представленный предыдущим фрагментом.

Таким образом, ***Maple 10*** 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*, поэтому останавливаться на этом не имеет смысла. Напомним лишь, что при создании пакетного модуля *процедурного* либо *модульного* типа в его разделе **option** необходимо кодировать опцию ***package,*** что позволит после сохранения его в ***Maple***-библиотеке в последующем обращаться к содержащимся в нем средствам принятыми в ***Maple*** способами.

Здесь же мы рассмотрим лишь создание пакетного модуля *табличного* типа, как первого наиболее массового типа модулей в среде *ранних* релизов ***Maple.*** Как можно будет впоследствии заметить, именно основные методы доступа к пакетным модулям определяются форматом, используемым при обращении к табличным объектам. Общая схема создания пакетного модуля табличного весьма проста и состоит в следующем. На первом этапе производятся тщательные тестирование, апробация и отладка процедур, предназначенных к включению в создаваемый *пакетный* модуль. Затем готовые процедуры погружаются в *табличную* структуру, в качестве *входов* которой являются *имена процедур* и *выходов* – их *определения*, т.е. создается таблица следующего вида:

***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. При этом, в качестве *выходов* таблицы наряду с *определениями* процедур могут выступать также вызовы процедур, функций или произвольные ***Maple***-выражения. На *третьем* заключительном этапе таблица сохраняется описанным выше способом в ***Maple***-библиотеке. Если библиотека логически связана с главной библиотекой пакета, то сохраненная в ней таблица и будет вашим пакетным модулем. Рассмотрим пример создания простого пакетного модуля *табличного* типа.

```
> restart;  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);
 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), Art = ( ( ) → `*`(args) / nargs^2 ),
 SV = (proc()
 description "a Maple procedure automatically generated by define()'
       ...
 end proc)
 ])
> TabMod[SV](64, 59, 39, 10, 17, 44), TabMod[Sr](64, 59, 39, 10, 17, 44);  ⇒ 1398, 38.8333
> TabMod[Ds](64, 59, 39, 10, 17, 44);
        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](64, 59, 39, 10, 17, 44);  ⇒ [Art, Ds, SV, Sr], 19.8948
> savelibname:= "C:/Program Files/Maple 10/LIB/UserLib": savelib(TabMod); restart;
> libname, with(TabMod);
 "c:/program files/maple 10/lib/userlib", "C:\Program Files\Maple 10/lib", [Art, Ds, SV, Sr]
> seq(F(42, 47, 67, 89, 96), F = [Art, SV, Ds, Sr]); restart;
                                  1130012352/25, 1705, 21.6462, 68.2000
> seq(F(42, 47, 67, 89, 96), F = [Art, SV, Ds, Sr]);
          Art(42,47,67,89,96), SV(42,47,67,89,96), Ds(42,47,67,89,96), Sr(42,47,67,89,96)
```

Первые три примера фрагмента иллюстрируют поэтапное определение ***TabMod***-таблицы, в которой входами являются имена процедур и выходами – их определения. При этом, следует отметить, т.к. ***define****-определение* функции в случае успешного завершения возвращает *NULL*-значение, то при необходимости включения определенной таким образом функции в таблицу следует использовать конструкцию, представленную *модульной* ***SV***-функцией фрагмента, т.е. имеющей в общем случае следующий формат:

Id_table[*Id_function*] := **define(***Id_f1*, *Id_f1*(...) = F(...)**)**, ***eval*(*Id_f1*)** {**;**|**:**}

При этом, идентификатор *Id_f1* может совпадать с идентификатором *Id_function*. Вызов функции **eval(TabMod)** возвращает содержимое таблицы **TabMod,** тогда как функции **print(TabMod)** – содержимое таблицы выводится на печать.

В следующем примере производится *вызов* погруженных в таблицу процедур ***SV*** и ***Sr*** на конкретном кортеже фактических аргументов с возвратом соответствующих значений, тогда как аналогичный вызов процедуры ***Ds*** возвращает результат *невычисленным*, если не считать весьма очевидных упрощений, обусловленных, прежде всего, использованием ***evalf***-функции. Объясняется это тем, что в такого типа *табличных* объектах не производится полного вычисления их входов в том смысле, что при использовании одним из выходов какого-либо входа таблицы либо иного вызова, он рассматривает данный вход неопределенным. Для устранения такой ситуации следует перед вызовами входов таблицы использовать вызов процедуры **with**.

Так, 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 же пакетный *модуль* ***Id*** находится в библиотечном файле, путь к которому определен в переменной ***libname*** пакета**,** то по вызову **with(Id)** производится его загрузка его в *рабочую область пакета* (**РОП**), обеспечивая доступ ко всем поддерживаемым им функциональным средствам. Тогда как по конструкциям формата:

*Id_Модуля***[***Id_Функции***]** и *Id_Модуля***[***Id_Функции***](***Фактические аргументы***)**

можно загружать в **РОП** только конкретную модульную функцию и вычислять 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. Однако, и в данном случае требуется выполнение приведенного выше условия – предварительный вызов **with**-процедуры.

Так, из *двух* последних примеров фрагмента, иллюстрирующих сказанное, хорошо видно, что после сохранения таблицы ***TabMod*** в ***Maple***-библиотеке, логически связанной с главной библиотекой пакета, выполнение вызова **with(TabMod)** делает полностью доступными все процедуры, определенные в пакетном модуле **TabMod**. С другой стороны, без вызова ***with*(*TabMod*)** процедуры пакетного модуля остаются неопределенными.

При создании пакетных модулей следует иметь в виду, что модульная таблица, обеспечивающая доступ к функциональным средствам модуля, имеет одно из двух основных представлений (*в разрезе вход – выход*), а именно:

Id-функции **= readlib(**`*Id-модуля/alias-функции*`**)** и *Id-функции* **= (***процедура/функция***)**

При этом, если в первом случае выход таблицы модуля определяет вызов соответствующего функционального средства, то во втором *выход* непосредственно определяет функцию, процедуру либо произвольное ***Maple***-выражение.

В первом случае имеет место полезное эквивалентное соотношение вида:

with(*Id-модуля***)[<***Функция***>](***Args***) ≡ readlib(**`*Id-модуля/alias-функции*`**)(***Args***)**

для конкретного вызова необходимой модульной функции, определяемой ее алиасом или именем, где *Args* - передаваемые в точке вызова фактические аргументы. Тогда как во втором случае допустимо использование только **with**-процедуры. В *общем* случае вызовы **with**-процедуры имеют два формата кодирования, а именно:

with(<*Module***>)** и **with(<***Module***>, P1, P2, …)**

В первом случае возвращается список экспортируемых пакетным модулем *Module* имен средств с активацией их в текущем сеансе (*загрузка в* **РОП**), тогда как во втором случае возвращается список экспортируемых пакетным модулем *Module* имен [***P1, P2, …***] с активацией их в текущем сеансе. В частности, по конструкции **op(with(***Module***, P1)(***Args***))** возвращается вызов пакетной функции/процедуры на заданных фактических *Args*-аргументах. Кстати, по вызову процедуры ***packages*()** возвращается упорядоченный *список* имен всех пакетных модулей, хоть один экспорт которых был активирован в текущем сеансе пакета.

Процедура ***ListPack*** [103] позволяет получать список *пакетных* модулей, содержащихся в главной ***Maple***-библиотеке пакета или в библиотеке, подобной главной библиотеке. Процедура обрабатывает все основные ошибочные и особые ситуации. Вызов процедуры ***ListPack*()** без фактических аргументов предполагает, что *пакетные* модули разыскиваются в *главной* ***Maple-***библиотеке, тогда как вызов процедуры ***ListPack*(F)** обеспечивает поиск в ***Maple***-библиотеке, имя которой или полный путь к ней определены факти-

ческим 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 и констатирует процедура. Следующий фрагмент представляет исходный текст процедуры и примеры ее применения.

```
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' ) 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') th
                n _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;
            fremove( 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, ∞ ) ), 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' = {op(v), cat(``, k[1 .. t - 1])}), `if`(search(k, "[", 't'),
        assign('v' = {op(v), cat(``, k[1 .. t - 1])}, '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

{*finance, Slode, PolynomialTools, tensor, plots, simplex, DEtools, group, student, polytools, Units, numapprox, geom3d, codegen, stats, diffalg, RealDomain, algcurves, numtheory, ScientificConstants, geometry, ListTools, combinat, Maplets, plottools, LinearOperators, PDEtools, inttrans, ExternalCalling, Ore_algebra, RandomTools, CodeGeneration, LargeExpressions, Groebner, VariationalCalculus, process, difforms, LibraryTools, CurveFitting, orthopoly, RationalNormalForms, linalg, powseries, LinearFunctionalSystems, LinearAlgebra, VectorCalculus, sumtools, genfunc, queue, GaussInt, Matlab, TypeTools, OrthogonalSeries, StringTools, Student, RootFinding, Spread, Sockets, Domains, liesymm, networks, SoftwareMetrics, padic, SolveTools, MatrixPolynomialAlgebra, XMLTools, SNAP, LREtools, priqueue, Worksheet, context, combstruct, MathML, SumTools*},
{*Elements, Tools, Examples, Standard, Converter, Calculus1, Natural, Hypergeometric, stats*[*fit*], *stats*[*anova*], *stats*[*describe*], *stats*[*statplots*]}

> ListPack("C:\\Program Files\\Maple 8\\Lib\\UserLib");

{ *SoLists*, *DIRAX*, *SimpleStat*, *ACC*, *AlgLists*}

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/процелур, как это иллюстрирует следующий простой пример:

> Kr[Sv]:= 3: Kr[S]:= proc() 'procname(args)' end proc: Kr[W]:= readlib(ifactors):
> with(Kr), Sv, S(42, 47, 67, 62, 89, 96), ifactors(2006);

[S, Sv, W], 3, S(42,47,67,62,89,96), [1, [[2, 1], [17, 1], [59, 1]]]

Это обстоятельство можно достаточно эффективно использовать при 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:

> 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)(64, 59, 39))/op(with(T, y)(64, 59, 39)) else
** op(with(T, z)(64, 59, 39)) end if;** $\Rightarrow$ 54

Наряду с отмеченными имеется еще целый ряд *весьма* интересных применений табличных структур ***Maple***-языка, полезных для практического программирования в среде пакета [8-14,41,103]. Мы же в качестве примера представим один простой и в ряде случаев полезный тип пакетных *модулей*. Предположим, что определения **4**-х процедур ***f1***, ***f2***, ***f3***, ***f4*** помещены в массив (*array*), который сохраняется в ***Maple***-библиотеке **Svetlana**.

Для обеспечения доступа к такого типа *нестандартным* пакетным модулям создается весьма простая процедура ***ArtMod*(L, f** {**, *args***}**),** первый формальный аргумент **L** которой определяет имя пакетного модуля, второй – имя **f** содержащейся в нем процедуры; тогда как остальные аргументы необязательны, определяя передаваемые **f**-процедуре фак-

тические аргументы ***args*.** Если в качестве второго фактического аргумента указано **`?`**-значение, то возвращается список имен процедур, содержащихся в **L**-модуле; при этом, все они становятся *активными* в текущем сеансе. Таким образом, процедуру ***ArtMod*** можно рассматривать в качестве некоторого аналога процедуры **with** пакета. Нижеследующий фрагмент представляет исходный текст процедуры, создание пакетного модуля **Art** и иллюстрирует вызовы процедур из модуля **Art** на конкретных фактических аргументах.

