МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

П.Ю. Шалимов

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ ЛИСП, ЭРЛАНГ

Утверждено редакционно-издательским советом университета
в качестве учебного пособия

Брянск 2010

ББК 65.29

Шалимов, П.Ю. Функциональное программирование на языках Лисп, Эрланг : учеб. пособие / П.Ю. Шалимов. – Брянск: БГТУ, 2010. – 191 с.

I8BN 5-89838-224-0

Рассмотрены теоретические и практические положения, а также принципиальные вопросы функционального программирования: управление памятью, типизация, виды вычислений, управление в функциональной программе, определение функций и организация повторяющихся вычислений. Дано системное представление об основах высокоуровневого программирования на примере одного из самых популярных его стилей – функционального программирования.

Учебное пособие предназначено для студентов всех формы обучения специальностей 010503 «Математическое обеспечение и администрирование информационных систем», 230105 «Программное обеспечение вычислительной техники и автоматизированных систем», а также может быть полезно для лиц, самостоятельно изучающих функциональное программирование.

Ил. 3. Библиогр.– 24 назв.

Научный редактор В.В. Конкин

Рецензенты: кафедра «Информатика и прикладная математика» Брянского государственного университета;
канд. техн. наук Л.И. Евельсон.

I8BN 5-89838-224-0

ПРЕДИСЛОВИЕ

Учебное пособие предназначено для изучения дисциплины «Функциональное программирование». Материал учебного пособия разбит по уровням семантической значимости на следующие группы: основной; дополнительный; главный. Основной материал определяется уровнем знаний, необходимым для полного усвоения учебной дисциплины в объеме, определяемым Государственным образовательным стандартом. Основной материал не выделяется шрифтом или обрамлением.

Дополнительный материал помещен в пункт «Примечания» и содержит сведения расширяющие кругозор читателя, справочный материал, а также информация междисциплинарного значения. При чтении учебного пособия дополнительный материал можно опустить без ущерба для качества знаний.

К главному материалу относятся сведения, имеющие фундаментальное значение в конкретном разделе учебной дисциплины. Главный материал выполнен жирным шрифтом. Кроме того, к главному материалу следует отнести содержимое пункта «Резюме», где в краткой форме излагаются основные положения каждого раздела дисциплины.

Курсивом выделяются термины (понятия) и определения. Многие определения выделяются как главный материал.

Большое внимание в учебном пособии уделяется примерам решения задач программирования, которые представлены исходными кодами, и результатами решения задачи на ЭВМ. В обоих случаях соответствующий материал выделяется одинаково – обрамлением в виде строковых отступов от основного текста. В некоторых случаях пояснительный материал может быть встроен в пример как комментарий к программе. Все, что отмечено таким образом, может быть перенесено в исходный код и выполнено в соответствующей инструментальной системе.

В учебное пособие также включены упражнения (контрольные вопросы и задачи) для самостоятельной работы, которые будут

способствовать более глубокому изучению функционального программирования. Изучать это учебное пособие следует, имея сводный доступ к компьютеру, на котором установлены инструментальные системы Лиспа и Эрланга.

Учебное пособие предназначено для студентов всех форм обучения специальностей 010503 – «Математическое обеспечение и администрирование информационных систем», 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем», а также может быть полезно для лиц, самостоятельно изучающих функциональное программирование.

В качестве требований к начальной подготовке следует только поставить условие – знакомство с императивным программированием и практическое владение, хотя бы одним императивным языком программирования.

ВВЕДЕНИЕ

Функциональное программирование занимает, в определенной степени, промежуточное положение между императивным программированием на языках типа Си, Паскаль и логическим программированием на Языке Пролог. Оно позволяет разработчику сосредоточиться на решении собственно прикладной задачи и обладает сравнительно высокой вычислительной эффективностью.

Функциональное программирование характеризуется следующими признаками, некоторые из которых впоследствии переместились в императивные языки, повысив их уровень.

Требование использования только чистых функций (без побочных эффектов), не является уникальным признаком функционального программирования, но именно здесь оно становятся обязательным.

Отказ от присваивания и любых структуроразрушающих действий, которые являются основой императивного программирования, также становятся обязательными.

Бестиповость, которая означает отсутствие какое-либо связывания синтаксических объектов, оперируемых программистом, с ячейками памяти, и в этом плане полное отрешение от аппаратной реализации вычислительной системы.

Использование функции высшего порядка возможно практически во всех императивных языках программирования, но в функциональной парадигме применение таких функций позволяет вывести разрабатываемый проект на новый уровень абстракции, что приводит к большей выразительности.

Карринг, как форма записи функции от нескольких аргументов, в виде синтеза функций от одного аргумента, являлся до недавнего времени уникальным свойством функциональных языков.

Ленивые (отложенные) вычисления (уникальный признак функционального программирования) заключаются в том, что вычисления откладываются до тех пор, пока не понадобится их результат. Ленивые вычисления логично вытекают из концепции

функциональных языков, находят теоретическую основу в лямбда-исчислении Черча.

Лямбда–выражения (анонимные функции), создаются в любом месте программы с контекстом, существующим на время вычисления. С использованием лямбда-выражений создается особый вид функционального объекта – лексическое замыкание.

Замыкание − механизм связывания кода функции с ее лексическим окружением, наряду с объектами, является одним из способов композиции программ и данных.

Сопоставления с образцом является мощным инструментом анализа структур, широко используется в логическом программировании. В некоторых функциональных языках программирования используется как основной способ определения функции.

Использование лямбда-выражений, замыканий и механизма сопоставления с образцом позволяет значительно повысить выразительность программного кода. Программист, использующий платформу Microsoft.NET с императивными языками программирования, может применять такие традиционно функциональные инструменты, как лямбда-выражения, замыкания, сопоставления с образцом.

Изучение функционального стиля программирования позволяет сформировать полноценное «программистское мировоззрение», на основе которого появится возможность решения практически любых задач и активной позиции в области теоретических исследований и разработок.

ГЛАВА 1

ОСНОВНЫЕ ПОЛОЖЕНИЯ ВЫСОКОУРОВНЕВОГО ПРОГРАММИРОВАНИЯ

Парадигмы программирования
Стили декларативного программирования
Функциональное программирование в разработке систем искусственного интеллекта
Функциональное программирование в телекоммуникационных приложениях
Сферы применимости
Понятие высокоуровневого программирования

Высокоуровневое программирование, несмотря на условность и относительность этого понятия, связывается в большинстве случаев с декларативной парадигмой программирования (п. 1.1). Декларативная парадигма может реализовываться посредством логического либо функционального программирования.

При определении места функционального программирования среди других стилей программирования, необходимо определить в чем именно выражается «сверхвысокий» уровень изучаемых языков программирования, ввести критерии и признаки высокоуровневого функционального программирования.

Функциональное программирование ориентировано на решение задач, которые можно отнести к задачам искусственного интеллекта. Задачи искусственного интеллекта являются трудноформализуемыми задачами, которые можно описать в терминах «что необходимо получить», но, зачастую, довольно сложно определить в терминах «как получить результат». Таким образом, все эти задачи могут относиться либо к традиционной сфере применения функционального программирования – системам

искусственного интеллекта, либо к сравнительно новой сфере – телекоммуникационным приложениям.

1.1. Парадигмы программирования

Существуют две парадигмы (основные концептуальные схемы) программирования: *императивная* и *декларативная*. Основной (традиционной) парадигмой программирования является императивное (процедурное) программирование. В императивной парадигме рассматривается класс языков программирования, в которых программа явно определяет способ получения результата, но не указывает ожидаемых свойств результата. Процедура получения результата имеет вид последовательности операций. Этим обусловливаются следующие характерные черты императивной программы:

- указание логики управления в программе;
- определение порядка выполнения операций;
- наличие операторов присваивания, выполняющих разрушающее присваивание.

Императивная парадигма основана на *фон-неймановской* вычислительной модели[1], параметрами которой являются:

- программа, состоящая из набора команд, которые выполняются последовательно;
- поименованные области памяти (концепция переменных как областей памяти, к которым можно обращаться по имени).

Концепция памяти как хранилища значений переменных, содержимое которого может обновляться операторами программы, является фундаментальной в императивном программировании.

Реализация императивной программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти − значений исходных данных − в заключительное, в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Императивные языки программирования характеризуются следующими особенностями:

- необходимостью явного управления памятью, в частности описанием переменных;

- малой пригодностью для символьных вычислений;
- отсутствием строгой математической основы;
- высокой эффективностью реализации на традиционных ЭВМ.

Одним из значимых классификационных признаков языка программирования является его уровень. Уровень языка программирования определяется семантической (смысловой) емкостью его конструкций и степенью его ориентации на программиста. Язык программирования уменьшает разрыв между методами решения различных задач человеком и вычислительной машиной. Чем более язык ориентирован на человека, тем выше его уровень. К императивным языкам программирования относятся ассемблеры и хорошо распространенные языки программирования высокого уровня, например такие, как Фортран, Паскаль, Си.

Принципиально другую вычислительную модель предполагает декларативная парадигма программирования.

При использовании декларативного языка в программе в явном виде указывают свойства результата, но не определяют порядок его получения. В идеальном случае декларативная программа будет состоять из предложений, описывающих необходимый результат.

Характерно, что в этом случае порядок предложений не имеет значения, так как в декларативной парадигме отсутствует концепция переменной как поименованной области памяти и явное управление. Декларативные языки не привязаны жестко к традиционной фон-неймановской модели вычислений. В большинстве случаев алгоритм получения необходимого результата может иметь высокую степень параллелизма.

Декларативная парадигма программирования реализуется с помощью одного из двух стилей программирования: *функционального* или *логического* программирования.

1.2. Стили декларативного программирования

Декларативная парадигма программирования реализуется с помощью одного из следующих стилей программирования: функционального или логического программирования.

Академик А. П. Ершов[2] в предисловии к [15] основную идею функционального программирования определил как «… способ составления программ, в которых единственным действием является

вызов функции, единственным способом расчленения программы на части является введение имени для функции, а единственным правилом композиции − оператор суперпозиции функции. Никаких ячеек памяти, ни операторов присваивания, ни циклов, ни, тем более, блок-схем, ни передачи управления». Основной конструкцией в функциональных языках является символьное выражение (S-выражение). К S-выражениям относятся скалярные константы, структурированные
объекты, функции, тела функций и вызовы функций.

Функция рассматривается как однозначное отображение из области определения функции в область значений функции, что полностью соответствует математическому определению функции.

Функциональный язык программирования включает следующие элементы**:**

- классы констант, которыми могут манипулировать функции;
- набор базовых (определенных в данной системе) функций, называемых примитивами;
- правила построения новых функций на основе примитивов;
- правила формирования выражений на основе вызовов функций.

Программа представляет собой совокупность описаний функций и выражений, которые необходимо вычислить. Заданное в программе выражение вычисляется посредством *редукции* − серии упрощений (параграф 9.1.2) − по следующим правилам:

- вызовы функций-примитивов заменяются соответствующими значениями;
- вызовы определенных программистом функций заменяются их телами, в которых параметры замещены аргументами.

В функциональном программировании не используется концепция «память как хранилище значений переменных», характерная для *фон-неймановской вычислительной архитектуры* (п.1.1.). Операторы присваивания отсутствуют, вследствие чего переменные обозначают не области памяти, а объекты программы. Это полностью соответствует понятию переменной в математике. В функциональном программировании отсутствуют существенные различия между константами и функциями, т. е. между данными и программами. В результате этого функция может быть значением вызова другой функции, а также элементом структурированного объекта. Число аргументов при вызове функции не обязательно должно совпадать с числом параметров, указанных при ее описании.

Таким образом, функциональные языки можно определить как языки сверхвысокого уровня по отношению к языкам высокого уровня.

Логическое программирование основывается на понятии *отношения (реляция),* поэтому существует другое название логического программирования – *реляционное программирование*. Программа логического программирования представляет собой совокупность определений отношений между объектами и цели.

Процесс выполнения логической программы рассматривается как процесс установления общезначимости логической формулы, построенной по правилам, установленным семантикой используемого языка. Результат вычисления является побочным продуктом этого процесса.

В логическом программировании необходимо только специфицировать факты, описывающие задачу, а не определять последовательность шагов, которые требуется выполнить. Это означает декларативный характер языков логического программирования, которые отличаются:

- высоким уровнем;
- строгой ориентацией на символьные вычисления;
- возможностью инверсных вычислений, при которых переменные в процедурах не делятся явно на входные и выходные.

Логическое программирование не эффективно с вычислительной точки зрения. Программы на языке логического программирования имеют небольшое быстродействие, так как вычисления осуществляются методом проб и ошибок, поиском с возвратами к предыдущим шагам.

Несмотря на это, логическое программирование предпочтительно именно с точки зрения приближения к декларативной парадигме программирования. Наиболее эффективно логическое программирование при решении неформальных задач, задач, алгоритм решения которых неизвестен или получение алгоритма сопряжено с большими затратами рабочего времени высококвалифицированного специалиста. В этом случае описание задачи в терминах того, что необходимо получить, приводит к цели с наименьшими затратами рабочего времени. Наибольшее число описанных задач относится к задачам искусственного интеллекта.

