Международная Академия Ноосферы

В.З. Аладьев

Основы программирования в Maple

Таллинн - 2006

УДК [658.012.011.56.005:681.3]+681.325.5-181.4
ISBN 9985-9508-1-X, 978-9985-9508-1-4

В.З. АЛАДЬЕВ. *Основы программирования в Maple. Таллинн*, 2006.

Книга вводит в программную среду известного математического пакета ***Maple***, представляющего собой одну из наиболее развитых современных *систем компьютерной алгебры* (**CAS**). Книга представляет достаточно детальное введение в среду встроенного ***Maple***-языка программирования, позволяющего пользователю не только четко представить все возможности пакета, но и решать в его среде достаточно сложные прикладные задачи из многих разделов техники, математики, физики, химии и других естественно-научных дисциплин, для решения которых пакет не имеет стандартных средств. Более того, ***Maple***-язык может оказаться весьма эффективным средством в системе преподавания указанных и, в первую очередь, математических дисциплин. Именно в данном направлении он может получить признание не меньшее, чем в *среде* многочисленных исследователей естественно-научных дисциплин, широко использующих математические методы.

Представленный в книге 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 возможности встроенного ***Maple***-языка. Ряд представленных в книге приемов может оказаться полезным при формировании эффективной концепции программирования в его среде.

Все это делает материал книги 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 числовых и символьных вычислений, работы с графическими объектами и программирования на высокоуровневом процедурном языке, прежде всего, задач, носящих математический характер с акцентом на символьных (*алгебраических*) вычислениях.

Книга является одним из немногих в отечественной литературе изданием по программированию в среде пакета ***Maple***, что и определяет ее место среди литературы по программным средствам для **ПК**, использующих операционную среду ***Windows***. Вместе с тем мобильность пакета позволяет использовать его многими другими популярными платформами и типами **ЭВМ**. Книга рассчитана на достаточно широкий круг специалистов, использующих в своей профессиональной деятельности **IBM**-совместимые **ПК**, а также ряд других компьютерных платформ для решения задач математического характера, а также студентов и учащихся по курсу «*Основы информатики и вычислительной техники*» физико-математических и других естественно-научных специальностей соответствующих университетов и колледжей.

Настоящая книга представляет собой авторский оригинал-макет конспекта *курсов* лекций по программированию в среде ***Maple***, проведенных в *ряде* университетов Прибалтики и СНГ, и не подвержен тщательному редактированию. Однако, на наш взгляд, представленный материал может быть достаточно полезен пользователям ***Maple***, для решения задач которых требуется их непосредственное программирование в среде пакета.

ISBN 9985-9508-1-X, 978-9985-9508-1-4

aladjev@yandex.ru, aladjev@gmail.com, valadjev@yahoo.com
http://www.aladjev.newmail.ru/, http://www.aladjev.narod.ru/

Содержание

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, Axiom, Reduce*** и ***Magma*** становятся все более популярными для решения задач преподавания математически ориентированных дисциплин, в научных исследованиях и 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 задач различного назначения.

В серии наших работ [1-20, 22-33, 39, 41-46, 47, 49, 50, 91, 103] достаточно детально рассмотрены такие математические пакеты как ***Maple, Reduce, MathCAD, Mathematica***. При этом, особое внимание нами уделелялось особенностям каждого из пакетов, его преимуществам и 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 пакеты ***Maple*** и ***Mathematica*** в качестве бесспорных лидеров (*на основе специального обобщенного индекса*) среди всех известных на сегодня современных **СКА**. Между тем, 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 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 пакет ***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 достаточно 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 органично с ней связанной. Комментированное описание данной Библиотеки представлено в нашей последней книге [103]. К ней же и прилагается данная Библиотека. Демо-версию нашей *Библиотеки* можно бесплатно загрузить с адреса, указанного в [91].

Более того, 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 для средств Библиотеки относительно релизов **6** - **10**.

В заключение данной преамбулы весьма кратко 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,*** 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*** довольно 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 быстро 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 своего опыта достаточно глубокого использования и апробирования обоих пакетов отмечу, 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 популярных 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 **3** 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, Sun Microsystems, Ford, Hewlett Packard, General Electric, Daimler-Chrysler* и др.

Резюмируя сказанное (*более детальный сравнительный анализ обоих пакетов может быть найден 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 достаточно приличную **СКА – *AXIOM,*** являющуюся на сегодня лидером среди **СКА** на европейском уровне. При этом, пакет ***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.

Довольно существенные ошибки и недоработки (*многие из них неоднократно отражались в наших книгах и статьях, а также во многих 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 книгах и статьях. Кое-что для усовершенствования совместимости нами было сделано (*в частности, посредством нашей библиотеки, представленной в книге* [103]), однако не все. Тем временем, для ***Maple*** релизов **9** и **10** была обнаружена несовместимость уже среди их клонов. Как 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 эти 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 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 последствиям, крайне нежелательным для ***Maple.*** Более того, нам кажется, что их действие уже начинает сказываться.

Ввиду сказанного, упомянутая Библиотека корректно работает с ***Maple*** релизов **6-8** и ***Maple*** **9-10** (*классический режим*), тогда как для ***Maple*** **9 - 10** (*стандартный режим*) некоторые библиотечные средства, использующие *стандартную* функцию ***system*****,** будут выпоняться некорректно или вызывать ошибочные ситуации, в ряде случаев непредсказуемые. В этой связи заинтересованный читатель в качестве довольно 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 *Библиотеки* [103,108].

Характеристика нашей библиотеки программных средств. Упоминаемая здесь *Библиотека* расширяет диапазон и эффективность использования ***Maple*** на платформе ***Windows*** благодаря содержащимся в ней средствам в *трех* основных направлениях: (**1**) *устранение ряда основных дефектов и недостатков*, (**2**) *расширение возможностей целого 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], представляющая разработанные нами средства и содержащая предложения по дальнейшему развитию пакета, в значительной степени стимулировала появление таких приложений как пакетные модули **FileTools, LibraryTools, ListTools** и **StringTools.** Между тем, и в свете данных приложений средства нашей *Библиотеки* существенно расширяют возможности пакета, во многих случаях перекрывая средства указанных пакетных модулей. Текущая версия *Библиотеки* содержит набор средств (*более **700** процедур и программных модулей*), ориентируемых на следующие основные виды обработки информации и вычисления [103,108]:

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 последней книге [103] и на страницах **http://www.aladjev.narod.ru/MapleBook.htm**, и **http://www.exponenta.ru/educat/news/aladjev/book2.asp**. Исходя из нашего многолетнего опыта использования пакета ***Maple*** релизов **4 - 10** и опыта наших коллег из университетов и академических институтов России, Эстонии, Белоруссии, Литвы, Латвии, а также ряда других стран, следует отметить, что многие из средств (*либо их аналоги*) нашей Библиотеки весьма целесообразно включить в стандартные поставки последующих релизов ***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**, **LibraryTools**, **ListTools** и StringTools. Однако, в этом отношении средства, представленные нами, существенно расширяют возможности пакета, во многих случаях превышая таковые из указанных пакетных модулей. В наст-ящее же время они доступны пользователям пакета в виде предлагаемой Библиотеки, функционирующей на платформах ***Windows*** и поддерживающей релизы **6 - 10** пакета. Данная Библиотека прилагается к нашей книге [103]. Средства Библиотеки в целом ряде случаев позволяют существенно упрощать программирование различных 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 **16** MB свободного пространства на жестком диске. Условия получения данной Библиотеки, прилагаемой к книге [103], и ее последующих обновлений приведены в тексте указанной книги. Сопутствующие материалы содержат немало дополнительной полезной информации, которая по тем либо иным причинам не 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 ***Maple*** и делать 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 умышленно с целью иллюстрации ряда особенностей и возможностей языка программирования ***Maple.*** Это будет весьма полезно при освоении *практического* программирования в ***Maple.***

Следует отметить, что поставляемые с Библиотекой файлы "**ProcUser.txt**" (*для* ***Maple*** **6 – 9**) и "**ProcUser10.txt**" (*для* ***Maple*** **10**), содержащие исходные тексты всех 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 возможность адаптации ряда процедур нашей ***Maple***-библиотеки к программной среде других пакетов. В частности, целый ряд процедур Библиотеки достаточно легко был 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 найдет среди средств Библиотеки полезные для своего творчества.

В предлагаемой книге рассматриваются основы 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 об-

суждаются 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 достаточно 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 детально рассмотрены в [103].

Двухуровневая лингвистическая поддержка пакета ***Maple*** обеспечивается такими языками программирования как ***C*** и ***Maple.*** В некоторых публикациях я встречал иную (*не совсем обоснованную на мой 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 моим 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 реализовать на его основе интуитивный прин-

цип 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 или 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 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*** 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 знание их и весьма предпочтительно.

Программирование в среде ***Maple***-языка в большинстве случаев не требует какого-то особого программистского навыка (*хотя его наличие и весьма нелишне*), т.к. в отличие от других языков универсального назначения и многих проблемно-ориентированных ***Maple****-язык* включает большое число математически ориентированных функций и процедур, позволяя только одним вызовом решать достаточно сложные самостоятельные задачи, например: решать си-

стемы 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*) средств, 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 **700** 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]. 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 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 среде ***Windows*** 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 знаний, 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. 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 средства ***Maple***-языка для работы с данными и структурами символьного, строчного, списочного, множественного и табличного типов, включая средства работы с выражениями строчного и символьного типов, множествами, списками и таблицами. Рассматриваются алгебраические правила подстановок для символьных вычислений и др.

Третья 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.

Четвертая глава достаточно детально рассматривает одну из ключевых *образующих* модульного программирования – *процедуры* и средства их организации в среде ***Maple***-языка. Представлены как базовые так и расширенные средства наряду с вопросами расширения функциональных средств ***Maple***-языка. Достаточно 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.

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

Шестая 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 здесь и такие важные средства пакета как *графические* и *обеспечения доступа* к файлам данных. Мотивировано данное решение тем, что эти средства достаточно 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, FORTRAN, PL/1, Pascal, Ada, Basic*** и т. д. В то же самое время ***Maple***-язык, ориентированный, прежде всего, на решение задач математического характера, предоставляет набор средств для доступа к файлам данных, которые могут вполне удовлетворить достаточно 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 и назначения.

Со всей определенностью можно констатировать, что *новые* пакетные модули **LibraryTools** и **FileTools** 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 достаточно 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 настоящей 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, это вполне уместно и в том потребительском контексте, что *основная* масса цитируемых моих изданий по ***Maple***-тематике может быть *совершенно бесплатно* получена с указанного в [91] *Internet*-адреса Белорусского университета. Конечно, там представлены издания, ориентированные на ***Maple* 5 – 7**, однако ввиду полной приемственности по основным 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. В будущем данная тематика не планируется в числе моих 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 смею надеяться, что и в таком виде оно представит определенный интерес для начинающих программистов в среде ***Maple***, да и не только.

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* Белорусского университета. Там же можно найти и массу полезных примеров.

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

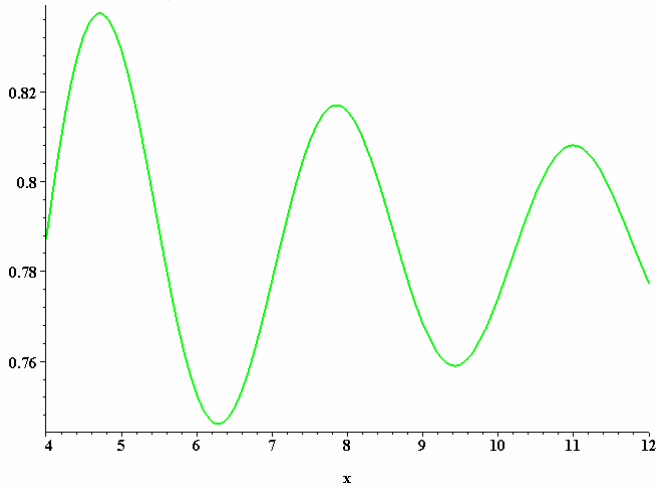
- **ПС** – программное средство;
- **ПО** – программное обеспечение;
- **ПК** – персональный компьютер;
- альтернативные элементы разделяются символом “ | “ и заключаются в фигурные скобки, например, {**A** | **B**} – **A** или **B**;
- смысл конструкции указывается в угловых скобках “**<** ... **>**“;
- 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;
- **ГМП** (**GUI**)- *главное меню пакета;* содержит группы основных 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 обозначения, понятия, сокращения и соглашения будут вводится по мере надобности по тексту книги. Тогда как с нашими последующими разработками и изданиями как по данной проблематике, так и научно-прикладной активности *Балтийского отделения Международной Академии Ноосферы* в целом можно периодически знакомиться на ***Web***-странице:

http://www.aladjev.newmail.ru

На этой же странице находятся ***email***-*адреса*, в которые можно отправлять все замечания, пожелания и предложения, относящиеся к материалу предлагаемой книги. Все они будут приняты с благодарностью и без нашего внимания не оставлены.

Таллинн, октябрь 31, 2006

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 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 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. Интерактивность же 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, библиотечных, модульных*) и прикладных (***Maple****-документов*) средств, созданных к настоящему времени пользователями из различных прикладных областей, прежде всего физико-математических и технических.

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 по-

льзователя, но и расширяющие средства самого пакета. Примеры такого типа представлены в нашей последней книге [103]. Именно поэтому далее представим 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 достаточно 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 на-писан 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 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 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.

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: находить минимакс выражения, решать системы диффуравнений или алгебраических уравнений, вычислять производные и интегралы, выводить графики сложных функций и др. Интерактивность же языка обеспечивает простоту его освоения и удобство редактирования и отладки прикладных ***Maple***-документов и программ. Реальная же мощь языка обеспечивается не только его управляющими структурами и структурами данных, но и всем богатством его функциональных (*встроенных, библиотечных, модульных*) и прикладных (***Maple***-*документов*) средств, созданных к настоящему времени пользователями из различных прикладных областей, прежде всего математических. Тут же уместно отметить, что значительная часть функциональных средств самого пакета написана и скомпилирована именно на ***Maple***-языке, позволившим создать достаточно эффективные относительно основных ресурсов **ЭВМ** загрузочные модули. Анализ этих средств является весьма неплохим подспорьем при серьезном освоении ***Maple***-языка.

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] 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 либо установкой в место ошибки **|**-*курсора*, как это иллюстрирует нижеследующий простой 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.

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*. Допустимые 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 представленного 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.

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 простой фрагмент иллюстрирует применение *нульарных* операторов языка**:**

```
> 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, обеспечивающей ***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 и для 32-битных **ЭВМ** составляет 524271 символов, а на 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 следующий 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 `;`
```

Сразу же следует отметить одно принципиальное отличие *строк* от *символов* ***Maple*-языка**. Если символам можно присваивать значения, то строки такой процедуры не допускают, вызывая ошибочные ситуации, например:

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

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 вычислительной 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 функциональными 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*

Кратко 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 использование 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. Данное средство можно использовать для комментирования текстов ***Maple***-программ, тогда как в интерактивном режиме особого смысла оно 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\nbbbbbbbbbbb\ncccccccccccc`;   # Перевод строки
A := aaaaaaaaaaa
     bbbbbbbbbbb
     cccccccccccc
> A:= `aaaaaaaaaa\a\nbbbbbbbbbb\a\ncccccccccccc`; # Звонки с переводом строки
A := aaaaaaaaaa•
     bbbbbbbbbb•
     cccccccccccc
> 1942.1949\1959\1962\1966\1972\1995\1999\2006;
                    1942.19491959196219661972199519992006
```

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 следует 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 ***kernelopts*(*dirsep*)** возвращается стандартный *разделитель подкаталогов*, принятый 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*, идентифицирующая 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* 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 ради удобства представления иллюстративных примеров программную строку и результат ее вычисления будем иногда представлять в следующем достаточно естественном виде

> *Maple-предложение* {**:**|**;**} ⇒ ***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
```

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 пакета, обеспечивающей вывод справочной информации по указанным аргументами средствам пакета. Например, по конструкции формата "**> ?integer**" выводится информация ***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 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 регистрозависимы, 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 предназначены, 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 символпм ` `, `  `, `  ` 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 вида:

***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 соотношение для обоих методов конкатенации:

***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, например:

```
> 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 обстоя-

тельством, 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 несколько 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 виде:

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 простом виде:

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*** - основной идентификатор, 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 книгах [10-12,91].

Близкой по назначению к ***alias***-функции является и встроенная 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 ***macro****-определения* 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 сказанное:

```
> 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]);
                                [42 x^3 sin(x)^3, 42 (a b + c)^3, 7375872]

> 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 **restart**-*предложению*, приводящего 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 (*процедуры, функции 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 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

```
> &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);
                                  (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', "(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;
                                  (a+b)/(c-d), (a+b)/(c-d), (a+b)/(c-d), (a+b)/(c-d), (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. Ли-

бо 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')** можно 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^3 + z/y - 99 x y + sin(t), z/y, -99 x y, x, y, z, t }

> 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 ***evaln***-функции 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 вычисление предложения ***Asv:=evaln(Asv)*** делает ***Asv***-идентификатор вновь *неопределенным*. *Вторым* способом приведения идентификатора к неопределенному состоянию является использование конструкции вида ***Id:= 'Id'***, как это иллюстрирует второй пример 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** – идентификатор, **B** - любое допустимое выражение и **C** - список, множество или 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];      # Maple 7 и выше
                                        [64]
> 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 констукциях, 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 идентификаторы 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 ***unassign***-процедуре 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 и 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.

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 ***protected***-атрибута. Если идентификатору не приписано 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*** языка ***Maple*** 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*) не рекомендуется делать.

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 выборку 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 ключевых слов как ***environment***, ***user***, ***alluser***, с назначением которых можно ознакомиться по справочной базе пакета.

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 сказанное:

```
> 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. 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 ***interface***-процедуры, 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 весьма простой фрагмент:

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

$V + G$

```
> 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');   ⇒  true
> x:= 'x': unassign('y'), map(is, [x, y], 'integer'), is(z, 'nonnegative');    ⇒  [false, false], true
```

До **7**-го релиза включительно *оперативно* переопределять режим идентификации ***assume***-переменных 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 ***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 либо к справке по пакету, либо к книгам [8-14,78-86,88,103,105].

Механизм *свойств*, определяемый ***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, определяющей *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.

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

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 либо 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 простой пример:

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

$$+,\ 3\sin(x),\ \frac{5}{32}\cos(y),\ \sqrt{AG^2+AS^2}\ TRG$$

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

• ***Целые*** (*integer*); представляют собой цепочки из одной или более цифр, максимальная длина которых определяется используемой платформой **ЭВМ**: 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);
```

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

$$-0.7722517003 + 1.867646862\,I,\ 1.961822660$$

Тогда как наша 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 этого 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, тогда как 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.

• ***Рациональные*** (*rational*); представляют собой числа, 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** есть 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 эту 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 истинность 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 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 и 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 ***convert***-функция 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 число **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 лишь 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(`123456`), map2(type,`123456`, [‘symbol’, ‘name’]); ⇒ 123456, 1, [*true, true*]

В отличие от *четвертого* 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 задач в среде пакета ***Maple***.

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. Кратко остановимся на каждой из данных 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. Ниже дано несколько примеров ее применения, а именно:

```
> typeseq("Kr",Art,6,14,"RANS",IAN,Tallinn,Moscow, '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 *ранжирования* 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 кодирования:

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 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 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***-функцией 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 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. Более того, 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 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 присвоение **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*** [103], обновляющую список на месте. Ее исходный текст и примеры применения приведены 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 ***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 полезной.

иски могут иметь различный уровень вложенности, определяя различного рода структу-ры 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 *listlist*-тип, играющий важную роль при организации индексированных структур (*массив*, *матрица 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. В свою очередь, наша библиотека [103] также содержит немало 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. 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 ***nops*** и ***op*** 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 следующий 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 процедуры **F** к каждому аргументу **f(x, y, z, ...)**-конструкции (*а в более общей постановке, к каждому операнду выражения*), как это иллюстрирует следующий простой фрагмент кодирования:

> 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)

Вместе с 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 иллюстратив-

ных примерах, лучше раскрывая принцип своего выполнения. Нами был определен ряд новых процедур ***map3***, ***map4***, ***map5***, ***map6***, ***mapN***, ***mapLS***, ***mapTab*** (*расширяющих возможности функций* ***map*** *и* ***map2***), описанных в книге [103] и включенных в состав прилагаемой библиотеки. ***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].

•***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. 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 "a .. 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: **M:=** ***array*****(1..3):** ***print*****(M);** $\Rightarrow [M_1, M_2, M_3]$. 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. Применение данных функций проиллюстрировано ниже. Ни один из трех аргументов ***array***-функции 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);
```

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

```
> 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:
```

$$\begin{bmatrix} 42 & 47 & 67 & 62 \\ 47 & 57 & 89 & 96 \\ 67 & 89 & 52 & 99 \\ 62 & 96 & 99 & 9 \end{bmatrix} \quad \begin{bmatrix} 42 & 47 & 67 & 62 \\ 47 & 57 & 89 & 96 \\ 67 & 89 & 52 & 99 \\ 62 & 96 & 99 & 9 \end{bmatrix}, \begin{bmatrix} 42 & 47 & 67 & 62 \\ 47 & 57 & 89 & 96 \\ 67 & 89 & 52 & 99 \\ 62 & 96 & 99 & 9 \end{bmatrix}$$

```
> 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*. При определения начальных значений элементов непосредственно в ***array***-конструкции они представляются в виде вложенного списка, структура которого должна строго соответствовать структуре определяемого массива, как это показано во фрагменте. В качестве встроенных *индексных* атрибутов пакета используются: *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. 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*).

Ниже обсуждение структур данных *array*-типа будет детализироваться и рассматриваться на протяжении последующих глав книги. При этом, следует иметь в виду тот момент, 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 следующего формата:

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

где назначение формальных аргументов функции полностью аналогично случаю ***array***-функции; 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 соответственно по функциям ***eval*** и ***print***. Следующий фрагмент иллюстрирует вышесказанное:

```
> МАПТ:= hfarray([evalf([sqrt(10), sqrt(17)]), evalf([sqrt(64), sqrt(59)])]);
                                  МАПТ := [3.16227766000000000   4.12310562600000008]
                                          [                                         ]
                                          [              8.      7.68114574799999960]
> 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);
                                  [3.16227766000000000   4.12310562600000008]
                                  [                                         ]
                                  [              8.      7.68114574799999960]
                                  [3.16227766000000000   4.12310562600000008]
                                  [                                         ]
                                  [              8.      7.68114574799999960]
> evalf(array([[sqrt(10), sqrt(17)], [sqrt(64), sqrt(59)]]));
                                      [3.162277660   4.123105626]
                                      [                         ]
                                      [      8.      7.681145748]
```

Структуры данных *hfarray*-типа широко используются с ***evalhf***-функцией, поддерживающей **МАПТ**, и вполне детально рассматриваются, например, в [11,12]. При этом, следует иметь в виду, что работа с такого типа массивами вне рамок **МАПТ** не эффективна, ибо требуется выполнение соглашений между **МАПТ** 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 типа *extended_numeric* (*обобщение типов numeric, undefined либо infinity, -infinity*). Для устранения этого 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.*). В общем случае **T**-таблица определяется конструкцией следующего простого формата кодирования:

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 либо множества, элементами которого могут быть как отдельные допустимые 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 аргумен-

та *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 пакетом 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 допускаемые ***Maple*** объекты, позволяя создавать достаточно сложные табличные структуры данных, примером чего может служить следующий простой пример:

```
> 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);
```

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

При этом, более простым способом создания таблицы является *непосредственное* присвоение индексированной переменной значений, как это иллюстрирует следующий 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 нашей 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 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 **with(T)** возвращается их отсортированный *лексикографически* список, в противном случае возникает ошибочная ситуация, как это иллюстрирует *пример* **4**.