```
> ArtMod:=proc (L::symbol, f::symbol) local a,k; [`if`(type(L,'list'), assign('a'=L), `if`(type(L,
  'array'), assign('a' = map(op, convert(L, 'list1'))), ERROR("structure type of module <%1>
  is invalid", L))), `if`(convert(args[2], 'string') = "?", RETURN(map(lhs, a), seq(assign(a[k]),
  k=1..nops(a))), [add(`if`(lhs(a[k]) = f, RETURN(rhs(a[k])(args[3..nargs])),1), k=1..nops(a)),
  ERROR("procedure <%1> does not exist in package module <%2>", f, L)])]  end proc;
```

```
ArtMod := proc(L::symbol, f::symbol)
local a, k;
    [`if`(type(L, 'list'), assign('a' = L), `if`(type(L, 'array'),
    assign('a' = map(op, convert(L, 'list1')))), ERROR("structure type of module <%1>
      is invalid", L))), `if`(convert(args[2], 'string') = "?",
    RETURN(map(lhs, a), seq(assign(a[k]), k = 1 .. nops(a)))), [add(
    `if`(lhs(a[k]) = f, RETURN(rhs(a[k])(args[3 .. nargs]))), 1), k = 1 .. nops(a)),
    ERROR("procedure <%1> does not exist in package module <%2>", f, L)])]
end proc
```

```
> Kr:=array(1..2,1..2, [[f1=(() -> `+`(args)/nargs), f2 =(() -> add(args[k]^2, k=1..nargs)/nargs)],
  [f3 = (() -> [`+`(args), nargs]), f4 = (() -> `*`(args))]]);
```

$$Kr := \begin{bmatrix} f1 = \left((\ ) \to \frac{\text{`+`}(\text{args})}{\text{nargs}} \right) & f2 = \left((\ ) \to \frac{\text{add}(\text{args}_k^{\,2}, k = 1 \ldots \text{nargs})}{\text{nargs}} \right) \\ f3 = ((\ ) \to [\text{`+`}(\text{args}), \text{nargs}]) & f4 = ((\ ) \to \text{`*`}(\text{args})) \end{bmatrix}$$

```
> savelibname:= "C:/Program Files/Maple 10/LIB/Svetlana": savelib(Kr); restart;
> 2006*ArtMod(Kr, f4, 42, 47, 67, 89, 96),  ArtMod(Kr, f3);   ⇒ 2266804778112, [0, 0]
> restart;  ArtMod(Kr, `?`),  2006*f4(64, 59, 39, 44, 10, 17);   ⇒ [f1, f2, f3, f4], 2209678648320
> 6*seq(k(64, 59, 39, 44, 10, 17), k = [f1, f2, f3, f4]);   ⇒ 233, 11423, [1398, 36], 6609208320
```

Представленный выше фрагмент иллюстрирует создание *пакетного* модуля **Kr,** использующего структуру *массива*, с последующим сохранением его в ***Maple***-библиотеке пользователя **Svetlana**. Остальные примеры фрагмента иллюстрируют использование процедуры ***ArtMod*** для того, чтобы обеспечить доступ к данному пакетному модулю. При вызове ***ArtMod*** процедуры поиск указанного ее *первым* фактическим аргументом модуля **L** производится *сначала* среди активных средств текущего сеанса пакета и только после этого в цепочке библиотек, определенной предопределенной ***libname***-переменной. В ряде случаев *подобные* нестандартные подходы к организации структур пакетных модулей оказываются значительно эффективнее стандартных, поддерживаемых средствами пакета, позволяя выполнять над структурами таких модулей необходимые символьные вычисления и преобразования. Наш опыт апробации и эксплуатации пакета ***Maple*** релизов с **4**-го по **10**-й со всей определенностью говорят в пользу данного утверждения.

В заключение данного раздела кратко напомним *способы* обращения к *пакетным* модулю и содержащимся в нем объектам в целом, предполагая, что модуль находится в библиотеке, логически связанной с *главной* библиотекой пакета:

M[*Function***](***args***), M:-** *Function*(*args*) – *вызов* пакетной функции на заданных фактических аргументах ***args*** при условии, что пакетный модуль **M** имеет модульную структуру, в противном случае второй формат вызова инициирует ошибочную ситуацию с диагностикой «Error, `**M**` does not evaluate to a module». *Первый* формат корректен для пакетного модуля любого типа

with(M) – возврат списка *имен* объектов, находящихся в модуле **M,** с их автивацией в текущем сеансе. После такого *вызова* доступ к функциям модуля **M** обеспечивается в обычном формате *Function*(***args***); при этом, появление предупреждений типа «Warning, the name **N** has been redefined» говорит, что одноименная с библиотечной процедура **N** пакетного модуля *заменила* собой библиотечное средство. Во избежание этого рекомендуется использовать *разовые* вызовы пакетных средств по конструкциям следующего формата **M[***Function***](***args***), M:-** *Function*(*args*)

with(M, f1, f2, ...) – возврат *списка имен* [***f1, f2, ...***] объектов модуля **M** с активацией в текущем сеансе приписанных им определений. После данного вызова доступ к функциям ***f1, f2, ...*** модуля **M** обеспечивается в обычном формате ***fj*(*args*)**

packages() – возврат упорядоченного (*согласно порядка обращения к модулям*) списка имен всех пакетных модулей, хоть один экспорт которых был активирован в текущем сеансе

ListPack*()** – возврат списка *всех* пакетных модулей, находящихся в *главной* ***Maple-библиотеке, ***ListPack*(F)** – возврат списка *всех* пакетных модулей, находящихся в ***Maple***-библиотеке, имя которой или полный путь к ней определены фактическим аргументом **F.**

Примеры нижеследующего фрагмента иллюстрируют вышесказанное:

```
> with(linalg);
   [BlockDiagonal,GramSchmidt, JordanBlock,LUdecomp, QRdecomp, Wronskian, addcol, addrow,
 ================================================================
  submatrix, sylvester, toeplitz, trace, transpose, vandermonde, vecpotent, vectdim, vector, wronskian]
> restart;  with(linalg, det, rank);    ⇒  [det, rank]
> restart;  m:= matrix(3, 3, [x, y, z, a, b, c, d, g, h]):  linalg[det](m), linalg[rank](m);
                                x b h - x c g - a y h + a z g + d y c - d z b, 3
> restart;  m:= matrix(3, 3, [x, y, z, a, b, c, d, g, h]):  linalg:- det(m), linalg:- rank(m);
Error, `linalg` does not evaluate to a module
> map(M_Type, [linalg, LinearAlgebra]);   ⇒  [Tab, Mod]
> restart;  N:=Matrix(3, 3, [[x, y, z], [a, b, c], [d, g, h]]):  LinearAlgebra:- Determinant(N),
LinearAlgebra:- Rank(N);    ⇒  x b h - x c g - a y h + a z g + d y c - d z b, 3
> 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();
Warning, the name changecoords has been redefined
Warning, the protected names norm and trace have been redefined and unprotected
Warning, the name arrow has been redefined
                                [plots, linalg, LinearAlgebra, plottools]
> restart; p:= x^3+10*x^2+17: m:=matrix(3, 3, [x, y, z, a, b, c, d, g, h]): norm(p, 1),
linalg[norm](m);    ⇒  28, max(|x| + |y| + |z|, |d + |g| + |h|, |a| + |b| + |c|)
> ListPack();                                  #  Maple 8
{simplex, polytools, padic, geom3d, student, tensor, geometry, PDEtools, powseries, plots, plottools,
 ================================================================
   Domains, liesymm, Sockets, SNAP, VectorCalculus,Units, Maplets,Worksheet, stats, SolveTools}
> ListPack("C:/Program Files/Maple 8/LIB/UserLib");
                          {ACC, boolop, DIRAX, SimpleStat, SoLists, AlgLists}
```

В заключение данного раздела сделаем еще одно существенное замечание. Процедура **with** пакета используется для обеспечения удобного доступа к *модульным* средствам пакета, в удобной форме на интерактивном уровне. Это - команда, которая обеспечивает функциональные возможности, аналогичные предложению **use,** но работает на интерактивном уровне и применима ко всем пакетным и программным модулям. При этом, предложение **use** обеспечивает средства только для работы с *программными* модулями.

Процедура **with** эффективна только на верхнем уровне и предназначена, 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, тогда как в ***Maple 10*** она не работает корректно в телах процедур либо модулей, выводя соответствующие предупреждения.

Наша процедура **With** [103] устраняет данный недостаток, позволяя корректно использовать вышеупомянутый формат в телах процедур и модулей. В целом ряде случаев это обеспечивает более удобное представление алгоритмов в среде ***Maple***-языка. Наряду с этим, вызов процедуры **With(P, Fo {, F1 {….}})** присваивает именам {**Fo, F1,…**} *protected*-атрибут, отсутствующий для стандартных средств пакета.

```
With := proc(P::{ `module`, package }, 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 ], "/_$Art17_Kr9$_" ) )
        , 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:= 59;
Error, attempting to assign to `display` which is protected
> animate:= 64;
Error, attempting to assign to `animate` which is protected
> Kr:=module() export sr, No; No:=() -> [[nargs], args]; sr:=() -> `+`(args)/nargs end module:
> With(Kr, sr, No);   ⇒ [sr, No]
> No(1, 2, 3), sr(1, 2, 3);   ⇒ [[3], 1, 2, 3], 2
> M:= module ()  end module: with(M);
Error, (in with) module `M` has no exports
> With(M);   ⇒ []
```

Более того, при отсутствии у модуля **M** экспортов вызов **with(M)** вызывает ошибочную ситуацию с диагностикой "module `%1` has no exports" для всех релизов ***Maple.*** Поэтому для поддержки более удобной работы с *экспортами* модуля рекомендуется или использовать ***exports***-функцию, возвращающую *NULL*-значение на модулях *без* экспортов, или нашу 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 этом случае производится загрузка в **РОП** всех его экспортов. Поэтому, для таких целей рекомендуется использовать процедуру ***tpacmod,*** имеющую формат вызова следующего вида: ***tpacmod*(P** {**, *Name***}**)**, где **P –** имя пакетного модуля и ***Name*** – имя его экспорта [41,103]. Вызов процедуры с одним аргументом **P,** в качестве которого допустимо только имя пакетного модуля, находящегося в ***Maple***-библиотеке, логически сцепленной с главной библиотекой пакета, возвращает список экспортов *модуля* **P.** Тогда как вызов процедуры ***tpacmod*(P,*Name*)** с двумя 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 **РОП** 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 пакетных модулей.