1.3. Функциональное программирование в системах искусственного интеллекта

Под термином *системы искусственного интеллекта(СИИ)* понимаются кибернетические системы, моделирующие некоторые стороны интеллектуальной деятельности человека: логическое, аналитическое мышление.

Задачей-максимум в области СИИ можно считать понимание принципов и механизмов интеллектуальной деятельности человека. Однако в идеале вряд ли такая задача достижима вообще. Практически работы в области СИИ проводились по некоторым задачам, каждая из которых предполагала имитирование отдельной, строго ограниченной области интеллектуальной деятельности. Ниже приводится перечень основных этих задач:

- *проблемы естественного языка*[3] (ЕЯ-проблемы), основная цель − общение с ЭВМ на естественном для человека языке;
- *экспертные системы* – системы обработки данных, основанные на знаниях и экспертных оценках в некоторой области;
- *представление знаний*[4] – вопросы представления информации для решения интеллектуальных проблем;
- *распознавание образов* – автоматическое наблюдение и идентификация (классификация) объектов;
- *доказательство теорем* – разработка общей системы доказательства и решения проблем;
- *программирование игр* – разработка специальных процедур поиска, оценки и выбора решений;
- *интеллектуальная робототехника* – исследования, направленные на снабжение машин измеряющими устройствами, аналогичными по своему действию механизмам ориентации в живом мире.

Программы на декларативных языках значительно ближе к логической спецификации задачи (ЛС), чем на любом императивном языке. В большинстве случаев это просто записывание логической спецификации задачи в терминах конкретного языка программирования.

В системах искусственного интеллекта решают сложные задачи, которые недостаточно понимаемы. Для большинства задач искусственного интеллекта не существует четко заданных

алгоритмических решений, но они могут быть исследованы с помощью механизмов символических рассуждений.

1.4. Функциональное программирование в телекоммуникационных приложениях

Системы искусственного интеллекта являются классической и исторически первой областью применения функционального программирования. В последнее время появились области, где применение функционального программирования позволяет эффективно и конкурентно (по сравнению с императивным программированием) решать задачи. Одна из таких областей – программирование телекоммуникационных приложений. Применение функционального программирования во многом определяется рядом особенностей и требований к инструментам программирования для телекоммуникационных приложений:

- распределенные вычисления – инструментальная система должна поддерживать работу на нескольких узлах вычислительной среды, под разными операционными системами;
- поддержка параллельных вычислений – создание приложений с большим числом параллельно работающих процессов при обеспечении требований по времени отклика системы;
- время отклика разрабатываемых систем должно быть минимальным и обеспечиваться уже инструментальными средствами разработки приложений;
- контролируемая (инкрементальная) загрузка кода обеспечивает поддержку процесса загрузки программы по частям с возможностью замены модулей;
- масштабируемость – требование увеличения производительности телекоммуникационной системы при добавлении вычислительных узлов;
- поддержка горячих изменений программного кода. Предполагается, что на работающей системе старый код может меняться на новый код без выключения. В определенный момент времени одновременно существуют обе программы[5];
- обеспечение надежности работы системы. Система должна работать постоянно, ее нельзя выключать даже для обслуживания и обновления программного обеспечения. При отказе процессы

автоматически переносятся на другие узлы и обратно при восстановлении работоспособности системы;

- высокая производительность при максимальных нагрузках. При достижении максимальной нагрузки недопустимо или нежелательно уменьшение пропускной способности системы.

Все требования обеспечиваются современными функциональными языками программирования, причем с эффективностью, превышающей эффективность императивных и очень популярных систем программирования[6]. При этом логика применения функционального программирования для решения задач может определяться следующей цепочкой рассуждений. Требования отказоустойчивости и надежности системы приводит к отказу от программирования с помощью побочных эффектов, т.е. к функциональной программе (см. главу 2). В языке Эрланг (Erlang) отсутствуют императивные включения и уменьшается возможность программирования с помощью побочных эффектов.

1.5. Сферы применимости

С учетом положений, изложенных в предыдущих параграфах, можно определить признаки функциональных задач, которые целесообразно решать с помощью средств функционального или логического программирования. Четких и однозначных рекомендаций по выбору стиля программирования в общем случае реальной функциональной задачи нет, поэтому можно сформулировать определенные эвристические правила, позволяющие определиться при выборе подходов к решению конкретной задачи. Языки функционального и логического программирования следует применять в следующих случаях: алгоритм решения задачи не известен, разработка алгоритма сопряжена со значительными трудозатратами, решение телекоммуникационных задач для систем с высокими требованиями к надежности, реализующие распределенные вычисления.

Функциональное программирование предоставляет в распоряжение программиста средства высокоуровневой разработки программного обеспечения, применяемые для решения специфичных задач, в основном задач искусственного интеллекта и символьных вычислений. Функциональное программирование нецелесообразно применять для решения численных задач, так как большинство этих

задач имеют достаточно хорошо разработанный алгоритм и требовательны к затратам вычислительных ресурсов (оперативной памяти и времени счета).

Телекоммуникационные приложения − новая сфера применения функционального программирования − своим появлением обязана языку Эрланг.

Разработчики программного обеспечения для мультиядерных процессоров применяют язык Эрланг в качестве основной инструментальной среды из-за его способности эффективно поддерживать параллельное программирование. Способность эффективно распараллеливать вычислительный процесс обусловливается функциональной парадигмой программирования и основывается на математической теории лямбда-исчисления Черча (п. 9.1.3).

1.6. Понятие высокоуровневого программирования

Традиционно программированием на языках высокого уровня считается императивное программирование с использованием языков Фортран, Паскаль, Си. В первом приближении уровень программирования определяется по принципу близости языка программирования к естественной для человека записи, поэтому, рассматриваемые языки программирования оказываются более высокого уровня, чем, например, язык Ассемблер, в котором команды языка соответствуют командам процессора. Использование языков высокого уровня позволяет абстрагироваться от аппаратной реализации компьютера, использовать лаконичные смысловые конструкции.

Однако кроме синтаксической записи понятие «уровень языка программирования» включает возможность семантического проектирования в модели, наиболее близкой к стилю мышления человека. По данному критерию уровень программ на императивных языках программирования будет значительно ниже уровня декларативных программ, так как требуется выписывать последовательный набор инструкций для получения решения.

Для классификации языков программирования по их уровню и четкого определения понятия «высокоуровневый язык» предлагаются

следующие критерии: «управление памятью», «поток управления», «образ мышления программиста».

Множество значений шкалы критерия «управление памятью» ограничено значениями «ручное» и «автоматическое». При ручном управлении программист явно выделяет память, связывая, например, символьное имя с конкретным типом данных, и явно ее освобождает. При автоматическом управлении память полностью абстрагируются от аппаратных характеристик вычислительной системы. Введенное программистом имя будет означать абстракцию решаемой задачи. Именно такое управление будет соответствовать высокоуровневому программированию. В дальнейшем будет показано, что автоматическое управление памятью обеспечивают бестиповые языки программирования, к которым относятся Лисп, Эрланг[7] (п.2.4, 4.3,4.4). Для бестиповых (крайний случай динамической типизацией) языков программирования верны следующие, кажущиеся на первый взгляд парадоксальными утверждения: переменная в разные моменты времени может представлять разные объекты и переменная в один момент времени представляет разные объекты[8](п. 4.4). Типы связаны со значениями, а не с переменными, поэтому понятие «высокоуровневое функциональное программирование» идентично понятию «бестиповое функциональное программирование».

Примечания

1. Фон Нейман Джон (1903 – 1957 гг.). Венгр по национальности. Окончил в 1926 г. Будапештский университет. Фон Нейман преподавал в Германии, в 1930 г. эмигрировал в США и стал сотрудником Принстонского института перспективных исследований. В 1946 г. вместе с Г.Гольдстейном и А.Берксом он написал и выпустил отчет «Предварительное обсуждение логической конструкции электронной вычислительной машины». Поскольку имя фон Неймана как выдающегося физика и математика было к тому времени более известно среди ученых, чем имена его соавторов, все высказанные положения в отчете приписывались ему. Более того, архитектура ЭВМ с последовательным выполнением команд в программе получила название «фон Неймановской архитектуры ЭВМ».
2. Ершов Андрей Петрович (1931−1988 гг). Один из зачинателей теоретического и системного программирования. Создатель сибирской школы информатики. Фундаментальные исследования А.П.Ершова в

области схем программ и теории компиляции оказали заметное влияние на его многочисленных учеников и последователей. Книга А.П.Ершова «Программирующая программа для электронной вычислительной машины БЭСМ» была одной из первых среди монографий по автоматизации программирования. В 80-х годах прошлого века он начал эксперименты по преподаванию программирования в средней школе, которые привели к введению курса информатики и вычислительной техники в средние школы страны.

3. Разработка естественно-языковых интерфейсов и машинный перевод. В настоящее время используется более сложная модель, включающая анализ и синтез естественно-языковых сообщений. Модель состоит из нескольких блоков: *морфологический анализ* − анализ слов в тексте; *синтаксический анализ* − анализ предложений, грамматики и связей между словами; *семантический анализ* − анализ смысла каждого предложения на основе некоторой предметно-ориентированной базы знаний; *прагматический анализ* −анализ смысла предложений в окружающем контексте на основе собственной базы знаний. Синтез включает аналогичные этапы, но несколько в другом порядке.
4. Первые исследования по представлению знаний включали модели, основанные на семантических сетях, концептуальных зависимостях, сценариях и фреймах. В дальнейшем появились модели на стохастическом, коннекционистском и генетическом представлении. [12]
5. При изучении планет Солнечной системы космический корабль Deep Space 1 должен был несколько дней работать под управлением программы, написанной на языке LISP. В полете была обнаружена ошибка в программном обеспечении, которую благодаря возможностям языка LISP удалось найти и исправить.
6. Язык программирования Эрланг, изучаемый в учебном пособии (глава 2), использует легковесные процессы (подпроцессы). Время создания подпроцесса и переключения контекста на два порядка меньше по сравнению с потоками .NET. Они взаимодействуют посредством асинхронной посылки сообщений, которая в свою очередь требует ресурсов в 50 раз дешевле, чем аналогичная операция в .NET.
7. Отметим, что «бестиповость» языка не означает, что отрицается полностью понятие «тип данных». Практически все бестиповые языки имеют мощную систему типов данных. Однако такие языки предполагают возможность разработки программы без связывания символьного имени с типом данных. Бестиповыми являются языки логического программирования в Единбургском диалекте.
8. Оба этих утверждения соответствуют только языку Лисп.

Резюме

- Императивная парадигма ориентируется на спецификацию способов получения решения, а декларативная − на спецификацию искомого решения.
- В функциональном программировании искомое решение описывают с помощью функций, в логическом программировании − логическими отношениями.
- Функциональное программирование используется в системах искусственного интеллекта и в телекоммуникационных приложениях.

ГЛАВА 2

ОБЛИК ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

Теоретические основы функционального программирования
Программирование в функциональных обозначениях
Виды вычислений в функциональных языках
Обзор функциональных языков программирования

Функциональное программирование основано на положении, что в результате каждого действия возникает значение. Значение становится аргументом следующего действия, и конечный результат задачи выдается пользователю. Программа в функциональном программировании состоит из логически расчлененных определений функций. которые состоят из организующих вычисления управляющих структур и из вложенных, часто вызывающих самих себя (рекурсивных) вызовов функций [6,11,12,13].

А.П. Ершов, в предисловии к книги Хендерсона П. [18], определял функциональное программирование как «способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции − оператор суперпозиции функции. Никаких ячеек памяти, ни операторов присваивания, ни циклов, ни, тем более, блок-схем, ни передач управления».

Положения, сформулированные А.П. Ершовым, составляют необходимые и достаточные условия функционального стиля программирования. Необходимыми являются положения, определяющие ограничения, которые должен принять программист и облик функционального инструмента программирования: бестиповый язык программирования, запрет разрушающих действий и соответствующих способов программирования.

2.1.Теоретические основы функционального программирования

Теоретическими основами функционального программирования являются строгий математический аппарат *лямбда-исчисления Чёрча* и *теория рекурсивных функций*.

Введенное в 1931 году математиком Алонзо Черчем[1], лямбда-исчисление оперирует всего тремя типами элементов [1,11]:

- символами, представляющими переменные и константы;
- скобками для группировки символов;
- обозначениями функций с использованием греческой буквы лямбда.

Лямбда-исчисление применяется для синтаксического описания свойств математических функций и обработки их в качестве правил (гл. 9).

Функциональная программа состоит из совокупности определений функций. Функции в свою очередь представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов (предложений, описывающих свойства результата в декларативной программе, гл. 1). Вычисления начинаются с вызова некоторой функции, которая вызывает функции, входящие в состав ее определения и т. д. в соответствии с иерархией определений и структурой условных предложений. Функции могут прямо или опосредованно вызывать сами себя. Каждый вызов возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается. Процесс повторяется до тех пор, пока запустившая вычисления функция не вернет конечный результат пользователю. В функциональном программировании не применяется присваивание и передача управления. Разветвление вычислений основано на механизме обработки аргументов условного предложения.