Относительно структур типа *массив* (*array*) и *таблица* (*table*) следует сделать одно существенное пояснение. Все используемые пакетом 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* данное вполне естественное свойство не соблюдается. Поэтому ***Maple***-язык располагает *специальной* процедурой ***copy***(<*Массив*/*Таблица*>), позволяющей создавать копии указанного массива/таблицы, модификация которых не затрагивает самого оригинала. Следующий 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].

В завершение рассмотрения структур данных, поддерживаемых пакетом, уместно будет 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 пределы традиционного университетского курса высшей алгебры. Основу теории матриц составляет алгебра квадратных матриц, для обеспечения которой ***Maple***-язык располагает рядом важных и полезных средств, поддерживаемых функциональными средствами - модулей **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. 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:

***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) -> Ф(k ,j),** такой, что **M[k, j]:= Fn(k, j).**

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 сказанное.

```
> 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.

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 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);
```

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

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.

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. 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)];
```

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

```
> 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 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 кодирования:

***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 подобно случаю определения матриц. По тестирующей 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.

• ***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 достаточно 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, в качестве единственного фактического аргумента использующей выражение типа {*array*, *matrix*, *table*}, можно получать (**nxp**)-размерность произвольной **M**-матрицы в виде [n, p]-списка [103], 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));
```

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

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.

Как таблица, так и общая функция ***array*(**{<ИФ>}{**,** <*Размерность*>}{**,** <НЗ>}**)** допускает *три* 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. 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***. Рассмотрим вкратце эти 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 соотношение **∀(k)∀(j)(k≠j)(M[k, j]=0)**. Посредством ***sparse***-аргумента определяется **M**-матрица, чьи неопределенные входы получают нулевые значения. Например, по ***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 фрагмент иллюстрирует 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 достаточно 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] или к авторскому оригинал-макету, который можно бесплатно скачать с университетского ***Web***-сайта http://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 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 определение 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. Сказанное иллюстрирует 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 [10-14,59,80,86,88,90,91].

• ***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 **LinearAlgebra**-модуля, 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].

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 ***rtablesize* = 0** определяет вывод любого *rtable*-объекта в виде шаблона, тогда как установка ***rtablesize=infinity*** определяет вывод *rtable*-объекта полностью безотносительно его размера, например:

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

$$Kr := \begin{bmatrix} 1..3\ 1\text{-D Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

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

> 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.

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 собственными аналогичными средствами, кратко рассматриваемыми ниже. Для вывода *rtable*-объектов можно использовать форматирующие функции ***printf***-группы, а именно: ***printf***, ***fprintf***, ***sprintf***, ***nprintf***, с опциями которых для этого случая можно ознакомиться по справке пакета. Аналогично тому 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 фрагмент, иллюстрирующий использование операций *rtable*-алгебры.

```
> 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;
```

$$\begin{bmatrix} 6 & 7 & 6 & 0 \\ 8 & 6 & 11 & 6 \\ 4 & 5 & 6 & 0 \\ 7 & 5 & 8 & 4 \end{bmatrix}, \begin{bmatrix} 0 & 44 & 91 & 49 \\ 93 & 90 & 76 & 86 \\ 0 & 72 & 42 & 0 \\ 93 & 67 & 0 & 94 \end{bmatrix}, \begin{bmatrix} 66 \\ 84 \\ 72 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> Vr:= rtable(1..4, random(47..99, 0.99), subtype=Vector[row], storage=sparse, datatype=integer);
```

$$Vr := [65, 56, 57, 90]$$

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

$$\left[\frac{6257455647305336}{16044855393609}, \frac{110021561430595}{327446028441}, \frac{21949362201384325}{64179421574436}, \frac{176823056706773}{327446028441}\right]$$

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

$$\begin{bmatrix} 195 & 218 & 195 & 99 \\ 243 & 195 & 330 & 195 \\ 155 & 174 & 195 & 99 \\ 218 & 174 & 243 & 155 \end{bmatrix}, \begin{bmatrix} 8744 & 14543 & 8713 & 9223 \\ 17949 & 25051 & 19787 & 21843 \\ 6696 & 10728 & 8045 & 6192 \\ 16554 & 17559 & 13555 & 20848 \end{bmatrix},$$

$$[34638221, 45669132, 33705248, 40740017]$$

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* ***rtable****-функции*). 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 функцией ***rtable***, 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*>) достаточно прозрачны и особых пояснений не требуют. Поэтому на данном вопросе детальнее останавливаться не будем, а приведем несколько примеров на прямое определение массивов.

```
> 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);
                                  [42   64   59]
                                  [64   47   39], symmetric , datatype  = integer , storage  = triangular upper ,
                                  [59   39   67]
                                      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, НЗ, <*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 об-

щему случаю ***rtable**-функции*, рассмотренной выше. Однако, среди них имеются и специфические для матричной структуры. Детальнее с описанием опций функции можно познакомиться 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;
```

$$\begin{bmatrix} -71 & -35 & 23 \\ -49 & 86 & 243 \\ -19 & 215 & 471 \end{bmatrix}, \begin{bmatrix} 53 & 47 & 43 \\ 65 & 62 & 92 \\ 77 & 46 & 69 \end{bmatrix}, \begin{bmatrix} 42 & 64 & 10 \\ 47 & 59 & 17 \\ 67 & 39 & 44 \end{bmatrix}$$

```
> 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});
```

$$M6 := \begin{bmatrix} 64 & 59 \\ 10 & 17 \end{bmatrix}$$

```
> 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 простого вида:

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

недостатком которой является невозможность определения 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] и в справке по пакету. С учетом сказанного *Matrix*-объекты (***NAG**-матрицы*) достаточно прозрачны и за более детальной 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 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, V 3;
```

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

```
> 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 простого вида:

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

недостатком которых является невозможность определения 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*) достаточно прозрачны и за более детальной информацией по их определению можно обращаться к справке (*например, оперативно по конструкции* **?Vector**).

Как уже отмечалось, 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 :=
```

$$\begin{bmatrix} a & b & c \\ x & y & z \\ 42 & 47 & 6 \end{bmatrix}$$

```
> A1[2]:= [Grodno, Tallinn]:  A1;
```

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

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.

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 представлены 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 целый ряд дополнительных возможностей.

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. Такая 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 **linalg** и **LinearAlgebra.** Средства первого модуля и его концепция были достаточно детально рассмотрены в наших книгах [8-14], тогда ка 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 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]'));
```

$$[131.7149459,\ -1.85747297 + 23.57676174\,\mathbf{I},\ -1.85747297 - 23.57676174\,\mathbf{I}]$$

```
> 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);
```

$$\begin{bmatrix} -87.8008999999999986 & -70.3864999999999981 & -68.8033999999999964 \\ 0.276677999999999980 & -28.9090999999999988 & -26.8831999999999988 \\ 0.283266000000000018 & -0.664429999999999965 & 29.0239000000000011 \end{bmatrix}, \quad \begin{bmatrix} 1.72892000000000001 \\ 1.38748000000000004 \\ 0. \end{bmatrix}$$

```
> 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));   ⇒ [-4.756244385, 5.948607367, -0.9376460018]
```

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 такой достаточно 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 процедур ***LinearSolve*** и ***LeastSquares*** модуля. Посредством ***Transpose***-процедуры результат решения системы линейных уравнений **A.X=B** получаем в виде вектора-*строки*. Последующие примеры фрагмента иллюстрируют применение различных функций **LinearAlgebra**-модуля. 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 **linalg**-модуля как в области определения самой функции, так и вне ее. Тогда как сог-

ласно 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 читатель отсылается к нашим книгам [13-14,29-33,39,41-43,45,46].

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 *Help*-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. Данные средства постоянно расширяются. На базе рассмотренных средств ***Maple***-языка пользователь имеет возможность программировать 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 нужды. Типичным примером такого подхода и является наша библиотека программных средств [41, 103]. 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.

● ***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 начального 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 следующего 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 ***randomize()***-вызов 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 случае 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 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))** или **(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.

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 рассмотренные типы *данных*, *структур данных* и *выражений*.

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

Согласно аксиоматике пакета 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 языка 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* или их *множество*. Булева функция {***type***|***typematch***} возвращает логическое ***true***-значение, если значение ***Maple***-выражения имеет *тип*, определяемый ее *вторым* аргументом, и ***false***-значение в противном случае. В случае определения в качестве *второго* аргумента *множества типов* {***type***|***typematch***}-функция 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, {***type***|***typematch***}-функцией существенно более

развиты, чем подобные им средства пакета ***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 {***type*** | ***typematch***}-функцией 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* (**ВС**); элементы **ВС** 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. Этот 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 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***, которая имеет весьма простой формат кодирования: ***whattype***(<*Выражение*>) и возвращает тип заданного *выражения*, если он является одним из нижеследующих:

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

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 ***whattype***-процедурой он идентифицируется. 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. Это обусловлено тем обстоятельством, что предварительно вызов процедуры ***whattype*(A)** вычисляет и/или упрощает выражение **A**. Для расширенного тестирования типов выражений служит уже рассмотренная в связи с данными и их структурами {***type*** | ***typematch***}-функция. 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)**, вызов которой 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.

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 сказанное:

```
> 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.

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 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.

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 ***hastype***-функции 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 сказанное, но и указывает на то, что, в частности, ***sqrt*-**функция не тестируется ***hasfun*-**функцией. Ряд дополнительных вопросов, относящихся к тестированию выражений, рассматривается несколько ниже**.**

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 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 можно 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 PLOT3Doptions PLOToptions polar POLYGONS polynom power pwlist radians radical rational ratpoly RealRange record relation Riccati roman RootOf SecondKind set signum sincos sqrfree StandardFunctions std stdle string Sum sum surd symbol system table tan 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* ***Maple 10***) следуют весьма широкие возможности ***convert*-функции**. Многие из них будут довольно детально рассмотрены в настоящей книге в различных контекстах, с другими можно детально ознакомиться по ***Help*-системе** пакета или 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 сводной 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);
```

$$1+\cfrac{x^2}{-6+\cfrac{x^2}{-\frac{10}{3}+\frac{11\,x^2}{126}}}$$

```
> 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');
```

$$-\frac{1}{2}I\,y\left(\mathbf{e}^{(x\,I)}-\frac{1}{\mathbf{e}^{(x\,I)}}\right)+x\left(\frac{1}{2}\mathbf{e}^{(y\,I)}+\frac{1}{2}\frac{1}{\mathbf{e}^{(y\,I)}}\right)$$

```
> convert([[V, 42, 64], [G, 47, 59], [Art, 89, 96]], 'matrix');
```

$$\begin{bmatrix} V & 42 & 64 \\ G & 47 & 59 \\ Art & 89 & 96 \end{bmatrix}$$

```
> convert([inch, ft, yard, miles, bushel], 'metric');
```

$$\left[\frac{127\,m}{5000}, \frac{381\,m}{1250}, \frac{1143\,m}{1250}, \frac{25146\,km}{15625}, 0.035238775147289395200 m^3\right]$$

```
> 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 рассмотренных выше средств конвертации *типов* (*форматов*) выражений в широком понимании данного термина представляется достаточно прозрачным и особых пояснений не требует. Более детальное ознако-

мление со средствами данного класса ***Maple***-языка рекомендуется проводить при непосредственной практической работе в его среде. Нами также был определен ряд полезных средств конвертации объектов ***Maple***-типа в объекты *rtable*-типа, и наоборот [103]. 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 релизова ***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*> либо в документации по пакету, например, [81-83,89]. Следует еще раз напомнить, что кодирование идентификаторов в пакете *регистро-зависимо*, поэтому необходимо правильно кодировать все используемые идентификаторы.

1.8. *Функции математической логики и средства тестирования*

Для решения задач математической логики, пропозиционального исчисления, а также организации *логических* конструкций, управляющих вычислительным процессом в ***Maple***-документе или ролграмме (*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 и/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 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 *подтипу* ***relation*** относятся выражения вида {**<** | **<=** | **=** | **<>**}, а к ***logical***-*подтипу* - выражения вида {**and**, **or**, **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 формата:

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

```
> 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. Например, 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 предназначены для работы с выражениями, используя двузначную булеву логику, т.е. без ***FAIL***. По конструкции **?Logic** читатель может детально ознакомиться со средствами данного модуля.

Рассмотренные в настоящем разделе средства ***Maple***-языка по обеспечению р-шения 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, базовыми тестирующими средствами являются функции ***type*** и ***typematch***, а также процедура ***whattype***. Третья из них возвращает поддерживаемый языком *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 языком пакета *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 упомянутыми средствами ***Maple***-языка типов относятся именно к *первой* группе - *внешним типам*. *Типы*, требующие для своего тестирования анализа (*возможно и рекурсивного*) всего структурного *дерева* выражений, относятся ко *второй* группе - *вложенным типам*. В качестве *вложенных* ***Maple***-язык рассматривает следующие **13** *типов* выражений:

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 или 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 вероятности. 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 сказанное:

> [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];
```

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

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

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

Механизм *шаблонов*, поддерживаемый ***match***-процедурой, позволяет устанавливать точную структуру ***Maple***-выражений на основе указанного 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];
```

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

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 или *возвращаются* ***whattype***-процедурой в результате тестиро-

вания ***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 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, функции ***typematch*** 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 фрагмент иллюстрирует ряд примеров применения тестирующих {***type*** | ***typematch***}-функций:

```
> [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];
                                  [true, [G = 19/60, G = 47/99, G = 10/89, G = 1/32, f = F(19/60, 47/99, 10/89, 1/32)]]
> [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];
                                  [true, [a = tan(x), b = sin(x), c = 64/59]]
> [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 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 следующий 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 установлено *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 ***conditional***-слова, 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 ***patmatch***-процедуры для структурно-типированного анализа 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 = √v, c = -ln(10) π - e^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 ***conditional***-слова. 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 отдельно. 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)]);
```

$$T := \left[(a::algebraic)\,(x::name)^{(n::integer)} = \frac{a\,x^{(n+1)}}{n+1}, \quad \sin(x::name)\cos(x::name)^{(p::integer)} = -\frac{\cos(x)^{(p+1)}}{p+1}, \right.$$

$$\frac{1}{(a::positive) + (b::positive)\,(x::name)^2} = \frac{\arctan\left(\frac{b\,x}{\sqrt{a\,b}}\right)}{\sqrt{a\,b}},$$

$$\left. \sin((n::integer)\,(x::name))^{(m::integer)} = \int \sin(n\,x)^m\,dx \right]$$

```
> 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 принцип создания функциональной **CT**-таблицы и последующего ее использования. Рассмотренные средства ***Maple***-языка 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~) I
```

Аппарат *шаблонов* ***Maple***-языка представляет собой достаточно 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. Именно третья глава книги и посвящена вопросам организации механизма процедур в среде ***Maple***-языка пакета.

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] (*либо бесплатно скачать авторский оригинал-макет книги с адреса* [91]), тогда как с введенными нами новыми важными типами можно ознакомиться в книге [103].

Глава 2. *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 приложений.

2.1. *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].

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 {*symbol*, *name*}-типа, 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 определения 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 простой пример:

> 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.

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 программировании. Их специфические свойства и особенности достаточно детально рассмотрены в [8-14,29,30].

Прежде всего, для объединения (*конкатенации*) строк и/или символьных значений предназначены ***cat***-функция и **||**-оператор *конкатенации*, имеющие следующие весьма простые форматы кодирования:

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 следующий 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 ***length***(<*Выражение*>)-функция 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 представления исходного выражения. Данный результат определяет своего рода *числовую меру* сложности выражения. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> 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 производится 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 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 пример 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:

```
> `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 следующие 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 либо 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, к которой привязывается 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 или *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, завер-

шалось ли содержимое *последней* успешно обработанной ***parse***-функцией строки (**;**)-разделителем либо нет. Следующий фрагмент иллюстрирует 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 ***Maple***-выражение/предложение. 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. В данном же случае имеет место несоответствие числа *открывающих* и *закрывающих* скобок ***Maple***-выражения/предложения, 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. Используя возможности, предоставляемые ***parse***-функцией, можно использовать их для организации простого механизма динамической генерации вычисляемых ***Maple***-выражений, включая 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(cat("", y, ":= () -> `+`(args):"), 'statement')
    else parse(cat("", y, ":= () -> `+`(args)/nargs:"), 'statement')
    end if
end proc
> A(2, Sr), Sr(64, 59, 39, 44, 10, 17);
```

$$() \to \frac{`+`(args)}{nargs}, \frac{233}{6}$$

```
> A(1, Summa), Summa(64, 59, 39, 44, 10, 17);   ⇒   () -> `+`(args), 233
F := (f::symbol, x::symbol, n::posint) → parse(
    cat("", f, " := (", seqstr(seq(x || k, k = 1 .. n)), ") -> ", "`+`(args)/nargs;"), 'statement')
> F(Kr, y, 10);
```

$$(y1, y2, y3, y4, y5, y6, y7, y8, y9, y10) \to \frac{`+`(args)}{nargs}$$

```
> 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 со-

зданию/модификации с последующим вычислением (*активацией*) в текущем сеансе пакета. 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], давая в результате **P2**-строку. И на заключительном этапе по вызову ***eval*(*parse*(P2))** получаем результат вычисления **P2**-выражения, доступный в текущем сеансе. Приведенный ниже *текст* процедуры ***Aproc*** [103], генерирующей расширенные процедуры, весьма прозрачен и превосходно иллюстрирует описанный механизм:

```
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 = cat( "(", seqstr( b ), ")" ));
    if A = [ ] then parse( a )

    elif b = NULL then
        d := cat( "(", seqstr( op( A )), ")" ); eval( parse( sub_1 ( c = d, a )))
    elif p ≤ 0 then
        d := cat( "(", seqstr( op( A ), b ), ")" );
        c := cat( "(", seqstr( b ), ")" );

        eval( parse( sub_1 ( c = d, a )))
    elif nops ( [ b ] ) ≤ p then
        d := cat( "(", seqstr( b, op( A )), ")" );
        c := cat( "(", seqstr( b ), ")" );
        eval( parse( sub_1 ( c = d, a )))

    else
        d := cat( "(", seqstr( op( [ b ][ 1 .. p ] ), op( A ), op( [ b ][ p + 1 .. -1 ] )), ")" );
        c := cat( "(", 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 импользовать 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 [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*» [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, могут быть существенно проще погружены 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 мето-

дов, 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 [103] 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. Надеюсь, 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 также значения типа {*symbol*, *name*}.

Функция ***sscanf*** сканирует указанную *Строку* [*конвертируя входящие в нее числа и подстроки согласно заданной форматирующей строки-конвертора* (*Формат*)] и производит их грамматический анализ в соответствии с ***Maple***-синтаксисом, возвращая список сканированных компонент указанной *первым* аргументом *строки*. *Форматирующая строка* состоит из конвертирующих **%**-*спецификаторов*, имеющих следующую достаточно 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. **7**.

Таблица 7

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 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, ":Прим"]
> 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:", 19.99, sqrt((Sv+Arn)/(Art+Kr)+Γ sqrt(G+V+109)), 180, 70]
> Kr:= "Paldiski 23 april 2006":  sscanf(Kr, "%s%d%s%d");     ⇒ ["Paldiski", 23, "april", 2006]
> scanf("%a");
> 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 иллюстрирует второй пример фрагмента для ***Maple 10***. 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, начиная с **6**-го релиза, *строчное* выражение является *индексируемым*, т.е. к отдельному его символу можно адресоваться по его позиции (например, **"abcddfg"[3], "abcddfg"[3..5];** ⇒ "c", "cdd"), то к такого типа выражениям применимы и многие стандартные средства, предназначенные для работы с индексированными структурами данных.

Ниже средства ***Maple***-языка 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 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] 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*) или список целых положительных.

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.

Если *второй* и *третий* фактические аргументы определяют списки, между которыми имеет место 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 ее 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("aaccccbccccccccccccbccccccccckccccckcccccccccccc", "cccc", 2);
                                    "aaccccbccccbccccckccccckcccccccc"
> Red_n("aaccccbbccccccccckccccckcccccccc", "cccc", 1);
                                                "aabbkk"
```

```
> Red_n("11111222222333333344444444455555555556666666666", [`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 [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 образец **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 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 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. Процедура ***swmpat*** имеет целый ряд полезных приложений в задачах обработки строк и символов [103]. Ниже представлены исходный текст процедуры и некоторые 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, m, p, insensitive, 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
> 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]
```

Наконец, отметим еще одну весьма полезную процедуру работы со *строчными* выражениями. Процедура ***nexts*(S,A,B {,r})** обеспечивает *поиск* в строке или символе, определенных фак-

тическим 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]. Ниже представлены исходный текст процедуры и некоторые примеры ее применения.

```
nexts := proc(S::{string, symbol}, A::{string, symbol, posint}, 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:="aaaccccacaaaccaaacaacddddccrtdrtbbaaabaaabaaahtyrebaaa": nexts(S,29,b), nexts(S,29,b,8);
                                    [29, 33], [29]
> nexts(S, "", b);
Error, (in nexts) arguments cannot be empty,
[aaaccccacaaaccaaacaacddddccrtdrtbbaaabaaabaaahtyrebaaa, , 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]
```

В главе **5** [103] рассматриваются программные средства, расширяющие возможности пакета ***Maple*** релизов **6-10** при работе с выражениями типов {*string*, *symbol*}. Представленные средства обеспечивают множество полезных процедур таких как специальные виды преобразования, сравнение строк или/и 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 другие. В ряде случаев приведенные 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 пакета.

2.2. *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:

***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:

```
> SL:= [`Example`, array(1..2, 1..2, [[V, G], [S, A]]), F(k)$k=1..3, Int(G(x), x=a..b)];
```

$$SL := \left[Example, \begin{bmatrix} V & G \\ S & A \end{bmatrix}, F(1), F(2), F(3), \int_a^b G(x)\,dx \right]$$

```
> SS:= {`Example`, array(1..2, 1..2, [[V, G], [S, A]]), F(k)$k=1..3, Int(G(x), x=a..b)};
```

$$SS := \{ Example, \int_a^b G(x)\,dx, F(2), F(3), F(1), \begin{bmatrix} V & G \\ S & A \end{bmatrix} \}$$

Результатом вычисления ***List***-структуры (***Set***-*структуры*), как правило, является выражение *list*-типа (*set*-*типа*), тестируемое функциями ***type***, ***typematch*** и процедурой ***whattype***, 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 либо 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 при 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 *множество* не имеет четкого упорядочения 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 **k**-й элемент и изменяется значение **k**-го элемента **L**-списка путем присвоения ему значения указанного *выражения*, тогда как по ***NULL****-значению* (**L[k]:=*NULL***) удалять **k**-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),** либо 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*>**}**} произво-

дится конвертация *последовательности* (*exprseq*) в структуру типа {*list*|*set*} соответственно либо 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 выделямого 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*), 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 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*>'})

производит 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], обеспечивающая тестирование принадлежности 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. 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:

$$charfcn[A](X) = \begin{cases} 1, \text{ если } X \text{ удовлетворяет } A\text{-спецификатору} \\ 0, \text{ если } X \text{ не удовлетворяет } A\text{-спецификатору} \\ \text{'charfcn[A](X)', в противном случае} \end{cases}$$

В качестве **A**-спецификатора может выступать: множество, действительное или комплексное значение, диапазон действительных или комплексных значений либо последовательность выражений любого из перечисленных типов. Тогда как смысл выражения "*удовлетворяет A-спецификатору*" определяется следующей табл. **8**.

Таблица 8

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 это 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 иллюстрирует использование характеристической ***charfcn***-функции 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 ***subsop***-функции, 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]. Пример применения процедуры:

> 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 формальный простой 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*** [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 нижеследующего 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'*** [103], 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 listlists*). 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 [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*), или 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: ***numeric***, **`<`**, ***lexorder*** и ***address***; при этом, две после-

дние определяют сортировку соответственно *лексикографическую* и *адресную* (*в соответствии со значениями адресов, занимаемыми в памяти элементами списка*). 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 полезных процедур [103], в частности, полезной во многих приложениях является процедура **SLj**, обеспечивающая сортировку *вложенных* списков на основе заданных позиций элементов их подсписков. Ниже представлены ее исходный текст и некоторые примеры 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 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. По вызову функции ***convert*(B, set)** можно конвертировать в структуру *set*-типа массив, таблицу либо произвольное **B**-выражение, *операнды* которого 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 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 теоретико-множественных функций для поддержки теоретико-множественных операций.