```
> tpacmod:= proc(P::package)  parse(cat("`if`(belong(Release(), 6 .. 8), `pacman`,
  `PackageManagement`):- ", convert(`if`(nargs=1,`pexports`,`pmember`)(`if`(nargs=1, args,
  op([args[2], args[1]]))), 'string')), 'statement')  end proc;
                tpacmod := proc(P::package)
                    parse( cat( "`if`(belong(Release(), 6 .. 8), `pacman`,
                      `PackageManagement`):- ", convert( `if`( nargs = 1, pexports, pmember )(
                     `if`( nargs = 1, args, op( [ args[ 2 ], args[ 1 ] ] ) ) ), 'string' ) ), 'statement' )
                end proc
> tpacmod(linalg);
[BlockDiagonal, GramSchmidt, JordanBlock, LUdecomp, QRdecomp, Wronskian, addcol, addrow, adj,
 ================================================================
  swaprow, sylvester, toeplitz, trace, transpose, vandermonde, vecpotent, vectdim, vector, wronskian]
> packages();   ⇒  []
> tpacmod(linalg, AVZ), tpacmod(linalg, det), tpacmod(linalg, diag);    ⇒  false, true, true
> tpacmod(SimpleStat, AGN), tpacmod(SimpleStat, Ds), tpacmod(SimpleStat, MCC);
                                        false, true, true
> packages();   ⇒  []
> tpacmod(stats);
                    [anova, describe, fit, importdata, random, statevalf, statplots, transform]
> packages();   ⇒  []
```

Из примеров фрагмента следует, что *вызовы* процедуры ***tpacmod*(M)** как для получения списка экспортов пакетного модуля **M,** так и для тестирования имени быть экспортом пакетного модуля не загружают ни экспортов пакетного модуля в **РОП,** ни модуля в целом. Данное средство во многих случаях работы с пакетными модулями оказывается в значительной степени весьма полезным.

Рассмотрев основные вопросы ведения *библиотек* пользователя в среде ***Maple,*** довольно важным представляется и вопрос анализа библиотек на предмет их эффективности, ответ на который в значительной мере может прояснить статистический анализ *динамики* функционирования библиотек в процессе их реального использования.

8.4. *Статистический анализ Maple-библи*отек

Создав собственную ***Maple***-библиотеку процедур с использованием вышеупомянутых подходов или каким-либо иным способом, естественно возникает задача ее оптимизации, в частности, с целью раскрытия частоты использования средств, содержащихся в ней, и основных ресурсов компьютера, используемых ими. В этом контексте, проблема оптимизации библиотек пользователя довольно актуальна. Для этих целей достаточно полезной представляется процедура ***StatLib*(L)** [41,103], обеспечивающая сбор основной статистики по заданной **L** библиотеке и возврату статистики для последующего *анализа*. Прежде всего, процедура ***StatLib*** предполагает, что анализируемая библиотека **L** находится в каталоге **LIB** с главной библиотекой ***Maple***. В процессе своего выполнения процедура ***StatLib*** требует некоторых дополнительных ресурсов памяти и времени.

Процедура ***StatLib*(L)** в качестве *первого* обязательного аргумента **L** использует *имя* анализируемой ***Maple***-библиотеки, расположенной в подкаталоге **LIB** *главного* каталога пакета, 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:

StatLib(L) – *инициация* сбора статистики по средствам библиотеки **L** в целом**;**
StatLib(L, 0) – *удаление* всех файлов со статистической информацией по анализируемой библиотеке **L;**
StatLib(L, 1) – *отмена сбора* статистики по библиотеке **L** с сохранением полученных результатов**;**
StatLib(L, P) – *инициация* сбора статистики по процедуре **P** из библиотеки **L;**
StatLib(L, T, 2) – *возврат* собранной статистики по библиотеке **L** либо по ее процедуре **P** в заданном разрезе **T;**
StatLib(L, T, 2, m) – *возврат* собранной статистики по библиотеке **L** либо по ее процедуре **P** в заданном **T** разрезе, характеризуемом числом **m;**
StatLib(L, T, N) – *возврат* собранной статистики по заданной процедуре **N** в требуемом разрезе {*calls, bytes, depth, maxdepth, time*}**;**
StatLib(L, *abend*) – *завершение* процедуры при возникновении критических ошибок**.**

Вызов процедуры ***StatLib*(L)** с одним фактическим аргументом **L** инициирует процесс сбора статистики по вызовам составляющих библиотеку **L** процедур. Данный процесс сбора может 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 *пяти* файлах, размещаемых в каталоге с библиотекой **L.** Имена этих файлов данных имеют одинаковый "**$@_**"**-**префикс. Вызов процедуры с нулевым значением второго фактического аргумента возвращает *NULL*-значение с удалением файлов сбора статистики для данной библиотеки **L.** Данный вызов рекомендуется выполнять перед инициацией нового процесса профилирования процедур с целью сохранения его результатов в каталоге с библиотекой **L.** При этом, существующие статистические файлы данных с ранее собранной информацией могут быть предварительно сохранены в другом каталоге либо остаться в старом под иными именами.

Вызов процедуры ***StatLib*(L, P)** инициирует процесс сбора статистики по *вызовам* процедур, составляющих библиотеку **L** и определяемых их **P**-списком имен. При этом, запрашиваемые к профилированию процедуры не обязательно должны принадлежать библиотеке **L** – они могут находиться в любой библиотеке, логически связанной с главной

библиотекой пакета, или быть активными в текущем ***Maple-***сеансе. В противном случае инициируется *ошибочная* ситуация с возвратом соответствующей диагностики. Это позволяет производить анализ как всей библиотеки **L** *в целом*, так и в *разрезе* составляющих библиотекуе процедур (*не совмещая оба эти процесса*), а также любых *активных* либо *доступных* процедур текущего сеанса пакета.

Процедура обеспечивает возврат статистической информации в следующих *пяти* разрезах, определяемых ключевыми словами **T = {*calls*, *bytes*, *depth*, *maxdepth*, *time***}, определяющими соответственно показатели: (**1**) *количество вызовов*, (**2**) *объем используемой памяти*, (**3**) *глубина* и (**4**) *максимальная глубина вложенности*, а также (**5**) *использованное время* в сек. Вызов процедуры ***StatLib*(L, T, 2)** возвращает массив числовых характеристик для всех профилируемых процедур библиотеки **L** {*либо процедур*, *определенных* **P**-*аргументом при вызове* ***StatLib*(L, P)**} на текущий момент в заданном **T**-разрезе (*например*, ***calls* –** *количество вызовов процедуры*), тогда как вызов процедуры ***StatLib*(L, T, 2, m)** возвращает массив для **T**-разреза тех профилируемых процедур, значения соответствующих **T**-разрезу характеристик которых не менее **m**-величины. При отсутствии таких строк у массива процедура возвращает ***lack***-значение, информируя о том, что на данный момент ни одна из профилируемых процедур не обладает соответствующей **T**-разрезу характеристикой со значением, не меньшим, чем **m.** Данный массив построчно отсортирован в порядке убывания значений характеристик; при этом, для *равных* значений строки массива сортируются лексикографически.

Наконец, вызов ***StatLib*(L, T, N)** процедуры (*если в качестве третьего аргумента* **N** *процедуры указано значение* {*symbol*, *string*}-*типа*) возвращает значение характеристики процедуры с именем **N** (*если таковая существует и ранее профилировалась*) в заданном **T**-разрезе. Например, вызов ***StatLib*(*UserLib*, *calls*, *RmDir*)** возвращает количество вызовов процедуры ***RmDir*** библиотеки **UserLib,** если данная библиотека или процедура предварительно профилировались. На остальных кортежах значений фактических аргументов возвращается *NULL*-значение. Для обеспечения больших надежности и сохранности собираемой статистической информации рекомендуется периодически ее сохранять посредством вызовов ***StatLib*(L, 1)** с последующим возобновлением *профилирования* требуемых средств вызовом ***StatLib*(L)** процедуры.

Процедура допускает два режима мониторинга результатов *профилирования* процедур – *динамический* и *разовый*. Динамический режим обеспечивает довольно удобную возможность мониторинга посредством описанных выше вызовов процедуры в процессе профилирования, т.е. между двумя *вызовами* процедуры ***StatLib*(L)** и ***StatLib*(L,1).** При этом, не нарушается сам процесс профилирования. Тогда как разовый режим предоставляет возможность проводить *выборочную* проверку результатов *предыдущего* профилирования без возобновления самого процесса профилирования. Данный подход позволяет более гибко производить мониторинг процесса профилирования процедур как внутри него, так и вне.

В некоторых *версиях релизов* **6-10** пакета использование процедуры ***StatLib*** может в ряде случаев вызывать критические ошибки, связанные, прежде всего, с *переполнением* пакетного стэка. В этом случае рекомендуется выполнить вызов ***StatLib*(L, abend)** с последующим выполнением предложения **restart** пакета. Данный прием позволяет сохранять в указанных статистических файлах по меньшей мере информацию по количеству вызовов профилированных процедур на момент аварийной ситуации. Последующий вызов ***StatLib*(L)** обеспечивает возобновление процесса профилирования с *прерванного* момента. Вышеупомянутые ошибочные ситуации могут быть в значительной степени объяснены следующим образом.

Наш достаточно длительный опыт использования ***Maple*** релизов **4-10** в различных приложениях, включая развитие средств, расширяющих основные средства пакета, со всей определенностью выявил *одно* довольно существенное обстоятельство. Многие из часто используемых стандартных ***Maple-***средств были *обеспечены* недостаточно развитой системой обработки специальных 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 *стэк* недостаточной глубины, или вышеупомянутая ситуация была вызвана (*в отсутствие каких-либо циклических вычислений*) ошибкой, имеющей *причину* некоторого другого характера. К сожалению, увеличение номеров релизов ***Maple*** пока сопровождается *уменьшением* их ошибкоустойчивости, да и не только этого.

Реализация алгоритма процедуры ***StatLib*** существенно *базируется* на представленных в книгах [41,103] процедурах ***DoF, Fremove, Plib, belong, _SL, tabar,*** а также на *специальных* процедурах ***profile*** и ***unprofile*** пакета для *профилирования* вызовов процедур. При использовании ***StatLib*** процедуры имеет место замедление вычислений и увеличение используемой памяти, *величина* которых зависит, прежде всего, как от количества профилируемых процедур, так и частоты их использования. Однако, ввиду относительно небольших библиотек пользователя (*до* **600 – 700** *процедур и частоте их использования не более* **600** *за сеанс, исключая циклические конструкции*) это обстоятельство не приводит к критическим ситуациям, связанным с использованием основных ресурсов компьютера и времени обработки. В частности, использование данного механизма для *нашей* библиотеки с процедурами (*свыше* **720**), представленными в книге [103] и прилагаемой к ней *Библиотеке*, не дает каких-либо оснований рассматривать описанный механизм профилирования в качестве причины достаточно серьезных дополнительных издержек основных ресурсов **ПК,** правда, эксперименты производились на **Pentium 4** с частотой **3** GHz, RAM **1** GB и HDD **120** GB.

Механизм использования ***StatLib*** процедуры сводится к следующему. В самом начале сеанса работы с пакетом выполняется вызов процедуры ***StatLib*****(L** {**, P**}**),** где **L** – имя анализируемой библиотеки пользователя, удовлетворяющей указанным выше условиям (**P-***аргумент может определять список имен процедур из библиотеки* **L** *или вне ее*). Перед завершением текущего сеанса работы выполняется вызов процедуры ***StatLib*****(L, 1),** который обеспечивает сохранение статистической информации в пяти специальных файлах с именами "**$@#_h**" (где **h** ∈ {*depth*, *calls*, *bytes*, *maxdepth*, *time*}), помещаемых в подкаталог с библиотекой **L.** В любой момент (*динамически либо разово*) посредством вызовов процедуры ***StatLib*****(L, T,** {**2**|***name***} {**, m**}**)** можно получать справку по характеристикам вызовов процедур библиотеки в указанных выше разрезах.

Наиболее эффективным режимом является следующий. Каждый очередной *сеанс* работы с пакетом начинается вызовом ***StatLib*****(L),** после которого производится текущая работа в среде пакета. Периодически рекомендуется производить *пары* вызовов {***StatLib*****(L, 1),** ***StatLib*****(L)**} для обеспечения надежности по сохранности результатов профилирования процедур библиотеки. Перед *завершением* текущего сеанса выполняется вызов процедуры ***StatLib*****(L, 1),** обеспечивая прекращение процесса профилирования и *сохранение* его результатов в упомянутых файлах. Просмотр результатов профилирования в упомянутых разрезах рассматривался нами выше. *Анализ собранной* статистической информации дает возможность улучшать как организацию библиотеки в целом, так и эффективность составляющих ее отдельных процедур, имеющих достаточно высокую частоту использования либо существенно использующих основные ресурсы компьютера.

Опыт использования процедуры ***StatLib*** со всей определенностью говорит о ее довольно высокой эффективности в случае решения проблем *оптимизации* библиотек пользователя. *Механизм* и *методы* использования процедуры ***StatLib*** достаточно подробно рассмотрены в наших предыдущих книгах [29-33,39,42-46,103]. Тогда как представленные ниже примеры достаточно наглядно иллюстрируют принципы и результаты применения процедуры ***StatLib***.