При таком подходе к программированию рекурсия является единственным способом организации повторяющихся вычислений. Подробно рекурсия как способ решения задач, механизм организации повторяющихся вычислений в функциональных программах, а также основы теории рекурсивных функций будут подробно рассмотрены в гл. 7**.**

В функциональной программе не должно быть:

- присваиваний;
- глобальных переменных;
- обработки исключений;
- функций с побочными эффектами.

Этот перечень следует из основного свойства функционального программирования – прозрачности по ссылкам.

2.2. Программирование в функциональных обозначениях

Прозрачность по ссылкам (*функциональность*) является фундаментальным свойством математических функций и может быть сформулировано следующим образом: значение функции зависит только от нее самой и аргументов вызова. Это означает, что каждое выражение определяет единственную величину, которую нельзя изменить ни путем ее вычисления, ни предоставлением различным частям программы возможности совместно использовать это выражение. Ссылки на некоторую величину эквивалентны самой величине, и возможность ссылаться на некоторое выражение из другой части программы не влияет на значение этого выражения.

Большинство императивных языков позволяют функции в процессе своего выполнения читать и изменять значения *глобальных переменных*. Такие функции называются *функциями с побочными эффектами*, поэтому если вызывать одну и ту же функцию несколько раз с одним и тем же аргументом, можно в результате получать различные результаты.

В функциональном программировании оператор присваивания отсутствует, объекты нельзя изменять и уничтожать, можно только создавать новые декомпозицией и синтезом существующих объектов. О ненужных объектах позаботится встроенный в язык *сборщик мусора* (удаление невостребованных объектов программы вынесено на уровень системы), поэтому все функции свободны от побочных эффектов.

Благодаря отсутствию побочных эффектов имеется еще одно преимущество – *параллелизм вычислений*. Можно предположить, что если все функции для вычислений используют только свои

параметры то можно вычислять независимые функции в произвольном порядке или, скажем, параллельно. На результат вычислений это не повлияет. Причем параллелизм может быть организован как на уровне компилятора с языка, так и на уровне архитектуры. В лямбда-исчислении доказывается теорема, которая подтверждает предположение на уровне математической теории (гл. 9).

2.3. Виды вычислений в функциональных языках

В большинстве императивных языков программирования все аргументы вычисляется, а затем передаются функции. Таким образом, аргумент передается по значению**,** подразумевая при этом, что только его значение передается в тело функции. Такое правило вычислений или механизм вызова называется *вызовом по значению***.** Его преимущество заключается в том, что реализация достаточно проста: сначала вычисляется аргумент, а затем вызывается функция. Недостатком является избыточность вычислений, когда значение аргумента не требуется вызываемой функции.

Противоположным вызову функции по значению является *вызов по необходимости***,** при котором все аргументы передаются функции в невычисленном виде и вычисляются только, если в них возникает необходимость внутри тела функции. Преимущество вызова по необходимости является отсутствие избыточных вычислений, если значение аргумента не понадобится, недостатоком – более сложная, по сравнению с вызовом по значению, реализация системы программирования. В этом случае функциям передаются не значения тех или иных аргументов, а выражения, которые при определенных условиях придется вычислять и применять в теле функции. Вызов по необходимости не характерен для императивных языков программирования и многих функциональных языков, а функциональные языки, поддерживающие вызов по необходимости, в большинстве своем снабжены конструкторами, обеспечивающими вычисление по схеме вызов по значению.

В теории функционального программирования рассматривают противоположные виды вычислений: *энергичное* и *ленивое***.**

Ленивое вычисление − стратегия планирования, при которой вычисления не выполняются до тех пор, пока не возникнет необходимость в их результатах.

Энергичное вычисление – стратегия, при которой вычисления выполняются, как только появляется возможность их выполнить.

Принцип энергичного вычисления можно сформулировать следующим тезисом: «на каждом шаге вычисляется все, что может быть вычислено». Не важно, пригодится ли в конечном случае полученный результат. Таким образом, всегда существует вероятность выполнения невостребованных вычислений.

Принцип ленивого вычисления: «не вычислять ничего, что не требуется в данный момент». Можно провести некоторую аналогию энергичного вычисления в функциональных языках с механизмом вызова по значению в императивных языках, а ленивое вычисление с механизмом вызова по необходимости. Однако между ними не существует тождественного равенства, и точное соотношение между этими терминами будет понятным, после изучения гл. 9. Если функциональный язык не поддерживает отложенные вычисления, то он называется *строгим (strict)* языком программирования, в отличие от *нестрогого (lazy)* языка с ленивой стратегией вычисления.

2.4. Обзор функциональных языков программирования

Современные функциональные языки подразделяются на две основные группы по принципу практической применимости: академические, практические. Академические отражают современное состояние научных и прикладных исследований в области теории программирования, Принцип практической применимости.

Множество функциональных языков программирования будет описываться на основании следующих критериев:

- вид вычислений;
- типизация;
- чистота;
- практическая применимость.

Критерий «вид вычислений» определяет поддержку отложенных вычислений и имеет значения *strict и lazy.*

Критерий «типизация» определяет момент, когда определяется тип переменной. Критерий имеет значения *статическая типизация* и *динамическая типизация.* Статическая типизация определяет связывание типа с переменной в момент ее объявления. При динамической тип связывается с переменной в момент присваивания значения. Статическая типизация способствует выработке более надежного кода, а программы с динамической типизацией обладают большей общностью.

Критерий «чистота» определяет наличие в функциональном языке возможностей для императивного программирования. Критерий имеет значения *чистый* (без императивных включений) и *не чистый* язык программирования.

Критерий «практическая применимость» определяет основную сферу приложения определенного языка программирования. Значение *академический* характеризует язык, который используется для научных и прикладных исследований в области программирования, а значение *практический* описывает язык, используемый в коммерческих проектах.

Перечень наиболее популярных функциональных языков программирования [8,11,13] приводится ниже с использованием следующих критериев: «общие сведения»; «типизация»; «вид вычисления»; «чистота».

Scheme. Диалект языка Лисп, предназначенный для научных исследований в области информатики и обучения функциональному программированию. Благодаря отсутствию императивных включений объем языка получился намного меньше, чем Common Lisp. Значения критериев: strict, динамическая типизация, чистый, академический.

Refal[4]. Семейство языков, разработанных В. Ф. Турчиным. Старейший член этого семейства впервые реализован в 1968 году в России. Содержит элементы логического программирования (сопоставление с образцом). Значения критериев: *lazy*, динамическая типизация, чистый, академический,

Miranda. Строго типизированный, поддерживает типы данных пользователя и полиморфизм. Разработан Тернером на основе более ранних языков SALS и KRC. Имеет ленивую семантику. Без императивных включений. Значения критериев: *lazy*, статическая типизация, чистый, академический.

Haskell. Широко применяется в научных исследованиях. В некоторых западных университетах используется в качестве

основного языка для изучения студентами. Один из наиболее мощных функциональных языков. Ленивый, чисто функциональный, типизированный язык. Haskell – отличный инструмент для обучения и проведения экспериментов со сложными функциональными типами данных. Программы, написанные на языке Haskell**,** имеют значительные размер объектного кода и невысокую скорость исполнения. Значения критериев: *lazy*, статическая типизация, чистый, академический.

Clean. Диалект Haskell, приспособленный к потребностям практического программирования. Как и Haskell, является ленивым чисто функциональным языком, содержит классы типов. Но Clean также содержит интересные особенности, которые не имеют эквивалента в Haskell. Например, императивные возможности в Clean основаны на *уникальных* типах, идея которых заимствована из линейной логики (*linear logic*). Clean содержит механизмы, которые позволяют значительно улучшить эффективность программ. Среди этих механизмов явное подавление отложенных вычислений. Значения критериев: *lazy*, статическая типизация, не чистый, академический.

ML(Meta Language). Разработан группой программистов во главе с Робертом Милнером в середине 70-х гг. в Эдинбурге (Edinburgh Logic for Computable Functions). Идея языка состояла в создании механизма для построения формальных доказательств в системе логики для вычислимых функций. В 1983 г. язык был пересмотрен и дополнен такими концепциями, как модули. Стал называться «стандартный ML» (Standard ML). Значения критериев: strict, статическая типизация, чистый, академический.

Common Lisp. Версия Лиспа, которая с 1970 г. может считаться стандартом языка, благодаря поддержке со стороны лаборатории искусственного интеллекта Массачусетского технологического института (гл. 3), бестиповый, энергичный, с большим набором императивных включений, допускающих присваивание, разрушение структур. Практический. На языке Лисп был написан векторный графический редактор *Автокад*. Благодаря набору этих свойств, а также то, что Лисп может быть отнесен к классике программирования, он выбран в качестве базового языка при изучении функционального программирования. Значения критериев: strict, динамическая типизация, не чистый, практический.

Erlang[5] строгий функциональный язык. Практический, ориентирован на разработку телекоммуникационных приложений. Имеет поддержу распределенных вычислений и отказоустойчивости. Разработан во второй половине 80-х в Ericsson Computer Science Laboratory. Использование большинства современных языков программирования (императивных) приводило к усложнению системы. За основу для разработки нового языка программирования были выбраны императивный язык Фортран, функциональный язык Лисп, и язык логического программирования Пролог. Отметим, что язык Эрланг семантически близок к Лиспу, а синтаксически к Прологу. Благодаря тому, что Эрланг практический язык, он не поддерживает ленивые вычисления, **карринговой**[6] формы представления функции и других присущих в основном академическим языкам особенностей (зачастую именно эти возможности и уменьшают надежность приложений). Указанные свойства языка (практичность, современность) предопределили то, что Эрланг выбран в качестве второго основного языка для изучения в данной работе. Значения критериев: strict, динамическая типизация, чистый, практический.

O'Caml (Objective Caml). Разработан и поддерживается INRIA (Французский национальный институт информатики и автоматики). Практический, поддерживает функциональный, императивный и объектно-ориентированный стили. O'Caml является диалектом языка ML (Meta Language). Существует возможность использовать «ленивые» функции. Значительная статическая типизация. Система типов в O'Caml более строгая, чем у традиционных статических языков. Не допускается «скрытое» преобразование типов и значения «неизвестного типа». В этом плане – полная противоположность Лиспу. Допускает возможность автоматического определения и явного указания типов. Не требует указания типов в тех случаях, когда их можно определить из контекста на этапе компиляции. Значения критериев: *lazy*, статическая типизация, не чистый, практический.

F#. Язык семейства языков Microsoft® .NET Framework. Разработан в отделе исследований корпорации Microsoft. Обладает достаточно высокой производительностью относительно большинства функциональных языков. Синтаксически совместим с O'Caml. Значения критериев: strict, статическая типизация, не чистый, академический.

Примечания

1. Черч Алонзо (Church Alonzo) (родился в 1903 г. в Вашингтоне) – американский логик и математик, проффесор Принстонского и Калифорнийского университетов. Внес большой вклад в развитие математической логики и теории автоматов. В математической логике большое значение имеет тезис ЧЕРЧА: понятие вычислимости по Тьюрингу является корректной формализацией нашего интуитивного понятия эффективной вычислимости. На русский язык переведена его книга «Введение в математическую логику» – М., 1960.
2. Аналогичная ситуация в логическом программировании, например «некоронованный» стандарт языка Пролог, так называемый «эдинбургский диалект» бестиповый, а одна из распространенных систем программирования, базирующаяся на Турбо-Прологе, является строго типизированной.
3. Следует отметить, что деление языков на чистые и нечистые дает тот же результат, что и классификация их на практические и академические. Практические системы используются для реального программирования, поэтому туда неизбежно приходится встраивать императивные включения для повышения эффективности создаваемых программ.
4. Основой Рефала является теория нормальных алгоритмов Маркова (реализованная во внутренней машине вывода), которая потенциально мощнее теории исчисления Хорновых дизъюнктов, воплощенной в Прологе.
5. Назван в честь Агнера Эрланга (**Агнер Краруп Эрланг 1878-1929**) - датского инженера, известного своим вкладом в создание математической теории массового обслуживания, которая используется при создании распределенных отказоустойчивых систем, работающих в реальном масштабе времени. В теории телетрафика для систем массового обслуживания с потерями Эрлангом определено выражение для вероятности отказа в обслуживании. Эрлангом (erlang) названа также единица измерения трафика (эрл). В общем случае интенсивность трафика в эрлангах находится как произведение среднего времени обслуживания вызова (продолжительности разговора) на среднее число вызовов в единицу времени. Его именем названо также распределение Эрланга, которое часто используется в теории надежности.
6. Карринг – преставление функции от нескольких переменных в виде конкатенации функций от одного переменного. Названа честь математика Хаскеля Карри. Является основной формой записи в лямбда-исчислении. Подробно гл. 5.

Резюме

Программа, написанная на функциональном языке, состоит:

- из неупорядоченного набора уравнений определяющих функции и значения;
- функций, определяемых рекурсивно и через другие функции и значения;
- значений, задаваемых как функции от других значений.

В функциональной программе не должно быть:

- присваиваний;
- удаления и уничтожения объектов;
- глобальных переменных;
- обработки исключений;
- функций с побочными эффектами.

Функциональному программированию присущи следующие признаки и свойства:

- отсутствие побочных эффектов;
- параллелизм;
- отложенные вычисления.