Для обеспечения работы с *множествами* ***Maple***-язык пакета поддерживает три базовых теоретико-множественных оператора/функции, имеющих простые 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, созданным еще для ***Maple V*** (*все наши издания по* ***Maple****-тематике 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 функций и процедур.

Табличные объекты. Табличный объект (*или 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. 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, ибо отсутствует порядок элементов множества, и следовательно порядок вхождений в результате не может 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 таком 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 процедуры ***detab*** [103].

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**. Ниже представлены исходный текст процедуры и некоторые примеры ее применения.

```
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, {'list', 'set'}),
            [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, {'list', 'set'}),
            [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 have table-type but had 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])
```

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 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 простой пример иллюстрирует организацию пакетного модуля на основе табличной структуры:

```
                    ST := table([Sr = ( ) → (`+`(args))/nargs),
                                 Dis = ( ) → sqrt(add((args_k − Sr(args))^2, k = 1 .. nargs)/nargs)
                                 ])
> with(ST), ST[Sr](64, 59, 39, 44, 10, 17), ST[Dis](64, 59, 39, 44, 10, 17);
                                  [Dis, Sr], 233/6, sqrt(14249)/6
> UpLib("C:/Program Files/Maple 8/Llib/UserLib", [ST]);
Warning, <ST> does exist and will be updated
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), ST[Sr](64, 59, 39, 44, 10, 17), ST[Dis](64, 59, 39, 44, 10, 17);
                                  [Dis, Sr], 233/6, sqrt(14249)/6
> 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*** [103] 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] 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 настоящей книги. Заинтересованный же читатель может обратиться к нашим книгам [41,42,45,46,103]. Для работы с выражениями типов {*list*, *table*, *set*} ***Maple***-язык располагает целым рядом средств, основные из которых были представлены или упомянуты в настоящем разделе. Ряд полезных средств для обработки такого типа выражений представляет и наша библиотека [103], демо-версию которой можно бесплатно загрузить с адреса [108].

Так, глава **6** [103] представляет средства, расширяющие возможности ***Maple-***языка при работе с объектами типов {*list*, *set*, *table*}. *Списочные* структуры играют чрезвычайно важную роль, определяя упорядоченные последовательности элементов. Начиная с **6**-го релиза ***Maple***, 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 пакета ***Maple***.

2.3. *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 последующем фрагменте.

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 лишь 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 *индексированных* 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^o + 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');
                                  (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))];
                                [3 a^2 b^2 / c + sqrt(a b d / c), 3 a^2 b^2 + sqrt(d^2)]
> 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^(π x^2 + 10 - π^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);
                                (3 h^4 - 5 h^2 + 3) s / h^2 + (-h^4 - 9 + h^6) / h^3, h^3 + 3 s h^2 - h - 5 s + 3 s / h^2 - 9 / h^3,
                                s^3 - s^2 h - 6 s + 6 s h^2 - 8 h^3, s^3 - 3 s^2 / h - 4 s + 3 s / h^2 - 9 / h^3 + 3 / h + h^3
> SV:= (a + b - 3)*x/(a + b) + a*y/(a + b) - b*(a + b)/z;  ⇒ SV := (a + b - 3) x / (a + b) + a y / (a + b) - b (a + b) / z
> simplify([algsubs(a + b = 1/h^2, SV), algsubs(a + b = 1/h^2, SV, [a, b], 'exact')]);
                                [-(-x z h^2 + 3 x z h^4 + y h^4 z b - y h^2 z + b) / (z h^2), (x z h^2 - 3 x z h^4 + y a h^4 z - b) / (z h^2)]
> 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, т.е. ***numboccur***-функцию.

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 рассмотренных 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].

Нижеслелующий 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*

Учитывая важность различного рода подстановок при работе с символьными и строчными выражениями – одними из основных составляющих символьных вычислений и обработки – нами был определен целый ряд средств данного типа, ориентированных на различные случаи приложений [41,103]. 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 фактического аргумента **S**.

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 типа {*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 **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*). Ниже представлен фрагмент с исходным текстом процедуры ***Sub_st*** и результатами ее применения.

```
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
```

Наряду с целым рядом средств, обеспечивающих разнообразную обработку *символьных* и *строчных* выражений, наша библиотека [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 следующий 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 соответствующими *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 мож-

но 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`, `=`, `<>`, `<`, `<=`.

Представленная ниже 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. Следующий фрагмент представляет исходный текст процедуры ***Use*** и примеры ее применения.

```
> 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 (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);
                                        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*1/10000000000000000000000001.0, (64 + Digits^a) end proc, a = 2);
                      proc () Digits := 40; 0.2000000000*10^(-22), 64 + Digits^2 end proc
> %();   ⇒ 0.2000000000 10 -22, 1664
```

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.

2.4. *Средства Maple-языка для обработки 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 символьных вычислений. Для этих целей ***Maple***-язык располагает специальной ***simplify***-процедурой 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. **9**:

Таблица 9

Тип	*Производится упрощение выражения, содержащего:*
`@`	*операторы*; как правило при работе с *обратными* функциями
Ei	*экспоненциальные* интегралы; ***Ei*(1, x*I)** → **-*Ci*(x) + *Si*(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. 9 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);
                                  { -59 x - 103     x < 0
                                  {     -39         x = 0
                                  {  59 x + 25      0 < x
> 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 e^(10^x)]
> G:= [64^(1/4), 180^(2/3), (-1999)^(1/3), sqrt(191)]:  simplify(G, radical);
                                  [2 sqrt(2), 180^(2/3), (1 + sqrt(3) I) 1999^(1/3)/2, sqrt(191)]
> 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 RootOf(_Z^2 - 13)^2 - RootOf(_Z^2 - 13) + 10, 49 - RootOf(_Z^2 - 13)]
> h:= (64*s - 59)/(s^2 + 10*s + 17):  [h, simplify(h, RootOf)];
                  [(64 RootOf(_Z^2 - 13) - 59)/(RootOf(_Z^2 - 13)^2 + 10 RootOf(_Z^2 - 13) + 17),
                   -251/40 RootOf(_Z^2 - 13) + 1009/40]
> 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)];
                                  [sqrt(64) sqrt(1/(x - 10)^2), 8/(x - 10)]
> [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);
                                  [-cosh(x), 2/cos(x)]
> simplify(57*x^4-52*x^3+32*x^2-10*x+3, {13*x^2+3*x+10 = 0});
                                  86523 x/2197 + 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 или к нашей книге [12], принимая во внимание, что и релиз накладывает свои особенности. Вместе с тем, представленной информации вполне достато-

чно для проведения многих важных упрощающих процедур над 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 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 языка для упрощения различного типа 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 функции ***expand*** *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 простой 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));
```

$\Rightarrow\ x^3 + 3x^2y + 3xy^2 + y^3 - 3x^2 + 6xy - 3y^2 + 10x + 10y$

```
> expand((z + 2)^2*((x + y)^2 - 3*(x - y)^2 + 10*(x + y)), x + y, x - y);
```

$$z^2(x+y)^2 - 3z^2(x-y)^2 + 10z^2x + 10z^2y + 4z(x+y)^2 - 12z(x-y)^2 + 40zx + 40zy + 4(x+y)^2 - 12(x-y)^2 + 40x + 40y$$

```
> 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:

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 вызова функции ***expand***{***off*** | ***on***}**()** описанное действие по санкционированию *раскрытия* функций ***expand***-функцией распространяется на *все* функции текущего сеанса, исключая *пользовательские* функции. Следующий довольно прозрачный фрагмент иллюстрирует применение указанных функций:

```
> P:= x -> (x + 10)^3 - sin(3*x): expand(P(y)); expand(expandoff()):
```

$$y^3 + 30y^2 + 300y + 1000 - 4\sin(y)\cos(y)^2 + \sin(y)$$

```
> expandoff(sin): expand(P(z));
```

$\Rightarrow\ z^3 + 30z^2 + 300z + 1000 - \sin(3z)$

```
> expand(sin(3*x)), expand(expandon()), expandon(), map(expand, [P(h), sin(3*t)]);
```

$$\sin(3x),\ \text{expandon}(\,),\ [h^3 + 30h^2 + 300h + 1000 - 4\sin(h)\cos(h)^2 + \sin(h),\ 4\sin(t)\cos(t)^2 - \sin(t)]$$

```
> P:= x -> (x + 10)^3 + sin(3*x)*exp(x + y): expandoff(): expand(P(t));
```

$$t^3 + 30t^2 + 300t + 1000 + 4\,e^{(t+y)}\sin(t)\cos(t)^2 - e^{(t+y)}\sin(t)$$

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 ***expandoff***-функцией, запрещающей разложение заданных функций ***Maple***-языка.

При использовании {***expandoff*** | ***expandon***}-функции следует иметь в виду, что в 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 нижеследующем 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))];
```

$$4\sin(x)\cos(x)^2 - \sin(x)$$

$$[4\sin(x)\cos(x)^2 - \sin(x), \sin(4x), 4\cos(x)^3 - 3\cos(x)]$$

```
> 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 либо 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 выражений, ряд, отношение либо функция, то факторизация применяется *рекурсивно* к его компонентам. Следующий прозрачный фрагмент иллюстрирует некоторые варианты применения ***factor***-процедуры для факторизации:

```
> 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);
```

$$[-57.(x - 1.20919)(x^2 + 0.296908x + 0.464281),$$
$$-57.(x + 0.148454 + 0.665013I)(x + 0.148454 - 0.665013I)(x - 1.20919)]$$
$$[(x - 1.50000 + 7.05337I)(x - 1.50000 - 7.05337I),$$
$$(x - 5.00000 + 2.64575I)(x - 5. - 2.64575I)]$$

```
> 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);
```

$$\frac{(x-2)^2(y-9)^2}{(x-1)^2(y-2)^2}$$

```
> factor([G(P1/P2) .. H(P2/P1), P1 <> P2]);
```

$$\left[G\left(\frac{(x-2)^2(y-9)^2}{(x-1)^2(y-2)^2}\right) .. H\left(\frac{(x-1)^2(y-2)^2}{(x-2)^2(y-9)^2}\right), (x-2)^2(y-9)^2 \neq (x-1)^2(y-2)^2\right]$$

Если в качестве значения **F**-аргумента выступают ***RootOf***-процедура, список или 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**-выра-

жение, для ***factors***-процедуры в качестве ее первого аргумента должен выступать только **V**-полином, а возвращается списочная структура следующего вида:

$$[G, [[M[1], k[1]], [M[2], k[2]], \ldots [M[n], k[n]]]] \ ; \qquad V = G * \prod_{j=1}^{n} M[j]^{k[j]}$$

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 или в **mod**-конструкции аналогично тому, как это было описано для ***Factor***-функции с очевидными изменениями. Следующий простой 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));
```

$$[1, [[x-1, 2], [y-2, 2]]], [1, [[x^4+1, 1], [x-2, 1]]],$$

$$\left[9, \left[\left[x^2 + \frac{\sqrt{3}}{3}, 1\right], \left[x^2 - \frac{\sqrt{3}}{3}, 1\right]\right]\right]$$

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 процедуру ***combine***, 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. **10**.

Таблица 10

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. **10** значений для второго фактического аргумента ***combine***-процедуры достаточно прозрачны и проиллюстрированы в нижеследующем фрагменте, однако по отдельным необходимы пояснения [8-14,39].

```
> combine(arctan(1/2) - arctan(10/17) + arctan(59/47));
```
$$\Rightarrow \arctan\left(\frac{491}{449}\right)$$
```
> combine(arctan(x) + arctan(y) - arctan(z), arctan, 'symbolic');
```
$$\Rightarrow \arctan\left(\frac{\frac{x+y}{1-x\,y}-z}{1+\frac{(x+y)\,z}{1-x\,y}}\right)$$
```
> 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');
```
$$\Rightarrow e^{(3\,x+y+x^2+10)}$$
```
> combine(a*ln(x)+b*ln(y)-ln(b-z)+ln(c+y)/2, ln, anything, 'symbolic');
```
$$\Rightarrow \ln\left(\frac{x^a\,y^b\,\sqrt{c+y}}{b-z}\right)$$
```
> 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), `in other cases`, exp(ln(x))):  combine(P_w);
```
$$\begin{cases} \ln(x\,e^{y}) & x<42 \\ \textit{in other cases} & x<64 \\ 28\,x\,\sin(2\,x) & 64\le x \end{cases}$$
```
> 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);
```
$$\Rightarrow \frac{\left(s^b-\frac{1}{2}\right)^3}{\left(s^a-\frac{1}{2}\right)^2}$$
```
> combine((3+sqrt(10))^(1/2)*(5-sqrt(10))^(1/2)*sqrt((3-4^(1/2))),'radical');
```
$$\Rightarrow \sqrt{(3+\sqrt{10})\,(5-\sqrt{10})}$$
```
> 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 необходимо в случае *символьных* аргументов ***arctan***-функций. 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-оп-

ределенные 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');
```

$$a^{(b+c+d)}\,(\ln(x+10)-\ln(y+17)+\ln(123)+\ln(y))\,\mathbf{e}^{(a\,x+b\,y)}$$

```
> 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 либо 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 $x1^p*x2^q*...*xn^k$. 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. Отмеченные случаи применения ***collect***-процедуры весьма наглядно иллюстрирует следующий 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(H(sqrt(|x (a - b)|))^2) y^2 + ln(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 внимание на таких часто используемых средствах как ***convert***, ***simplify***, ***factor***, ***collect***, ***map***, ***expand***, ***combine***. При этом следует иметь в виду, что в задачах *упрощения* (*а в более жестком понимании и канонизации*) выражений используются 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 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 *выше* в связи с другим контекстом. Наряду с этим для формальных преобразований используются и другие ранее рассмотренные средства ***Maple***-языка; здесь же представлены 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 выше 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 или 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 крайней мере для одной из пар <*Выражение*, *Переменная*>. Следующий фрагмент иллюстрирует применение ***depends***-процедуры для установления факта 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' = {op(Z), S[k]}))
        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, z, cos(y), e^x, ln(x), tan(z), y, Art, Kr, Sv, Ag, a, x, b}
> 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, ln(x), y, Art, c, Av, d, tan(y), G(x), S(z), x, b, γ}
```

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 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*, *Catalan*, *FAIL*, *Pi*}. 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] решает данную задачу. Вызов процедуры ***Indets***(**W**) возвращает множество, срдержащее все *независимые* переменные алгебраического **W**-выражения, удовлетворяющие вышеупомянутому, 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);   ⇒   {x, h, t, sin(x), VS(t), VA(r), Kr, Art, AG(h, x), sqrt(Art^2 + Kr^2), r}
> Indets(W);   ⇒   {x, h, t, Kr, Art, r}
```

В отличие от предыдущей процедуры вызов процедуры ***indetval***(**A**) [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 использовать ***factor***(***normal***(**V**))-конструкцию. В качестве **V**-выражения могут выступать список, множество, диапазон, ряд, 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]
> normal(Pf2);   ⇒
```

$$\frac{3\,x+2}{2\,x+1}$$

```
> map(normal, [F(1 + 1/y), [1 + 1/x, x + 1/x], z + 1/z .. z + z/(1 + z)]);
```

$$\left[F\left(\frac{y+1}{y}\right), \left[\frac{x+1}{x}, \frac{x^2+1}{x}\right], \frac{z^2+1}{z} .. \frac{z\,(2+z)}{1+z}\right]$$

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. ***Maple***-язык распознает над ***RootOf***-конструкциями операции упрощения (***simplify***), дифференцирования (***diff***), интегрирования (***int***), численных вычислений (***evalf***), разложения в ряд (***series***) и некоторые другие. Следующий 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 ***convert***-функции можно конвертировать **I**-константы и радикалы в ***RootOf***-конструкции.

По ***root***-процедуре, имеющей два эквивалентных формата кодирования следующего 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 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);
> assume(y < 2, y >= 0):  root[5](x^5*(y - 2)^4, 'symbolic');
> 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);
```

$$\Rightarrow (a\sim + b\sim y\sim + 10\, y\sim^2)^{1/3}$$

$$\Rightarrow x\,(2 - y\sim)^{4/5}$$

$$\Rightarrow \frac{x\left(\left(x^2+n\right)\left(z^3-m\right)\right)^{(1/3)}}{z^3-m}$$

$$[\sqrt{64+42\,I}, \sqrt{10}, 4.478839135, 6.387470130 - 4.618416900\,I, \sqrt{\gamma}, \sqrt{Catalan}\,]$$

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 ***Maple***-языка ситуаций рекомендуется использовать конструкции следующего вида ***trigsubs*(*simplify*(V, 'trig'))**, как это хорошо иллюстрирует нижеследующий достаточно простой фрагмент:

```
> trigsubs(0);  trigsubs(AVZ);
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'));
```

$$\Rightarrow \{\tan, \sec, \sin, \cos, \cot, \csc\}$$

$$\left[\sin(x), -\sin(-x), 2\sin\left(\frac{x}{2}\right)\cos\left(\frac{x}{2}\right), \frac{1}{\csc(x)}, -\frac{1}{\csc(-x)}, \frac{2\tan\left(\frac{x}{2}\right)}{1+\tan\left(\frac{x}{2}\right)^2}, \frac{-1}{2}\,I\,(e^{(x\,I)} - e^{(-I\,x)})\right]$$

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 строчное значение {`*found*`|`*not found*`}. 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`]
> 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 следующий 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());   ⇒ 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));   ⇒ sqrt(π x^2 + σ y^2 + γ) sin(x π) + cos(σ y) + 2006
> subs([a=Pi, b=sigma, c=gamma], Proc(x, y));    ⇒ 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);   ⇒ (x, y) → sqrt(sin(z) x + x π y) + sin(z) x π
```

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***}, но подобно ***algsubs*-процедуре** не производит математических преобразований обрабатываемого выражения, требуя для этих целей ***eval*-функции**. Следующий фрагмент иллюстрирует применение ***applyrule*-процедуры** для преобразования выражений**:**

```
> algsubs(x*h(x)=a*x, x*h(x));
Error, (in algsubs) cannot compute degree of pattern in x
> 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)
> 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) 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 ***Maple*-выражение** на предмет отношения его к типу "*функция*"**.** Как видно уже из следующего 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 вида. 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* x_j-переменной произвольной $F(x_1, ..., x_n)$-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 данных 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 полезной тестирующей представляется также и 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*) переменным, заданным 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:

$$nops(F(x_1, ..., x_n)) \Rightarrow n \qquad op(F(x_1, ..., x_n)) \Rightarrow x_1, ..., x_n$$
$$op(0, F(x_1, ..., x_n)) \Rightarrow F \qquad op(k, F(x_1, ..., x_n)) \Rightarrow x_k \quad (k > 0)$$

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))** [103] возвращает *true*-значение, если **F** является функцией либо процедурой **n**-арности; в противном случае возвращается *false*-значение. Более того, через *глобальную* переменную ***'n-arity'*** возвращается реальная арность произвольной функции/процедуры **F**, например:

```
> 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, а именно:

***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 либо приписываются **A**-атрибуты **B**-выражению, либо 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:

```
> [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 пакетную их 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;   ⇒ 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);
```

$$\sqrt{\cos(\ln(x))^2} + H(\mathbf{e}^x) - \frac{G(\sin(x) + \cos(x))}{Art(x) + Kr(x)}$$

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:

$$\mathbf{\mathit{seq}(A(k),\ k = [B(x)]) \equiv \mathit{seq}(A(k),\ k = \{B(x)\}) \equiv (A@B)(x) \equiv A(B(x))}$$
$$\mathbf{\mathit{seq}(A(k),\ k = x) \equiv \mathit{seq}(A(k),\ k = B(x)) \equiv A(x)}$$

Активные и пассивные формы функций. Наряду с вычисляемыми в указанном выше смысле функциями, идентификаторы которых 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* функций/процедур (***Diff***, ***Int***, ***Limit***, ***Sum***, ***Produc***, ***Normal*** *и др.*), суть которых нетрудно усмотривается 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(%));
```

$$\int \sqrt{\frac{1+x}{1-y}}\,dx$$

$$\frac{2\,(1+x)\sqrt{-\frac{1+x}{y-1}}}{3}$$

```
> Product((1 + x)^k, k = 1 .. 10);  value(%);
```

$\Rightarrow\ \prod_{k=1}^{10} (1+x)^k$

$$(1+x)^{55}$$

```
> Limit((1 + 1/j)^j, j = infinity);  value(%);
```

$\Rightarrow\ \lim_{j\to\infty} \left(1 + \frac{1}{j}\right)^j$ **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 *окончательного* 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.

Пассивные функции/процедуры тестируются и нашей процедурой ***ParProc*** [103], как это иллюстрирует следующий весьма простой фрагмент:

```
> 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))

$$-\frac{\sqrt{-\frac{1+x^2}{-1+x^2}}\,(-1+x^2)\sqrt{1-x^2}\,\sqrt{1+x^2}\,\mathrm{EllipticE}(x,I)}{\sqrt{-(1+x^2)(-1+x^2)}\,\sqrt{1-x^4}}$$

> 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

$$\frac{\mathrm{EllipticE}(x,I)\sqrt{1-x^2}\,\sqrt{-\frac{1+x^2}{-1+x^2}}}{\sqrt{1+x^2}}$$

> 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

$$y(x)=\frac{1}{2920}\sin(3x)-\frac{27}{2920}\cos(3x)+\frac{1}{8}\sin(x)-\frac{1}{8}\cos(x)+\_C1\,e^{x}$$
$$+\_C2\,e^{\left(-\frac{x}{2}\right)}\sin\left(\frac{\sqrt{3}\,x}{2}\right)+\_C3\,e^{\left(-\frac{x}{2}\right)}\cos\left(\frac{\sqrt{3}\,x}{2}\right)$$

> eval(infolevel); ⇒ 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 дис-

циплин 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].