```
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], {'bytes', 'maxdepth', 'calls', 'time', 'depth'})
         and (args[3] = 2 or type(args[3], 'symbol')), true, false), 4 = `if`(
        nargs = 4 and member(args[2], {'bytes', 'maxdepth', 'calls', 'time', 'depth'})
         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 = ['bytes', 'calls', 'depth', 'maxdepth', 'time']),
        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 = ['bytes1', 'calls1', 'depth1', 'maxdepth1', 'time1'],
        Z = [seq(cat(profile_, R[h]), h = 1 .. 5)], G = [ ]);
    `if`(nargs = 2 and args[2] = 0, RETURN(WARNING("datafiles with statistic
        s 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 lib
        ary <%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);
```

Procedures	*Procedure's bytes*
SLj	3591160
tabar	2097700
Red_n	355168
Case	176172
type/nestlist	115256
Search	74416
Adrive	64752
StatLib	60944
Plib	42456
Subs_all1	38856
type/dir	35676
CF	35372
belong	33132
sub_1	28392
type/file	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, 63], [Gal, 58], [Sv, 38], [Arn, 42], [Art, 16], [Kr, 9]], 2);

[[*Kr*, 9], [*Art*, 16], [*Sv*, 38], [*Arn*, 42], [*Gal*, 58], [*Vic*, 63]]

> 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 out of directory with library **<UserLib>**

Ввиду пояснений, сделанных выше, примеры данного фрагмента достаточно прозрачны и не требуют каких-либо дополнительных пояснений. В частности, представленная процедура ***StatLib*** весьма эффективно использовалась для улучшения *функциональных* характеристик пользовательской *Библиотеки* **UserLib,** содержащей *процедуры*, представленные в наших книгах [41,103].

Процедуры, представленные в [41,103] и прилагаемой к ней *Библиотеке*, 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 целом ряде важных приложений. Отмеченные и другие предпосылки, обусловленные вышеупомянутыми процедурами, позволяют довольно существенно автоматизировать обработку пользовательских библиотек, наряду с расширением возможностей, имеющих дело с сохранением *процедур* и *программных* модулей во внешней памяти компьютера. В целом же, средства, методы и приемы, представленные в книгах [41,103], предназначены как для повышения эффективности *применения* пакета ***Maple*** в различных приложениях, требующих программирования, так и для самого освоения программирования в его программной среде.

9. *Средства Maple, не рассматриваемые в настоящей книге, но полезные для приложений*

В данном разделе представлена краткая характеристика средств пакета, по целому ряду причин не рассматриваемых в настоящей книге, однако представляющих *несомненный* интерес для разработчиков ***Maple***-приложений. Основной причиной этого явилось то, что в настоящей книге представлен базовый материал разработки ***Maple***-приложений без привлечения каких-либо средств извне.

Механизм *внешних вызовов* (*external calling*) позволяет интегрировать в ***Maple*** программы, откомпилированные в среде таких языков как ***C, Fortran*** или ***Java.*** Большое количество **ПС** можно получать с интернета, обеспечивающих решение самых *разнообразных* задач из различных приложений. Механизм *внешних вызовов* позволяет выполнять эти средства в ваших ***Maple***-приложениях. С этой целью встроенная функция ***define_external*** обеспечивает связь с *внешней* функцией (*например*, *из* ***dll***-*библиотеки*), формируя ***Maple***-процедуру, выполняющую роль интерфейса с внешней функцией. В некотором роде механизм *внешних вызовов* является расширением возможностей пакетных функций ***system*** и ***ssystem,*** рассматриваемых в настоящей книге. Сам ***Maple*** использует механизм внешних вызовов для имплантирования в свою среду численных библиотек ***NAG,*** включая модуль **LinearAlgebra** для решения задач линейной алгебры. Детально в данной книге механизм внешних вызовов нами не рассматривается, приведем лишь сам *принцип* реализации такого механизма на весьма простом примере.

Рассмотрим простую функцию сложения двух целых чисел, написанную на **C:**

```
int add(int x, int y)
        {return x + y;}
```

сохраненную в файле «**add.c**» с последующей компиляцией в ***dll***-библиотеку «**add.dll**», расположенную по адресу «**c:/user/add.dll**». Для *обеспечения* последующего обращения к этой функции из ***Maple***-среды формируем *вызов* функции ***define_external,*** включающий следующие параметры:

– имя функции в ***dll***-библиотеке (*например*, ***add***)
– имя самой ***dll***-библиотеки (*например*, ***add.dll***) или класс, содержащий подпрограмму
– типированные аргументы и результат возврата (*например*, *integer*[4])

В настоящее время механизм *внешних* вызовов поддерживает соглашения по вызову только для языков ***C, Fortran*** и ***Java.*** При этом, автоматическая генерация оболочки вызова поддерживается только для языка **C.** В нашем конкретном случае механизм *внешнего* вызова принимает следующий вид:

```
> add_c := define_external('add', LIB="C:/user/add.dll", 'a'::integer[4], 'b'::integer[4],
  RETURN::integer[4]): add_c(10, 17);   ⇒ 27
```

После этого процедура ***add_c*** может использоваться подобно обычным ***Maple***-процедурам. В обшем случае вызов функции ***define_external*** имеет следующий формат:

define_external**('***имя функции***', a1::<***описание***>, ..., an::<***описание***>, LIB="***путь к dll-library***")**

При этом, если вместо аргументов **aj** (**j=1..n**), типированных своим *описанием*, кодируется только одно слово **MAPLE,** то предполагается, что внешняя функция принимает поток струк-ур данных ***Maple*** без преобразования. Представленный механизм *внешних вызовов* предполагает, что вы имеете ***dll***-библиотеку, а также полное описание входящих в нее функций, по меньшей мере тех, интерфейс с которыми вы планируете создавать.

Для повышения эффективности использования механизма внешних вызовов, начиная с ***Maple 7,*** с пакетом поставляется модуль **ExternalCalling,** содержащий процедуры для эффективного использования встроенной функции ***define_external.*** Заинтересованный читатель может детально ознакомиться с механизмом *внешних* вызовов в справке по пакету, например, по вызову **?external_calling.**

Начиная с ***Maple 8,*** пакет дополнен модулем **Maplets,** содержащим *набор* средств для создания *графического интерфейса пользователя*. Модуль **Maplets** обеспечивает *возможность* создания произвольных графических интерфейсов пользователя, называемых *маплетами* (***maplets***), для доступа к вычислительной среде пакета. **Maplets**-технология позволяет встраивать кнопки, панели инструментов, меню, всплывающие меню, перетаскиваемые кнопки, окна для построения графики и обеспечивать работу других элементов. И при этом, пользователи могут выполнять вычисления и *другие* операции, включая работу с графическими объектами, без использования стандартного **GUI.** В этом *отношении* пользователь может создавать графический интерфейс под свои конкретные приложения. Приложения, созданные посредством **Maplets,** используют ***Java***-технологию, однако пользователь не должен быть знаком с ***Java.*** Ему потребуется лишь: (**1**) знание основ программирования в ***Maple*** и (**2**) общее знакомство со средствами модуля **Maplets.**

Пользователь для создания ***Maplet***-приложения может использовать или средства модуля **Maplets,** или ***Maplet***-конструктор (*Maplet Builder*). При этом, *второй* подход допустим только в среде стандартного клона ***Maple.*** К недостаткам (*возможно*, *временным*) данного средства можно отнести существенно меньшую *мобильность* относительно *стандартного* **GUI,** необходимость наличия ***Java***-среды и немалое число недоработок. Здесь средства для создания ***Maplet***-приложений не рассматриваются и заинтересованный читатель за информацией может обратиться к справке по пакету либо получить вводную довольно полезную информацию по адресу **http://maplets.exponenta.ru/intro.htm.**

Наконец, вкратце упомянем еще одно средство **OpenMaple,** обеспечивающее доступ к библиотекам ***Maple*** из других ***Windows***-приложений. Начиная с ***Maple 9,*** оно обеспечивает прямой доступ к ***Maple***-библиотекам из внешних программ на основе механизма **API** (*application programming interface*). **OpenMaple** представляет собой набор функций, которые предоставляют *откомпилированным* программам доступ к программам и структурам данных ***Maple.*** В определенной мере это средство, обратное к вышеупомянутому механизму внешних вызовов (*externalcalling*), обеспечивающему доступ к *внешним* функциям и структурам данных в среде ***Visual Basic, Java, C, MATLAB*** и ***FORTRAN.*** Использование средств **OpenMaple** *наиболее* эффективно и естественно из ***C***-среды, однако достаточно опытные программисты могут использовать данный механизм также из среды ***Java*** и ***Visual Basic.***

Так, используя **OpenMaple API** in ***Maple 9.5,*** пользователь в режиме реального времени может вызывать ***Maple***-программы *изнутри* программ, написанных на языках ***Java, C*** и ***Visual*** **Basic.** В данном случае **OpenMaple** представляет собой *набор* программ, написанных на языках ***Java, C*** и ***Visual Basic,*** и обеспечивающих программам, написанным на этих языках, доступ к ***Maple***-программам как пакетным, так и пользовательским. С примерами такого рода можно ознакомиться в стандартных *поставках* пакета, начиная уже с релиза ***Maple 9.5.*** Из вышесказанного нетрудно усмотреть, почему данная тема не рассматривается нашей книгой. Это проблема, скорее, пользователя из иной программной среды, желающего использовать те или иные средства пакета ***Maple,*** например, *средства* для работы с графическими объектами либо для символьных вычислений. Таким образом, собственно для пользователя пакета ***Maple*** именно средства обеспечения внешних вызовов и маплетов представляются нам наиболее интересными и полезными.

Заключение

Настоящая книга вводит специалистов, ученых, преподавателей и студентов в различные аспекты программирования в среде известного математического пакета ***Maple*** релизов **6 – 11**. В ряде источников ***Maple*** определяется как система *компьютерной алгебры* (**CAS**), использовался данный термин и нами. Однако мы остановились именно на термине «***пакет***» и вот почему. В нашем понимании программное средство, именуемое «*пакетом*», предоставляет собственную среду программирования, ориентированную, прежде всего, на наиболее эффективную реализацию в ней задач, относящихся к области приложений пакета. При этом, перед пакетом, как правило, не ставится цели универсализации в том смысле, чтобы программируемые в его среде средства были выполняемыми непосредственно в операционной среде **ПК**, например, на уровне {***exe, com***}-файлов. А именно такой возможностью и не обладает ***Maple***.

Книга является непосредственным продолжением наших предыдущих книг по проблематике пакета, изданных в России, Белоруссии, Эстонии, Литве и США. Основное наше внимание уделено основам и принципам программирования в среде ***Maple***-языка пакета, позволяющим эффективно использовать нашу *Библиотеку* [41,103,109], прежде всего. Наряду с этим, книга может послужить неплохим *введением* в программную среду пакета ***Maple*** релизов **6 – 11**. Здесь упомянут и релиз **11** пакета, появившийся на рынке в самом начале 2007. И это совсем не лишне, ибо программная среда пакета является его основой и наименее обновляемой компонентой, пролонгируясь от релиза к релизу.

Данное программное обеспечение, выполненное на инновативном уровне, организовано как пользовательская библиотека, логическое соединение которой с главной ***Maple***-библиотекой позволяет использовать содержащиеся в ней средства на уровне стандартных средств пакета. *Библиотека* содержит хорошо-разработанное программное обеспечение (*набор более* ***730*** *процедур и модулей*), которое хорошо дополняет *уже* существующее программное обеспечение пакета с ориентацией на самый *широкий* круг пользователей, существенно *увеличивая* его применимость и эффективность. Опыт использования данной *Библиотеки* как отдельными пользователями, так и в ряде университетов и исследовательских институтов в России, Белоруссии, Латвии, Литве, Украине, Германии и т.д. подтвердил ее *хорошие* функциональные характеристики при решении разнообразных физико-математических задач. Во многих случаях представленные в ней дополнительные процедуры и модули иллюстрируют как полезные *приемы* программирования, так и элементы *методологии* и *методов* программирования в среде пакета.

Библиотека предназначена для ***Maple*** *релизов* **6-11,** функционирующего на платформах типа ***Windows*** **95/98/98SE/*ME/NT/XP*/2000/2003,** однако ASCII- файл с исходными текстами всех библиотечных средств, поставляемый с библиотекой, позволяет *легко* адаптировать ее к операционным платформам, отличным от ***Windows***. При этом, наборы исходных текстов всех библиотечных средств и справочных страниц, составляющих ее справочную базу данных позволяют легко обновлять библиотеку или создавать на ее основе собственные библиотеки. Прилагаемые файлы ***ProcLib_6_7_8_9.mws*** и ***ProcLib_10.mws*** содержат ***Maple***-документы, обеспечивающие автоматическую инсталляцию библиотеки в среде ***Maple*** релизов **6-10**. Наконец, в поставляемый комплект *Библиотеки* входит набор ее стандартных вариантов для релизов **6-10,** легко устанавливаемых на **ПК** простым копированием без традиционной инсталляции.

Средства, представленные в *Библиотеке*, расширяют диапазон и эффективность применения пакета на платформе ***Windows*** благодаря новациям в *трех* основных направле-

ниях, а именно: (**1**) устранение ряда основных дефектов и недостатков, (**2**) расширение возможностей целого ряда стандартных средств, и (**3**) пополнение пакета новыми средствами, которые расширяют возможности его программной среды, включая средства, существенно повышающие уровень совместимости релизов **6 – 10**. Основное внимание было уделено *дополнительным* средствам, созданным в процессе практического использования и апробации ***Maple*** релизов **4 – 10,** которые по ряду характеристик *существенно* расширяют возможности пакета, делая работу с ним намного более легкой. Значительное внимание уделено средствам, обеспечивающим более *высокий уровень* совместимости пакета релизов **6 – 10**. Опыт наш и наших коллег по использованию вышеупомянутого программного обеспечения для различных приложений и обучения подтвердил его достаточно высокие эксплуатационные характеристики.

Наконец, следует отметить, что ряд наших книг и статей на пакету, представляя средства, созданные нами, и содержа предложения по дальнейшему развитии ***Maple*** стимулировал развитие таких приложений как модули **FileTools, LibraryTools, StringTools** и **ListTools**. Все это позволяет надеяться, что представленная книга, а также другие цитируемые в ней материалы окажутся достаточно полезными для широкой аудитории пользователей пакета, как новичков, так и уже имеющих опыт работы с пакетом ***Maple***.

И в заключение кратко о том, как создавалась *Библиотека*, упомянутая выше. Информация об этом позволит более *адекватно* оценить ее место в программной среде ***Maple*** и ее основные назначения для пользователей пакета различного уровня. Работая, в основе своей, в фундаментальных областях естествознания (*математика*, *кибернетика*, *математическая биология и др.*), мы, между тем, значительное *внимание* уделяли и такому прикладному направлению, как компьютерная техника с акцентом на ее программном обеспечении (*операционные системы*, *языки программирования*, *математические и статистические пакеты*, *системы компьютерной алгебры*, СУБД, САПРы *и т.д.*). Работа в данном направлении заключалась как в разработке программных средств различного назначения (*как системных*, *так и прикладных*), так и в проведении курсов лекций различного *уровня*, а также подготовке серий книг различной направленности, изданных в СССР, Эстонии, Литве, Белоруссии и США.

Как правило, работа с конкретным программным средством велась по пяти основным направлениям: (**1**) *освоение* на основе *всесторонней* апробации, (**2**) *применение* к решению различных задач и проектов математического, статистического и инженерно-физического характера, (**3**) *чтение* соответствующих курсов, (**4**) *выработка* рекомендаций по эффективному использованию средства, его особенностям и недостаткам, включая создание собственных средств, расширяющих, дополняющих и исправляющих стандартные средства, и (**5**) *подготовка* различного рода изданий (*книги*, *статьи*, *сборники и др.*) наряду с консультативной активностью. Естественно, подобная концепция предполагает весьма серьезную *творческую* активность в данной области, наиболее импонирующую нашей натуре и представлениям.