Упражнения

1. Чему равен возраст функционального программирования (определить по самому старшему языку)?
2. Какой функциональный язык программирования самый молодой?
3. Какой критерий функциональности следует считать главным при определении соответствия программы принципам функционального программирования?

4. Каким образом образовываются новые функции? С помощью какой операции и на основании чего?
5. Какие признаки функциональной программы можно считать вторичными (вытекающими) из главного признака?
6. Каких механизмов не должно быть в функциональной программе?
7. Почему в функциональной программе не должно быть уничтожения объектов?
8. Как можно охарактеризовать функциональную систему программирования с позиций управления памятью?
9. Какие механизмы из перечисленных ниже, можно отнести к побочным эффектам программирования? Что полностью соответствует принципу функциональности:

- присваивание значения переменной;
- удаление структуры;
- вызов функции;
- печать результатов выполнения программы;
- запись в файл результатов выполнения некоторой операции;
- формирование новой структуры из имеющихся структур и скалярных объектов на основе анализа и копирования?

10. Что понимается под строго типизированным языком программирования? Для чего вводится строгая типизация?

11. Какие положительные черты сопутствуют строгой типизации в языках программирования и какие положительные черты вытекают из «бестиповости» языка программирования?

ГЛАВА 3

ЯЗЫКИ ПРОГРАММИРОВАНИЯ ЛИСП И ЭРЛАНГ

История Лиспа и Эрланга
Символьные выражения Лиспа
Типизация в Лиспе
Система примитивов Лиспа и статические связи
Символьные выражения Эрланга
Базовые функции Эрланга и основные операции

Язык программирования Лисп обладает рядом новаторских решений, позволяющих провести аналогию новой разработки языка программирования с изобретением [12,13]. В облике этого языка программирования просматриваются признаки существенных отличий, необходимых для признания его изобретением: программы и данные представляются в виде основного структурного объекта – списка; динамическая проверка типов и позднее связывание; автоматическое и динамическое управление памятью.

Язык Эрланг, разработанный спустя более двух десятков лет после языка Лисп, также обладает рядом новаторских решений, одно из которых состоит в поддержке **легковесных процессов**[1], что ранее было присуще только операционным системам[24].

3.1. История Лиспа и Эрланга

Язык Лисп (Lisp) был создан в течение 1959−1962 г.г. группой авторов под руководством Дж. Маккарти[2] в Массачусетском технологическом институте. Слово LISP является аббревиатурой от «List Processing» (обработка списков). Одной из особенностей Лиспа, которая отличает его от большинства других языков, является то, что в нем поддерживается только один составной тип данных – список. Оригинальный язык Лисп, разработанный Дж. Маккарти, был чисто функциональным (в смысле прозрачности по ссылкам). Появившиеся в последующие годы диалекты содержат многие императивные особенности, например конструкции для выполнения разрушающего

присваивания. Однако в каждом из этих диалектов существует чистое подмножество языка Лисп, на котором можно писать функциональные программы.

Самым популярным диалектом языка Лисп является диалект Common Lisp, и может считаться стандартом языка, благодаря солидной поддержке со стороны лаборатории искусственного интеллекта Массачусетского технологического института. С появлением ПЭВМ были разработаны следующие реализации языка Лисп.

- система PC Scheme фирмы Texas Instruments (компилирующая);
- система LISP фирмы Microsoft (компилирующая);
- система LISP фирмы Norell (интерпретатор);
- система Mulisp 85 фирмы Microsoft (интерпретатор).

Кроме того, имеются интерпретаторы языков Clisp, WinLisp, XLisp, которые принадлежат ветви Common Lisp иерархического дерева диалектов Лиспа, а также Scheme для чисто функционального программирования. Версия Лиспа под названием Автолисп, на которой разработан векторный графический редактор (являющийся инструментом компьютерного проектирования) Автокад, также принадлежит к ветви Common Lisp.

Язык Эрланг разработан во второй половине 80-х г.г. в лаборатории информатики (Computer Science Laboratory) компании Ericsson. В результате исследований свойств известных языков программирования (в качестве основных на последней стадии исследований рассматривались языки Фортран, Лисп, Пролог) был разработан язык, который наиболее подходит для телекоммуникационных приложений. При этом функциональное программирование было признано наиболее подходящим для данного вида задач, так как упрощает разработку телекоммуникационных приложений (глава 1). Язык Эрланг является одним из немногих практических функциональных языков, разработанных для промышленных применений, применяется во многих областях, помимо телекоммуникаций.

Инструментальная система Эрланга доступна в виде свободно распространяемого программного обеспечения Erlang/OTP. OTP[2] (Open Telecom Platform) представляет собой набор библиотек, содержащих решения многих типичных сетевых и телекоммуникационных задач.

3.2. Символьные выражения Лиспа

Основной конструкцией в функциональных языках является символьное выражение (S-выражение). К символьным выражениям относятся скалярные константы и структурные объекты. К структурным объектам в Лиспе относятся: функции; определения функций; вызовы функций; вычислимые формы. Подробно структура S-выражения представлена на рис. 1. Скалярные (простейшие) объекты называются *атомами***.** Из них строятся все структурные объекты программы. К атомам относятся символы и константы. Символ языка Лисп – это имя, состоящее из букв, цифр и специальных знаков. Примеры символов:

Lisp
conned
symbol15
privet_14

Символы могут состоять из прописных или строчных букв, однако интерпретатор XLISP, как и большинство других интерпретаторов, их отождествляет. Символы могут применяться для обозначения других объектов программы, т.е. в качестве *переменных.* В отличие от символов числа, а также логические величины T (истина) и *NIL* (ложь) представляют только самих себя, не используются для обозначения других объектов, следовательно, являются константами.

Отметим, что в интерпретаторе XLISP символ T может быть применен в качестве переменной. Следует обратить внимание на особую значимость символа NIL. С одной стороны, это системно предопределенная константа, обозначающая логическое значение «ложь», с другой стороны, этот символ может обозначать *пустой список* (список, не содержащий ни одного элемента)**.**

Единственный структурный объект в чистом Лиспе – список. **Список – это структура, которая либо пуста (NIL), либо состоит из головы и хвоста. Хвост также является списком.**

Следует обратить внимание на элементы рекурсии (гл. 7), которые видны в приведенном декларативном определении списка.

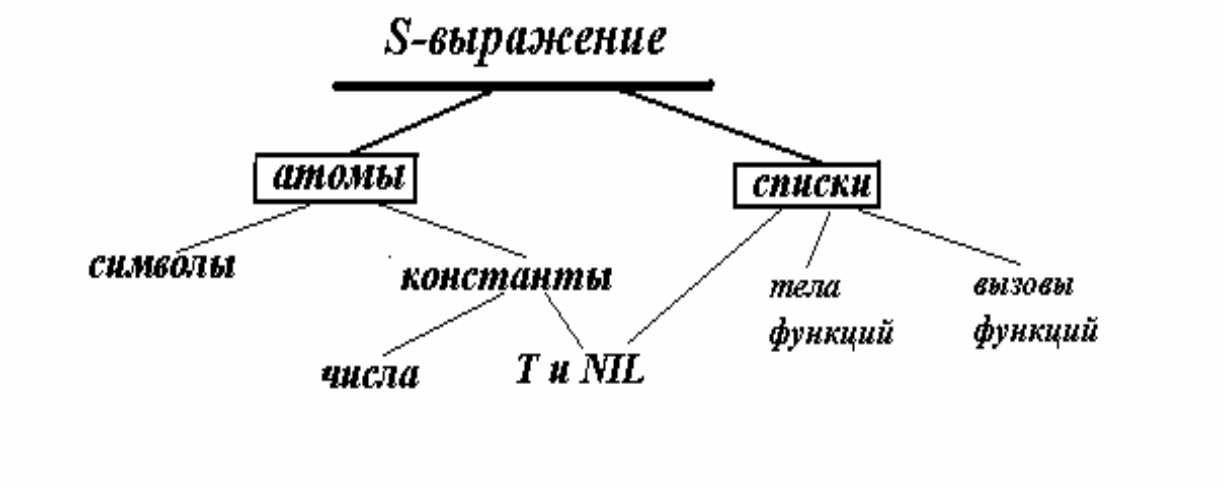


Рис. 1. Структура символьных выражений Лиспа

Список в языке Лисп представляется последовательностью элементов, заключенной в круглые скобки. Элементами списка могут быть списки. В этом случае получается *многоуровневый список*. Вот примеры списков на языке Лиспе.

1. (+ 5 2)
2. ‘(a в c)
3. ‘(a (т к) о ((к о)))
4. (defun max (x1 x2) (cond ((> x1 x2) x1) (t x2)))
5. (max 5 3)

Первый список в данной интерпретации является вызовом функции сложения двух чисел.

Эти примеры иллюстрируют существенное свойство языка Лисп. Данные и программы (функции) представляются единообразно в виде списков. Таким образом, четвертый пример можно в зависимости от окружения рассматривать как список, состоящий из атомов «defun»,«max», подсписка (x1 x2).

Второй – одноуровневый список, состоящий из атомов «a,в,c». Третий – многоуровневый список, состоящий из двух атомов «a, о» и двух подсписков (т к) и ((к о)), причем второй подсписок состоит из единственного элемента списка (к о).

Четвертый – многоуровневый список, который интерпретатором XLISP будет распознан как определение функции нахождения максимального значения. Наконец, пятый список – это вызов функции нахождения максимального значения с аргументами 5 и 3.

Списками в языке Лиспе представляются программы и данные. В Лиспе принята единообразная (имя функции и аргументы записываются внутри скобок) *префиксная нотация*. Вычисляя

значение вводимого выражения, интерпретатор по первому символу *S-выражения* определяет, что представляет собой данное выражение (функция или данные). Если введенное выражение – вызов функции, то первый символ списка будет распознан как имя функции, а остальные элементы списка – как аргументы вызова.

```
>(- 10 5)     ; вызов системно определенной функции вычитания
 5            ; вычисленное значение
>             ; приглашение к вводу следующей команды
;  символ комментария
>(2 3 4)
error: bad function - 2
;  2   константа и не может использоваться в качестве имени функции
1)>           ; приглашение к вводу после первой ошибки.
```

Если первым элементом списка введен символ, который может быть использован в качестве имени функции, но с ним не связано каких-либо действий (функция не определена), то диалог с интерпретатором может быть следующий.

```
>(a 3 5)
error: unbound function - A
```

В вызов функции может входит несколько вложенных вызовов. В этом случае порядок вычисления определяется принятой в конкретном языке программирования стратегией вычисления. В Коммон Лиспе (и XLISP соответственно) сначала вычисляются выражения, являющиеся аргументами внешнего вызова, например в выражении

(F0 (f1 arg11 arg12) (f2 arg21 arg22))

Сначала будут определены значения вызовов (f1 arg11 arg12) и (f2 arg21 arg22)), результаты которых будут аргументами функции F0. Следует отметить, что такой порядок вычислений определяется понятием «энергичная» стратегия вычисления (на каждом шаге вычисляется все то, что может быть вычислено на этом шаге). Ранее было показано, что в зависимости от вычислительного окружения, одно и то же *S-выражение* может быть определено интерпретатором как вызов функции (и вычислено), и как данные, например:

```
>(* 2 3)
6                    ; вычислено значение функции
>(quote (* 2 3)) ; форма quote заблокировала вычисления
(* 2 3)              ; возвращено не вычисленное выражение
>’(* 2 3)
; апостроф перед выражением заменяет форму quote
(* 2 3)
 ; возвращено не вычисленное выражение
```

Таким образом, *quote* можно рассматривать как функцию с одним аргументом, причем эта функция возвращает этот аргумент. Интерпретатор, считывая начинающееся с апострофа выражение, автоматически преобразует его в вызов функции quote.

```
>’(a ‘(b c))
(a (quote (b c)))
```

Символ будет представлять самого себя, если он предварен апострофом. Если этого не сделать, то символ будет воспринят интерпретатором как переменная, и интерпретатор будет пытаться определить значение переменной.

```
>a
error: unbound variable - A  ; несвязанная переменная -A
1)>’a
A                                ; символ A
```

Константы апострофом не предваряются. Хотя и ошибки в этом не будет.

```
>t
T
>nil
NIL
>’t
t
>’nil
nil
```

3.3 Типизация в Лиспе

Типы данных в Лиспе сопровождают сами объекты программы и не связаны с именами объектов данных. Следовательно, переменные в разные моменты времени могут представлять разные типы данных.

Переменная в языке Lisp может представлять одновременно разные объекты программы.