Для целей преобразования математических функций (*известных ядру пакета*) из одного типа в другой служит 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 данном случае ***convert***-функция имеет формат кодирования вида:

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 всегда 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 ***GAMMA***-*функцию*; 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 согласно 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 либо 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 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));
```

$$\Rightarrow\quad R := \begin{cases} x\,a^{x} & x \le 0 \\ x\ln(x) & -x < 0 \text{ and } -59 + x \le 0 \\ x\,\mathbf{e}^{x} & -x < -59 \text{ and } x - 64 \le 0 \\ \mathbf{e}^{x}\cos(x) + \sqrt{Art + Kr} & otherwise \end{cases}$$

```
> 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***-процедуру 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 ни в функцию Хэвисайда, ни 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 ***convert***-функции 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'));
```

$$2\,I\,\sqrt{x^2 + 1 - \mathrm{erf}(x)}\;\mathbf{e}^{(x^2)}$$

```
> 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);
```

$$R := \begin{cases} 64 & g \le 42 \\ 59 & g - 47 \le 0 \text{ and } -g < -42 \\ 39 & g - 67 \le 0 \text{ and } -g < -47 \\ 2006 & otherwise \end{cases}$$

```
> convert(R, 'Heaviside');
```

$$64 - 5\,\mathrm{Heaviside}(-42+g) - 20\,\mathrm{Heaviside}(g-47) + 1967\,\mathrm{Heaviside}(g-67)$$

```
> convert(%, 'piecewise');
```

$$\begin{cases} 64 & g < 42 \\ undefined & g = 42 \\ 59 & g < 47 \\ undefined & g = 47 \\ 39 & g < 67 \\ undefined & g = 67 \\ 2006 & 67 < g \end{cases}$$

```
> convert(Heaviside(t), 'piecewise');
```

$$\begin{cases} 0 & t < 0 \\ undefined & t = 0 \\ 1 & 0 < t \end{cases}$$

```
> simplify(convert(Sum(17*k/(k + 10), k = 0 .. n), 'hypergeom'));
```

$$-17\,(\text{hypergeom}([1, n-1], [n], 1)\,n^3 - 2\,\text{hypergeom}([1, n-1], [n], 1)\,n^2 - 18\,\text{hypergeom}([1, n-1], [n], 1) + 9\,\text{hypergeom}([1, n-1], [n], 1)\,n + \text{hypergeom}([2, n-1], [n], 1)\,n^2 + 10\,\text{hypergeom}([2, n-1], [n], 1)\,n - 11\,\text{hypergeom}([2, n-1], [n], 1) - 5\,\text{hypergeom}([2, 2], [3], 1)\,n^3 + 25\,\text{hypergeom}([2, 2], [3], 1)\,n^2 - 40\,\text{hypergeom}([2, 2], [3], 1)\,n + 20\,\text{hypergeom}([2, 2], [3], 1))\,/\,((n^2 - 3n + 2)(n - 2))$$

```
> convert(hypergeom([3, 5], [3/2], h), 'StandardFunctions');
```

$$\frac{-16h^3 + 216h^2 + 2346h + 919}{1024\,(-1+h)^6} + \frac{1}{1024}\,\frac{(-1680h^2 - 1680h - 105)\sqrt{1-h}\,\arcsin(\sqrt{h})}{(-1+h)^7\sqrt{h}}$$

```
> 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'));
```

$$\pi x\,\text{AiryAi}(1, x^2) + y\,\text{AiryBi}(x) + y\,x\,\text{AiryBi}(1, x)$$

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 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 достаточно 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 представляют интерес прежде всего. 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, теория эволюции и развития, адаптивные и динамические системы, 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 методами, проблематикой 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 результатов, а также основных тенденций развития данного направления исследований. Представленные материалы позволяют получить *единую* картину составляющих частей **ТОС** и ее приложений со всеми основны-

ми их взаимосвязями. На фоне предложенной архитектуры рельефнее вырисовывается 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 иде-нтичными. 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 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 $t \geq 0$.

Состояние всей однородной среды называется *конфигурацией* (**КФ**) **1-ОС** и представляет собой набор текущих состояний всех составляющих ее единичных автоматов. А именно, кон-

фигурация **1-ОС** есть любое отображение **КФ: $Z^1 \to A$** и **C(A, 1)** обозначает множество всех таких конфигураций относительно **Z^1** и **A**, т.е. **C(A,1) = {КФ | КФ: $Z^1 \to A$}**. 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**) **$\sigma^{(n)}$**, 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 **$\sigma^{(n)}$: $A^n \to A$**; в дальнейшем для **ЛФП** классических структур будем использовать следующие основные обозначения:

a1a2 ... an → 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 **$\sigma^{(n)}$: $0^n \to 0$**, т.е. структуры с ограничением на скорость передачи информации в них.

Одновременное применение **ЛФП $\sigma^{(n)}$** к текущей конфигурации щаблона соседства каждого единичного **z**-автомата **1-ОС** определяет *глобальную функцию перехода* (**ГФП**) **$\tau^{(n)}$** структуры, переводящую текущую **КФ c ∈ C(A, 1)** в следующую **КФ c $\tau^{(n)}$ ∈ C(A, 1)** структуры. Именно для программной реализации *глобальной функции перехода* на основе заданной *локальной функции* и была создана процедура ***HS_1***, представленная ниже. Следующий фрагмент представляет исходный текст процедуры и примеры ее применения для *генерации* конечных конфигураций из некоторой начальной конфигурации структуры **1-ОС**.

```
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 con \
        tain 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 10 do assign('S' = HS_1(S, {a, b, c}, c, 3)), S end do;

baabbccbcbabbaab
accbaababacbabcbab
baaaccbcbccbcbaacbab
accaaaababbabaccabcbab
baabaaabcbabbccabaaacbab
accbccabaacbaabbaccaabcbab
baaabbbaccabccbabcababaacbab
accabacbcabaabacbacacbccabcbab
baabaccbbcaccbccbcccabbbbaaacbab
accbccaacacaaabbabbcbaaccbcaabcbab

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 вышелен **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 нестандартные процедуры ***belong*** и ***`convert/set1`*** из нашей библиотеки [103], с которыми можно ознакомиться в демо-версии библиотеки, доступной по адресу, указанному в [108].

2.5. *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 и/или 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.* **11**).

Таблица 11

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);
```

```
                                    2      2 1/2    S + A    Art
                                sqrt(Art  + Kr  )  ,  --------,  -----
                                                      V + G    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);
```

$$\sqrt{a + \pi\,b}, \mathbf{e}^{(\sqrt{a + \pi\,b})}\,h - \frac{S + A}{V + G}, zx$$

```
(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 ***print***-функци длинные выражения выводятся в терминах %-*меток*, как это иллюстрирует предпоследний пример фрагмента.

Наконец, наиболее развитым средством *форматирования* результатов вычисления ***Maple***-выражений 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:

%{<*Флажки*>}{<*Длина*>}{.<*Точность*>}<*Код*>

Кроме параметра "*Код*", остальные параметры %-спецификации необязательны. В качестве значений для *флажков* используются символы, имеющие следующую смысловую нагрузку:

(**+**) - 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, либо *нулями* при указании **0**-флажка. Параметр "*Точность*" **%**-спецификатора определяет количество цифр, выводимых после десятичной точки числа, либо 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. **12.**

Таблица 12

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, а именно: $Код = \begin{cases} d, & if\ \alpha\ contains\ no\ decimal\ point \\ \{E \mid e\}, & if\ 10^{p} < \alpha \le 10^{-4} \\ f, & otherwise \end{cases}$, где **p** - значение "*Точности*"
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 весьма простой 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 либо 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 и делая ее доступной для использования в текущем сеансе.

Если форматируемое значение имеет *hfarray*-тип, то выводятся все элементы массива. При этом форматирование его отдельных элементов определяется значениями параметров "*Длина*" и "*Точность*" %-спецификатора.

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.** Данная возможность *может* быть весьма полезным подспорьем при оформлении вывода *форматируемых* конструкций, как это иллюстрирует следующий весьма 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  |
--------------------------------------------------------
| 57 | 52 | 32 | 10 |  3  | 37  |
--------------------------------------------------------
> printf(`%s<Комментарий>%s`, `Выражение_1`, `Выражение_2`);
Выражение_1<Комментарий>Выражение_2
> printf(`Произвольный комментарий`);
Произвольный комментарий
> print(`Произвольный комментарий`);
                                             Произвольный комментарий
> G:="SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS”:
> printf(%);      ⇒ NULL
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
> nprintf(%);     ⇒ NULL
           SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
> 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 функции ***sprintf*("**<*Строка*>**") -** *возврат* произвольной ***Maple***-строки, т.е. оба средства в данном 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 [12]. Принцип действия ***printf***-процедуры распространяется и на две остальные функции ***sprintf*** и ***fprintf*** группы 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 ***printf***-группы 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|"
> nprintf(%);  ⇒ 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
> sprintf(`%a`, "String"),  sprintf(`%a`, `Symbol`);  ⇒ ""String"", "Symbol"
> nprintf(`%a`, "String"),  nprintf(`%a`, `Symbol`);  ⇒ "String", Symbol
```

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/Books_2006.65":  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/Books_2006.65" 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 "**a**". 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));
```

$$\Rightarrow \sqrt{\mathrm{evalf}(\pi\, Art^2 + \gamma\, Kr^2)}$$

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 3. *Базовые управляющие структуры Maple-языка*

Для описания *произвольного* вычислительного алгоритма рассмотренных выше конструкций ***Maple***-языка явно недостаточно по причине отсутствия средств по *управлению* вычислительным процессом. Настоящая глава и служит целям рассмотрения данных средств ***Maple***-языка, делающих его универсальным языком программирования высокого уровня.

3.1. *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*, *подпрограмм*)**;** 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 [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 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, теоретически *любой* сложности корректная структурированная программа 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 подход позволяет отказаться в программах от использования меток и безусловных переходов. Структура таких 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 иллюстрационных 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 иного порядка. В дальнейшем предложения языка будем называть в соответствии с их определяющим назначением, например: предложения *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 здесь 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 упомянутой ***assign***-процедуры:

```
> [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 ***assign***-процедуры. При этом, последний пример фрагмента демонстрирует использование *невычисленных* идентификаторов, обеспечивающих корректность вычислений. Тогда как третий 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, устраняющих, в том числе, и проблему ее *несовместимости* относительно релизов **6** – **10** пакета.

3.2. *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** <ППп> **end if** {**;**|**:**}
(2) if <ЛУ> **then** <ППп1> **else** <ППп2> **end if** {**;**|**:**}
(3) if <ЛУ1> **then** <ППп1> **elif** <ЛУ2> **then** <ППп2> **else** <ППп3> **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* (**ППп**) представляет собой управляющую структуру типа следования, предложения которой завершаются {**;**|**:**}-разделителем; для последнего предложения **ППп** 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 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 **ППп**, 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 **ППп**, 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 **ППп1**, в противном случае выполняется следующая за ключевым **else**-словам **ППп2**. Замечание относительно вывода промежуточных результатов для случая первого формата **if**-предложения сохраняет силу и для *второго* формата кодирования.

Третий формат **if**-предложения несет смысловую нагрузку, легко усматриваемую из следующей общей **R**-функции, поясняющей принцип выбора выполняемой **ППк** в зависимости от цепочки *истинности* предшествующих ей ЛУj (**j = 1 .. k**) в предложении:

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

А именно: **if**-предложение *третьего* формата возвращает **R**-результат выполнения **ППк** тогда и только тогда, когда справедливо следующее определяющее соотношение:

$$(\forall j \mid j \le k\text{-}1)\ (\textit{evalb}(\text{ЛУj}) \Rightarrow \textit{false})\ \textbf{and}\ (\textit{evalb}(\text{ЛУk}) \Rightarrow \textit{true})$$

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 простой фрагмент иллюстрирует применение всех 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 иллюстрационных фрагментов ***Maple***-языка.

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

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

ЛУk - *логические условия* и **ПП** - последовательность предложений. Данного формата **if**-конструкция выполняется корректно, но результат возвращается без вывода. Его можно использовать по {%|%%|%%%}-предложению либо 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. Следует 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 [8-14,39].

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 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 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*** [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 ***Maple***-языка пакета.

3.3. *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.* **3-23** [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. Нижеследующий фрагмент иллюстрирует принципы выполнения всех вариантов первого формата **while_do-**предложения 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 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 либо внешнему относительно его циклу, то ***next***-значение позволяет управлять счетчиком вы-

полнения 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 ***MS D0S + 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 используемого для предложения *второго* формата завершающего его разделителя.

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 задач 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.

3.4. *Специальные типы циклических управляющих структур Maple-языка*

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 ошибкам 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;   ⇒ 3320413933267719290880
> 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 *невычисленнои формате*. Более того, три последние 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 воспринимаемое 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.

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 программы 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.

Глава 4. *Организация механизма процедур в Maple-языке*

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. 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 различного назначения.

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.

4.1. *Определения процедур в Maple-языке и их типы*

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 пользователей. 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*>**;**
  global <*Последовательность идентификаторов*>**;**
  options <*Последовательность параметров*>**;**
  description <*Описание*>**;**
  <***Т Е Л О процедуры***>
end proc {**;**|**:**}

Описательная часть определения процедуры

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 в разделе **4.5.** 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. Секции **local, global, options** и **description** составляют *описательную* часть определения ***Maple-***процедуры, которая в общем случае может и отсутствовать. Управлять выводом тела процедуры можно посредством установки ***verboseproc-***переменной ***interface-***процедуры ***Maple-***языка. *Минимальной* конструкцией, распознаваемой ядром в качестве *процедуры*, является структура следующего весьма простого вида**:**

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 тоже 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 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 процедурной скобкой*).

Вторым способом определения процедур является использование *функционального* оператора (**->**), позволяющего представлять процедуры в нотации классической операции *отображения*, 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 или **if**-предложением языка, например:

```
> 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. Аналогично *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.* **3-26** [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 данному 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 *типированными* 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* **F(x_1, ..., x_n)**. Следующий простой 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 спо-

собом **S**-процедуры (*в ее теле использована* ***cat***-*функция языка*), 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 процедура ***unapply*** 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 -> x, x -> 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 довольно 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 простой фрагмент:

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

$$SV := (\ ) \to \left[\mathrm{nargs},\ \frac{\sqrt{\sum_{k=1}^{\mathrm{nargs}} \mathrm{args}_k^{\,2}}}{\mathrm{nargs}}\right]$$

```
> [SV(64, 39, 59, 44, 10, 17), evalf(SV(64, 39, 59, 44, 10, 17), 6)];   ⇒  [[6, sqrt(11423)/6], [6., 17.8130]]
> VS:= () -> evalf([nargs, sqrt(sum(args['k']^2, 'k' = 1..nargs))/nargs], 6);
```

$$VS := (\ ) \to \mathrm{evalf}\left(\left[\mathrm{nargs},\ \frac{\sqrt{\sum_{'k'=1}^{\mathrm{nargs}} \mathrm{args}_{'k'}^{\,2}}}{\mathrm{nargs}}\right], 6\right)$$

```
> 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, базиру-

ющегося 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(GS, GS(x::integer)=interp(R, S, x)); GS(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);   ⇒ √100, 10, H(5√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));
                                  √4001558, DD(64, 59, 39, 44, 17, 10)
> restart;  define(R, R(x::even, y::odd) = sqrt(x^2 + y^2)),  R(10, 17);   ⇒ √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);   ⇒ √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 два принципиальных момента:

(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 фрагмента иллюстрирует (*в ряде случаев полезную*) возможность рекурсивного использования *define*-функции 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, что механизм *пользовательских* функций, поддерживаемый пакетом ***Mathemaica*** [6, 7], представляется нам существенно более гибким при реализации алгоритмов обработки.

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.

4.2. *Формальные и фактические аргументы Maple-процедуры*

Формальный аргумент процедуры в общем случае имеет вид **<*Id*>::<*Тип*>**, 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 выводом 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*.** В случае 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 допустимыхе ***Maple*-**выражений, 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 заданного для соответствующего ему *формального* аргумента, возникает ошибочная ситуация с возвратом через ***lasterror***-переменную соответствующей диагностики и с выводом ее в текущий документ, 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 рекомендуется 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 подход программной обработки 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 нижеследующий 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 must 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 must 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/множест-

ва **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 программного 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 [103].

Для организации процедуры наряду с предложениями, описывающими непосредственный алгоритм решаемой задачи (*а в ряде случаев и для обеспечения самого алгоритма*), ***Maple***-язык предоставляет ряд важных средств, обеспечивающих функции, управляющие выполнением процедуры. В *первую* очередь, 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 сказанное:

```
> 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 определять, 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 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 может ***procname*** 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*. Исходные тексты процедур нашей библиотеки, прилагаемой к книге [103], предоставляют неплохой иллюстративный материал по использованию переменных ***args***, ***nargs*** и ***procname*** процедуры в различных ситуациях.

4.3. *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 работы с ядром пакета 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. В противном случае 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.

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 в процедуре **P** соответственно. Тогда как 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:  H:= proc() local k; 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];   ⇒ [64, 109772628480, 64]
> G:=proc() [seq(args[k],k=1..nargs), mul(args[n], n=1..nargs)/add(args[p], p=1..nargs)] end proc:
> k:=64:  n:= 95:  p:= 99:  G(1, 2, 3, 4, 5, 6, 7, 8, 9):  [k, n, p];   ⇒ [64, 95, 99]
> k:=64:  P:= () -> [seq(args[k], k=1..nargs)]:  P(1, 2, 3), k;   ⇒ [1, 2, 3], 64
> k:=64:  P:= () -> [add(args[k], k=1..nargs)]:  P(1, 2, 3), k;   ⇒ [6], 64
> k:=64:  P:= () -> [mul(args[k], k=1..nargs)]:  P(1, 2, 3), k;   ⇒ [6], 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 *первый* 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 работы с пакетом, включая тело процедуры (*глобально переопределять ее можно внутри любой* ***Maple***-*конструкции*), то *локальная* переменная *областью определения* имеет лишь *тело* процедуры и вне процедуры она полагается *неопределенной*, если до того не была определена вне процедуры *глобально* переменная с тем же идентификатором. Данный механизм имеет глубокий смысл, ибо позволяет локализовать действия переменных рамками процедуры (*в общем случае черного ящика*), не влияя на общий вычислительный процесс текущего сеанса работы с пакетом. Примеры предыдущего фрагмента наглядно иллюстрируют *практическую* реализацию описанного механизма локализации переменных в ***Maple***-процедурах.

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 втором 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:

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

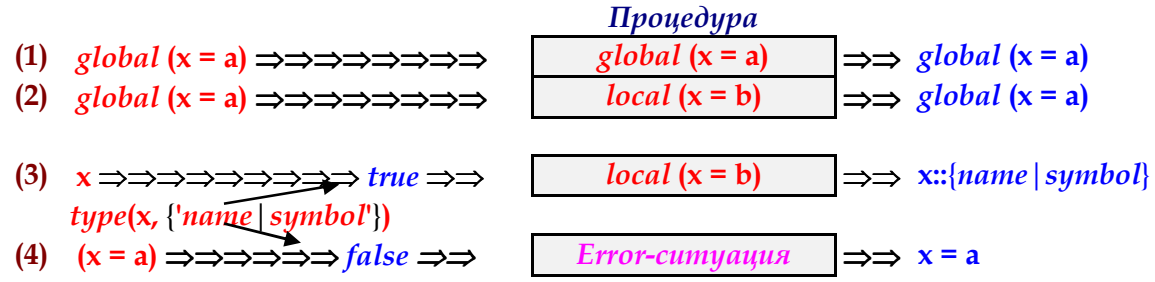
При этом, для таких объектов как процедуры, модули, таблицы и массивы (*включая матрицы и векторы в смысле* ***Maple***, *а не* ***NAG***) кодирование их имен в невычисленном формате *необяза-*

тельно, 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* или *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*, ***sum***, ***product***), 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 ***Maple***-языка 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 же 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))];
```

$$\left[64,\ \frac{y^{(n+1)}\,n^2-2\,y^n\,n^2+y^{(n-1)}\,n^2-y^{(n+1)}\,n+y^{(n-1)}\,n+2\,y^n-2}{(y-1)^3}\right]$$

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 рекурсивная цепочка выражений**,** вычисление которой реализует полностью рекурсивную 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 рекурсивной 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.

4.4. *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**, **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 ее 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 {***name***, ***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 {***options***|***option***}-секции, 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***, ***Copyright***, ***trace***, ***arrow***, ***operator***, ***remember*** и ***call_external***. При этом, перечень опций может зависеть от 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 ***remember***-опцию.

Для проверки процедур могут быть полезны и 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 фрагмент:

```
> ParProc(MkDir), ParProc(came);  map(ParProc, ['add', march, goto, iostatus, seq]);
```

$$\left[\begin{array}{c} Arguments = (F::\{symbol, string\}) \\ locals = (cd, r, k, h, z, K, L, \Lambda, t, d, \omega, \omega 1, u, f, s) \end{array}\right], \left[\begin{array}{c} Arguments = (E::anything) \\ locals = (f, h) \\ globals = (\_\_Art\_Kr\_) \end{array}\right]$$

[*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 8/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 8/lib/userlib"}]

```
> ParProc1(Sockets, 't'),  t;
Warning, module Sockets is in library [package, Maple, {"C:\\Program Files\\Maple 8/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 8/lib"}]

> 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 находятся в прилагаемой к книге библиотеке. 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(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
```

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 независимо 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 и 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 ***Maple-***процедурой, определенной любым из *трех* рассмотренных в разделе **3.1** способов, и кроме того, вычисленной. Следующий простой фрагмент иллюстрирует сказанное**:**

```
> 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 шести компонент *определения* процедуры, приписаного ***Proc***-переменной. В табл. **13** представлены **n**-значения для указанной конструкции и их смысловая нагрузка:

Таблица 13

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 6 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 шести компонент приведенной ***IAN*-процедуры**.

Как следует из вышесказанного, *тело процедуры* рассмотренными средствами не идентифицируется в качестве ее компоненты и *доступ* к нему возможен иными средствами, рассматриваемыми нами несколько ниже. Здесь же уместно отметить лишь ***dismantle*(P)-процедуру**, обеспечивающую вывод структуры данных, определяемой **P**-выражением. Процедура 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 рассмотренной в предыдущем примере ***IAN*-процедуре:**

```
> 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 возв-

ратиться к рассмотрению ***remember***-параметра **option**-секции, предварительно дав дополнительную полезную информацию.

Набор из процедуры ***dismantle*** и четырех встроенных функций ***assemble***, ***addressof***, ***pointto*** и ***disassemble*** 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 данного типа представлены и в [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 работы с пакетом в специальной ***remember***-таблице. Использование данного *механизма* требует определенных пространственных затрат, 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 ***remember***-таблица, которую можно получать по уже рассматриваемой конструкции ***op*(4, *eval*(Proc))**. 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 ядром наличия в ее *таблице* ***remember*** *аналогичного* вызова, *производится* возвращение соответствующего результата без повторного выполнения *тела* процедуры, т.е. *входы* в таблицу не дублируются. Целый ряд процедур/функций ***Maple***-языка 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 компромиса, 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 используется процедура ***forget*(Proc)**. Вызов ***forget*(Proc{, A}{, 0})** позволяет удалять из ***remember***-таблицы *входы* для определяемой *пер-*

вым 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 ***Help***-системе 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 выполнения *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 ***Share***-модулях. 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 исходный текст процедуры [12,103] и примеры ее применения:

```
Remember_T := proc (NP::{name, symbol})
local k, p, h, G, F, S;
global __T;
    if not type(NP, procedure) then ERROR("<%1> is not a procedure", NP)
    elif nargs < 2 then RETURN(op(4, eval(NP)))

    else assign('__T' = op(4, eval(NP)), F = cat([libname][1][1 .. 2], "\$$$vgs"))
    end if;
    for k from 2 to nargs do
        assign('G' = "", 'S' = "");
        if whattype(args[k]) = `=` then __T[op(lhs(args[k]))] := rhs(args[k])

        else
            for p to nops(args[k]) - 1 do
                G := cat(G, convert(args[k][p], 'string'), ",")
            end do;
            G := cat("'__T[", cat(G, convert(args[k][nops(args[k])], 'string')),

                "]';");
            assign('S' = convert(__T[args[k]], 'string')),
                assign('S' = cat(S[1 .. 4], S[6 .. -2]));
            assign('G' = cat(cat(S, ":="), G)), writebytes(F, convert(G, 'bytes')),
                close(F);

            read F;
            __T[seq(args[k][p], p = 1 .. nops(args[k]))] := %, fremove(F)
        end if
    end do;
    unassign('__T'), op(4, eval(NP))
end proc
> P3:= proc() options operator, arrow, remember; evalf(sum(args[k], k=1..nargs)/nargs) end proc:
[P3(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 аргументом должен выступать идентификатор процедуры, ***remember***-таблица которой модифицируется. Остальные фактические аргументы кодируются в следующем простом формате:

$[x_1, x_2, \ldots, x_n]$ = <*Значение*> либо $[x_1, x_2, \ldots, x_n]$

где *первый* формат определяет необходимость замены *выхода* на $(x_1, x_2, \ldots, x_n)$-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 процедуры ***Remember_T*** не рассматривается и оставляется читателю в качестве достаточно полезного упражнения.

В ряде случаев она оказывается весьма *полезным* средством, в первую очередь, при необходимости более эффективного использования *памяти* при работе с *рекурсивными* процедурами и часто используемыми процедурами в циклических конструкциях. Пример в нижеследующем фрагменте иллюстрирует строго линейную зависимость числа входов в ***remember***-таблицу 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) = 67.16666667, (30, 6, 99, 350, 520) = 201., (10, 17, 39, 44, 59, 64) =
                                          38.83333333])
> 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];
                                  [2178309, 0.]
> nops([indices(op(4, eval(Fib1)))]);
                                  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 упомянутого фрагмента*).