Именно данный подход к каждому **ПС,** с которыми мы имели дело, и позволил создать полезные средства в данном направлении. Так, в 1976 была создана операционная система **MINIOS** (*оптимизированная версия OS IBM/360 для младших моделей ЕС ЭВМ*), ***ПСОИ*** (*параллельная система обработки информации для ЕС ЭВМ/IBM 360/370*), **СУБД MINOKA** (*оптимизированная версия db ms IMS*) и др. Издав в 1991 г. книгу по **MathCAD,** первую в СССР вводящую отечественного читателя в область математических пакетов, затем были подготовлены книги по таким средствам как ***Reduce***, ***Mathematica*** и ***Maple***. Именно на последнем пакете наше внимание и задержалось на более длительное время. Обусловлено это было, прежде всего, тем, что именно данный пакет использовался нами и

нашими коллегами из Литвы и Беларуси в ряде приложений математического и инженерно-физического характера.

Издание в 1996-1998 *одних* из *первых* в стране книг по пакетам ***Mathematica*** и ***Maple 5*** породило немало писем в наш адрес с целым рядом очень интересных вопросов по этим (*конкурирующим между собой и во многом подобным*) пакетам. На сегодня в *общей* сложности их более **1500**. Основная масса носила и носит довольно тривиальный характер, однако немало встречается и вопросов, требующих достаточно серьезной проработки. Ни один из вопросов не остался без нашего внимания. Так вот, среди этой массы писем целый ряд содержал вопросы, *решение* которых и инициировало создание многих из представленных в [103] процедур. Наша особая благодарность авторам писем, чьи вопросы позволили оформить их в качестве отдельных задач, полезных как для практического применения, так и в учебных целях.

Наконец, немало процедур было инициировано проведением целого ряда *курсов* по пакету ***Maple*** различного уровня, проведенных в **2001 – 2006** для преподавателей и докторантов ряда университетов, а также научных сотрудников академических институтов СНГ, Прибалтики и др. Таким образом, наша активность по использованию пакета, работа с письмами читателей наших книг и проведение серии курсов – вот *три* основных источника, стимулировавших появление *Библиотеки*, прилагаемой к настоящей книге.

Суммируем теперь по *внутреннему* оформлению библиотечных средств. Так как *Библиотека* ориентирована как для применения по своему основному назначению в качестве дополнения к уже имеющимся средствам ***Maple*** (*новые средства*, *расширение и улучшение стандартных и др.*), так и для использования в учебном процессе по курсу «*Программирование в среде пакета Maple*» в качестве практического справочного материала. Во втором случае нами используется следующая методика. На первом этапе читается замкнутый курс по встроенному ***Maple***-языку пакета, включая сопутствующие темы: основные типы структур данных, *основные* встроенные функции пакета, библиотечные процедуры, средства доступа к данным и т.д. Изложение сопровождается решением (*совместным со слушателями*) наиболее типичных примеров по всем темам курса. На втором этапе слушателям выдаются задания на применение полученных ими знаний и навыков для решения задач, аналогичных находящимся в данной библиотеке процедурам, но с одним непременным условием, чтобы их решение максимально отличалось от имеющегося в качестве контрольного. Опыт показывает, что такой подход дает довольно неплохие результаты, а именно.

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. Ряд часто используемых процедур библиотеки, ориентированных на *массовое* применение при программировании различных приложений, оптимизирован. Тогда как многие, обладая функциональной полнотой, на которую они и были ориентированы, между тем, в *полной* мере не оптимизированы, что предоставляет слушателю достаточно широкое *поле* для его творчества как по оптимизиции процедуры, так и по созданию собственных аналогов, постоянно контролируя себя готовым, отлаженным и корректно функционирующим прообразом. Более того, используемые в процедурах полезные, эффективные (*а в целом ряде случаев и нестандартные*) приемы программирования позволяют более глубоко и за более короткий срок освоить программную среду пакета. Использование же во многих процедурах обработки особых и ошибочных ситуаций дает возможность *акцентировать* уже на ранней стадии внимание на таких важных компонентах создания программных средств, как их надежность, мобильность и ошибкоустойчивость.

Наконец, работая с *Библиотекой*, слушатель не только имеет прекрасную возможность освоить многие из ее средств для своей текущей и последующей работы с пакетом, но и проникается концепцией эффективной организации *своих* собственных ***Maple***-библиотек, включающих средства, обеспечивающие его профессиональные интересы.

Представленная в [100] *Библиотека* содержит далеко не все разработанные нами средства, ориентированные на работу в среде ***Maple***. В нее вошли лишь средства, ориентированные на достаточно широкое использование при программировании в среде пакета и базирующиеся исключительно на основных стандартных средствах. Значительная же часть наших разработок выполнена в виде отдельных пакетных *модулей*, ориентированных на специальные приложения в естественных науках и поставляемых на коммерческой основе. Естественно, данные модули достаточно существенно используют и средства упомянутой здесь *Библиотеки*. Есть надежда, что и читатель найдет среди средств нашей *Библиотеки* полезные для своего творчества.

В настоящей книге, невзирая на ее характер, представлен ряд *полезных* приемов и рекомендаций по программированию в ***Maple***, намного больше такого типа информации можно найти в наших книгах [41-43,103] и в прилагаемых к ним архивах, содержащих исходные тексты большого количества процедур. Однако, немало различного рода нюансов работы в среде ***Maple*** осталось и *вне* нашего поля зрения, следовательно для вполне приличного освоения (*mastering*) требуется достаточно серьезная творческая *наработка* с пакетом и не на уровне высоко интеллектуального калькулятора, а вполне *реальное* программирование приложений в его среде, обеспечивающее вас как *расширенной* средой программирования, так и стимулирующее более активно знакомиться с ее как возможностями, так и недостатками.

По нашей *Библиотеке* и содержащимся в ней средствам *уместно* сделать несколько замечаний общего характера. Наша *Библиотека* является наглядной иллюстрацией следующего, с позволения сказать, *технологического* процесса использования ***Maple***. В процессе применения пакета для решения различных прикладных задач, проведения различных курсов по пакету и т.д. постепенно нарабатывался определенный *набор законченных* процедур, реализующих часто используемые алгоритмы. Впоследствии из данного набора выбирались наиболее интересные как с точки зрения *приложений*, так и с точки зрения обеспечения *учебного* процесса примерами, содержащими полезные *приемы* программирования в среде пакета. Для удобства данный набор был оформлен в виде библиотеки, аналогичной главной ***Maple***-библиотеке, с простыми способами логической связи ее с главной библиотекой, позволяя использовать ее средства на логическом уровне, аналогично стандартным средствам пакета.

По мере расширения средства библиотеки достаточно широко использовались как для создания различных приложений, так и в учебных целях. Более того, простота подключения к пакету и внутренняя полнота *Библиотеки* делают ее достаточно *мобильной*, позволяя передавать вместе с приложениями, ее использующими. При этом, ввиду и учебной направленности *Библиотека* для ряда средств содержит несколько *версий*, представляющих различные методы реализации и иллюстрирующие различные приемы программирования. Некоторые процедуры, решая поставленные задачи, между тем, не подвергались тщательной оптимизации и работа в этом направлении представляет весьма неплохой стимул для освоения практического программирования на отлаженных примерах, решающих вполне конкретные и нужные задачи. Сама же *Библиотека* дает неплохой пример организации *собственной среды* программирования, дополняющей и расширяющей ***Maple***.

В книгах [41,103] и настоящей представлены средства вышеупомянутой *Библиотеки*, которые как расширяют, так и улучшают стандартные средства пакета релизов **6 – 10**. Эти средства достаточно широко используются и при работе с пакетом в интерактивном режиме, и при программировании в его среде различных задач. Они представляют *несомненный* интерес при *программировании* различных задач в среде ***Maple***, как упрощая сам процесс программирование, так и делая его более прозрачным. В целом же ряде *случаев* предложенные выше средства упрощают работу с пакетом после различных аварийных завершений.

Библиотека **UserLib** прилагается к нашим книгам [41,103], ее демо-версию можно бесплатно загрузить с одного из следующих вебсайтов:

http://aladjev-maple.narod.ru/DemoLib.zip
http://writers.fultus.com/aladjev/book01.html

Тогда как архив с *Библиотекой* и сопутствующими ей материалами можно загружить с одного из следующих вебсайтов:

http://www.aladjev.newmail.ru/DownLoad/UserLib6789.zip
http://writers.fultus.com/aladjev/source/UserLib6789.zip

Еще на одном моменте следует акцентировать *внимание*. К сожалению, с *новациями* в новые релизы привносится и немало ошибок, недоработок и несуразиц. Имеются и просто *вопиющие* несуразицы, когда решаемые в младших релизах ***Maple*** задачи, не решаются в более старших. Примеров тому *немало* и они довольно активно дебатируются пользователями на различных форумах и в группах по пакету. Прискорбно, что *подобное* игнорирование общ-принятых требований к качественному программному обеспечению весьма негативно сказывается на достаточно хорошем в целом пакете.

Поэтому, при обнаружении подобных ситуаций убедительно просим в *любой* из наших адресов выслать четкое описание ситуации (*желательно с примерами в виде* ***mws****-файлов; в Subject-строке следует указать* "*Problems with Maple*"). Это позволит нам не только оказать посильную вам помощь в устранении возникшей ситуации, но и в случае необходимости разработать средства, обеспечивающие *устранение* недоработок и ошибок ***Maple,*** обнаруженных при выполнении нашей *Библиотеки* в среде ***Maple 10–11*** на ***Windows-***платформе. К сожалению, данная непростая ситуация является проблемой не нашей *Библиотеки*, а вопросов обеспечения устойчивого и качественного программирования в среде пакета ***Maple*** в целом. При этом, следует сделать *одно* весьма существенное замечание. В настоящее время мы активно не занимаемся ***Maple***-тематикой, поэтому в наш адрес не ***рекомендуется*** отсылать сообщения следующих двух типов:

(**1**) по частным ошибкам (*например, некорректное вычисление конкретного интеграла и др.*), коими пакет изобилует и количество которых с появлением новых релизов по меньшей мере не уменьшается. В противном случае, только этим нам и пришлось бы заниматься. Бесполезной же констатацией ошибок и несуразиц ***Maple*** мы не считаем нужным заниматься. Такого типа вопросы адресуются в ***MapleSoft,*** хотя это и не очень продуктивно.

(**2**) по *вопросам* применения ***Maple*** для решения конкретных задач (*тематика просто может выходить за рамки наших интересов и компетентности*). Подобные *вопросы* можно обсуждать на соответствующих форумах или в группах в *Internet*.

Нами будут гарантированно рассматриваться лишь вопросы, непосредственно относящиеся к функционированию в том или ином релизе пакета ***Maple*** нашей *Библиотеки*, а также вопросы, носящие общий и концептуальный характер по программной среде пакета и его организации. Именно на такие *вопросы* следует ожидать нашей *реакции* в той или иной форме.

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*.- М.: Изд-во ФилинЪ, 1998, 496 с., ISBN 5-89568-068-2

3. ***Аладьев В.З., Хунт Ю.Я., Шишаков М.Л.*** Основы информатики: *Учебное пособие*. **2**-е изд.- М.: Изд-во ФилинЪ, 1999, 520 с., ISBN 5-89568-068-6

4. ***Аладьев В.З., Гершгорн Н.А.*** Вычислительные задачи на персональном компьютере.- Киев: Изд-во Тэхника, 1991, 248 с.

5. ***Аладьев В.З., Тупало В.Г.*** Алгебраические вычисления на компьютере.- М.: Минтопэнерго, 1993, 251 с., ISBN 5-942-00456-8

6. ***Аладьев В.З., Хунт Ю.Я., Шишаков М.Л.*** Математика на персональном компьютере.- Гомель: Изд-во ФОРТ, 1996, 498 с., ISBN 3-420-614023-3

7. ***Аладьев В.З., Шишаков М.Л.*** Введение в среду пакета ***Mathematica* 2.2.-** М.: Изд-во *ФилинЪ*, 1997, 362 с., ISBN 5-89568-004-6

8. ***Аладьев В.З., Ваганов В.А., Хунт Ю.Я., Шишаков М.Л.*** Введение в среду математического пакета ***Maple V.-*** Минск: Изд-во IAN Press, 1998, 452 с., ISBN 14-064256-98