Это стало возможным благодаря тому, что тип данных в Лиспе определяется во время выполнения программы (динамическая проверка типов, гл. 2). Поэтому Лисп определяют как бестиповый язык программирования, что ни в коем случае не означает, что на Лиспе нет системы типов данных. При записывании на Лиспе арифметических выражений следует иметь в виду, что тип результата арифметического выражения определяется типами его аргументов.

```
>(/ 20 3)
6
>(/ 20.0 3.0)
6.66667
```

В первом случае применение функции деления с целыми аргументами привело к целочисленному делению (аналог операции DIV в процедурных языках), во втором случае − деление с вещественными аргументами. Если попытаться применить вещественную функцию к целым аргументам, то будет выдано сообщение об ошибке.

```
>(sqrt 9)
error: bad integer operation
>(sqrt 9.0)
3
>(/ 5 3)
1
>(/ 5.0 3.0)
1.6666
```

Пример. Вычислить значение выражения $\sqrt{10}$-sin0.57-cos3.14

```
>( - (sqrt 10.0) (sin 0.57) (cos 3.14))
3.62264
```

В большинстве процедурных языков сложение, вычитание, деление, умножение определены как инфиксные операции. В Лиспе аналогичные действия выполняются функциями с переменным числом параметров. Указываются имя функции и аргументы, как элементы списка. В последующих главах будет показано, как символ может одновременно представлять (быть связанным) и скалярный объект, и функцию, и список свойств. Побочным эффектом применения динамического определения типов и позднего связывания является необходимость вводить проверку типов на этапе вычисления. С этой целью в бестиповые языки программирования введены функции для проверки типов на этапе вычислений.

3.4. Система примитивов языка Лисп и статические связи

Любой функциональный язык программирования включает следующие элементы:

- классы констант, которыми могут оперировать функции;
- набор базовых функций (примитивов), которые используются без предварительного определения;
- правила построения новых функций на основе базовых.

Первый диалект Лиспа предназначался только для обработки списков и включал только пять примитивов. Вот они:

- CAR−выделение головы списка;
- CDR−выделение хвоста списка;
- CONS−создание списка из головы и хвоста;
- ATOM−проверка (является ли S−выражение атомом)
- EQ−проверка (эквивалентности двух символов).

Четвертый и пятый примитивы являются предикатами.

Предикат−это функция, которая проверяет некоторое условие, являющееся его аргументом, и возвращает в качестве значения либо NIL, либо произвольное значение, отличное от NIL (например, T).

Функция CAR возвращает в качестве значения голову списка. Аргументом функции должен быть список. Возвращает функция в общем случае S-выражение.

(CAR список) -> S-выражение.

```
>(CAR ‘(1 2 3))
1
>( CAR ‘(((1)) 2 3))
((1))
>( CAR ‘a)
error: bad argument type − A
; атом a не является списком.
>(car ‘())
NIL
; голова пустого списка − пустой список
>(car ‘(nil))
NIL
; голова списка, состоящего из атома nil − пустой список
>(car ‘(car cdr cons))
CAR
; атом car является головой списка
```

Функция CDR возвращает хвостовую часть списка. Аргументом функции должен быть список, возвращает функция также список.

(CDR список) -> список.

```
>(cdr ‘(1 2 3))
(2 3)
>(cdr ‘((a) ((s d)) (d) f))
(((S D)) (D) F)
>(cdr ‘(1))
NIL
```

Хвостом списка, состоящего из одного элемента является пустой список

```
>(cdr ‘())
NIL
;Хвостом пустого списка является также пустой список.
>(cdr ‘a)
error: bad argument type – A
; Функция CDR определена только для списков
```

Функция CONS формирует список из двух аргументов. Первый аргумент – S-выражение. В результирующем списке он станет головой списка. Второй аргумент должен быть списком. Он будет хвостом результирующего списка.

(CONS S- выражение список) -> список

```
>(cons ‘x ‘(y z))
(X Y Z)
>(cons ‘(1 2 3) ‘(4 5 6))
((1 2 3) 4 5 6)
>(cons (/ 10.0 3) ‘(= 10 / 3))
(3.333333 = 10 / 3)
; первый аргумент − без блокировки вычислений
>(cons ‘(1 2 3) nil)
((1 2 3))
>(cons ‘() ‘(1 2 3))
(NIL 1 2 3)
; атом NIL − голова списка
>(cons 1 nil)
(1)
; список из одного атома
>(cons nil nil)
(NIL)
; список из атома NIL
```

Функции CAR, CDR называются *селекторами*, так как они разделяют список, а функция CONS соответственно называется *конструктором.*

Селекторы CAR,CDR можно комбинировать для выделения произвольного элемента списка, например для выделения элемента X из списка ((1 2) ((X)) y) можно написать вызов

```
>(car (car (car (cdr ‘((1 2) ((X)) y)))))
X
```

или комбинацию

```
>(caaadr ‘((1 2) ((X)) y))
X
```

Здесь комбинация вызовов заменяется вызовом одной функции, которая в общем виде выглядит следующим образом:

(C!!!!R список)

Вместо знаков ! необходимо подставить «a» или «d», как заменители соответствующих вызовов CAR или CDR.

Ранее было показано, что тип объекта аргумента вызова функции в Лиспе определяется на стадии выполнения программы. Это является характерной чертой языков программирования с динамической проверкой типов. С одной стороны, такой подход соответствует декларативной парадигме программирования, но требуется проверка типа аргумента на этапе исполнения программы. Если тип аргумента вызова не соответствует требованиям функции, то выдается сообщение об ошибке. В связи с этим необходимо опознавать тип символьного выражения.

Перед применением селекторов CAR и CDR может понадобиться определить тип аргумента. Для этого используется предикат ATOM.

```
(atom S−выражение)
>(atom 1)
T
>(atom ‘(1))
NIL
>(atom (/ 10.0 3))
t
```

Значение арифметического выражения является атомом.

Предикат EQ сравнивает два символа и возвращает в случае успеха T.

```
>(eq ‘q ‘q)
t
>(eq () nil)
t
Этот предикат не используется для сравнения чисел и списков
>(eq 2.0 2.0)
NIL
>(eq ‘(1 2) ‘(1 2))
NIL
```

NULL − проверяет, является ли аргумент пустым списком.

```
>(null ())
T
>(null nil)
T
>(null ‘(1))
NIL
>(null (cdr ‘(1)))
T
```

EQL−помимо возможностей EQ, позволяет сравнивать однотипные числа. Его не следует применять для сравнения списков.

```
>(eql 2.0 2.0)
T
>(eql 2.0 2)
NIL
```

Предикатом EQUAL, кроме того, можно пользоваться для сравнения списков.

```
 >(equal ‘equal ‘equal)
 T
 >(equal ‘(1 2 3) ‘(1 2 3))
 T
 ; числа следует проверять с использованием предиката =
 >(= 2 2.0)
 T
```

Предикат NUMBERP проверяет, является ли его аргумент числом.

```
>(numberp 2)
T
>(numberp ‘a)
NIL
>(numberp ‘(1 2))
```

NIL

Помимо этих примитивов, в Лисп-систему включается определенное число встроенных функций (от нескольких десятков в XLISP до тысяч в Коммон Лиспе или Интерлиспе). Эти функции облегчают программирование в Лиспе. Вот некоторые из них.

Функция NTH выделяет n−й элемент списка.

```
>(nth 2 ‘(0 1 2 3))
2
```

Функция LAST выделяет последний элемент списка.

```
>(last ‘(1 2 3))
3
```

Функция LIST создает список из элементов.

```
>(list 1 2 3)
(1 2 3)
```

Функция APPEND предназначена для слияния списков.

```
>(append ‘(1 2 3) ‘(4 5 6))
(1 2 3 4 5 6)
```

Статические связи в Лиспе позволяют реализовать императивный подход программирования. Символ может выступать в роли переменной, т. е. представлять другой объект. Так, если символ X связан с объектом Y, то интерпретатор на запрос

```
>X  ; выдаст ответ
Y   ; Y−значение X
```

Изначально (при загрузке интерпретатора) у символов никакого значения нет. Связывание символа со значением возможно при помощи функции SET.

```
>(SET NAME VALUE)
```

Функция **SET** вычисляет значение символа NAME и связывает его со значением VALUE. После связывания на значение символа можно сослаться, записав его без апострофа.

```
>(set ‘x ‘y)
Y
  ; возвращенное функцией значение
>x
Y
 ; значение символа x
>(set (car ‘(z c)) 2)
 ;  с головой списка−символом z связывается константа 2
2
>z
2
>(set (car ‘(a s d)) (car ‘(a q z)))
;  с головой первого списка связывается голова второго
A
>a
;  значением символа может быть сам этот символ
A
```

Установленные связи действуют до окончания сеанса работы или до нового переопределения связей. Наряду с функцией SET**,** в Лиспе определена функция SETQ, которая применяется аналогично SET**,** но не вычисляет значение своего первого аргумента. Следовательно, при связывании символа отпадает необходимость в блокировке его вычисления.

```
>(setq w 2)
2
>w
2
```

Наряду с этими функциями, в интерпретаторе XLISP определен предикат **BOUNDP** для проверки факта связывания символа, являющегося его аргументом. Он истинен, если атом имеет значение.

```
>(boundp ‘w)
```

T

Функции SET и SETQ обладают побочным эффектом, состоящим в установлении связи между символом и значением. В гл. 1 было декларировано, что функциональное программирование − это программирование без побочных эффектов. Таким образом, рассмотренные только что функции не совсем чистые с точки зрения функциональной парадигмы, поэтому их принято называть *псевдофункциями*. В чистом подмножестве Лиспа Scheme они отсутствуют.

Функция EVAL является *интерпретатором* Лиспа (в гл. 8 будет показано, что эта функция используется вместе с функцией APPLY)**.** Она может вычислить любое правильно построенное Лисповское выражение. Вызов этой функции неявно присутствует в диалоге с интерпретатором (диалог с интерпретатором – это диалог с этой функцией). Нередко становится необходимым вызов этой функции в диалоге с Лисп-системой, например для снятия эффекта блокировки вычислений.

```
>(setq w ‘(car ‘(a b c)))
(CAR (QUOTE (A B C)))
>(eval w)
A
>(setq q ‘(eval w))
(EVAL W)
>(eval q)
A
```

Таким образом, функции EVAL и QUOTE отменяют действие друг друга. QUOTE − отмена вычисления, EVAL − отмена отмены вычисления.

Показанные функции SET и SETQ могут применяться в императивном программировании на Лиспе. Необходимость в таких решениях может определяться требованием увеличения вычислительной эффективности программы, например для организации итерационных вычислений (гл. 6,7). Поэтому императивные механизмы включаются в большинство языков программирования или диалектов. Возможности применения в практическом программировании только «чистых», лишенных

императивных включений языков невелики: такая программа при прочих равных условиях будет менее эффективна по критерию «время выполнения» и потребляет больше оперативной памяти (п. 4.1). Кроме того, следует учесть, что с помощью псевдофункций реализуются ввод и вывод во внешний поток, определение функций (гл. 5,6,).

3.5. Символьные выражения Эрланга

Объекты, которыми манипулируют программы, будем называть символьными выражениями (Expressions), по аналоги с понятиями Лиспа. Ниже перечислен список нескольких, наиболее значимых с точки зрения функционального программирования, позиций в структуре символьных выражений:

- вычислимые выражения (Expression Evaluation);
- термы (Terms);
- переменные (Variables);
- образцы (Patterns);
- сопоставления (Match);
- вызовы функций (Function Calls).

В основе типов данных Эрланга лежит понятие терма. Эти выражения можно разделить на скалярные объекты (числа, атомы, логические значения) и структурные объекты (кортежи, списки, записи). К термам также относятся:

функциональные объекты (fun);

идентификатор порта (Port Identifier);

идентификатор процесса (Pid).

Атомы записываются как последовательность букв, цифр и знаков подчёркивания, начинающейся со строчной буквы, либо как последовательность произвольных символов, заключенных в кавычки (эти же правила применяются для имён функций). **В отличие от Лиспа атомы не используются для именования других объектов.**

Числа подразделяются на два типа: целые и вещественные. Числовые константы записываются традиционно, за исключением нескольких особенностей:

- запись $<символ> имеет значением код символа (`$X` = 88);

- запись <основание>#<значение> используется для целых недесятичных чисел по основанию от двух до шестнадцати (16#FFFF = 65535).

Логические значения представлены атомами `true` и `false`, которые могут применяться как обычные атомы. Определены стандартные логические функции not, and, or и xor. Инструментальная среда Эрланга[3] содержит интерпретатор, позволяющий вычислять значения выражений, синтаксически близких обычной математической записи. В конце каждого законченного выражения ставиться точка. Следующий пример иллюстрирует основные особенности записи чисел и логических значений в Эрланге.

```
1> $A.

65

2> 16#a1.

161

3> true>false.

true

4> true and false.

false

5> true or false.

true

6> 'то же атом'.

'то же атом'
```

Переменные предназначены для именования других объектов. Они записываются в виде последовательности букв, цифр и знаков подчёркивания, обязательно начинающейся с прописной буквы или знака подчёркивания. Значение переменной может быть присвоено только однажды (запрет разрушающего присваивания). Переменная,

которой не присвоено значение называется *несвязанной* и попытка обратиться к ней приводит к ошибке. Переменная в Эрланге замещает имя объекта, а не указывает на место в памяти компьютера, что полностью сответсвует требованиям функционального стиля и декларативной парадигме программирования.

```
1> X=5.

5

2> X.

5

3> X=6.

ПОПЫТКА РАЗРУШАЮЩЕГО ПРИСВАИВАНИЯ

=ERROR REPORT==== 23-Nov-2004::09:47:38 ===

Error in process <0.29.0> with exit value: {{badmatch,6},[{erl_eval,expr,3}]}

** exited: {{badmatch,6},[{erl_eval,expr,3}]} **

4> Y.

ПОПЫТКА ОБРАЩЕНИЯ К НЕСВЯЗАННОЙ ПЕРЕМЕННОЙ

** 1: variable 'Y' is unbound **
```

Форма записи списков в языке Эрланг практически полностью заимствована из языка Пролог. Конструктор списков имеет вид [<голова>|<хвост>]. Пустой список обозначается []. Стандартная форма записи списка $[e_1|[e_2|...[e_n|[]]...]]$ в большинстве случаев заменяется более удобной формой $[e_1,e_2,...e_n]$.

```
1> [].

[]

2> X=[e1|[e2|[e3|[]]]].

[e1,e2,e3]
```

Если все элементы списка будут представлены целыми положительными числами, то такой список может быть воспринят как строка.

```
1> [65,66,67,68,69].

"ABCDE"

2> [65,66,67,atom,69].

[65,66,67,atom,69]

3> [65,66,67,68,69,253,254,255].

"ABCDEэюя"
```

Такая особенность Эрланга позволяет не рассматривать строки как еще один тип данных и соответственно применять для их обработки функции для работы со списками.