Для модификации ***remember***-таблицы произвольной ***Proc***-процедуры, определение которой содержало ***remember***-*параметр*, наряду с приведенной выше процедурой ***Remember_T*** 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 данных 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 простая **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 раз-*

рыва, 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 в случае его отсутствия. С этой целью выражение с особенностями оформляется процедурой с *параметром* ***remember*** в ее **option**-секции, процедура вычисляется и сразу же описанным выше способом производится 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 = ∞])
```

Этот фрагмент иллюстрирует еще *один важный* момент, связанный с процедурами, а именно. С каждой процедурой *независимо* от наличия в ее определении параметра ***remember*** ассоциируется ***remember***-таблица, сохраняющая историю присвоений по конструкциям ***Proc*(ФА):=** <*Значение*> и ***assign*('*Proc*(ФА)',** <*Значение*>**)**, где ***Proc*** **-** процедура, определенная *любым* допустимым способом (*см. раздел* **3.1**), 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 = 647, 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 (*см. раздел* **3.1**), 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/функции, с которой она ассоциирована.

4.5. *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 привоено определение процедуры*) вызывается подобно функции по конструкции следующего формата**:**

Proc**(**<*Фактические аргументы*>**)**

возвращая на переданных ей *фактических аргументах* соответствующее *значение***,** определяемое одним из вышеуказанных способов.

Из указанных механизмов возврата результата вызова процедуры наиболее часто используемым является использование ***RETURN*****-**функции или **return**-предложения форматов**:**

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]=Аргумент, T[1,2]=Тип);
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, 19.99, F(x), a*b, n .. p, array(1 .. 3, [10, 17, 39]));
```

Аргументы	*Тип*
RANS	*symbol*
"IAN"	*string*
2006	*integer*
19.99	*float*
$F(x)$	*function*
$a\ b$	*
$n\ ..\ p$	..
[10, 17, 39]	*array*

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 функцию ***RETURN***. 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, а 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 **return**-предложение имеет существенно меньшие выразительные возможности по представлению вычислительных алгоритмов в среде пакета.

Следует отметить, что, начиная с ***Maple 6***, разработчики объявили о том, что ***RETURN***-функция является устаревшим средством 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 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 сказанное:

```
> LO:= proc() local k; for k to nargs do if whattype(args[k]) <> 'float' then return 'LO(args)' end
if end do end proc:   LO(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 фрагмента 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-иденти-

фикатору требуемого значения, которое доступно сразу же после вызова процедуры. Следующий простой фрагмент иллюстрирует вышесказаное:

```
G := proc (L::list, x::evaln, y::evaln, z::evaln)
local a, b, c, k;
    `&ma`(a, b, c, [ ]), seq(`if`(type(L[k], 'float'), assign('a' = [op(a), L[k]]), `if`(
    type(L[k], 'integer'), assign('b' = [op(b), L[k]]),
    `if`(type(L[k], 'fraction'), assign('c' = [op(c), L[k]]), NULL))),
        k = 1 .. nops(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;
                    50.21094042, [19.42, 65.0, 16.10], [2006, 64, 350], [10/17, 1/33]

G1 := proc (L::list, x::evaln, y::evaln, z::evaln)
local a, b, c, d, k;
    `&ma`(a, b, c, d, [ ]), seq(`if`(type(L[k], 'float'), assign('a' = [op(a), L[k]]),
    `if`(type(L[k], 'integer'), assign('b' = [op(b), L[k]]), `if`(
    type(L[k], 'fraction'), assign('c' = [op(c), L[k]]),
      `if`(4 < nargs, assign('d' = [op(d), L[k]]), NULL))), k = 1 .. nops(L)),
    assign(x = a, y = b, z = c), evalf(sqrt(`+`(op(L)))),
    `if`(4 < nargs, assign(args[5] = d), NULL)
  end proc
> G1([2006, 64, 10/17, 19.42, A, 59/1947, 350, 65.0, V, 16.10, Z], a, v, z, 'h'),  a, v, z, h;
          sqrt(A + V + Z + 2521.138538), [19.42, 65.0, 16.10], [2006, 64, 350], [10/17, 1/33], [A, V, Z]
```

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 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 список 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 обоих процедурах рекомендуется обратить внимание на использование нашего оператора/процедуры **&ma** [103], обеспечивающего *присвоение* одного и того же выражения последовательности переменных, и ***assign***-процедуры для присвоения значений фактическим аргументам, через которые обеспечиваются *вторичные* возвраты.

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 сделано *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 результата вызова ***Maple***-процедуры через ее фактические аргументы позволяет наряду со стандартным или на основе ***RETURN***-функции механизмами организовывать *дополнительные* (*вторичные*) выходы, определяемые наличием или отсутствием при вызове процедуры соответствующих аргументов-идентифи-

каторов. Такая организация возврата результатов вызова широко практикуется в библиотечных процедурах как собственно самого пакета ***Maple***, так и наших [103].

Механизм возврата результатов вызова ***Maple***-процедуры через ее фактические аргументы позволяет наряду со *стандартным* или на основе ***RETURN***-функции (**return**-*предложения*) механизмами организовывать и дополнительные выходы, определяемые наличием или отсутствием при вызове процедуры соответствующих аргументов-*идентификаторов*. 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]. 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 необязательными *ключевыми аргументами* в качестве механизмов возврата результатов процедура может 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 достаточно 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

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 я не вижу 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 *непрерыв-*

ность 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 есть «*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 непросто 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 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
```

Фрагмент содержит подпроцедуру ***conv***, результат вызова которой обеспечивает *конвертирование* возвращаемого процедурой ***P1*** выражения в *тип*, определенный после заголовка процедуры. Возможно, в некоторых случаях такой прием может оказаться полезным.

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* процедуры с выводом 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 пакетом автоматически, пользователь имеет возможность как производить обработку ситуаций, определяемых спецификой вызываемой процедуры, так и в определенной мере перехватывать обработку *ошибочных* ситуаций, стандартно обрабатываемых ядром пакета. В следующем разделе рассматриваются средства обработки *ошибочных* ситуаций, имеющих особый смысл именно для процедурных и модульных объектов.

4.6. *Средства обработки ошибочных ситуаций в Maple*

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 ядром пакета ситуации такого рода возвращаются в *предопределенную* ***lasterror***-переменную, значение которой определяет *последнюю* обнаруженную в текущем сеансе ошибку и доступно текущему документу для необходимой обработки возникшей ситуации. Значение ***lasterror***-переменной определяется только в момент возникновения ошибочной ситуации и сразу же после загрузки пакета она является неопределенной. К ***lasterror***-переменной непосредственно примыкает и ***tracelast***-функция, обеспечивающая вызов последней ошибочной ситуации из *стэка ошибок*. 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 фрагмент 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 пустому ***traperror*()**-вызову функции. 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 *аварийных* 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 функцией ***traperror*: *interrupted*** (*прерванное вычисление*), ***assertion failed*** (*генерируется при активном* ***ASSERT***-*механизме*), ***out of memory*** (*недостаток памяти*), ***stack overflow*** (*переполнение стэка*), ***object too large*** (*объект слишком велик*). Это объясняется невозможностью восстановления на момент возникновения указанных ситуаций.

При этом следует иметь в виду, что функция ***traperror*(V)** не связывает с **V**-выражением *ошибочной* 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*) либо *упрощения* выражения. Следующий простой фрагмент иллюстрирует вышесказанное:

```
> 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 этих 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 ядра 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.
```

$$Time := ( ) \rightarrow evalf\left(map2\left(`*`, time( ), \left[\frac{1}{60}, \frac{1}{3600}\right]\right), 2\right)$$

```
> Time();   ⇒ [0.50, 0.0083]
```

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 вида "*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. В качестве примера применения ***timelimit***-функции приводится простая ***Timetest*(t, x, V, p)**-процедура, возвращающая ***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 **p**-аргумент процедуры возвращается список, первый элемент которого определяет *время вычисления* искомого выражения, а второй - *результат* вычисления либо ***undefined***-значение соответственно.

```
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, [0.514, undefined]
> Timetest(0.5, 10^8, G, p), p;   ⇒ false, [0.517, undefined]
```

В приведенном фрагменте ***Timetest***-процедура используется 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 позволяют 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 выводом 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 следующего вида:

error {**Msg** {, **p1, p2, …, pn**}} {**:** | **;**}

производится немедленный выход из процедуры в точке его выполнения с выводом 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 такой нет, то выводится сообщение о неспецифицированной ошибке. Если параметр имеет вид '**%n**', то в выходном 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, 3 and 7 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 ***Maple 6.*** 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 простой 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 выводом соответствующего сообщения. После этого на основе анализа значения ***lasterror***-переменной производится выбор пути дальнейших вычислений. Последний пример фрагмента иллюстрирует вывод ошибочного сообщения, связанного с определенным соотношением между аргументами, для *непоименованной* процедуры пользователя. При этом, использованы как ***ERROR***-функция, так и эквивалентное ей **error**-предложение.

Следует отметить, к сожалению, ***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 следующий достаточно простой фрагмент:

```
> 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
```

Это же, в свою очередь, не позволяет обрабатывать такого типа ошибки средствами пакета. В свою очередь, другого типа недоработки пакета 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 данной 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 [103], например, путем корректировке **try**-предложения:

```
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. Подобная практика не соответствует принципам создания качественного **ПО**, не говоря уже о самом престиже разработчиков.

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 простой 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 **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-***языка. В этой связи пользователю предоставляется возможность организации *собственных* механизмов обработки как ошибочных, так и особых ситуаций, связанных с алгоритмом вычислений или передаваемыми процедуре аргументами. Как это иллюстрирует следующий достаточно простой фрагмент**:**

```
VK := proc ()
local k;
      for k to nargs do
            if whattype( args[ k ] ) ≠ 'float' then
                  error  "%-1 argument has type different from 'float'" , k
            end if
      end do ;
      "Body of procedure"
end proc
> VK(6.4, 5.9, 10.2, 3, 17);
Error, (in VK) 4th argument has 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);   ⇒  "Body of procedure"
```

Процедура ***VK*** программно обрабатывает ситуацию получения аргумента *некорректного* типа на основе **error**-предложения с выводом соответствующей диагностики.

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]. Здесь же мы лишь сделаем одно замечание относительно реализации функций ***RETURN***, ***ERROR*** и предложений **return**, **error**. В настоящее время эти средства попарно функционально эквивалентны, однако синтаксис их использования при создании процедур и программных модулей различен, а именно. В пакете ***Maple 5*** и ниже использовались встроенные функции ***RETURN***, ***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, синтаксис которых 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*" аварийную ситуацию, предоставляя возможность более гибкой ее обработки. Простой фрагмент иллюстрирует вышесказанное:

```
H := proc ()
local k, h, z;
      assign(z = 64 + sqrt(sum(args[k], k = 1 .. nargs))),
            assign(h = traperror(z/`+`(args)));
      if z = 0 then return `Null-arguments`
      elif h ≠ "numeric exception: division by zero" then return h
      else z/(59 + sum(abs(args[k]), k = 1 .. nargs))
      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 ре-

зультаты на момент прерывания, что может оказаться весьма полезным при отладке, например, рекурсивных процедур.

Механизм try-предложения. Для *программной* обработки ошибочных ситуаций, отражаемых в ***lasterror***-переменной пакета, ***Maple***-язык располагает встроенной функцией ***traperror***, рассмотренной выше, и процедурами ***stoperror*** и ***unstoperror***, а также **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.

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 вывода 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 к следующему. После получения управления **try**-предложение инициирует выполнение блока **A** предложений ***Maple***, в которых мы пробуем 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 или* (**3**) *закрывающая скобка* **`end try`**). В этом 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 ***ERROR***-*функцию* (**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.

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; в качестве строк совпадения **catch**-блоки используют *диагностику* по ошибкам, сформированным на основе предложений **error** и **return**:

```
> F:= x -> `if`(x <= 42, sin(x), `if`(x > 42 and x <=47, VG(x), `if`(x > 47 and x <=76, Sv(x), `if`(x > 76
and x <= 89, Artur(x), `if`(x <= 96, Kristo(x), Arn(x))))));  seq(F(x), x = [42, 45, 53, 77, 95, 100]);
                    sin(42), VG(45), Sv(53), Artur(77), Kristo(95), Arn(100)
```

```
F1 := 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 ≤ 89 then error "Branch_4"
        elif 89 < x and x ≤ 96 then error "Branch_5"
        else error "Branch_6"
        end if
    catch "Branch_1": sin(x)

    catch "Branch_2": VG(x)
    catch "Branch_3": Sv(x)
    catch "Branch_4": Art(x)
    catch "Branch_5": Kristo(x)
    catch "Branch_6": Arn(x)
    end try
end proc
> seq(F1(x), x=[42, 45, 53, 77, 95, 100]);   ⇒ sin(42), VG(45), Sv(53), Artur(77), Kristo(95), Arn(100)
```

На наш взгляд, данный фрагмент достаточно прозрачно иллюстрирует механизм *инициирования* **catch**-блоков, обеспечивающий довольно 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 (*данная 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:

• *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 книге [103] представлен целый ряд интересных применений **try**-предложения для обработки различного типа ошибочных и особых ситуаций.

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, либо 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 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.

4.7. *Расширенные средства Maple-языка для работы с процедурами*

Рассмотрим теперь вопросы использования механизма процедур несколько детальнее, учитывая прикладную значимость процедур. 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 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 последующим ее вычислением и вызовом. Тогда как второй пример фрагмента иллюстрирует использование определенной вышеуказанным способом ***G*-**процедуры в вычислительной конструкции. Приведенный выше способ *определения* процедуры оказывается весьма полезным в целом ряде приложений. Между тем, *вложенные* процедуры можно создавать и на основе стандартного определения. Следующий простой фрагмент иллюстрирует применение обоих способов для определения вложенных процедур, генерируемых основной процедурой в зависимости от условий**:**

```
WM := proc (x::numeric , y::numeric , z::integer )
      `if`( not  member(z, {10, 17}),
            ERROR("impermissible 3rd argument <%1>", z), `if`(z = 10,
            assign(AG, proc (x, y) [x, y, evalf(sqrt(x×y), 6)] end proc  ),
            assign(AG, 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]]
> eval(AG);
                          proc (x, y) [x, y, evalf(sqrt(x*y),6)] end proc

WM1 := proc (x::numeric , y::numeric , z::integer )
local  AV, AG;
      `if`( not  member(z, {10, 17}),
            ERROR("impermissible 3rd argument <%1>", z), NULL );
      if z = 10 then  AG := proc (x, y) [x, y, evalf(sqrt(x×y), 6)] end proc
      else  AG := proc (x, y) [x, y, evalf(sqrt(x + y), 6)] end proc
      end if
end proc
> [WM1(59, 64, 10), AV(64, 10)];
                    [proc (x, y) [x, y, evalf(sqrt(x*y),6)] end proc, AV(64,10)]]
```

```
WM2 := proc (x::numeric , y::numeric , z::integer )
global AV, AG;
      `if`( not member(z, {10, 17}),
            ERROR( "impermissible 3rd argument <%1>" , z ), NULL );
      if z = 10 then AG := proc (x, y) [x, y, evalf( sqrt(x×y ), 6 )] end proc
      else AG := proc (x, y) [x, y, evalf( sqrt(x + y ), 6 )] end proc
      end if
end proc
> [WM2(59,64,10)(10,17), AG(59, 10), AV(64, 17)];  ⇒ [[10, 17, 13.0384], [59, 10, 24.2899], AV(64,17)]
```

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 вычисленное *определение* 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. **14.** 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 14

	Proc:= proc() ... end proc	***assign(Proc, proc() ... end proc)***
global	***global***	***global***
local	***local***	***local***
не декларирована	***local***	***global***

Представленный в табл. **14** 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, в противном случае 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 перемен-

ным, 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 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 же 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 работы с ядром пакета, 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 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 := proc (a, b, c)
local x, y, z;
global X, Y, Z;
    x := a×(X^2 + Y^2 + Z^2)^(1/2);
    y := (x + b)×exp(Z)/(ln(X) + sin(Y));
    z := x×y/(a×x + b×y);
    x×y/(x + z)
end proc
```

```
> X:= 6.4:  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 не сложных 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 ***userinfo***-функции 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 опе-

ративную 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 ***profile*(P1,...,Pn)**-процедура, которая в случае успешного вызова возвращает значение ***NULL*** и по которой устанавливается режим *мониторинга* вызовов **Pj**-процедур, определяемых ее фактическими аргументами. При этом, следует иметь в виду, что попытка вызова ***profile***-процедуры для уже находящейся в режиме ***profile***-мониторинга процедуры или функции вызывает ошибочную ситуацию, как это иллюстрирует пример нижеследующего фрагмента. По вызову ***profile*()** производится *мониторинг* вызовов всех процедур и функций в текущем сеансе работы с пакетом. 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 и/или функциям. Повторное применение ***unprofile***-процедуры вызывает ошибочную ситуацию. Следующий фрагмент иллюстрирует применение рассмотренных средств для мониторинга пользовательской ***VSV***-процедуры и некоторых встроенных функций ***Maple***-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 назначению 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 сказано при рассмотрении создания библиотек пользователя.

В заключение настоящего раздела отметим, что более искушенный пользователь может при работе с *процедурами* воспользоваться специальными процедурами ***procbody*** и ***procmake*.** По вызову процедуры ***procbody*(*Proc*)** возвращается специальная *невычисленная* форма процедуры с ***Proc*-**именем, которая может быть протестирована и модифицирована, *не затрагивая* исходной ***Proc*-**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:  AGN(a, b, c, d, e, f, g, h, p ,t);
```

$$\frac{a}{10}+\frac{b}{10}+\frac{c}{10}+\frac{d}{10}+\frac{e}{10}+\frac{f}{10}+\frac{g}{10}+\frac{h}{10}+\frac{p}{10}+\frac{t}{10}$$

```
> AGN1:= procbody(AGN);  procmake(AGN1);
```

$$AGN1 := \&proc\left(\&expseq(\ ),\&expseq(\ ),\&expseq(\ ),\&expseq(\ ),\ \frac{+\ \&function\ \&expseq(\&args_{-1})}{\&args_0},\&expseq(\ ),\&expseq(\ ),\&expseq(\ )\right)$$

```
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 наглядно иллюстрирует следующий 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, от отображаясь в ***lasterror*-**переменной пакета. Принципиально, данная новация не столь 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.

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 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]
> P1:= 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:
> P1(64, 59, 39);   ⇒ [[64, 59, 39], 3, 0, 0, undefined]
> P1(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* процедур «*сверху-вниз*».

4.8. *Расширение функциональных средств Maple-языка*

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 средства главной ***Maple***-библиотеки пакета.

Каждое уникальное расширение средства (*типа* ***type*** *или* ***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 ***type***-функцией.

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 дает 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.

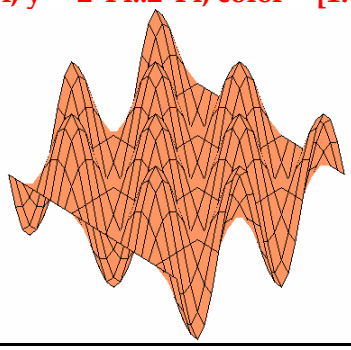
Все встроенные процедуры и большинство библиотечных процедур, допускающие пользовательские расширения, используют *классический* механизм, основанный на конкатенации имен, а именно. Расширяемой процедуре ***name*** дают новые функциональные возможности, определяя выражение (*как правило, процедура*) с именем формата `` `name/new` ``, где ***new*** – имя расширения. Например, новый тип для бинарных выражений (*распознаваемый стандартной* ***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/aaa` `` 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/aaa`:= proc(x::numeric) if x < 0 then true elif x > 0 then false else FAIL end if end proc:
> type(0, 'aaa');
Error, result from type `aaa` 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 переменную ***_ColorType*** 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 вызов процедуры возвращает ***false***-значение, то *переменная* ***_ColorType*** возвращается неопределенной. Процедура `` `type/color` `` имеет 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 := 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]
> 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 случаев. Ниже представлен ряд других примеров применения *классическго* механизма для расширения стандартных средств пакета, с которыми рекомендуется разобраться в качестве довольно полезного упражнения.

```
                    type/float_list := proc (G)
                          `if`(type( G, 'list'),
                          `if`(member(false, { op(map( type, G, 'float' )) } ), false, true ), false )
                    end proc
> 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 rig    \
hts 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 ))),
                  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);
            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 = "" then return false
                else assign('rm' = came(sv(rm)), 'im' = came(sv(im)))
                end if
            catch : return interface(warnlevel = h), false
            end try ;
            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);
                                        a + b   a + b   a + b   a + b   a + b
                                        -----, -----, -----, -----, -----
                                        c − d   c − d   c − d   c − d   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;
                                        a + b   a + b   a + b   a + b   a + b
                                        -----, -----, -----, -----, -----
                                        c − d   c − d   c − d   c − d   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". Между тем, 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** или применения оператора **&ma** возвращает ***NULL****-значение* с выполнением указанных назначений. В целом ряде приложений оператор **&ma** достаточно полезен.

Наконец, *последний* пример фрагмента иллюстрирует определение пользовательского оператора **&ma** [103], обеспечивающего *многократные* присвоения одного и того же выражения переменным или 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 языком в качестве ключевых слов для идентификации *пользовательских* операторов.

Пользовательский оператор можно использовать в качестве *унарного префиксного*, *бинарного инфиксного* операторов или *вызова* процедуры/функции. В любом из указанных случаев производится вызов процедуры, определение которой 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 процедуры **&**<*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:

$$\&\langle Символ\rangle\ (x_1, x_2, \ldots, x_n) \equiv \&\langle Символ\rangle(x_1, x_2, \ldots, x_n) \qquad (n \geq 1)$$

Простой фрагмент ниже 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)];
```

$$\left[-\frac{b\,c}{c^2+d^2}-\frac{a\,d}{c^2+d^2}-\left(\frac{a\,c}{c^2+d^2}-\frac{b\,d}{c^2+d^2}+h\right)I,\ -\frac{b\,c}{c^2+d^2}-\frac{a\,d}{c^2+d^2}-\left(\frac{a\,c}{c^2+d^2}-\frac{b\,d}{c^2+d^2}+h\right)I\right]$$

```
> `&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]
> a  &Kr  b;
Error, invalid input: &Kr expects its 1st argument, x, to be of type numeric, but received a
> `&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 достаточно 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 один полезный пример.