9. ***Аладьев В., Ваганов В., Хунт Ю., Шишаков М.*** Программирование в среде мтематического пакета ***Maple V***.- Гомель: TRG & Salcombe, 1999, 470 с., ISBN 4-10-121298-2

10. ***Аладьев В., Ваганов В., Хунт Ю., Шишаков М.*** Рабочее место для математика.- Гомель-Таллинн: International Academy of Noosphere, 1999, 605 с., ISBN 3-42061-402-3

11. ***Аладьев В.З., Богдявичус М.А.*** Решение математических и физико-технических задач с пакетом ***Maple V.-*** Вильнюс: Technics Press, 1999, 686 с., ISBN 9986-05-398-6

12. ***Аладьев В.З., Шишаков М.Л.*** **АРМ** математика.- М.: Лаборатория Базовых Знаний, 2000, 751 с. + CD, ISBN 5-93208-052-3

13. ***Аладьев В.З., Богдявичус М.А. Maple 6:*** Решение математических, статистических и инженерно-физических задач.- М.: Изд-во БИНОМ, 2001, 850 с., 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, 2001-2002, CD

15. ***Aladjev V.Z., Bogdevicius M.A.*** Use of package ***Maple V*** for solution of physical and engineering problems **//** Intern. Conf. *TRANSBALTICA-99*, Technics Press, 1999, Vilnius.

16. ***Aladjev V.Z., Hunt U.J.*** Workstation for mathematicians **//** Int. Conf. *TRANSBALTICA-99*, Technics Press, April 1999, Vilnius, Lithuania.

17. ***Aladjev V.Z., Hunt U.J.*** Workstation for mathematicians **//** Int. Con. **"**Perfection of Mechanisms of Management**",** Institute of Modern Knowledge, April 1999, Grodno, Byelorussia.

18. ***Aladjev V.Z., Shishakov M.L.*** Programming in Package ***Maple V*** **// 2nd** Int. Conf. **"**Computer Algebra in Fundamental and Applied Researches and Education**".-** Minsk, 1999.

19. ***Aladjev V.Z., Shishakov M.L.*** A Workstation for mathematicians **// 2nd** Int. Conf. **"**Computer Algebra in Fundamental and Applied Researches and Education**".-** Minsk, 1999.

20. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** Educational computer laboratory of the engineer **//** Proc. **8**th Byelorussia Mathemat. Conf., vol. **3,** Minsk, Byelorussia, 2000.

21. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** Applied aspects of theory of homogeneous structures **//** Proc. **8**th Byelorussia Mathemat. Conf., vol. **4,** Minsk, Byelorussia, 2000.

22. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** Modelling in program environment of the mathematical package ***Maple*** **//** Int. Conf. on Math. Modeling *MKMM-2000*.- *Herson*, 2000.

23. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** A workstation for solution of systems of differential equations **// 3rd** Int. Conf. **"**Differential Equations and Applications**".-** *Sant-Petersburg*, 2000

24. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** Computer laboratory for engineering researches **//** Intern. Conf. *ACA-2000*.– Saint-Petersburg, Russia, 2000.

25. ***Aladjev V.Z., Bogdevicius M.A., Hunt U.J.*** A Workstation for mathematicians **/** *Lithuanian* Conf. *TRANSPORT-2000*.- Vilnius: Technics Press, April 2000, Lithuania.

26. ***Аладьев В.З.*** Компьютерная алгебра **//** Альфа, № **1,** 2001, Гродно, ГрГУ, Беларусь.

27. ***Aladjev V.Z.*** Modern computer algebra for modeling of the transport systems **//** Int. Conf. *TRANSBALTICA-2001*.- Vilnius: Technics Press, April 2001, Lithuania

28. ***Aladjev V.Z., Shishakov M.L., Trokhova T.*** Workstation for the engineer-mathematician **//** Proc. of the GSTU, № **3,** 2000, pp. 42-47, Gomel State University, Gomel, Byelorussia

29. ***Aladjev V.Z., Bogdevicius M.A.*** *Special Questions of Operation in Environment of the Mathematical* ***Maple*** *Package*.- Tallinn-Vilnius: Vilnius Gediminas Technical University, 2001

30. ***Aladjev V.Z., Vaganov V.A., Grishin E.*** *Additional Functional Tools of Mathematical Package* ***Maple 6/7.-*** Tallinn: International Academy of Noosphere, 2002, ISBN 9985-9277-2-9

31. ***Аладьев В.З.*** Эффективная работа с ***Maple 6/7.-*** М.: Лаборатория Базовых Знаний, 2002, 334 с. + CD, ISBN 5-93208-118-X

32. ***Аладьев В.З., Лиопо В.А., Никитин А.В.*** Математический пакет ***Maple*** в физическом моделировании.- Гродно: Гродненский госуниверситет, 2002, ISBN 3-093-31831-3

33. ***Aladjev V.Z., Vaganov V.A.*** *Computer Algebra System* ***Maple:*** *A New Software Library*.- Tallinn: International Academy of Noosphere, 2002, 420 p.+ CD, ISBN 9985-9277-5-3

34. ***Аладьев В., Веетыусме Р., Хунт Ю.*** Общая теория статистики:- Таллинн: TRG & SALCOMBE Eesti Ltd., 1995, 201 с., ISBN 1-995-14642-8

35. ***Аладьев В.З., Хунт Ю.Я, Шишаков М.Л.*** Курс общей теории статистики:- Гомель: Изд-во БЕЛГУТ, 1995, 201 с., ISBN 1-995-14642-9

36. ***Аладьев В.З., Хунт Ю.Я, Шишаков М.Л.*** Вопросы математической теории классических однородных структур.- Гомель: Изд-во БЕЛГУТ, 1996, ISBN 5-063-56078-5

37. ***Ефимова М., Хунт Ю.*** Геометрия рисования: *Графический пакет* ***AutoTouch*** (*под. ред. акад.* ***В.З. Аладьева***).- Гомель: *Российская Академия Ноосферы*, 1997, ISBN 7-14-064254-7

38. ***Aladjev V.Z., Hunt U.J., Shishakov M.L.*** *Scientific-Research Activity of the Tallinn Research Group:* Scientific Report during 1995-1998.- Tallinn-Moscow: TRG, 1998, 80 p.

39. ***Aladjev V.Z., Bogdevicius M.A., Prentkovskis O.V.*** *New Software for Mathematical Package* ***Maple*** *of Releases* ***6, 7*** *and* ***8***.- Vilnius: Vilnius Gediminas Technical University, 2002, 404 p.

40. ***Aladjev V.Z., Hunt U.J., Shishakov M.L.*** *Mathematical Theory of the Classical Homogeneous Structures*.- Tallinn-Gomel: TRG & Salcombe Eesti Ltd., 1998, 300 p., ISBN 9-063-56078-9

41. ***Aladjev V.*** *Computer Algebra Systems:* A New Software Toolbox for ***Maple***.- Palo Alto: Fultus Publishing, 2004, ISBN 1-59682-000-4

42. ***Aladjev V.*** *Computer Algebra Systems:* A New Software Toolbox for ***Maple***.- Palo Alto: Fultus Publishing, 2004, ISBN 1-59682-015-2, Adobe Acrobat eBook (pdf)

43. ***Aladjev V. et al.*** *Electronic Library of Books and Software for Scientists, Experts, Teachers and Students in Natural and Social Sciences*.- Palo Alto: Fultus Publishing, 2005, CD

44. ***Aladjev V.Z.*** *Interactive Course of General Theory of Statistics*.- Tallinn: International Academy of Noosphere, the Baltic Branch, 2001, CD with Booklet, ISBN 9985-60-866-6

45. ***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

46. ***Aladjev V.Z., Bogdevicius M.A., Vaganov V.A.*** *Systems of Computer Algebra:* A New Software Toolbox for ***Maple***. **2nd** *edition*.**-** Tallinn: International Academy of Noosphere, 2004

47. ***Aladjev V.Z., Bogdevicius M.A.*** *Computer algebra system* ***Maple:*** A new software toolbox **//** **4th** Int. Conf. *TRANSBALTICA-03*, Technics Press, April 2003, Vilnius, pp. 458-466.

48. ***Aladjev V.Z., Barzdaitis V., Bogdevicius M.A., Gecys S.*** The solution of the dynamic model of asynchronous engine by finite elements method **// 4th** Int. Con. *TRANSBALTICA-03*, Technics Press, April 2003, Vilnius, Lithuania, pp. 339-352.

49. ***Aladjev V.Z.*** *Computer Algebra System* ***Maple:*** A New Software Library **//** Int. Conf. "Computer Algebra Systems and Their Applications", *CASA-2003*, Saint-Petersburg, 2003.

50. ***Aladjev V., Bogdevicius M., Vaganov V.*** Systems of Computer Algebra: A New Software Toolbox for ***Maple*** **//** Int. Conf. on Software Engin. Res. and Practice, 2004, Las Vegas

51. http://www.aladjev.newmail.ru, http://www.geocities.com/noosphere_academy

52. ***Owen D.B.*** *Handbook of Statistical Tables*.- London: Addison-Wesley Publishing Co., 1963

53. ***Kelley T.L.*** *The Kelley Statistical Tables*.- Cambridge: Harvard University Press, 1948.

54. ***Голоскоков Д.П.*** Уравнения математической физики. Решение задач в системе ***Maple***.- Санкт-Петербург: Изд-во Питер, 2004

55. ***Васильев А. Н. Maple 8.*** Самоучитель:- М.: Диалектика, Вильямс, 2003.

56. ***Кирсанов М.*** Решебник. Теоретическая механика.- М.: Физматлит. 2002.

57. ***Очков В.*** Физические и экономические величины в ***Mathcad*** и ***Maple***.- М.: ФиС, 2002

58. ***Говорухин В., Цибулин В.*** Компьютер в математическом исследовании: ***Maple, MATLAB, LaTeX.-*** Санкт-Петербург: Изд-во Питер, 2001

59. ***Матросов А. Maple 6:*** Решение задач высшей математики и механики.- *Санкт-Петербург*: Изд-во БХВ-Петербург, 2001

60. ***Манзон Б. Maple V*** Power Edition.- М: Изд-во ФилинЪ, 1998

61. ***Прохоров Г., Леденев М., Колбеев В.*** Пакет символьных вычислений ***Maple***.- М: Изд-во Петит, 1997

62. ***Говорухин В., В.Цибулин В.*** Введение в ***Maple***. Математический пакет для всех.- М.: Изд-во Мир, 1997

63. ***Statistical Tools for Finance and Insurance /*** *Eds.* **P. Cizek, W. Hardle, R. Weron.-** Berlin: Springer-Verlag, 2004, ISBN 3-540-22189-1

64. ***Good R.*** *Permutation, Parametric, and Bootstrap Tests of Hypotheses*.- N.Y. Springer, 2005.

65. ***Scherer B, Martin R.*** *Introduction to Modern Portfolio Optimization with* ***NUOPT & S-Plus***.- Berlin: Springer-Verlag, 2005, ISBN 0-387-21016-4

66. ***New Developments in Classification and Data Analysis /*** *Eds.* **M. Vichi et al.-** Paris: Springer-Verlag, 2005, ISBN 3-540-23809-3

67. Zivot, Wang J. *Modeling Financial Time Series with* ***S-Plus***.- N.Y.: Springer-Verlag, 2004.

68. ***Statistical Tools for Finance and Insurance /*** *Eds.* **P**. **Cizek et al.-** Springer-Verlag, 2004.

69. ***Dekking F.M. et al.*** *A Modern Introduction to Probability and Statistics*.- Springer, 2005.

70. ***Лакин Г.Ф.*** Биометрика.- Москва: Изд-во «Высшая школа», 1990

71. ***CRC Standard Mathematical Tables and Formulae /****Ed.* **D. Zwillinger.-** Springer, 1995

72. ***Encyclopedia of Statistical Sciences /****Eds.* **S. Kotz & N. Johnson,** v. **1-9**.- N.Y.: Wiley, 1995

73. ***Aladjev V.Z., Haritonov V.N.*** *General Theory of Statistics*.- Palo Alto: Fultus Corp., 2004.

74. ***Balakrishnan N., Chen W.*** *CRC Handbook of Tables for Order Statistics*.- Springer, 1997

75. ***Кильдишев Г.С.*** Общая теория статистики.- Москва: Изд-во «Статистика», 1980

76. ***Portela A., Charafi A.*** *Finite Elements Using* ***Maple:*** *A Symbolic Programming Approach*.- London- Berlin-Paris: Springer, 2002, 320 p. + CD, ISBN 3-540-42986-7.

77. ***Cyganowski S., Kloeden P., Ombach J.*** *From Elementary Probability to Stochastic Differential Equations with* ***Maple***.- Berlin-London: Springer-Verlag, 2002, 310 p., ISBN 3-540-42666-3.

78. ***Corless R.M.*** *Essential* ***Maple 7:*** *An Introduction for Scientific Programmers*.- Springer-Verlag, 2002, 305 pp., ISBN 0-387-95352-3.

79. ***Monagan M. et al. Maple 6:*** *Programming Guide*.- Waterloo: Waterloo Maple Inc., 2000

80. ***Redfern M., Betounes D.*** *Mathematical Computing: An Introduction in Programming Using* ***Maple***.- Hattiesburg: Springer-Verlag, 2002, 420 pp.

81. ***Maple 8*** *Learning Guide*.- Toronto: Waterloo Maple Inc., 2002, 308 pp.

82. ***Maple 8*** *Introductory Programming Guide*.- Toronto: Waterloo Maple Inc., 2002, 380 pp

83. ***Maple 8*** *Advanced Programming Guide*.- Toronto: Waterloo Maple Inc., 2002, 382 pp.

84. ***DeMarco P. et al. Maple*** *Advanced Programming Guide*.- Waterloo Maple Inc., 2005

85. ***DeMarco P. et al. Maple*** *Introductory Programming Guide*.- Waterloo Maple Inc., 2005

86. ***Abell M., Braselton J. Maple*** *by Example*, **3**rd Edition.- Berlin: Springer, 2005

87. ***Corless R.*** *Symbolic Recipes: Scientific Computing with* ***Maple.-*** Waterloo Maple Inc., 2005

88. ***Adams P. et al.*** *Introduction To Mathematics With* ***Maple.-*** Waterloo Maple Inc., 2004

89. ***Maple 9.5*** *Getting Started Guide*.- Waterloo Maple Inc., 2004

90. ***Enns R., McGuire G.*** *Computer Algebra Recipes* + CD *with* ***Maple*** *Softcover*.- Berlin, 2006

91. **http://www.grsu.by/cgi-bin/lib/lib.cgi?menu=links&path=sites**

92. ***Aladjev V.Z.*** Recent Results in the Mathematical Theory of Homogeneous Structures **/** Trends, Techniques and Problems in Theoretical Comp. Science **//** Lecture Notes in Computer Science, Band **281.-** Heidelberg: Springer-Verlag, 1986, p. 110-128.

93. ***Aladjev V.Z.*** Homogeneous Structures in Mathematical Modeling **//** Proc. Sixth Intern. Conf. on Mathem. Modelling.- Sant-Louis: Washington University, USA, 1987.