В особый тип данных выделены кортежи[4] (tuples), которые можно определить как единый упорядоченный набор объектов, доступ к которым осуществляется указанием их номера (индекса), поэтому аналогом кортежа в языке Эрланг является массив, такой тип данных следует считать императивным включением. Кортежи записываются как последовательности термов, разделённые запятыми и заключенные в фигурные скобки. Возможен кортеж, не содержащий ни одного элемента.

```
1> {1,2}.

{1,2}

2> X={1,2,3}.

{1,2,3}

3> X.

{1,2,3}
```

Размер кортежа

4> size(X).

3

Извлечение элемента с номером 2 их кортежа X

5> element(2,X).

2

6> Z={}.

{}

7> Z.

{}

3.6. Базовые функции Эрланга и основные операции

Синтаксис выражений соответствует обычной математической записи, основная форма записи инфиксная. Каждое введённое выражение должно завершаться точкой. Базовые (аналоги примитивов Лиспа) и библиотечные функции Эрланга определены в модулях (п. 5.5). Вызов функции в интерпретаторе в общем случае имеет вид

имя модуля **:** имя функции (аргумент1, аргумент2 …)**.**

Для базовых функций имя модуля erlang, которое можно не указывать.

Базовая функция модуль

1> abs(-5).

5

Базовая функция – голова списка

2> hd([1,2,3,4]).

1

Базовая функция – голова списка

3> erlang:hd([1,2,3,4]).

1

В модулях описаны функции, имеющие одинаковую тематическую направленность. Так, в модуле math математические функции, в модуле string – функции для работы со строками.

1> math:sin(3.14).

1.59265e-3

2> math:cos(3.14).

-0.999999

3> string:len("12345").

5

7> math:sqrt(10.0)-math:sin(0.57)-math:cos(3.14).

3.62264

Значения выражений вычисляются в соответствии с видом вычислений энергичных языков.

Одним из основных отличий Эрланга от большинства языков программирования (кроме языка логического программирования Пролог) является *операция* ***сопоставления с образцом***[5] (Pattern Matching).

Образцом считается любой структурный терм, содержащий несвязанную переменную.

Смысл операции сопоставления X=Y в следующем:

- Если левая часть выражения X представляет свободной переменной, то она получает значение правой части Y (действие аналогичное присваиванию);
- Если на месте левой части X находится образец, то сравнивается структура со структурой терма правой части Y и при совпадении

структур происходит связывание всех переменных образца с соответствующими элементами терма.

В качестве переменных образца, могут выступать *анонимные переменные.* Анонимные переменные сопоставляются с любым термом:

```
1> A=5.

5

2> A.

5

3> [H|T]=[1,2,3,4,5].

[1,2,3,4,5]

4> H.

1

5> T.

[2,3,4,5]

6> [1,H2|T2]=[1,2,3,4,5,6].

[1,2,3,4,5,6]

7> H2.

2

8> T2.

[3,4,5,6]
```

Образец с анонимной переменной

```
9> [H3|_]=[a,s,d,f].

[a,s,d,f]
```

10> H3.

a

Образец списка из двух элементов

11> [H4,H5]=[z,x].

[z,x]

12> H4.

z

Между термами в Эрланге введены отношения отличия, определяющие следующий порядок:
число < атом < функциональный объект < идентификатор порта < идентификатор процесса < кортежа < списка.

1> 4==4.0.
2> True
3> 2> 4=:=4.0.
4> false
3> 4/=5.0.
true
4> 4/=4.0.
false

Сравнение термов (Term Comparisons) реализуется следующим набором операций:

Знак операции	Описание операции
==	Равный
/=	Не равный
=<	Меньше или равный
<	Меньше
>=	Больше или разный
>	Больше чем
=:=	В точности равный (только сравнение однотипных)
=/=	В точности не равный (только сравнение однотипных)

Для обработки списков в Эрланге существуют функции, аналогичные лисповским селекторам car и cdr:

hd(список) – терм;

tl(список) – список.

```
1> X=hd([1,2,3,4]).
1
2> Y=tl([1,2,3,4]).
[2,3,4]
3> Z=[a,[5,g,s],[[[f]]]].
[a,[5,g,s],[[[f]]]]
4> G=hd(tl(hd(tl(Z)))).
g
```

В отличие от Лиспа в Эрланге ряд действий можно выполнять с помощью операций. Кроме перечисленных базовых функций работы со списками, в Эрланге есть ряд списочных операций:

- список ++ список = список – конкатенация списков (аналог функции append);
- список -- список = список – деконкатенация списков.

```
1> X=[1,2,3,4].
[1,2,3,4]
2> Y=[5,6,7].
[5,6,7]
3> Z=X++Y.
[1,2,3,4,5,6,7]
4> Z.
[1,2,3,4,5,6,7]
5> V=[6,7].
[6,7]
6> C=Z--V.
[1,2,3,4,5]
```

Примечания

1. Легковесный процесс (lightweight process). Еще это же понятие

обозначается как поток или нить (thread). В отечественной литературе встречается понятие «подпроцесс», которое в дальнейшем и будет использоваться в данной работе. Подпроцессы применяются для распараллеливания программ, для них не создается отдельное адресное пространство, но которые на многопроцессорных системах также распределяются по процессорам. Создание подпроцессов и управление ими – одна из функций операционной системы, но в языке Эрланг, а примерно через 10 лет и в языке Java введены средства для выполнения этих действий. В языке программирования С возможно прямое использование этого механизма для распараллеливания программ, но посредством вызова соответствующих системных функций.

2. Дж. Маккарти (J. McCarthy) − один из основателей современного программирования. В 1958−1961 гг., будучи профессором Массачусетского технологического института, разработал язык Лисп на основе алгебры списочных структур, лямбда-исчисления и теории рекурсивных функций.
3. Erlang/OTP доступен бесплатно в виде open source (www.erlang.org). OTP(Open Telecom Platform) представляет собой набор библиотек, содержащих решения многих типичных сетевых и телекоммуникационных задач. Erlang/OTP распространяется под собственной лицензией EPL (Erlang Public License).
4. В кортеже существенны не только элементы, но и порядок, в котором они располагаются. В математике кортежем является вектор, заданный проекциями на оси. Кортеж заключается в угловые скобки: $< a_1 ,a_2, a_3, ..., a_n >$ кортеж длиной n или упорядоченная *n-ка*. Кортежи рассматриваются как тип данных еще в нескольких функциональных языка. Пример – академический язык Hope /9/. Однако там кортеж рассматривается как набор связанных величин или объектов, и в этом смысле он более походит на список.
5. Помимо языка Пролог операция сопоставления с образцом включена в состав основных механизмов языка Рефал, история которого начинается с 1968 г.

Резюме

Язык программирования Лисп, являясь исторически первым функциональным языком программирования, имеет следующие отличительные черты:

- динамическое определение типов на этапе исполнения программы;
- основной объект программы – символьные выражения;
- к символьным выражениям относятся атомы (символы, константы) и списки (определения функций, вызовы функций);

- основная структура в чистом языке – список, с помощью которого представляются и программы, и данные;
- к примитивам относятся функции для работы со списками и функции для проверки типа символьного выражения;
- включение в функциональный язык программирования конструкций для императивного программирования является средством увеличения эффективности программы;
- при решении практических задач программисту всегда приходится сталкиваться с дилеммой «выразительность – эффективность».

Основные отличия языка Эрланг заключаются в следующем:

- переменная относиться к символьным выражениям;
- значение переменной может быть присвоено только один раз;
- в основе структуры типов данных лежит понятие терма;
- между типами данных существует отношение равенства;
- введены понятия образца и механизм сопоставление с образцом.

Упражнения

1. Вычислите следующие функции с помощью интерпретаторов XLisp и Erlang.

```
5.64*(4.643-3.64)/(7.8/(4.6*(56-36+34)))
sqrt(abs(sin(6.67/(sqrt(5.68/3.21)))))
 (467-111-234)*((56+23)/(34+32+75))
tan(exp(54/32)+1./(sqrt(abs(cos(56/78)))))
 (77.77-56.33+64.22/56.88)/(54*(67-43))
 (exp(44/(ctg(55/88))))*(sqrt(abs(67/(45-88))))
```

2. Выделите с помощью комбинации вызовов списочных примитивов элементы x, y, z из следующих списков, а также все элементы, символы которых состоят из двух знаков.

```
((a s d (f  x (g y (((z))) u) (t tr tt ) (yq (rq (rn (((gd)))))))))
(a d (f (h (x ((((y)))) (z (uu (jj))) ff ) hh)))
(((vv) (((tt))) gg (c (x (y (((z)))))))
((a f ((y)) u ((x))) (i (f (uu (us ((hd)) z)))))
```

(a (w ((uu ((y)) (((x)) ((z))) (f us)) (g uf))))
(((f x (y (((z))) u) (((tr)) t) ((wq (n (((gd)))))))))

3. Запишите списки, приведенные в предыдущем задании в нотации языка Erlang, и выполните выделение соответствующих элементов.

4. Приведенный ниже список является определением функции

(defun defun (defun)
(* defun defun)
)

Установлены следующие статические связи:
(setq z1 (car ‘defun))
(setq z2 (car (cdr ‘defun)))
(setq z3 (car (cddr ‘defun)))
(setq z4 (caar (cddr ‘defun)))
Что будет значением следующих запросов к интерпретатору Лиспа ?
>z1
>z2
>z3
>z4
Проверьте свой ответ с помощью интерпретатора.
5. Установлена статическая связь
(setq defun 'defun)
Что будет значением следующего вызова?
>(defun 3)

6. Проверьте с помощью интерпретатора языка Лиспа результаты выполнения следующих запросов:

>(eq ‘(x y z) ‘(x y z))
>(equal ‘(x y z) ‘(x y z))
Попытайтесь найти объяснение полученным результатам. Подробный разбор полученного ответа можно найти в следующей главе.

ГЛАВА 4

ОРГАНИЗАЦИЯ ПАМЯТИ ФУНКЦИОНАЛЬНЫХ ЯЗЫКОВ

Неразрушающее функциональное программирование
Разрушающее императивное программирование
Автоматическая сборка мусора
Полиморфные символы Лиспа

При функциональном программировании не требуется явное управление памятью (ее выделение и освобождение). Кроме того, языки программирования с динамическим определением типа, в том числе Лисп (п. 3.4) позволяют не связывать явно имя объекта программы с конкретным типом данных [11,13]. Это обеспечивается мощным, имеющим принципиальные отличия от императивных языков программирования механизмом управления памятью функциональных языков.

Именно благодаря знанию основных принципиальных отличий этого механизма обеспечивается успешное программирование на Лиспе в функциональном стиле [6,12,17,22,23]. В учебном пособии в качестве иллюстративного материала используются программы на языке Common Lisp, содержащем императивные включения, хотя существует чистая версия Лиспа (Sheme) Такой выбор обоснован практичностью языка программирования. Примеры программирования в нефункциональном стиле показаны для иллюстрирования определенных свойств и механизмов функциональных языков, таких как принципы организации памяти (п. 4.1, 4.2.), особенности функциональной обработки данных (п. 4.2.), автоматическая сборка мусора (п. 4.3.).

4.1. Неразрушающее функциональное программирование

Одним из значимых положений функционального программирования является отказ от структуроразрушающего

присваивания. В функциональных языках структуры не разрушаются, а только создаются путем анализа и копирования из существующих структур и атомов. При этом старые структуры (на основе которых были созданы новые) также остаются в памяти. В связи с этим значимым является организация памяти функциональных языков. Укажем основные принципы этой организации на логическом уровне:

- в системе существует только один атом с конкретным именем;
- все введенные в систему имена остаются до конца сеанса (если их специально оттуда не убрать);
- с каждым именем связан список системных свойств;
- списки создаются на основе так называемых «*списочных ячеек*»;
- каждая списочная ячейка состоит из двух полей.

В левом поле списочной ячейки находится указатель на объект, являющийся головой списка, а в правом поле – указатель на хвост списка. Указатель (ссылка) на первый элемент списка является *указателем на список*. Таким образом, список – это цепочка из списочных ячеек, связи которой представляются ссылками из поля CDR левой ячейки на поле CAR правой. В полях CAR записаны ссылки на элементы списка. В поле CDR последней ячейки записан указатель на NIL.