```
&Shift := proc ()
local k;
    `if`(nargs < 3, ERROR("incorrect quantity <%1> of actual arguments" , nargs),
        `if`( not (type(args[1], 'symbol') and type(args[2], 'list')), ERROR("\
        incorrect type of the first argument <%1> and/or the second argument   \
        <%2>", args[1], args[2]), `if`(nops(args[2]) ≠ nargs − 2, ERROR("in\
        correct 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  (Art, [a, b, c, d, g, s, q], x, y, z, t, u, r, h);   ⇒ Art(x+a, y+b, z+c, t+d, u+g, r+s, h+q)
> &Shift  (Art, kr, x, y, 64, z);
Error, (in &Shift) incorrect type of the first argument <Art> and/or the second argument <kr>
```

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 фрагмент представляет определение оператора и некоторые результаты его применения для реализации указанного выше оператора *функционального сдвига* **&Shift.**

Читателю в качестве *весьма* полезного упражнения рекомендуется**,** используя описанный метод**,** запрограммировать несколько **&**-операторов, определяющих какие-либо интересные нестандартные операции над данными и/или структурами данных рассмотренных 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 простой фрагмент**:**

```
> PT[Sr]:= () -> `+`(args)/nargs:  PT[H]:= () -> sqrt(`*`(args)):  PT[G]:= () -> (`+`(args))^2:
> type(PT, 'table'), whattype(eval(PT)), eval(PT);
```

$$true,\ table,$$

$$\mathrm{table}\left([H = (() \to \sqrt{\texttt{`*`}(\mathrm{args})}),\ G = (() \to \texttt{`+`}(\mathrm{args})^2),\ Sr = \left(() \to \frac{\texttt{`+`}(\mathrm{args})}{\mathrm{nargs}}\right)]\right)$$

```
> PT[G](64, 59, 39, 10, 17, 44), Sr(64, 59, 39, 10, 17, 44), PT[H](64, 59, 39, 10, 17, 44);
```

$$54289,\ Sr(64, 59, 39, 10, 17, 44),\ 16\sqrt{4302870}$$

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 [8-14,29,30,41,103].

4.9. *Иллюстративный пример оформления Maple-процедуры*

Представленное выше описание структурной организации ***Maple***-процедуры иллюстрируется нижеследующим фрагментом, отражающим основные *ее* элементы, и позволяет непосредственно приступать с учетом ранее рассмотренного материала к созданию, *на первых порах*, относительно несложных пользовательских процедур различного назначения. В основу иллюстративной процедуры положим одну практически важную задачу создания цепочки каталогов любого уровня вложенности, которая имеет многочисленные приложения при программировании доступа к внешним файлам данных пакета. Хорошо известно, что встроенная функция ***mkdir***, находящаяся в библиотеке **iolib** пакета, не позволяет создавать цепочку каталогов уровня вложенности более одного, например:

```
> mkdir("C:/temp/Grodno/Grsu");
Error, (in mkdir) file or directory does not exist
> mkdir("C:/temp/Tallinn");
Error, (in mkdir) directory exists and is not empty
```

При этом, вызов ***mkdir***-функции вызывает ошибочную ситуацию не только при попытке создать цепочку каталогов уровня вложенности более одного, но и инициирует ошибочную ситуацию (*не вполне некорректную*) даже при указании уже существующего и пустого каталога, последнего в цепочке каталогов. В целях расширения возможностей стандартной функции ***mkdir*** нами и была создана процедура ***MkDir***, предлагаемая в качестве примера.

Процедура ***MkDir*** [103] предназначена для создание цепочки каталогов любого уровня вложенности и/или пустого закрытого файла. Процедура имеет следующий формат вызова:

MkDir(*dirName* {, 1})

где ***dirName*** – имя, цепочка каталогов или полный путь (*может быть строка или символ*) и **1** – (*необязательный*) индикатор создания файла данных.

Процедура ***MkDir*** 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 допускает кодирование фактического аргумента ***dirName*** строкой или символом следующего формата:

<*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 обратная наклонная черта (*backslash*), то она должна удваиваться, т.к. строки ***Maple*** используют этот символ в качестве управляющего (*escape*) символа.

Если при вызове процедуры ***MkDir*(*dirName*, 1)** был указан второй фактический аргумент **1**, то процедура рассматривает последний элемент цепочки, определенной *первым* аргументом ***dirName***, как файл, который будет создан как *пустой закрытый файл* данных. Так, вызов процедуры **MkDir(«c:/Temp/Dir/ArtKr169.txt», 1)** создает как цепочку каталогов **«c:/Temp/Dir»** (*в пределах отсутствующих компонент*), так и *пустой закрытый* файл **«c:/Temp/Dir/ArtKr169.txt»**.

Необходимо обратить внимание на следующее обстоятельство. Файловая концепция *базовой* операционной системы ***MS DOS*** специально не различает непосредственно файл и каталог. Поэтому, даже ***MS DOS*** команда ***mkdir*** не может создать подкаталог в каталоге, содержащем файл того же названия. Для устранения данного недостатка, процедура ***MkDir*** использует искусственный прием, состоящий в следующем. Если при создании требуемого подкаталога процедура обнаруживает присутствие файла того же названия в каталоге предыдущего уровня, то создаваемый подкаталог получает новое имя, сформированное из заданного имени путем добавления к нему префикса «_». Аналогичная ситуация имеет место, если при созда-

нии требуемого файла ***MkDir*** обнаруживает присутствие подкаталога с тем же именем; об этом выводится соответствующее *предупреждения*. В общем случае это предупреждение имеет следующий вид «Path element <%1> has been found, it has been renamed on <%2>».

Успешный вызов процедуры ***MkDir*(F)** сохраняет текущий каталог неизменным и возвращает полный путь к требуемому каталогу или файлу данных, заданному фактическим аргументом **F**. Вызов процедуры ***MkDir*({`` | ""})** возвращает ***NULL***-значение, т.е. ничего. Более того, если требуемый путь отсутствует, то он создается с выводом соответствующего предупреждения вышеупомянутого вида, если это необходимо. Неудачный вызов вызывает особую ситуацию. В отличие от стандартной ***Maple***-функции ***mkdir***, успешный вызов ***MkDir***-процедуры всегда возвращает требуемый полный путь. Возвращаемый путь имеет стандартную нотацию ***Maple*** в нижнем регистре клавиатуры. Данный подход позволяет упрощать программирование многих задач, имеющих дело с доступом к файлам данных. Процедура ***MkDir*** существенно расширяет возможности стандартной функции ***mkdir***, которая позволяет создавать только каталоги одного уровня вложенности. Процедура во многих случаях позволяет существенно упрощать программирование задач обработки *элементов* файловой системы компьютера и задач, имеющих дело с доступом к файлам различных типа и назначения. В представленном ниже фрагменте приведен *исходный* текст процедуры и примеры ее применения для создания цепочек каталогов и/или пустых файлов данных.

```
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
```

Типичные примеры применения процедуры:

```
> MkDir("C:/Temp/rans/ian\\VGTU/AGN\\Art/Kr\\Svet/Arne");
                        "c:\temp\rans\ian\vgtu\agn\art\kr\svet\arne"
> MkDir("C:/Temp/rans/ian\\VTU/AGN\\Art/Kr\\Svet/Arne\\RANS.ian/", 1);
                    "c:\temp\rans\ian\vtu\agn\art\kr\svet\arne\rans.ian"
> MkDir("C:/Temp/rans/ian\\VTU/AGN\\Art/Kr\\Svet/Arne\\RANS.ian/");
Warning, Path element <rans.ian> has been found, it has been renamed on _rans.ian
                    "c:\temp\rans\ian\vtu\agn\art\kr\svet\arne\_rans.ian"
> MkDir("C:/temp/rans/ian/vtu/agn/art/kr/svet/arne/_rans.ian/Art.Kr", 1);
                "c:\temp\rans\ian\vtu\agn\art\kr\svet\arne\_rans.ian\art.kr"
> type("C:/temp/rans/ian/vtu/agn/art/kr/svet/arne/_rans.ian/Art.Kr", 'file'),  iostatus();
                                    true,  [0, 0, 7]
> MkDir("c:\\temp\\avz\\ASV\\art\\kristo\\svet\\aaa");  type(%, 'dir');
Warning, Path element <avz> has been found, it has been renamed on <_asv>
                        "c:\temp\_avz\asv\art\kristo\svet\aaa"
                                    true
> MkDir("C:\\temp\\avz\\ASV\\art\\kristo\\svet\\Order.txt",1);  type(%, 'file');
Warning, Path element <avz> has been found, it has been renamed on <_asv>
                        "c:\temp\_avz\asv\art\kristo\svet\order.txt"
                                    true
> MkDir("C:"), MkDir("C:/"), MkDir("C:/Temp\\"), MkDir("A:/Academy/Rans_IAN", 1);
                        "c:\", "c:\", "c:\temp", "a:\academy\rans_ian"
> MkDir("C:/Temp/Temp/Temp\\Temp/Temp"),  map(MkDir, [``, ""]);
                        "c:\temp\temp\temp\temp\temp", []
> MkDir("c:/Temp/Temp/Temp\\Temp/Temp", 1);   ⇒ "c:\temp\temp\temp\temp\_temp"
Warning, Path element <temp> has been found, it has been renamed on <_temp>
```

Однако, несмотря на отмеченные преимущества процедуры ***MkDir***, мы все же рекомендуем избегать *дублирования* имен файлов и подкаталогов в том же самом каталоге. Это обеспечит более простые алгоритмы обработки элементов файловой системы компьютера и обработки ошибочных ситуаций, возникающих в процессе доступа к файлам данных различных типа и назначения. Процедура существенно упрощает программирование задач, имеющих дело с доступом к файлам данных различных типов, характера и назначения. Данная процедура существенно используется в целом ряде процедур нашей библиотеки [103], работающих с файловой системой компьютера. Она превосходно зарекомендовала себя при решении многих прикладных задач и уже даже по ее описанию следует вполне *однозначный* вывод о целесообразности включения данного средства (*либо его аналога*) в стандартные поставки ***Maple***. В

настоящее время данная процедура прошла *апробацию* в целом ряде университетов Европы. Наряду с представленной ***MkDir***-процедурой, носящей *общий* характер и *расширенные* функции по работе с файловой системой компьютера, в нашей библиотеке [103] имеются еще две подобные процедуры ***MkDir1*** и ***MkDir2***, проще, но с более ограниченными функциями и *без* средств обработки особых и ошибочных ситуаций.

Читателю в качестве полезного упражнения рекомендуется разобраться с приведенной процедурой. При этом, рекомендуется обратить внимание на использование **try**-предложения и ряда процедур {***CF***, ***Empty***, ***null***} и новых типов {*dir*, *file*}, поддерживаемых библиотекой [103] и полезных в практическом программировании в среде ***Maple***-языка.

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/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
> y(64, 59), convert(a = b, `proc`);   ⇒  √7577, 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 полезных, эффективных (*а в ряде случаев и нестандартных*) приемов программирования в ***Maple*** можно найти в архиве, поставляемом с книгой [103] или *бесплатно* скачать с *адреса*, указанного в [91]. В заключение настоящей главы кратко остановимся на вопросах отладки ***Maple***-программ.

4.10. *Элементы отладки Maple-процедур и функций*

Проблема отладки может возникать либо 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 этом 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, как 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 вызова процедуры {***untrace***|***undebug***}**(P1,P2, ..., Pn)** 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], начальную версию книги можно бесплатно получить в [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 же *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 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**-секций которого производится "**DBG>**"-метками. Отмена режима отладки производится по команде {**stop**|**quit**|**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**, **where**, **showstat**, **showstop**, **list**, **stopat**, **stoperror** и др., обеспечивающих целый ряд важных отладочных функций, с которыми детально можно ознакомиться в книгах [8,9] или в определенной степени по **Help**-системе пакета, а также по поставляемой с пакетом документации [78-84]. Там же достаточно детально можно ознакомиться 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 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 часто расширяют функции отладчика.

Прежде всего, в целях удобства установки в процедурах контрольных точек рекомендуется пронумеровать составляющие их предложения, что обеспечивает команда **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 и формальные аргументы про-

цедуры. 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 **unstopat**-команды. 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**. 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 ядра пакета. В противном 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 данном 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] либо [91] можно найти ряд весьма поучительных примеров применения рассмотренных **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 **DEBUG**-функции 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 требуемыъх 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**) или *вывод* значений с их *возвратом* (*формат* **2**), как это наглядно иллюстрирует достаточно прозрачный фрагмент, представленный ниже.

```
> 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*(*Proc*)** производится вывод протокола результатов семантического анализа ***Proc***-процедуры, включая ее *никогда* не выполняемые предложения. Следующий простой фрагмент иллюстрирует результат такого применения ***maplemint***-процедуры 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. Именно таким 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 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; *неиспользуемые* 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 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 процедура, как* ***Proc*** *из последнего примера не доступна 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 иллюстрирует нижеследующий 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 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. Детальнее 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 довольно сложных и эффективных программных средств.

Рассмотрев *процедурные* объекты ***Maple***-языка, обеспечивающие достаточно высокий уровень модульности программирования в среде пакета, представим теперь новый тип (*начиная с* **6***-го релиза*) объектов языка, не только повышающих уровень модульности разрабатываемых в среде ***Maple*** программных средств, но и в значительной мере обеспечивающих *объектно-ориентированную* технологию программирования в его среде. Такими объектами являются *программные модули*, представление которых предварим небольшим, но полезным экскурсом в недавнее прошлое программирования.

Глава 5. *Организация программных модулей Maple-языка*

5.1. *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.

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 понятие *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***-языком 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 или 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. Такая 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].

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 программных 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** (*задачи линейной алгебры*), **stats** (*задачи статистического анализа 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]. 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 программных модулей и их использования в ***Maple***-программах иллюстрируются соответствующими примерами.

5.2. *Организация программных модулей Maple-языка*

Организация *программного модуля* (**ПМ**) очень напоминает организацию ***Maple***-процедуры, достаточно детально рассмотренной в предыдущей главе. В общем случае организация (*структура*) программного модуля имеет следующий 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 ключевым словом **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 **local**, **global**, **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 работы с пакетом 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* или 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 типа 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 неизвестным (***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 поименования 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 поименованно-

го *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 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 [12-14,29-33,39,45,103]. Достаточно полезная процедура ***mod21*** решает данную проблему конвертирования модулей. Ранее мы уже представляли ее аналог, но 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 данном 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 ниже достаточно прозрачно иллюстрируют вышесказанное.

```
mod21 := proc (M::`module`)
local a, b, c, d, t, p, k, v, sf, h, v;
    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(v = interface(warnlevel)), null(interface(warnlevel = 0)),
                assign('h' = cat(MkDir(args[2]), "\", M, ".mod1"));
            (proc () null(interface(warnlevel = v)); save M, h end proc )( ), h;
            WARNING("module <%1> has been converted into the first typ \

                e, and 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 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 или несколько строк, составляющих единое описание модуля. Следующий 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 QR ()  local  Sr;  export  Dis;  options load = Sr; Dis:= () -> sqrt(sum((args[k] -
Sr(args))^2, k = 1..nargs)/nargs);  Sr:= () -> sum(args[k], k = 1..nargs)/nargs;  end module:
> 6*QR:- Dis(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* или *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 справочной системе 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 работы с пакетом, 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:

<*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 доступными в текущем сеансе и к ним можно обращаться без ссылок на содержащий их модуль, например:

```
> 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 либо через **global**-декларацию, либо 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. отмеченный прием работает лишь при условии вычисления *определения* модуля в текущем сеансе. Следовательно, описанный выше прием достаточно искусственен и ограничен, и носит 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 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(M), With(M1);   ⇒  [], []
```

Данное 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 работы с пакетом, что иллюстрирует следующий простой 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.* **6.37** [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 отличающимся 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 фрагмент иллюстрирует применение ***thismodule***-переменной:

```
> 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 или экспортируемую переменную ***ModuleLoad***, то определенная ею процедура вызывается, когда модуль читается из ***Maple***-библиотеки его содержащей. Если модуль имеет локальную либо экспортируемую переменную ***ModuleUnload***, то определенная ею процедура вызывается, когда модуль более 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 пакета по **?module.** Как следует из их описания, особо принципиально существенного данные переменные не несут, однако в ряде случаев могут оказаться достаточно полезными.

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;
```

Параметризация позволяет создавать модули, легко настраиваемые на конкретные условия применения, что обеспечивает их гибкость и мобильность при программировании задач из различных приложений. Следующий простой фрагмент иллюстрирует применение данной конструкции для *параметризации* конкретного программного модуля, определенного в процедуре ***SveGal:***

```
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    36465
                                                    64,  12007462
```

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** 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 задач в среде ***Maple*** и в целом ряде случаев существенно упрощают программирование.

5.3. *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 объекто 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*>**)** или 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***.

При этом, если в *файле* определено ***m***-расширение имени, то файл сохраняется во *внутреннем* ***m***-формате пакета, в противном случае используется *текстовый* 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 пакетом 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 *весьма* 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. Между тем, 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 использование **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 обоих 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.

Загрузка сохраненного по **save**-предложению файла производится 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 работы с ядром пакета, 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 же загрузки ***m***-файла информации не выводится и возвращается ***NULL***-значение. Примеры следующего фрагмента иллюстрируют применение **read**-предложения для *загрузки* файлов обоих форматов (*входного и внутреннего*):

```
> restart;  read("C:/Temp/Sr");  5*Sr(64, 59, 39, 10, 17);
                         Sr := ( ) -> `+`(args) / nargs
                                   189
> 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 и/или функций, а также сохранения часто используемых ***Maple***-конструкций. Независимо от формата (***m***-*формат либо* ***ASCII***-*формат*) сохраненного по **save**-предложению 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*** [103].

Вызов процедуры ***saveall*****(F)** возвращает *реальный* путь к принимающему файлу данных (*имя которого или путь к нему определены фактическим аргументом* **F**), содержащему все вычисленные в текущем сеансе *имена* в виде последовательности предложений присвоения. 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, например:

```
> 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"
```

В целом ряде случаев процедура ***saveall*** 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 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*** 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 фай-

лах данных с произвольными путями к ним. Примеры фрагмента, представленного ниже, иллюстрируют некоторые наиболее типичные применения ***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:

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*. 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. Ниже представлен фрагмент с исходными текстами обоих процедур и примерами их применения для сохранения/чтения программных модулей.

```
savem1 := proc(F::{string, symbol}, M::mod1)
local a;
    if not (map(type, {args[2 .. -1]}, 'mod1') = {true}) then
        error " modules should be of the first type"
    end if ;
    if "" || F[-2 .. -1] ≠ ".m" then a := F || ".m" else a := F end if ;
    seq(assign(args[k] = convert(eval(args[k]), 'string')), k = 2 .. nargs);
    (proc() save args, a end proc )(args[2 .. -1]), a
end proc
```

```
readm1 := proc (F:file)
local b, c, k;
      assign( b = {  }, c = 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; 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:
> savem1("C:/temp/Academy", M, M1, M2);   ⇒   "C:/temp/Academy.m"
> 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`
> readm1("C:/temp/Academy.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);
                                              183589120, 183589120
> M1:- y(64, 59, 39, 17, 10, 44),  y(64, 59, 39, 17, 10, 44);   ⇒   233/6, 233/6
> M2:- z(64, 59, 39, 17, 10, 44),  z(64, 59, 39, 17, 10, 44);   ⇒   233, 233
```

В качестве *весьма* полезного упражнения читателю рекомендуется рассмотреть организацию *обоих* процедур. Целый ряд полезных средств для сохранения программных модулей в файлах *внутреннего* ***Maple***-формата, а также для работы с файлами *данного* формата представлен в нашей книге [103] и в прилагаемой к ней библиотеке для ***Maple*** релизов **6 – 10**.

Глава 6. *Создание и работа с библиотеками пользователя*

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 разичного 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 [103], ориентированной на ***Maple*** релизов **6 – 10**. Можно ознакомиться с данными средствами и по демо-версии 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. Между тем, настоящая книга не содержит описания даже *базовых* средств пакета для доступа к файлам данных. Вместо этого *рекомендуется* обратиться либо к нашей книге [12] или бесплатно скачать исходные тексты наших книг по ***Maple***-тематике с адреса **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 достаточной пролонгированности представленных в них средств вполне приемлемы и для последующих релизов пакета.

6.1. *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. **1;** библиотека содержит три главных файла “***Maple.hdb***”, “***Maple.ind***” и “***Maple.lib***”, тогда как наличие некоторых других *файлов* зависит от текущего релиза пакета (*например,* ***Maple*** **6** *и* **7** *содержат файл* “***Maple.rep***”).

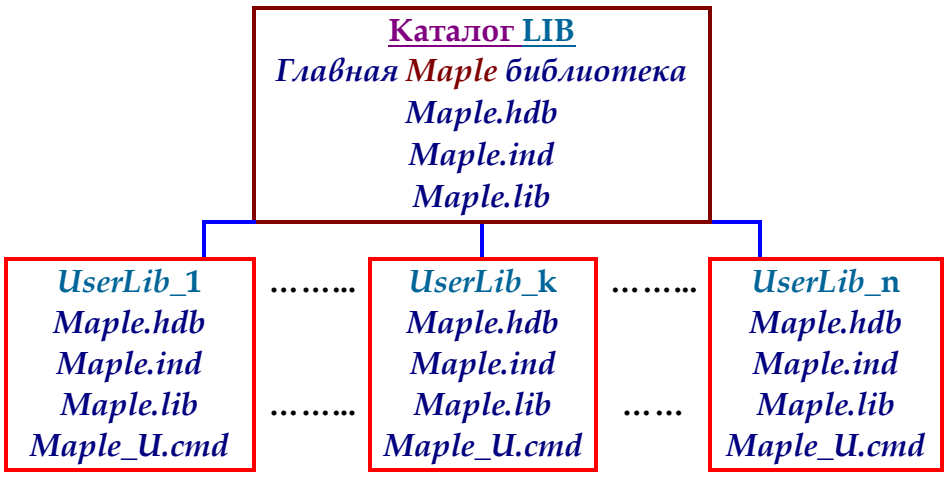


Рис. **1.** Принципиальная файловая организация главной ***Maple*** библиотеки и пользовательских библиотек, аналогичных главной библиотеке

В отличие от предыдущих релизов в ***Maple 10*** *главная* и другие библиотеки пакета организационно устроены несколько иначе, а именно: 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. **1**). 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 *дополнительно* появиться три дополнительных файла "***Maple.rep***", "***elpam.ind***" *и* "***elpam.lib***", чье *описание* может быть найдено в наших предыдущих книгах [29-33,39]. Ниже под *термином* `***Maple-****библиотека*` будет пониматься любая библиотека, структурно и организационно подобная главной ***Maple*** библиотеке.

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.

Для обеспечения работы с библиотеками пользователя, подобными ***Maple***-библиотеке, нами был создан целый ряд средств, описание которых можно найти в книге [103] и в *прилагаемой* к ней 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 и в ***Maple 10*** – **16** процедур. Однако целый

ряд *функций* по работе с библиотеками так и не был задействован. Наш же набор средств наряду с наиболее массовыми представляет средства для восстановления *поврежденных* библиотек и средства их оптимизации. Однако здесь мы не ставим целью представить упомянутые средства, а рассмотрим лишь стандартный подход к созданию библиотек пользователя.

Этап 1. Прежде всего, предполагается, что у пользователя имеется *набор* готовых и отлаженных процедур и/или программных модулей, которыми он желает наполнить *вновь* создаваемую библиотеку, подобную ***Maple***-библиотеке пакета. На *первом* этапе создается пустая библиотека, используя встроенную *iolib*-*функцию* (*ранее она была на уровне утилиты*) ***march*****,** 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 операционной среды **ПК,** 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 **ПК** 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 пользователем и ядром* ***Maple***), 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.