94. ***Aladjev V.Z.*** Recent Results in the Theory of Homogeneous Structures **//** Parallel Processing by Cellular Automata and Arrays.- Amsterdam: North-Holland, 1987, 31-48.

95. ***Aladjev V.Z.*** Unsolved Theoretical Problems in Homogeneous Structures **//** Mathem. Res., Band **48.-** Berlin: Akademie-Verlag, 1988, p. 33-49.

96. ***Aladjev V.Z.*** Survey on Some Theoretical Results and Applicability Aspects in Parallel Computation Modeling **//** Journal New Generation Comput. Systems, **1,** no. **4,** 1988.

97. ***Aladjev V.Z.*** A Solution of the Steinhays`s Combinatorical Problem **//** Appl. Mat. Letters, no. **1,** 1988, p. 11-12.

98. ***Aladjev V.Z.*** Recent Results in the Mathematical Theory of Homogeneous Structures **//** New Trends in Computer Sciences.- Amsterdam: North-Holland, 1988, p. 3-54.

99. ***Aladjev V.Z.*** An Algebraical System for Polinomial Representation of **K**-Valued Logical Functions **//** Applied Mathem. Letters, no. **3,** 1988, p. 207-209.

100. ***Aladjev V.Z.*** Interactive Program System for Modelling of Homogeneous Structures **//** **7th** Intern. Conf. on Mathem. and Comp. Modelling, Chicago, USA, 1989.

101. ***Aladjev V.Z. et al.*** *Theoretical and Applied Aspects of Homogeneous Structures* **//** Proc. Inter. Workshop *PARCELLA-90*.- Berlin: Akademie-Verlag, 1990, p. 48-70.

102. ***Aladjev V.Z.*** Homogeneous Structures: Theoretical and Applied Aspects **//** The **8**th Int. Conf. on Mathem. and Comput. Modelling, Washington University, USA, 1991.

103. ***Аладьев В.З.*** Системы компьютерной алгебры: ***Maple:*** *Искусство программирования*.- М.: Лаборатория Базовых Знаний, 2006, 792 с., ISBN 5-93208-189-9

104. ***Aladjev V.Z.*** *Encyclopedia of Classical Homogeneous Structures* (*in preparation*)

105. ***Solving Problems in Scientific Computing Using Maple and MATLAB /*** *Eds.* ***Gander W.*** *and* ***Hrebicek J.-*** Zurich-Brno: Springer-Verlag, 2006, 476 p., ISBN 3-540-21127-6

106. ***Kay S.*** *IntuitiveProbability and Random Processes using* ***MATLAB.-*** London: Springer, 2006

107. ***Betounes D.*** *Differential Equations: Theory and Applications with* ***Maple.-*** N.Y., 2001

108. http://www.aladjev-maple.narod.ru/DemoLib.zip

109. http://www.aladjev.newmail.ru/DownLoad/UserLib6789.zip или **http://writers.fultus.com/aladjev/source/UserLib6789.zip**

110. ***Aladjev V.Z., Bogdevicius M.A. Maple:*** *Programming, Physical and Engineering Problems.* CA: Palo Alto, Fultus Corp., 2006, 404 pp., ISBN 1-59682-080-2

111. ***Аладьев В.З.*** Основы программирования в ***Maple.-*** Таллинн: Международная Академия Ноосферы, 2006, 300 с., ISBN 9985-9808-1-X, 978-9985-9508-1-4, **http://www.aladjev-maple.narod.ru**

112. http://www.maplesoft.com/books/books_detail.aspx?isbn=1596820004

113. http://www.maplesoft.com/books/books_detail.aspx?isbn=1596820802

114. ***Pinter J.D.*** *Applied Nonlinear Optimization in Modeling Environments*.- N.Y.: CRC Press, 2007, ISBN 0849316235

115. ***Wang F.*** *Physics with* ***Maple:*** *The Computer Algebra Resource for Mathematical Methods in Physics*.- N.Y.: John Wiley & Sons, 2006, ISBN 3527406409

116. ***Enns R.H., McGuire G.C.*** *Computer Algebra Recipes: An Introductory Guide to the Mathematical Models of Science*.- London: : Springer, 2006, ISBN 0387257675

117. Putz J. ***Maple*** *Animation*.- N.Y.: CRC Press, 2003, ISBN 1-584-88378-2

118. ***Herman E.A., Pepe M.*** *Visual Linear Algebra*.- N.Y.: Wiley Press, 2005, ISBN 0-471-68299-3

119. ***Howell R.W., Mathews J.H.*** *Complex Analysis for Mathematics and Engineering*.- N.Y.: Jones & Bartlett Publ., 2006, ISBN 0763737488

120. ***Rasinariu C., Aratyn H.*** *A Short Course in Mathematical Methods with* ***Maple.-*** London: World Scientific Publ. Co., 2005, ISBN 9812564616

121. ***Proceedings of Maple Conference 2005 /*** *Ed.* ***Kotsireas I.S.-*** Waterloo: MapleSoft Inc., 2005

122. ***Aziz A.*** *Heat Conduction with* ***Maple.-*** N.Y.: R.T. Edwards Publ., 2005, ISBN 1-930217-15-3

123. ***Lopez R.*** *Advanced Engineering Mathematics with* ***Maple.*** Electronic Book.- Waterloo: MapleSoft Inc., 2005, ISBN 1-894511-86-7

124. ***DeMarco P. et al. Maple*** *Introductory Programming Guide*.- Waterloo: MapleSoft, 2005

125. ***Baker B., Bradie B., Packer A.*** *An Introduction to Numerical Analysis*.- London: Pearson Education Publ., 2005, ISBN 0131911716

126. ***Lee H.J., Schiesser W.E.*** *Ordinary and Partial Differential Equation Routines in* ***C, C++, Java, Fortran, Maple*** *and* ***MATLAB.-*** London: Chapman & Hall/CRC, 2003, ISBN 1584884231

127. ***Rovenskii V., Rand O.*** *Symbolic Analytic Methods in Anisotropic Elasticity with Symbolic Computational Tools*.- N.Y.: Birkhauser Press, 2003, ISBN 0-8176-4272-2

128. ***Singh K.*** *Engineering Mathematics Through Applications*.- N.Y.: Springer, 2003

129. ***Abel M. et.al.*** *Statistics with* ***Maple.-*** N.Y.: Elsevier Science & Technology, 2002

130. ***Borowski E. et. al.*** *Interactive Math Dictionary: The Math Resource on CD-ROM*.- Berlin: Springer-Verlag, 1998, ISBN 3-540-14650-4

АЛАДЬЕВ ВИКТОР ЗАХАРОВИЧ

Аладьев В.З. родился 14.06.1942 в г. Гродно (*Беларусь*). После успешного завершения **2**-й *средней* школы (*Гродно*) в 1959 поступил на **1**-й курс физико-математического факультета Гродненского университета, в 1962 был переведен на отделение «*Математики*» Тартусского университета (*Эстония*). В 1966 успешно закончил Тартуский университет по специальности «*Математика*»**.** В 1969 поступил в аспирантуру Академии Наук Эстонии по специальности «*Теория вероятностей и математическая статистика*»**,** которую *успешно* закончил в 1972 сразу по двум специальностям «*Теоретическая кибернетика*» и «*Техническая кибернетика*»**.** Ему была присвоена докторская степень по математике за первую монографию «*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 занимал ответственные посты в ряде проектно-технологических и исследовательских организаций г. Таллинна. В 1991 г. **Аладьев В.З.** организовал научно-прикладную фирму VASCO Ltd., а с конца 1992 г. становится вице-президентом Salcombe Eesti Ltd.

Аладьев В. является автором более **350** научных и научно-технических работ (*включая* **65** *монографий, книг и сборников статей*), опубликованных в бывшем *СССР*, *России*, *ФРГ*, *Белоруссии*, *Эстонии*, *Литве*, *Украине*, *ГДР*, *Чехословакии*, *Венгрии*, *Японии*, *США*, *Голландии*, *Болгарии* и *Великобритании*. С 1972 г. является референтом и членом редколлегии международного математического журнала «*Zentralblatt fur Mathematik*» и с 1980 г. – членом **IAMM**. Им создана Эстонская школа по математической теории однородных структур, результаты которой получили международное признание и легли в основу нового раздела современной математической кибернетики.

В 1993 **Аладьев В.** 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 г. **Аладьев В.** по совокупности научных работ в области *кибернетики* избран академиком Российской Академии Космонавтики по отделению «*Фундаментальных исследований*»**,** в сентябре же 1994 г. он избирается академиком Российской Академии Ноосферы по отделению «*Информатики*». В сентябре 1995 **Аладьев В.** избирается действительным членом *Российской Академии Естественных Наук* (***РАЕН***) по отделению «*Ноосферные знания и технологии*», а в 1998 – академиком Российской Экологической Академии.

В ноябре 1997 **Аладьев В.** избран академик-секретарем *Балтийского* отделения Российской Академии Ноосферы, объединяющего ученых и специалистов трех 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 **Аладьев В.** избирается ее Первым вице-президентом. В конце 1999 **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 др.

БОЙКО ВАЛЕРИЙ КОНСТАНТИНОВИЧ

Бойко В.К. родился 10.09.1950 в городе Гродно (*Беларусь*). В 1967 окончил среднюю школу и поступил на физико-математический факультет Гродненского государственного педагогического института имени Я. Купалы, который с отличием окончил в 1971 г. С 1971 по 1976 г.г. 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 с 1976 по 1983 г.г. работал ассистентом, старшим преподавателем, доцентом кафедры дифференциальных уравнений Гродненского государственного университета имени Я. Купалы.

В 1983 г. **Бойко В.К.** оканчивает курсы французского языка и с 1984 по 1987 г.г. по направлению Министерства Образования СССР работает доцентом кафедры математики строительного института в городе Эль-Аснам (*Алжир*).

По возвращении **Бойко В.К.** продолжает работу *доцентом* на кафедре дифференциальных уравнений Гродненского государственного университета имени Я. Купалы. Активно занимается научной работой, является ответственным исполнителем ряда научных тем, работает заместителем декана по научной работе. С 1998 г. и по настоящее время является деканом факультета математики и информатики Гродненского государственного университета имени Я. Купалы.

Основные научные результаты **Бойко В.К.** относятся к теории 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 в теоретических исследованиях, а также при решении конкретных 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 факультет переименован в факультет *математики* и *информатики*.

Бойко В.К. является автором более **70** научных и научно-методических работ (*в том числе* **1** *монография и* **5** *методических указаний*), опубликованных в СССР, Беларуси, России, Украине и Польше.

РОВБА ЕВГЕНИИ АЛЕКСЕЕВИЧ

Ровба Е.А. родился 01.04.1949 в деревне Андрошевщина 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 по 1979 г.г. работал старшим преподавателем, доцентом, заведующим кафедрой математического факультета Гомельского государственного университета.

В 1979 **Ровба Е.А.** переехал на работу в Гродненский государственный университет им. Я. Купалы (**ГрГУ**), где работал с 1979 г. доцентом, а с 1993 г. заведовал кафедрой теории функций, функционального анализа, вероятностей и прикладной математики. В 1999 защитил докторскую диссертацию на тему «*Интерполяция и ряды Фурье в рациональной аппроксимации*», которая легла в основу одноименной монографии, опубликованной в 2001 г. Работал заместителем декана и деканом математического факультета, проректором университета по учебной работе. С 1986 по 1988 г. – консультант Министерства образования Республики Куба. С 1996 по 1998 г. работал деканом Гродненского филиала специального факультета бизнеса и информационных технологий **БГУ**. С 1998 по 2000 годы – ректор Института повышения квалификации и переподготовки руководящих работников Гродненского университета. С 2000 г. – профессор, первый проректор университета. В 2005 г. был назначен ректором **ГрГУ**.

Основные научные исследования **Е.А. Ровбы** относятся к теории рациональных приближений функций. Им построены интерполяционные и интегральные (*на отрезке*) рациональные операторы, обладающие классическими с точки зрения конструктивной теории функций аппроксимационными характеристиками. Найден подход к построению обобщений квадратурных формул Гаусса на базе ортогональных систем рациональных функций и интерполяции. Получены оценки уклонений рациональных операторов Фурье и Валле-Пуссена на классах функций, имеющих дробную **n**-ю производную ограниченной вариации. Впервые интерполирование по специальным системам узлов применено 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. Ровбы** оказала большое влияние на развитие образовательной среды Гродненской области (*Беларусь*). **Ровба Е.А.** инициировал создание в Гродно полнокровной самофинансируемой образовательной структуры в сфере последипломного образования Гродненского филиала Специального факультета бизнеса и информационных технологий **БГУ**. Полученный опыт был *успешен* и стал основой для создания еще одной инновационной образовательной структуры – Института повышения квалификации и переподготовки руководящих работников (*ныне Институт последипломного образования* **ГрГУ**). Уже будучи первым проректором университета, **Е.А. Ровба** способствовал открытию и становлению лицеев, гимназий, созданию новых учебно-методических объединений в Гродненской области.

Ровба Е.А. является автором более **100** научных и научно-методических работ (*включая* **2** *монографии и* **4** *учебных пособия*), опубликованных в СССР, Беларуси, Болгарии, Кубе, России, Великобритании и Украине.