Функция CONS создает списочную ячейку и записывает в соответствующие поля указатели на голову и хвост списка.

```
>(setq X (cons 'a NIL))
(A)
```

Добавление еще одного атома A1 в список приведет к образованию еще одного списка.

```
>(setq Y (cons ‘a1 x))
(A1 A)
```

Старый список (связанный с атомом X) остается в памяти. Исходя из рассмотренных принципов организации памяти функциональных языков программирования, можно ввести понятие «*физическое равенство* структур» и «*логическое равенство»*. Под физическим равенством будет пониматься, являются ли **сравниваемые структуры одними и теми же объектами (состоят одних и тех же списочных ячеек). Логическое равенство**

предполагает сравнение структуры списков и объединяемых в список объектов.

```
>(setq w (cons ‘A1 ‘(A)))
(A1 A)
>(eq Y W)
NIL
>(eq (cdr Y) (cdr W))
NIL
```

Предикат EQ проверяет соблюдение физического равенства объектов. Следовательно, внешне одинаковые списки (A1 A) оказались физически разными объектами, т. е. состоящими из разных списочных ячеек. Селекторы CAR и CDR выбирают и возвращают значение соответственно из левого и правого полей списочной ячейки. Список, связанный с переменной X, входит в состав списка Y. Объединение списков функцией CONS не меняет структуру списков. Результатом объединения будет список ((A1 A) A1 A).

```
>(setq Q (cons W Y))
```

Предикат EQ определяет, являются ли его аргументы физически одними и теми же объектами (атомами и списочными ячейками).

```
>(eq (cdr q) y)
T
```

Он проверяет совпадение физических указателей. В отличие от него предикат EQUAL сопоставляет совпадение структурного построения списков и совпадение атомов, входящих в список.

```
>(eq ‘(x y z) ‘(x y z))       ; два физически разных списка
NIL
>(equal ‘(x y z) ‘(x y z))   ; логически идентичные списки
T
```

Логически идентичные (равенство в смысле EQUAL) списки могут быть физически разными (равенство в смысле EQ). В некотором смысле можно утверждать, что логическое равенство имеет значение, прежде всего, в функциональном программировании

(где структуры анализируются и копируются), а физическое равенство – в императивном (где наблюдается прямое манипулирование памятью).

Наряду со списками в Лиспе существует и другая структура, которая является *точечной парой* (dotted pair). Точечная пара получается, если вторым аргументом функции CONS будет атом (в общем случае − элемент, отличный от списка)

```
>(cons ‘a ‘d)
(A . D)
```

Поле CDR списочной ячейки может ссылаться не только на список (в результате формируется список) или атом (получается точечная пара), но и на другую точечную пару. При этом создается структура, символьное выражение которой записывается как (X Y (Z .W)). В частном случае, если вместо W стоит NIL, то получатся обычный список. Когда поле CDR точечной ссылается на список или точечную пару, возможно полное или частичное приведение к списочной нотации.

```
>(setq W ‘(X Y (Z . NIL)))
(X Y (Z))
```

В Эрланге также возможна структура, аналогичная лисповской точечной паре. Так, если в конструкторе списков вместо второго аргумента поставить не списочный элемент.

```
; точечная пара
1> X=[1|2].
[1|2]
; список
2> Y=[1|[2]].
[1,2]
```

Использование точечных пар в практическом программирование очень редкий случай. Вся работа со списками базируется на обработке списков в основном с применением рекурсии (гл. 7).

4.2. Разрушающее императивное программирование

Во многих диалектах Лиспа существуют функции, которые позволяют изменить содержимое конкретного поля списочной ячейки (переписать указатель). Такими функциями в диалекте XLISP являются:

- *(RPLACA X Y)* −меняет указатель в поле CAR головы списка, связанного с атомом X на значение объекта Y.
- *(RPLACD X Y)* −меняет указатель в поле CDR головы списка, связанного с атомом X на значение объекта Y.

```
>(setq W ‘(A S D))
(A S D)
>(rplaca W ‘Z)
(Z S D)
>w
(Z S D)

>(rplaca a ‘(1 2 3))
((1 2 3) W E)
>(setq x ‘(1 2 3 4))
(1 2 3 4)
>(rplacd x ‘(5 6 7))
>(1 5 6 7)
```

Отметим, что помимо псевдофункций SET и SETQ, позволяющих связать символ со значением, в XLISP существует обобщенная функция присваивания SETF, которая записывает в конкретную ячейку памяти соответствующее значение.

```
>(setq w ‘(a s d))
(A S D)
>(setf (car w) ‘z) ; записать указатель на Z, в ячейку памяти, указывающую на голову списка W
Z
>w
(Z S D)
```

В XLISP существует структуроразрушающая псевдофункция − конструктор NCONC.

(NCONC LIST1 LIST2)

Она позволяет объединить два списка: LIST1 и LIST2, минуя процедуру копирования исходных списков. В данном случае происходит переписывание указателя в поле CDR последней списочной ячейки списка LIST1 со значения NIL на первую списочную ячейку списка LIST2.

```
>(setq x '(a s d))
(A S D)
>(setq y '(z c v))
(Z C V)
>(setq w (nconc x y))
(A S D Z C V)
>w
(A S D Z C V)
```

Наряду с созданием списка W (что и предполагалось при вызове NCONC) произошло изменение значения символа Y(что не предусматривалось).

```
>y
(A S D Z C V)
```

Применение структуроразрушающих псевдофункций является, несомненно, нефункциональным приемом программирования, нарушающим декларативный смысл программы и позволяющим увеличить вычислительную эффективность программы.

4.3. Автоматическая сборка мусора

Функциональные языки программирования являются языками сверхвысокого уровня (по отношению к императивным языкам высокого уровня типа Паскаль, СИ). Применительно к теме данной главы это означает, что в функциональных языках время жизни объектов несущественно с точки зрения программиста, данные описываются на абстрактном уровне. Программист освобожден от необходимости планировать размещение в памяти компьютера программ и данных. Соответственно все эти мероприятия по управлению памятью перекладываются на систему управления

памятью. Поскольку память для новых элементов выделяется автоматически, то неиспользуемые элементы должны автоматически удаляться из памяти. Так, результирующий список (Z S D) в примере из п.3.2 можно получить и другим образом:

```
>(setq w ‘(a s d))
(A S D)
>(setq w (cons ‘z (cdr w)))
(Z S D)
```

После внесений соответствующих изменений появилась списочная ячейка (ссылающаяся на атом A), не включенная в новую структуру. Эта списочная ячейка продолжает занимать место в памяти, но она будет недоступна для программиста, так как ссылки на нее утеряны. Такие списочные ячейки называются *мусором*.

Одной из основных (не присущих императивным языкам) задач для системы управления памятью функциональных языков является удаление таких ячеек. Процесс поиска и удаления неиспользуемых ячеек называется ***сборкой мусора***[1]. При выполнении функциональной программы непрерывно генерируются структуры данных.

Генерируемые структуры состоят из набора отдельных ячеек. В зависимости от реализации конкретного языка программирования ячейки могут быть фиксированного и переменного размеров.

Каждый из этих способов организации памяти имеет свои преимущества и недостатки. Преимущества фиксированного размера организации области памяти заключаются в простоте управления, из-за того, что границы ячеек можно легко определить. При фиксированном размере ячеек памяти все структуры должны быть представлены списками, что характерно для Лиспа. Система управления памятью, основанной на ячейках переменного размера, более сложная, но она позволяет сделать постоянным время доступа к элементам из-за использования обычных методов индексации.

Первой задачей управления памятью будет нахождение свободного пространства в куче для размещения новых ячеек. При любом способе организации кучи эта задача может быть решена либо с помощью так называемого свободного списка, либо стека. Свободный список является списком пустых ячеек. На начало свободного списка указывает регистр свободного списка. Ячейки в

свободном списке связаны стандартным для всех списков способом − с помощью хвостового слова каждой ячейки.

Организация области памяти программы с переменным размером ячейки требует создания нескольких свободных списков − по одному для каждого размера ячейки, и соответственно несколько регистров свободного списка. Когда программе требуется свободная ячейка, то выбирается первая из списка свободных ячеек. При этом изменяется значение регистра свободных ячеек так, чтобы он по-прежнему указывал на первую свободную ячейку.

Все ячейки кучи принадлежат к одной из следующих категорий:
- рабочие ячейки (занятые данными):
- свободные ячейки;
- мусорные ячейки.

Сборщик мусора предназначен для обнаружения мусорных ячеек и перевод их в категорию свободных ячеек. Сборщики мусора по типу организации рабочего процесса относятся к двум основным категориям. Первая категория − *старт/стоп* сборщик. Этот тип сборщиков мусора требует приостановки вычислительного процесса пользователя для очистки памяти. Второй тип позволяет вести вычислительные процессы пользователя и сборки мусора параллельно - *сборщик реального времени*.

Такой тип сборщика мусора встроен в систему управления памятью Эрланга, что объясняется ориентированность этого языка программирования на телекоммуникационные приложения, которые в большинстве своем являются системами реального времени. Кроме того, система управления памятью Эрланга не предоставляет в распоряжение разработчика средства разрушающего программирования. Так, попытка простого присваивания новых данных переменной, имеющей значение приведет к ошибке.

```
1> X=5.
5
2> X=2+3.
5
3> X=6.
** exception error: no match of right hand side value 6
```

Несмотря на отсутствие явных императивных механизмов, программа на Эрланге, как и любая функциональная производит

мусор[2]. Так, по завершении рекурсивного процесса все сгененерированные структуры становятся мусором.

4.4. Полиморфные символы Лиспа

Под термином «*полиморфные символы*» будет пониматься способность символа одновременно представлять разные объекты программы. Наиболее распространен **полиморфизм**[3] в объектно-ориентированном программировании, в основном по отношению к соответствующим функциям. В гл. 2 было показано, что символ может представлять значение и быть функцией одновременно. Кроме того, с символом может быть связан особым образом организованный список свойств.

Одним из основных характерных признаков функциональных языков программирования является то, что переменные обозначают не области памяти, а объекты программы. Это полностью соответствует понятию переменной в математике. Следовательно, один и тот же символ можно связать с различными объектами программы (значением, определением функции, и.т.п.). Объекты, с которыми связан символ, будут составлять его *системные свойства*. Наряду с встроенными системными свойствами (т.е. управляющими работой интерпретатора) в Лиспе существует возможность связать с символом список свойств, определенный пользователем. Применение списка свойств дает программисту возможность реализовывать различные формализмы представления знаний в системах искусственного интеллекта.

Свойства символа записываются в хранимый вместе с символом список свойств, который может быть либо пуст, либо содержать произвольное число свойств. В общем виде список свойств строится по следующему принципу.

(свойство1 значение1..свойство_i значение_i..свойство_n значение_n)

Так, с символом volga_24_10 может быть связан следующий список свойств:

(Vmax= 145 Color white)

Для присваивания, чтения и корректировки списка свойств в Лиспе существуют специальные функции[4].

1.Функция чтения списка свойств

(symbol-plist *symbol*)

Здесь *symbol* − символ, список свойств которого необходимо прочитать, например:

```
>(symbol-plist ‘vaz-2101)
NIL
```

Список свойств пуст.

2.Функция присваивания нового свойства или изменения значения существующего свойства.

(putprop *symbol znak svoist)*

Здесь *symbol* − символ, список свойств которого необходимо корректировать;

znak- значение, которое должно быть присвоено свойству *svoist*.

```
>(putprop ‘vaz-2101 140 ‘v=)
140
```

Теперь можно прочитать список свойств

```
>(symbol-plist ‘vaz-2101)
(v= 140)
```

Добавление нового свойства функцией putprop осуществляется с головы списка (т.е. стандартно).

```
>(putprop ‘vaz-2101 5 ‘vmest)
5
>(symbol-plist ‘vaz-2101)
(vmest 5 v= 140)
```

Удаление свойства из списка свойств осуществляется функцией (remprop *symbol svoist*)

```
>(remprop ‘vaz-2101 ‘v=)
NIL
>(symbol-plist ‘vaz-2101)
(vmest 5)
```

Если удаляемое свойство в списке отсутствует, то функция также возвращает NIL. Прочитать значение свойства из списка свойств можно функцией

(get *symbol svoist*)

>(get ‘vaz-2101 ‘vmest)
5

Проиллюстрировать полиморфизм символа с учетом связей со списком свойств, можно следующим примером. Символ «car» становится переменной

```
> (setq car 5)
5
> car
5
```

Проверка связи «car» со спискам свойств

```
> (symbol-plist 'car)
NIL
```

Запись значений в список свойств символа «car»

```
> (putprop 'car 'vaz-2110 'Name=)
VAZ-2110
> (symbol-plist 'car)
(NAME= VAZ-2110)
```

Запись значений в список свойств символа «car»

```
> (putprop 'car 155 'V=)
155
```

Символ «car» представляет список свойств

```
> (symbol-plist 'car)
(V= 155 NAME= VAZ-2110)
```

Символ «car» − имя функции

```
> (car '(1 2 3))
1
Символ «car» − переменная
2> car
5
```

Примечания

1. Среди императивных языков программирования похожий механизм управления памяти у языка Java, он обладает сборщиком мусора, который автоматически определяет, когда объект более не используется, и разрушает его. Сборщик мусора позволяет уменьшить число проблем, которые программист должен отслеживать, и упростить исходный код. Язык Java появился через 30 лет после Лиспа.
2. Следует учитывать структуру мусора. Так, мусором может быть любой структурный объект, но атом остается до конца работы программы. Атом мусором не будет считаться и уборщику недоступен. Поэтому не рекомендуется произвольно использовать функции генерации атомов (например list_to_atom).
3. Слово «полиморфизм» греческого происхождения − много форм. Применялось до программирования в разных областях науки, например в биологии, химии. В программировании полиморфизм применяется в основном к функциям, когда одна и та же функция применяется к аргументам различных типов (есть одна функция (тело кода) и несколько ее интерпретаций). Возможен другой случай, когда имеется множество различных функций с одним именем. Такая ситуация называется перегрузкой, или полиморфизмом вида ad hoc. Между этими крайними случаями находятся переопределяемые и отложенные методы. Рассматриваются также полиморфные переменные, которые содержат значения, относящиеся к различным типам данных. В данном параграфе рассматривается совершенно иной пример полиморфизма, когда символ одновременно представляет совершенно разные объекты программы: является переменной, функцией, представляет список свойств.
4. Функции списка свойств символа предназначались для представления одного из основных формализмов представления знаний – фреймов.