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 или средствами ***DOS***, ***Windows***, или в среде самого ***Maple***, для чего существует *iolib*-функция ***mkdir***, имеющая следующий 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 8"
> mkdir("UserLib1");  currentdir("UserLib1");  currentdir();
                              "C:\Program Files\Maple 8\UserLib1"
```

Первый пример фрагмента определяет текущий каталог пакета (*как правило*, *это основной каталог пакета*), тогда как второй пример иллюстрирует *создание* в нем *нового* **UserLib1** каталога 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 ***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 либо ее переопределением. Более того, установленный по ***currentdir***-функции каталог становится *текущим* только для *внешних* файлов пакета, т.е. данная функция используется с функциями *доступа* к внешним файлам данных пакета.

Между тем, по ***mkdir***-функции мы имеем возможность создавать *цепочки* каталогов лишь одного уровня *вложенности* либо только один последний 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]. 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 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 обратная наклонная черта (*backslash*), то она должна удваиваться, т. к. строки ***Maple*** используют этот символ в качестве управляющего (*escape*) символа.

Создав по встроенной функции ***mkdir*** или по нашей процедуре ***MkDir*** каталог под создаваемую библиотеку (*в нашем изложении это каталог* "C:/program files/maple 8/UserLib1"), теперь мы можем в нем создать «*пустую*» библиотеку с именем **UserLib1**. Делается это по вызову:

march('create', "C:\\Program Files\\Maple 8\\UserLib1", <*Размер*>**)**

где аргумент <*Размер*> определяет *размер* создаваемой библиотеки. Размер библиотеки определяется количеством содержащихся в ней ***m***-файлов с процедурами и модулями. При этом, практически, имеется возможность сохранения в такой библиотеке *числа* ***m***-файлов, примерно равного *удвоенному* числу, заданному вторым фактическим аргументом. О ***m***-файлах будет идти речь несколько ниже. Таким образом, по вызову формата:

march('create', "C:\\Program Files\\Maple 8\\UserLib1", 350)

мы создаем *пустую* библиотеку **UserLib1,** предназначенную для сохранения *порядка* **700 ПС**.

В библиотеке создаются два *макетных файла* ***"Maple.ind"*** (*индексный*) и ***"Maple.lib"*** (*с* ***m****-файлами программных средств*), впоследствии *обновляемые* при *каждом* помещении в библиотеку новых программных средств 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.

Этап 2. На данном этапе необходимо в текущем сеансе вычислить *все определения* ***Maple***-объектов, помещаемых в библиотеку, например:

```
> Sr:= () -> `+`(args)/nargs;  Ds:= () -> sqrt(sum((args[k] - Sr(args))^2, k = 1..nargs)/nargs);
```

$$Sr := (\,) \to \frac{`+`(\mathrm{args})}{\mathrm{nargs}}$$

$$Ds := (\,) \to \sqrt{\frac{\sum_{k=1}^{\mathrm{nargs}} (\mathrm{args}_k - \mathrm{Sr}(\mathrm{args}))^2}{\mathrm{nargs}}}$$

```
> Sr(64, 59, 39, 10, 17, 44),  Ds(64, 59, 39, 10, 17, 44);
```

$$\Rightarrow \frac{233}{6}, \frac{\sqrt{14249}}{6}$$

После этого *две* процедуры ***Sr*** и ***Ds*** готовы для включения в созданную библиотеку **UserLib1**.

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 8\\UserLib1":  savelibname;
```

savelibname

"C:\Program Files\Maple 8\UserLib1"

Этап 4. На этом этапе производится непосредственно *сохранение* в библиотеке подготовленных выше процедур. Делается это вызовом процедуры ***savelib*, имеющей формат вызова:

***savelib*(N1, N2, ...)**

где **Nj** – *имена* сохраняемых в библиотеке ***Maple***-объектов. Для нашего примера это 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 фактическим аргументом, ***Maple***-объектов, точнее соответствующих им ***m***-файлов, например:

```
> march('list', "C:\\Program Files\\Maple 8\\UserLib1");
```

[["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.

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 пакета. Для нашего случая текущее состояние ***libname*** есть:

```
> libname;
```

⇒ "c:/program files/maple 8/lib/userlib", "C:\Program Files\Maple 8/lib"

определяющее, что пакет располагает логически сцепленной с главной ***Maple***-библиотекой "C:\Program Files\Maple 8/lib" и *библиотекой* "c:/program files/maple 8/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 8\\UserLib1":  libname;
```

"c:/program files/maple 8/lib/userlib", "C:\Program Files\Maple 8/lib",
"C:\Program Files\Maple 8\UserLib1"

В результате переопределения наша библиотека **UserLib1** получила наимеьший 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:

```
> Sr(64, 59, 39, 10, 17, 44),  Ds(64, 59, 39, 10, 17, 44);
```

$$\frac{233}{6}, \frac{\sqrt{14249}}{6}$$

Из примера следует, что *созданная* библиотека (*в рамках двух 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 **UserLib1**-библиотеки выполнять вышеуказанное переопределение ***libname***-переменной, что, естественно, неудобно.

Во избежание этого рекомендуется поступить следующим образом. Пакет 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***) поместить файл *"**Maple.ini**"* с единственной строкой следующего содержания:

libname:= "c:/program files/maple 8/lib/userlib", "C:/Program Files/Maple 8/lib", "C:/Program Files/Maple 8/UserLib1":

либо *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 пользователя **UserLib1**, средства которой становятся доступными при *каждой* загрузке пакета наравне со стандартными средствами ***Maple***.

При этом, подкаталог **USERS** предполагается по *умолчанию*, но эта установка может быть пересмотрена при инсталляции пакета. 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* соответственно с {*высшим* | *первым* | *вторым*} уровнем приоритета, 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**, чей ***"Maple.ini"***файл используется пакетом *всех* релизов **6** – **10**, однако для **9**-го релиза этот 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 или их вызовов. В конкретном случае нашей библиотеки [103] 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**, процедурами и/или модулями, *имена* **P** которых могут быть представлены как в единственном числе, так и списком или множеством. Перед вызовом процедуры ***uplib*(L, P)** определения сохраняемых в **L**-библиотеке модулей и процедур **P** должны быть предварительно вычислены. В следующем фрагменте приведен исходный текст процедуры ***uplib***, реализованной *однострочным экстракодом*, и пример ее применения для *обновлений* **UserLib1**-библиотеки модулем:

```
> uplib:= (L::{string, symbol}, P::{procedure, `module`, list({procedure, `module`}),
set({procedure, `module`})}) -> op([assign('savelibname' = L), savelib(`if`(type(P, {'list', 'set'}),
op(P), P))]);
```

$$uplib := (L::\{string, symbol\}, P::\{procedure, `module`, list(\{procedure, `module`\}), set(\{procedure, `module`\})\}) \rightarrow op([assign('savelibname' = L), savelib(`if`(type(P, \{'list', 'set'\}), op(P), P))])$$

```
> M:= module() export x; x:= () -> `*`(args)/nargs end module:
> uplib("C:\\Program Files\\Maple 8\\UserLib1", M);
> restart;  M:- x(64, 59, 39, 10, 17, 44), with(M);  x(64, 59, 39, 10, 17, 44),  eval(x);
```

$$183589120, [x]$$

$$183589120, (\ ) \rightarrow \frac{`*`(args)}{nargs}$$

```
> march('list', "C:\\Program Files\\Maple 8\\UserLib1");
     [["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]]
```

В данном фрагменте вычисляется определение процедуры ***uplib*** и программного модуля **M**, предназначенного для *сохранения* в существующей (*созданной на предыдущих этапах*) библиотеке **"C:\\Program Files\\Maple 8\\UserLib1"**. По вызову **uplib("C:\\Program Files\\Maple 8\\UserLib1", 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* **UserLib1**-библиотеки *модулем* **M**. При этом предполагалось, что на этапе **5** созданная **UserLib1**-библиотека была *логически* сцеплена с главной ***Maple***-библиотекой (*через **libname**-переменную*) посредством *файла* ***"Maple.ini"*** в **USERS**-каталоге пакета. Последний пример фрагмента представляет вывод нового состояния **UserLib1**-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].

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 8/Sr2.m");
> march('add', "C:\\Program Files\\Maple 8\\UserLib1", "Sr2.m", Sr2);
> march('list', "C:\\Program Files\\Maple 8\\UserLib1");
> restart;  3*Sr2(64, 59, 39, 43, 10,17);   ⇒ 116
```

Последующий вызов процедуры ***Sr2*** после **restart**-предложения подтверждает корректность выполненной операции *обновления* **UserLib1**-библиотеки процедурой ***Sr2***. При этом предполагалось, что на этапе **5** созданная **UserLib1**-библиотека была *логически* сцеплена с главной ***Maple***-библиотекой пакета.

По вызову функции **march('delete',** <*Путь к библиотеке*>**,** <*Имя объекта*>**)** делается *удаление* из библиотеки, определенной *вторым* аргументом, *объекта*, указанного *третьим* аргументом, например:

```
> march('delete', "C:\\Program Files\\Maple 8\\UserLib1", Ds);
> march('list', "C:\\Program Files\\Maple 8\\UserLib1");
  [["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"*** библио-

теки. Для *освобождения* этого места (*уплотнения библиотеки*) можно использовать вызов функции **march('pack',** <*Путь к библиотеке*>**).** С другими 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.

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 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 8\\UserLib1"**). Однако здесь ситуация несколько отлична от традиционной, а именно. Справочные *страницы* по средствам ***Maple***-библиотеки находится в справочной базе в виде файла ***"Maple.hdb"***, расположенного, как правило, в том же каталоге, что и сама библиотека. Каждый такой файл базы данных содержит одну или 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 формата **INTERFACE_HELP(...)**. Например, запрос **INTERFACE_HELP('display', topic=helpman)** выводит *справочную* страницу по процедуре ***helpman***. В общем случае запрос **INTERFACE_HELP** имеет следующий формат:

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** или 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 кодировать в виде **library=libname[k],** где **к** – *порядковый* номер библиотеки в цепочке

библиотек, отраженных в ***libname***-переменной. Детальнее с вопросами работы с функцией **INTERFACE_HELP** можно ознакомиться по вызову **?help,update**.

С учетом сказанного, для создания *начальной справочной* базы для нашей **UserLib1**-библиотеки используем следующий **INTERFACE_HELP**-вызов:

```
> INTERFACE_HELP('insert', topic= "UserLib", text= TEXT("Help on my means in UserLib.
Created 4.10.2006"), library= "C:\\Program Files\\Maple 8\\UserLib1");
```

В результате этого вызова в каталоге **"C:\\Program Files\\Maple 8\\UserLib1"** 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.

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 (*рекомендую взять за основу оформление страниц* (*разделов*) *справочной базы самого пакета; именно таким образом оформлена справочная база нашей библиотеки* [103]) 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 отображается 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, *справочная* система пакета организована достаточно эффективно и удобна для практического использования при работе с ***Maple***.

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:

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/savem1.mws`,
`C:\\Program Files\\Maple 8\\UserLib1`);
```

по которому в справочную базу нашей **UserLib1-**библиотеки помещается *справочная* страница под *именем* ***savem1*****,** создаваемая на основе одноименного ***mws***-файла, расположенного по адресу, указанному вторым фактическим аргументом. При этом, в случае *завершения* 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 формальные 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 оответствия; 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 формируют 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 *нескольких* 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.

```
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 argume\
nt 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 t \
            ype <%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 8/LIB/userlib", ["DoF1"], ["C:\\temp/dof1.mws"]);
  # вставка справочного раздела по процедуре DoF1 в справочную базу библиотеки UserLib
Warning, Work has been done!
> helpman(insert, "C:/Program Files/maple 8/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 8/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 8/LIB/userlib", "AFdes");  # вывод справочного раздела по процедуре AFdes
> helpman(delete, libname[1], Art_Kr);  # удаление справочного раздела Art_Kr из справочной базы библиотеки, путь к которой находится в начале цепочки библиотек libname-переменной
```

Выше 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.

Таким образом, нами представлены достаточно *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 библиотек указанного типа.

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 (*раздела*)

8 – *удаление* из справочной базы библиотеки **L** страницы (*раздела*), указанной *третьим* фактическим 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 при наименьшем приоритете первой

10 – *создание логического сцепления* библиотеки **L** с *главной* ***Maple***-библиотекой; данное сцепление действует до переопределения ***libname***-*переменной*, до **restart**-предложения или до перезагрузки пакета. Вызов процедуры возвращает текущее состояние ***libname***-переменной.

Ниже представлен исходный текст процедуры и примеры ее применения на все случаи:

```
> ulibrary:= proc(O::{1,2,3,4,5,6,7,8,9,10}, L::{symbol, string}) 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, "\"]"))
while true 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 5.10.2006"), 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-type 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;

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 he\
            lp on my means located in UserLib. The library has been created \
            5.10.2006"), 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 symbo \
            l-type 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 M\
            aple-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"
```

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 библиотекой выводит соответствующие предупреждения. Процедура ***ulibrary*** обрабатывает *основные* особые и ошибочные ситуации. 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.

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 пакетом ***Maple.***

6.2. *Специальные способы создания библиотек в среде Maple*

Прежде всего, *созданные* и *отлаженные* процедуры и программные модули можно сохранять в текстовых файлах во *входном* формате ***Maple***-языка. В этом случае они впоследствии читаются **read**-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:

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 или 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]. 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 определение процедур и модулей, находящихся в файле СФ, вычисляется и они становятся доступными *текущему* сеансу. 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 **read**-предложению 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 т.д.*). 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.

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 и сразу же становятся доступными *подобно* стандартным средствам пакета по **read**-предложению. 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 полезную 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 программными средствам (*процедурами и/или программными мо-*

дулями). 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. Сохранение *объектов* производится в *режиме дописывания* (*APPEND*), позволяя *сохранять* (*архивировать*) *все* версии программных 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.

```
> simplel:= proc(L::{string, symbol}) local a, b, c, f, mkf; mkf:=proc (f::{string, symbol}) local a, b, c, d, cc;  cc:= proc(d::{string, symbol}) local a, b, c, k; assign(a=[], b="" || d, c=17),`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("argument 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;
```

```
simplel := proc (L::{string , symbol})
local a, b, c, f, mkf;
    mkf := proc (f::{string , symbol})
        local a, b, c, d, cc;
            cc := proc (d::{string , symbol})

                local a, b, c, k;
                    assign(a = [ ], b = "" || d, c = 17), `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("argu\
            ment 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 def\
    ine 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 inp\

        ut 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 simpel  ", 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/Maple_u.cmd");
Error, (in simplel) datafile <C:/Temp/Maple_u.cmd> had been not created by procedure simplel
```

Представленный выше фрагмент содержит исходный текст и примеры применения процедуры ***simplel***. Представленная процедура может оказаться полезным средством при организации простых библиотек пользователя, несущих черты и *архива* 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] 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 или 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

<*Имя таблицы*>**[**<*Имя процедуры*>**](...)**

то к *программному модулю* по конструкции формата

<*Имя модуля*> **:-** <*Имя экспорта*>**](...)**

только после вызова процедуры **with(**<*Имя таблицы*>**)**. Следующий *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
```

$$T := \mathrm{table}\left([P = \left(( ) \to \frac{\texttt{`+`}(\mathrm{args})}{\mathrm{nargs}}\right),\right.$$

```
    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"  )
    ])
> 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 *вполне* 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 в практическом программировании приложений в среде пакета.

В свете представленного выше подхода к организации пользовательских библиотек на *основе* табличной организации может оказаться достаточно полезной процедура ***SoftTab***, *исходный* текст которой и примеры применения представлены нижеследующим фрагментом:

```
> SoftTab:= proc(T::symbol, F::{string, symbol}, 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 nargs > 2 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;
```

```
SoftTab := proc (T::symbol, F::{string, symbol}, R::symbol)
local a, b, s, k;
    b := ( ) → ERROR("procedure call does not contain procedures and/or mo \
        dules 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 - []
```

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 в ней 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 или 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 "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 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 довольно 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 [103]:

* *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 либо обновлении библиотеки пользователь должен определить только ее имя (*если библиотека будет расположена в каталоге* **LIB** *пакета*) или *полный* путь к *ней*, наряду с множеством или списком *имен* ***Maple***-объектов, сохраняемых в библиотеке, и чьи *определения* были *вычислены* в текущем ***Maple***-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 [103] 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 отношении они в некоторых случаях 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]. 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.

```
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`, 'table', 'procedure'}), P[k],
        assign('b' = {k, op(b)}))}), 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
```

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, либо с отсутствием сохраняемых *объектов*. При возникновении подобных ситуаций возвращается соответствующая диагностика.

Процедура обеспечивает вышеупомянутые функции для *библиотек* 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***).

```
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`, 'table', 'procedure'}), assign('a' = [op(a), N[k]]),
        WARNING("<%1> is not a procedure and not a module, and not a table" ,

        N[k])), k = 1 .. n));
    `if`(nops(a) = 0,
        ERROR("procedures, modules or tables do not exist for saving" ),
        assign(b = NLP(L)[1]));
    for k to nops(a) do

        if member(a[k], b) or Search1(cat(a[k], `:-`), a[k], 'd') and d = ['left']
        then WARNING("<%1> does exist and will be updated" , a[k])
        else WARNING("<%1> does not exist and will be added" , a[k])
        end if
    end do ;

    assign(c = savelibname ), assign('savelibname' = h), savelib(op(a));
    unassign('savelibname'), assign(savelibname = c),
        WARNING("Library update has been done!" )
end proc
```

Успешный *вызов* процедуры ***UpLib*(F, N)** обновляет ***Maple***-библиотеку, указанную *первым* аргументом **F**, процедурами, таблицами и/или пакетными модулями, чьи имена определены *вторым* фактическим **N** аргументом (*список имен*). Предполагается, что **F** *библиотека* аналогична главной библиотеке пакета. Успешный вызов процедуры возвращает ***NULL***-значение с выводом соответствующих сообщений. Процедура обрабатывает *основные* ошибочные ситуации, связанные с отсутствием библиотеки или с отсутствием *сохраняемых* объектов. При возникновении таких ситуаций инициируются соответствующие *ошибки*. Процедура ***UpLib*** зарекомендовала *себя* в качестве удобного и эффективного средства обновления ***Maple***-библиотек. Нами она регулярно используется для оперативного обновления ***Maple***-библиотек.

6.3. *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 ***Maple***, можно оперативно по *запросу* ***?index,package***. Во многих изданиях и технической документации по ***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 здесь не обсуждаемых. В частности, она позволяет вести *независимую* разработку с ориентацией на конкретный круг приложений, допускает использование *одноименных* со стандартными средств и т.д. Организационно пакетные модули в настоящее время бывают трех типов – *процедуры*, *программные модули* (*как правило второго типа*) и *таблицы*. С помощью нашей процедуры [103] можно проверять тип пакетного модуля; ее применение в среде ***Maple 10*** дает следующий результат:

```
> 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]]
```

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*. В главах **4** и **5** 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 лишь рассмотрим *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)), SV:  eval(TabMod);

table([Art = ( ( ) → `*`(args)/nargs^2 ), SV = SV,
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[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 8/LIB/UserLib":  savelib(TabMod);  restart;
> libname, with(TabMod);
    "c:/program files/maple 8/lib/userlib", "C:\Program Files\Maple 8/lib", [Art, Ds, SV, Sr]
> seq(F(42, 47, 67, 89, 96), F = [Art, SV, Ds, Sr]);  restart;
                  1130012352/25, SV(42, 47, 67, 89, 96), 21.6462, 68.2000

> seq(F(42, 47, 67, 89, 96), F = [Art, SV, Ds, Sr]);

                  1130012352/25, SV(42, 47, 67, 89, 96), 0.200000 (5. (Sr(42, 47, 67, 89, 96) - 42.)^2
                    + 5. (Sr(42, 47, 67, 89, 96) - 47.)^2 + 5. (Sr(42, 47, 67, 89, 96) - 67.)^2
                    + 5. (Sr(42, 47, 67, 89, 96) - 89.)^2 + 5. (Sr(42, 47, 67, 89, 96) - 96.)^2)^(1/2), 68.2000
```

Первые три примера фрагмента иллюстрируют поэтапное определение таблицы ***TabMod***, 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 способом функции в таблицу следует использовать конструкцию, представленную *модульной* ***SV***-функцией фрагмента, т.е. имеющей в общем случае следующий формат кодирования:

Id_table[*Id_function*] := define(*Id_f1*, *Id_f1*(...) = F(...)), *Id_f1* {;|:}

При этом, *идентификатор Id_f1* может совпадать с идентификатором *Id_function*. Вызов функции **eval(TabMod)** *возвращает* содержимое таблицы **TabMod**, а 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 ее *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***. Ес-

ли же *пакетный модуль* **Id** находится в библиотечном файле, путь к которому определен в переменных пакета ***libname*** и ***savelibname*****,** то по вызову **with(Id)** производится *загрузка* его в *рабочую область пакета* (**РОП**), обеспечивая доступ ко всем поддерживаемым им функциональным средствам. Тогда как по конструкциям следующего 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**. Однако, *неопределенной* остается процедура, предварительно заданная по ***define***. Следовательно, если определенная в текущем *сеансе* таблица ***TabMod*** допускает такое определение своего *выхода* (*активируя процедуру в текущем сеансе*), то в условиях *пакетного* модуля это *недопустимо*. *И второе*, без вызова ***with*****(*****TabMod*****)** процедуры пакетного модуля, имеющие вхождения явно не определенных в текущем сеансе выражений, остаются *неопределенными*.

При создании *пакетных* модулей следует иметь в виду, что *модульная* таблица, обеспечивающая доступ к функциональным средствам модуля, имеет *одно* из *двух базовых* 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.

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 фактическим аргументом **F.**

Механизм **with**-процедуры, в частности, состоит в том, что ее вызов непосредственно влечет за собой вычисление (*всех или заданных*) выходов соответствующей модульной таблицы, в ка-

честве 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* практическом программировании в среде 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;   ⇒ 54
```

Наряду с отмеченными имеется еще *целый* ряд 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 *фактические* 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))]]);
```

$$Kr := \begin{bmatrix} f1 = \left(( ) \to \frac{\text{`+`}(\text{args})}{\text{nargs}} \right) & f2 = \left(( ) \to \frac{\text{add}(\text{args}_k^{\,2}, k = 1 \,..\, \text{nargs})}{\text{nargs}} \right) \\ f3 = (( ) \to [\text{`+`}(\text{args}), \text{nargs}]) & f4 = (( ) \to \text{`*`}(\text{args})) \end{bmatrix}$$

```
> savelibname:= "C:/Program Files/Maple 8/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 ***ArtMod*** процедуры 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 пакета. В целом 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 пакетному модулю и содержащимся в нем объектам в целом, предполагая, что *модуль* находится в ***Maple***-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 данного 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 функция/процедура **N** *пакетного модуля заменила* собой *библиотечное* средство. Во избежание этого рекомендуется использовать *разовые* вызовы пакетных средств по конструкциям 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, adj,adjoint
================================================================================
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 assigned name GramSchmidt now has a global binding
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, diffalg,
================================================================================
LREtools, 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 ***Maple***, 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 или 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* или *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.

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 этих целей *рекомендуется* использовать процедуру ***tpacmod***, имеющую формат вызова следующего вида: ***tpacmod*(P** {**, *Name***}**)**, где **P** – имя пакетного модуля и ***Name*** – имя его *экспорта* [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();  ⇒ []
```

Из примеров фрагмента следует, что вызовы процедуры ***tpacmod*** как для выяснения списка *экспортов* пакетного модуля, так и для тестирования имени быть *экспортом* пакетного модуля не загружает ни экспортов пакетного модуля в **РОП**, ни модуля в целом.

6.4. *Статистический анализ* Maple-библиотек

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 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. Прежде всего, данная процедура предполагает, что анализируемая **L** библиотека расположена 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 ***Maple***-библиотеки, расположенной в подкаталоге **LIB** главного каталога пакета, которая логически связана с главной библиотекой пакета. Другие ее формальные аргументы (*в количестве до* **3**) являются дополнительными и назначение кортежей их значений определяется как:

StatLib(L) – *инициация* сбора статистики по средствам библиотеки **L** в целом;
StatLib(L, 0) – *удаление* всех файлов со статистической информацией по библиотеке;
StatLib(L, 1) – *отмена* сбора статистики по библиотеке **L** с сохранением результатов;
StatLib(L, P) – *инициация* сбора статистики по процедуре **P** из библиотеки **L**;
StatLib(L, T, 2) – *возврат* собранной статистики по библиотеке **L** или по ее **P** процедуре в заданном разрезе **T**;
StatLib(L, T, 2, m) – *возврат* собранной статистики по библиотеке **L** или по ее **P** процедуре 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 ***StatLib*** *при* возникновении *критических ошибок*.

Вызов процедуры ***StatLib*(L)** с одним фактическим **L** аргументом инициирует процесс сбора статистики по вызовам составляющих **L** библиотеку 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 **L** библиотекой. 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 **L** библиотеки. Данный вызов рекомендуется выполнять перед инициацией нового процесса *профилирования* процедур с целью сохранения его результатов в каталоге с **L** библиотекой. 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 **L** библиотеку и определяемых их **P**-списком имен. При этом, запрашиваемые к профилированию процедуры не обязательно должны принадлежать **L** библиотеке – они могут находиться в любой библиотеке, *логически* связанной с основной 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.

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 **L** библиотеки {*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 достаточно 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 дает 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* определен-

ностью 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, либо ***Maple*** имеет стэк недостаточной глубины, либо 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 книге [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*) данное 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* **700**), 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* **L** *библиотеки 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 **L** библиотекой. 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 или существенно использующих базовые 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]. Представленные ниже примеры достаточно хорошо 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 statist\
        ics have been removed out of directory with library <%1>", L),
        unprofile(),
        op(map(Fremove, [bytes1, calls1, depth1, maxdepth1, time1])), unp())
        , NULL);
    if nargs = 2 and args[2] = 1 then
        `if`(type(eval(profile_calls), 'table'), assign(
            'calls2' = eval(cat(profile_, R[2])),
            'bytes2' = eval(cat(profile_, R[1])),
            'depth2' = eval(cat(profile_, R[3])),
            'maxdepth2' = eval(cat(profile_, R[4])),
            'time2' = eval(cat(profile_, R[5]))),
            ERROR("profiling does not exist"));
        (proc()
                save maxdepth2, maxdepth1;
                save calls2, calls1;
                save bytes2, bytes1;
                save depth2, depth1;
                save time2, time1
            end proc)(), RETURN(unprofile(), unp(),
            WARNING("profiling of library <%1> has been completed", L))
    else NULL
    end if;
```

```
if nargs = 3 and belong(cat(``, args[2]), R) and
member(whattype(args[3]), {'symbol', 'string'}) then
    assign(''`$Art16_Kr9`'' = eval(cat(profile_, args[2])),
        pf = cat(K, "/$@_", args[2])), `if`(
        type(eval(`$Art16_Kr9`), 'table'), NULL, `if`(DoF(pf) = 'file', [
        (proc() read pf end proc)(),
        assign(''`$Art16_Kr9`'' = eval(cat(``, args[2], `2`)))],
        RETURN("a profiling information does not exist")));
    assign(a = [cat(``, args[2]), `$Art16_Kr9`[cat("", args[3])]]), RETURN(
        `if`(type(a[2], 'numeric'), a, "procedure has been not profiled"),
        `if`(type(eval(profile_proc), 'symbol'), unp(), NULL))
else NULL
end if;
if 3 ≤ nargs and belong(cat(``, args[2]), R) and args[3] = 2 then
    `if`(nargs = 4 and type(args[4], 'numeric'), assign(a = args[4]),
        assign(a = 0));
    assign(pf = cat(K, "/$@_", args[2]),
        ''`$Art16_Kr9`'' = eval(cat(profile_, args[2]))), `if`(
        type(eval(`$Art16_Kr9`), 'table'), NULL, `if`(DoF(pf) = 'file', [
        (proc() read pf end proc)(),
        assign(''`$Art16_Kr9`'' = eval(cat(``, args[2], `2`)))],
        RETURN("a profiling information does not exist")));
    RETURN(
        tabar(`$Art16_Kr9`, 'Procedures', cat(`Procedure's `, args[2]), a),
        `if`(type(eval(profile_proc), 'symbol'), unp(), NULL))
else NULL
end if;
`if`(K ≠ false, [assign(P = march('list', K)), assign('h' = nops(P))], ERROR(
    "library <%1> does not exist or is not linked with the main Maple library",
    L));
`if`(nargs = 2 and type(args[2], 'list'({'symbol'})), assign(S = args[2]), assign(
    S = [
    seq(`if`(P[k][1][1 .. 2] ≠ ":-", cat(``, P[k][1][1 .. -3]), NULL), k = 1 .. h)
    ]));
for k in S do
    try `if`(type(eval(k), 'procedure'), assign('G' = [op(G), k]), NULL)
    catch "": NULL("Exception handling with program modules")
    end try
end do;
`if`(G = [], ERROR("procedures ordered for profiling do not exist both in \
    library <%1> and in libraries logically linked with the main Maple libra \
    ry", 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, данная процедура весьма эффективно использовалась для *улучшения функциональных* характеристик пользовательской библиотеки **UserLib**, содержащей процедуры, представленные в нашей книге [103].

Процедуры, представленные в [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. Эти и другие предпосылки, обусловленные вышеупомянутыми процедурами, позволяют существенно автоматизировать обработку пользовательских библиотек, наряду с расширением возможностей, имеющих дело с сохранением *процедур* и *программных* модулей во внешней памяти компьютера. В целом, средства, методы и приемы, представленные в книге [103], предназначены как для повышения эффективности применения ***Maple*** в различных приложениях, требующих программирования, так и для самого освоения программирования в среде пакета ***Maple***.

Заключение

Настоящая книга вводит специалистов, ученых, преподавателей и студентов в *различные* аспекты программирования в среде известного математического пакета ***Maple*** релизов **6 – 10**. 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 программирования в среде ***Maple***-языка пакета, позволяющим *эффективно* использовать *нашу библиотеку*, прилагаемую к книге [103], прежде всего. Наряду с этим, книга может послужить неплохим введением в программную среду пакета ***Maple*** релизов **6 – 10**.

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* **700** *процедур и программных 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 целом 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 дополнительные процедуры и модули иллюстрируют как полезные *приемы* программирования, так и элементы *методологии* и *методов* программирования в среде пакета.

Библиотека предназначена для ***Maple*** *релизов* **6-10,** 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* направлениях, 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 – 9,** 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 исполь-

зованию 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, ListTools** и **StringTools**. 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.*), я, между тем, значительное внимание уделял и такому прикладному *направлению*, как компьютерная техника с акцентом на ее программном обеспечении (*операционные системы, оболочки, 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 серьезную творческую активность в данной области, наиболее импонирующую нашей натуре и представлениям.

Именно данный подход к каждому **ПС**, с которым я имел 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 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 этот пакет использовался мной и моими 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 **1200**. 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 читате-

лей наших книг и проведение серии курсов – вот три *основных* источника, стимулировавших появление библиотеки, упоминаемой в настоящей книге и приложенной к книге [103].

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 весьма неплохие результаты, а именно.

Библиотека в совокупности с главной ***Maple***-библиотекой обладает полнотой в том отношении, что любое ее средство использует либо 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 [103] *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 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 библиотеки полезные для своего творчества.

В настоящей книге, невзирая на ее начальный 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, поэтому для вполне приличного освоения (*mastering*) требуется достаточно серьезная творческая наработка с пакетом и не на уровне высоко интеллектуального калькулятора, а реальное программирование приложений в его среде, обеспечивающее вас как средой программирования, так и стимулирующее более активно знакомиться с ее как возможностями, так и недостатками.

Перечень программных средств, находящихся в Maple-библиотеке [103]

&ma, &Shift, _0N, _1N, _mMfile, _nnn, _ON, _rd, _SL, A_Color, ACC:-acc, ACC:-Ds, ACC:-Sr, Aconv, ACP, Adrive, AFdes, AlgLists:-&, AlgLists:-&+, AlgLists:-&-, AlgLists:-&/, AlgLists:-&^, Algsubs, All_Close, AnalT, andN, Animate2D, Animate3D, Aobj, Apmv, Aproc, Arobj, ArtMod, asceleton, Assign, assign6, assign67, assign7, Atr, AtrRW, avm_VM, belong, Bit, Bit1, blank, boolop:-&andB, boolop:-&impB, boolop:-&notB, boolop:-&orB, boolop:-&xorB, BootDrive, braces, Builtin, came, Case, Catch_Vir, CCF, CCM, CDiag, CDM, cdt, CF, CF1, CF2, cfdd, CFF, CFF1, cfln, Chess, ChkPnt, clib, cliblink, Close, Clr, Cls, clsd, cmf, cmlib, cmtf, cnvtSL, CoefTaylor, coldig, com6_9, com_exe1, com_exe2, CompStr, Con_Mws, conlltab, conmlib, conSA, conSV, ContLib, convert/Array, convert/list1, convert/listlist1, convert/lowercase, convert/module, convert/proc, convert/rlb, convert/set1, convert/ssll, convert/string1, convert/symbol1, convert/TEXT, convert/uppercase, conwf, Cookies, CorMod, CorMod1, CrNumMatrix, CS, csearch, ctab, CureLib, Currentdir, d2plot, D_ren, DAAF, DAclose, DagTag, DAopen, DAread, dcemod, ddll, DDT, DeCod, decode, DeCoder, DefOpt, DEL_F, delel, delf, delf1, delres, delsc, deltab, DestLm, detab, Dialog, difff, diffP, Dir, Dir_ren, DIRAX:-conv, DIRAX:-delete, DIRAX:-empty, DIRAX:-extract, DIRAX:-insert, DIRAX:-new, DIRAX:-printd, DIRAX:-replace, DIRAX:-reverse, DIRAX:-size, DIRAX:-sortd, DirE, DirF, DirFT, Dist_rand_n_2, DoF, DoF1, dslib, dsps, dt, DT, DUlib, E_mail, ecsolve, email, Empty, EntS, Etest, etf, Evalf, ewsc, exeP, expLS, ExprOfString, Extract, extrcalls, ExtrF, extrname, extrS, extrmws, F_alias, F_Analys, F_atr, F_atr1, F_atr2, F_ren, F_T, F_test_Ds, F_Type, Fac, Fappend, FBBcopy, FBcopy, FBfile, FD, Fend, Fequal, FFP, ffp, fileinfo, filePM, Find, FindFSK, fiolib, FLib, FLib1, FLL, FmF, fminimax, FNS, Fnull, FO_state, Fopen, fopen1, FOR_DO, fpalind, fpalind1, fpathdf, frame_n, Fremove, frss, FSSF, FT_part, FT_part1, FT_restr, FT_subs, FTabLine, FTabLine1, FTcopy, FTmerge, ftpd, Ftype, Fword, gelist, Gener, GenFT, getable, GG, gpp, gpp1, Heap, helpman, hidemws, histo, Histo, holdof, howAct, HS_1, HS_1_GF, HS_1D, IAN_REA, Iddn, Iddn1, IdR, IDS, IF, If, Images, Imaple, Indets, indetval, inel, Ins, Insert, insitudelL, insituls, insL, InstUlib, INT, intaddr, Interval, IntHelp, intproc, intt, InvL, InvList, InvT, IO_proc, IOproc, IOsave, IP, Is_Color, isDir, ISDS, IsFempty, isFile, isflo, IsFtype, isLnkMws, ismLib, isMSDcom, IsOpen, IsOpen1, IsOpen2, IsOpenF, isplabel, IsPtf, isRead, KCM, Kernels, KL, KL1, Kr_Mesh, Kvantil, LatexI, LGD, LibElem, LibElem1, LibElem2, LibLink, LibLink1, LibUser, Linear_Const, ListPack, LnFile, LO, LocalnL, logbytes, Lprot, Lrare, LRM_NRM, LSF, lsf, LSL, LT, LTfile, M_Turing, M_Type, MA, MAM, MAM1, map3, map4, map5, map6, mapleacs, MapleLib, mapLS, mapN, mapTab, marcmf, MatrSort, maxl, Mem, MEM, Mem1, memberL, membertol, mergenf, MiniMax, minl, MinMax, minmax3d, MkDir, MkDir1, mkdir1, MkDir2, MkDir3, mkfile, mlist, mlsnest, mmf, MmF, mmp, mod21, mod3, mod_proc, ModFile, ModProc, modproc, mpl_proc, MPL_txt, mPM, MSDcom, MSK, mtf, Mulel, mws789_6, Mwsin, mwsname, MwsRtb, N_Cont, NDLN, NDP, NDT, nexts, Nint, nlcvector, nLine, NLP, nmmlft, NonaLP, Nstring, null, nulldel, NumOfString, numres, nvalue, occur, OP, Open1, open2, OpenLN, orN, p3listlist, parmod, ParProc, ParProc1, parvar, Path, pathtf, Pclause, Pclause1, perml, permlib, pf3minmax, pfminmax, Pind, Plib, plotdefopts, plotpw, plotTab, plotu, plotvl, Pnd, Pnd1, PNorm, Polyhedra, POOS, porf, Porshen, Pos, PP, prestart, primetest, PSubs, Pul_Bal, Pul_Bal_Cor, Pulsar, Q2plot, QFline, Qsubstr, Queue, quotient, rand_Histo, rcs, Read, Read1, read3, Readarray,*

readdata1, readm1, readmp, ReadProc, reconf, Red_n, redL, redL1, RedList, redlt, redss, Reduce_T, RegMW, Release, Release1, relml, Remember_T, renmf, reprolib, Residue, Resl, ResRtb, Rev, Rfact, rID, Rlss, RmDir, rmdir1, Rmf, rName, Root, rp2f, RS, RSLU, Rssl, Rt_s, RTab, Rtable, RTfile, rtfnin, rtftab, RTsave, S_D, SA_text, sarray, Save, save1, Save1, Save2, save2, save3, savead, saveall, savem, savem1, savema, savemp, SaveMP, savemu, SaveProc, sblist, SD, SD_S, SD_S1, SDF, Search, Search1, Search2, Search3, Search_D, Search_D1, Search_D2, searchL, SEQ, seq1, seq2, seqstr, seqstr1, seqstr2, seqstr3, sext, sextr, sextr1, sfd, sHisto, sident, simplel, SimpleStat:-ACC, SimpleStat:-CC, SimpleStat:-CR, SimpleStat:-Ds, SimpleStat:-FD, SimpleStat:-LRM_NRM, SimpleStat:-LT, SimpleStat:-MAM, SimpleStat:-MCC, SimpleStat:-PCC, SimpleStat:-Sko, SimpleStat:-SR, SimpleStat:-Weights, SL, SLD, SLj, SLK, sllj, SLS, sls, Slss, Smart, Smart_Plot, sMf, SN, SN_6, SN_7, SN_8, SN_9, sof, SoftTab, SoLists:-intersect, SoLists:-minus, SoLists:-sublist, SoLists:-union, sortkL, SortL, sorts, sorttf, spfn, Sproc, SQHD, SS, SSet, SSF, ssf, SSN, ssortL, sspos, sstr, STACK, statf, StatLib, stpm, Sts, stype, sub_1, Sub_all, Sub_list, SUB_S, Sub_st, Subs_All, Subs_all1, Subs_all2, subseqn, Subset, subSL, subsLS, Suffix, swmpat, swmpat1, Sys_Env, System, T_Font, T_SQHD, T_SQHT, T_test_AV, tabar, TabList, Tcounter, Test, tpacmod, tpr, transmf7_6, trConv, TRm, truncn, ttable, type/arity, type/assignable1, type/binary, type/boolproc, type/byte, type/color, type/complex1, type/digit, type/dir, type/dirax, type/file, type/file1, type/fpath, type/heap, type/letter, type/libobj, type/lower, type/Lower, type/mla, type/mlab, type/mlib, type/mod1, type/nestlist, type/nonsingular, type/package, type/path, type/plot3dopt, type/plotopt, type/realnum, type/rlb, type/sequent, type/setset, type/ssign, type/Table, type/upper, type/Upper, Type_D, typeseq, U_test_MW, UbF, uglobal, Ulib, ulibpack, ulibrary, undef, Uninstall, UpLib, uprtable, Use, User_pfl, User_pflM, User_pflMH, UserC, Users, usertype, uvlama, V_Solve, V_Str, varsort, verdel, VisM, Vol, Vol_Free_Space, VTest, WARNING, WD, WDS, Weight_LF, Weights, Weights_L, winver, With, writedata1, WS, WT, X_test_VW, xbyte, xbyte1, xNB, xorN, xpack, XSN, XTfile

Вышеприведенные процедуры и пакетные модули (*числом более* ***700***) организованы в пользовательскую ***Maple***-библиотеку, снабженную справочной базой, подобной базе пакета. После логической связи библиотеки с главной ***Maple***-библиотекой средства, содержащиеся в ней, могут использоваться аналогично стандартным средствам пакета. Архив с версиями библиотеки для ***Maple*** релизов **6 – 10** на ***Windows***-платформе может быть загружен с нашего вебсайта, указанного в книге [103]. В процессе использования ***Maple*** как в различных физико-математических приложениях, так и в проведении курсов по современным *системам компьютерной алгебры* мы производим регулярное обновление библиотеки, которое также можно получать с вышеуказанного вебсайта.

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 пере-

давать 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] 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 программирование, так и делая его более прозрачным. В целом ряде случаев предложенные выше средства упрощают работу с пакетом после различных аварийных завершений.

Библиотека **UserLib** прилагается к нашим книгам [41,103], тогда как ее демо-версию можно бесплатно загрузить с одного из следующих вебсайтов:

http://www.aladjev-maple.narod.ru
http://writers.fultus.com/aladjev/book01.html

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 пакета 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 общепринятых требований к качественному программному обеспечению весьма *негативно* сказывается на достаточно хорошем в целом пакете современной компьютерной алгебры.

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 пакета, обнаруженных при выполнении *нашей* библиотеки в среде ***Maple 10*** на платформе ***Windows***. 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 пакета (*например, невычисление или некорректное вычисление конкретного интеграла и т.п.*), 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 пакета мы не считаем нужным заниматься. Такого типа вопросы следует адресовать *MapleSoft team*, хотя, *на наш взгляд*, это и не очень продуктивно.
(**2**) по вопросам применения пакета для решения конкретных задач (*тематика может просто 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.

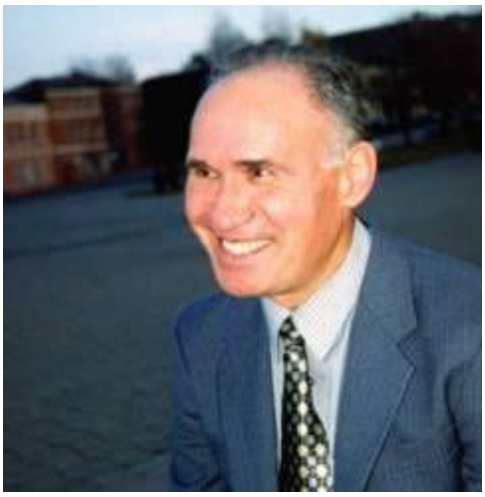
Литература

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.З., Ваганов В.А., Хунт Ю.Я., Шишаков М.Л.*** Программирование в среде математического пакета ***Maple V***.- Гомель: TRG & Salcombe, 1999, 470 с., ISBN 4-10-121298-2
10. ***Аладьев В.З., Ваганов В.А., Хунт Ю.Я., Шишаков М.Л.*** 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. + CD, ISBN 5-93308-085-X
14. ***Aladjev V.Z., Bogdevicius M.A. Interactive Maple:*** *Solution of Mathematical, Engineering, Statistical and Physical Problems*.- Tallinn-Vilnius, International 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, Lithuania.
16. ***Aladjev V.Z., Hunt U.J.*** Workstation for mathematicians // Intern. Conf. *TRANSBALTICA-99*, Technics Press, April 1999, Vilnius, Lithuania.
17. ***Aladjev V.Z., Hunt U.J.*** Workstation for mathematicians // Intern. Conf. "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: BGU Press, 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: BGU Press, 1999.
20. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** Educational computer laboratory of the engineer // Proc. 8th Byelorussia Mathemat. Conf., vol. **3**, Minsk, Byelorussia, 2000.
21. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** Applied aspects of theory of homogeneous structures // Proc. 8th 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 V*** // Proc. Intern. 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 **//** Proc. Int. Conf. *TRANSBALTICA-2001*.- Vilnius: Technics Press, April 2001, Lithuania.
28. ***Aladjev V.Z., Shishakov M.L., Trokhova T.A.*** 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, 215 p.
30. ***Aladjev V.Z., Vaganov V.A., Grishin E.*** *Additional Functional Tools of Mathematical Package* ***Maple 6/7.-*** Tallinn: International Academy of Noosphere, 2002, 325 p., 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, 416 с., 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.З., Веетыусме Р.А., Хунт Ю.Я.*** 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, 151 с., ISBN 5-063-56078-5
37. ***Ефимова М.О., Хунт Ю.Я.*** Геометрия рисования: *Графический пакет* ***AutoTouch*** (*под. ред. акад.* ***В.З. Аладьева***).- Гомель: *Российская Академия Ноосферы*, 1997, 72 с., 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., ISBN 14-064298-56
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, ISBN 1-59682-013-6
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***. *Second edition*.- Tallinn: International Academy of Noosphere, 2004, 462 p.
47. ***Aladjev V.Z., Bogdevicius M.A.*** *Computer algebra system* ***Maple:*** A new software toolbox **//** **4**th 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 **//** **4**th Intern. Conf. *TRANSBALTICA-03*, Technics Press, April 2003, Vilnius, Lithuania, pp. 339-352.
49. ***Aladjev V.Z.*** *Computer Algebra System* ***Maple:*** A New Software Library **//** Intern. Conf. "Computer Algebra Systems and Their Applications", *CASA-2003*, Saint-Petersburg, Russia, 2003.

50. ***Aladjev V., Bogdevicius M., Vaganov V.*** Systems of Computer Algebra: A New Software Toolbox for ***Maple*** // Intern. Conf. on Software Engin. Res. and Practice, *SERP'04*, 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*** *and* ***S-Plus***.- Berlin: Springer-Verlag, 2005, ISBN 0-387-21016-4
66. ***New Developments in Classification and Data Analysis*** / *Eds*. **M. Vichi et al.-** Roma-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.-** Berlin: Springer-Verlag, 2004.
69. ***Dekking F.M. et al.*** *A Modern Introduction to Probability and Statistics*.- Berlin: Springer, 2005.
70. ***Лакин Г.Ф.*** Биометрика.- Москва: Изд-во «Высшая школа», 1990
71. ***CRC Standard Mathematical Tables and Formulae*** /*Ed*. **D. Zwillinger.-** Berlin: Springer, 1995
72. ***Encyclopedia of Statistical Sciences*** /*Eds*. **S. Kotz & N. Johnson,** vol. **1-9**.- N.Y.: Wiley, 1995
73. ***Aladjev V.Z., Haritonov V.N.*** *General Theory of Statistics*.- Palo Alto: Fultus Corporation, 2004.
74. ***Balakrishnan N., Chen W.*** *CRC Handbook of Tables for Order Statistics*.- Berlin: 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*.- Ontario: 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: Springer, 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 **//** Mathematical 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. Mathem. 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 **//** **7**th Intern. Conf. on Mathem. and Comp. Modelling, Chicago, USA, 1989.
101. ***Aladjev V.Z. et al.*** *Theoretical and Applied Aspects of Homogeneous Structures* **//** Proc. Intern. Workshop *PARCELLA-90.-* Berlin: Akademie-Verlag, 1990, p. 48-70.
102. ***Aladjev V.Z.*** Homogeneous Structures: Theoretical and Applied Aspects **//** The **8**th Intern. 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 (Cellular Automata)* (*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.: Springer, 2001
108. **http://www.aladjev-maple.narod.ru/DemoLib.zip**



Аладьев Виктор Захарович

Аладьев В.З. родился 14.06.1942 в г. Гродно (*Западная Беларусь*). После успешного завершения **2**-й *средней* школы (*Гродно*) в 1959 поступил на **1**-й курс физико-математического *факультета* Гродненского университета, а 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 ЭССР по *специальности* "*Теория вероятностей и математическая статистика*"**,** которую успешно закончил в 1972 г. 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 (*Эстония*). В 1991 **Аладьев В.З.** организовал научно-прикладную фирму *VASCO Ltd*., а с *конца* 1992 становится вице-президентом совместной фирмы *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*, *Белоруссии*, *Эстонии*, *Литве*, *Украине*, *ФРГ*, *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 – членом **IAMM** (*International Association for Mathematical Modelling*, **USA**). 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.З.** по совокупности научных работ в области *кибернетики* избран *академиком* Российской Академии Космонавтики по отделению "*Фундаментальных исследований*"**,** в сентябре 1994 г. он избирается *академиком* Российской Академии Ноосферы по отделению "*Информатики*". В сентябре 1995 **Аладьев В.З.** 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.З.** 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.З.** 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*" (**http://icraen.narod.ru/OrgList3.htm**).

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.

http://www.aladjev.newmail.ru, **http://www.geocities.com/noosphere_academy**
aladjev@yandex.ru, **valadjev@yahoo.com**, **aladjev@gmail.com**
Phone +(372) 63 56 078; GSM +(372) 56 50 4256; Fax +(372) 60 07 969