Резюме

- В функциональном программировании новые структуры создаются на основе имеющихся структур, но старые также остаются в памяти в неизменном виде.

- Основной принцип формирования новых структур – анализ и копирование списочных ячеек.
- Структуры могут быть логически идентичными, но физически разными объектами.
- В языке Лисп есть стуктуроразрушающие функции, которые переписывают ссылки.
- Структуроразрушающие функции нарушают функциональность программы, поэтому они могут применяться в программах, ограниченных по затратам памяти и машинного времени.
- Характерные черты организации памяти в функциональных языках – полная автоматизация процессов размещения программ и данных в памяти.
- При выполнении программы появляются мусорные ячейки (занятые, но недоступные для вычислительного процесса пользователя). Для того чтобы сделать их доступными для процессов пользователя, в систему управления памятью встроены автоматические сборщики мусора.
- Символ Лиспа в один и тот же момент времени быть переменной, представлять функцию, иметь не пустой список свойств.
- Язык Эрланг, являясь также как Лисп языком с динамической типизацией, не допускает переопределения значения переменной, а попытка выполнить такое действие приводит к сообщению об ошибке.
- Язык Эрланг является более чистым с функциональной точки зрения языком.

Упражнения

1. В чем состоит различие между функциями SETQ и SETF? Проиллюстрируйте ответ.

2. Что будет значением следующих запросов к интерпретатору Лиспа?

```
>(eq ‘(1 2) ‘(1 2))
>(eql ‘(1 2) ‘(1 2))
```

Почему?

3. Возможна ли ситуация, при которой списки физически идентичные, но логически разные?
4. Какой эффект (помимо экономии памяти) может получиться при использовании точечных пар? Приведите пример.
5. Что с точки зрения языка программирования выполняют селекторы CAR и CDR на физическом уровне?
6. Благодаря каким особенностям Лиспа, реализуется присущий ему полиморфизм символа?
7. Какой сборщик мусора (сборщик старт/стоп или реального времени) используется в предложенной Вам системе программирования?
8. Как можно объяснить необходимость встраивания в функциональный язык программирования сборщиков мусора? К каким основным положениям декларативного программирования восходит эта необходимость?
9. Почему функциональная программа должна генерировать структуры, но не разрушать их? Как это свойство связано с основными принципами функционального и декларативного программирования?

ГЛАВА 5

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

Лямбда-выражения Лиспа
Определение функции в Лиспе
Формы записи функции нескольких переменных
Анонимные функции в Эрланге
Функции и модули Эрланга

Как отмечалось, единственный способ расчленения на части программы в функциональном программировании является введение имени для функции (гл. 1) [12]**.**

В данной главе приводятся понятия, традиционные для функционального программирования и применяемые в платформе .Net: карринг, лямбда-выражение.

5.1. Лямбда-выражения Лиспа

Представление функции в функциональном программировании основано на понятии *лямбда-выражения*, перешедшем во многие функциональные языки из *лямбда-исчисления* Черча (гл. 6). Лямбда-выражение представляет собой определение *анонимной* (безымянной) функции. В Лиспе лямбда-выражение записывается следующим образом:

(lambda (p1 p2 …pn) F)

Здесь

(p1 p2 …pn) – список *формальных параметров* называемый *(лямбда- список);*

F – *тело лямбда–выражения*, которое может быть представлено произвольной формой и значение которой может вычислить интерпретатор Лиспа.

Так, лямбда–выражение, соответствующее определению функции вычисления среднего арифметического значения двух чисел, в Лиспе может быть записано в следующем виде:

(lambda (x y) (/ (+ x y) 2))

Здесь тело лямбда–выражения F= (/ (+ x y) 2).
Применение лямбда–выражения к аргументам (*фактическим параметрам*) осуществляется с помощью вызова лямбда–выражения (*лямбда-вызова*). Лямбда–вызов в Лиспе имеет следующий вид:

((lambda (p1 p2 …pn) F) f1 f2 … fn)

Здесь f1 f2 … fn – фактические параметры, заменяющие соответствующие формальные параметры p1 p2 …pn в теле F лямбда–выражения. Лямбда–вызов, вычисляющий среднее арифметическое чисел 6 и 14, записывается в Лиспе следующим образом:

```
>((lambda (x y) (/ (+ x y) 2)) 6  14)
10
```

Вычисление лямбда-вызова происходит в несколько этапов.

- На первом этапе вычисляются значения выражений f1 f2 … fn и полученные значения связываются с соответствующими формальными параметрами p1 p2 …pn.
- На втором этапе с учетом новых связей вычисляется тело лямбда-вызова F и это значение возвращается в качестве значения всего лямбда-вызова.
- После этого восстанавливаются связи формальных параметров, существовавшие до вычисления лямбда-вызова.

Аналогичный порядок организации вычислений характерен для всех Лисп-систем, базирующихся на Коммон Лиспе. Как фактические параметры f1 f2 … fn, так и тело лямбда-выражения F могут быть в общем случае представлены любыми вычислимыми выражениями, в том числе и лямбда–выражениями. Следующие два примера показывают лямбда-вызовы, в которых другие лямбда-вызовы находятся на месте тела F лямбда-выражения

```
>((lambda (x) ((lambda (y) (cons x y)) '(2 3))) 1)
(1 2 3)
```

и на месте фактических параметров f1 f2 … fn.

```
>((lambda (x) (cons 1 x)) ((lambda (y) (cons y '(3))) 2))
(1 2 3)
```

5.2. Определение функции в Лиспе

Лямбда-выражения применяются в практическом программировании:

- когда функция должна быть сформирована в результате вычислений;
- функции необходимо в качестве аргумента передать другую функцию.

Кроме того, в гл. 8 будет показано, как лямбда-выражения применяются при программировании генераторов. Сразу после вычисления значения лямбда-вызова само лямбда-выражение пропадает, и повторное использование лямбда-выражения с другими аргументами вызова возможно лишь при его связывании с символом.

Для связывания символа с лямбда-выражением в Коммон Лиспе существует функция DEFUN (define function). Поэтому функцию DEFUN называют функцией определения функции. В общем виде

```
( defun symbol (p1 p2 …pn) F)
```

Здесь symbol – символ, который должен быть связан с лямбда – выражением (имя определяемой функции); (p1 p2 …pn) – лямбда – список; F – связываемое лямбда – выражение.

При этом для удобства в выражении F сам атом лямбда может быть опущен, например определение функции, вычисляющей среднее арифметическое двух чисел:

```
>(defun sred_ar (x y) (/ (+ x y) 2))
```

После ввода предыдущей фразы символ sred_ar будет связан с лямбда-выражением, что дает возможность применить лямбда-выражение к фактическим параметрам.

```
>(sred_ar  6 14)
10
```

Проверить наличие связи символа с лямбда-выражением можно с помощью предиката symbol-plist.

```
>(symbol-plist ‘sred_ar)
t
```

В Коммон Лиспе символ одновременно может быть связан со значением и лямбда-выражением. При этом конкретная интерпретация символа будет определяться его позицией и контекстом использования. Это иллюстрируется следующим примером.

```
>(setq x 5)
5
>(defun x (x) (* x x x))
X
>(x x)
125
```

С символом в Коммон Лиспе могут быть одновременно связаны: значение (форма set, п. 3.4); список свойств (форма putprop, п. 4.4); лямбда-выражение.

Функция DEFUN языка Коммон Лисп обладает *механизмом ключевых слов,* допускающим различные трактовки аргументов вызова функции. К наиболее значимым ключевым словам относятся:

- *необязательные параметры*;
- *параметр, связываемый с хвостом списка переменной длины*;
- *ключевой параметр*.

Описание необязательных аргументов определяется директивой &optional. В общем виде описание функции с необязательными параметрами имеет вид

(defun symbol (p1 p2 …pn &optional (n1 $n1_0$) (n2 $n2_0$) … (nn nn_0)) F)

Здесь p1 p2 … pn – обязательные параметры; n1 n2 … nn – необязательные параметры; $n1_0$ $n2_0$ … nn_0 – значения по умолчанию соответствующих параметров.

Если в вызове функции на месте необязательных параметров отсутствуют аргументы вызова, то формальные параметры n1 n2 …

nn будут связаны с соответствующими им значениями по умолчанию $n1_0$ $n2_0$ … nn_0, например определение функции, возвращающей точечную пару.

```
(defun g (x &optional (y ‘a)) (cons x y))
>G
>(g ‘q)
(Q . A)
>(g ‘q ‘x)
(Q . X)
```

Если в описании функции значения по умолчанию отсутствуют, то соответствующие необязательные параметры будут связаны со значением NIL.

```
>(defun t1 (x &optional y) (cons x y))
T1
>(t1 ‘a)
(A)
>(t1 ‘a ‘q)
(A . Q)
```

Функциям, параметры которых описаны с помощью ключевого слова &REST, можно передавать переменное число параметров. В этом случае &REST-параметр будет связан с хвостом списка аргументов переменной длины. Механизм ключевых слов Коммон Лиспа позволяет писать более эффективные, а главное, более выразительные программы. Следующий пример показывает, как с помощью ключевого слова &REST можно определить функцию, аналогичную описанной в главе 3 функции LIST.

```
>(defun list* (x &rest y) (cons x y))
LIST*
>(list* 1)
(1)
>(list* 1 2 3 4 5 6)
(1 2 3 4 5 6)
```

Кроме необязательных параметров(&optional) и параметров, связываемых с хвостом списка аргументов переменной длины(&rest), в Коммон Лиспе применяются также *ключевые параметры*(&key). Использование &key-параметров предоставляет пользователю возможность перечислять фактические параметры при вызове функции, не обращая внимания на их порядок в лямбда-списке определения функции. Тогда в вызове функции указывается имя формального параметра, перед которым указывается двоеточие и через пробел его значение.

```
>(defun f1 (&key x1 x2 x3) (/ (+ x1 x2) x3))
F1
>(f1 :x3 10.0 :x2 10.0 :x1 10.0)
2
```

5.3. Формы записи функции нескольких переменных

В параграфе 4.1. показано, как могут комбинироваться вызовы функций от нескольких переменных. Задача построения функций от нескольких переменных в функциональных языках решается по-разному. Первый способ представления таких функций соответствует общепринятому в математике:

$F(x_1,x_2, \ldots x_N)$

При таком способе F интерпретируется как функция от N аргументов. Это равносильно общепринятому в Лиспе. Аналогичная нотация предполагается и для *кортежной формы* представления функции.

Кортежем называется набор связанных величин или объектов. При этом говорят о кортеже из N – элементов (N-элементном кортеже). Кортежная форма представления функции нескольких переменных принята в языке Лисп и в большинстве императивных языков. Не существует понятия кортежа из одного элемента. В этом случае рассматривают функцию одного аргумента. Такая разница между списками аргументов и кортежами несущественна.

Карринговая форма представления функций принципиально отличается от кортежной. Идея карринговой формы (*карринг*) состоит в обработке функции от k аргументов как конкатенации k функций от одного аргумента. Название «карринг» произошло от имени математика H. Curry. В честь Карри[2] назван один из самых

современных языков функционального программирования Haskell, широко используется в исследованиях за рубежом. Отличительные особенности языка – строгая типизация, возможность отложенных вычислений. Карринг позволяет связывать аргументы с функцией и ожидать, пока остальные аргументы не появятся позже. Каррингова форма представления функций является основной в лямбда-исчислении Чёрча. Его следует использовать, когда очевидно, что с ним код будет более ясным, чем без него. Каррингова функции являются простейшим примером функций высшего порядка − функцией, результат и аргумент которой сам является функцией (гл. 8). В практическом программировании на Лиспе каррингова функции используются редко. Ниже показан пример определения функции в каррингова виде на Лиспе.

```
 (defun f (x)
       (defun s (y)
             (defun q (z)
 (/ (+ x y) z))))
F
>(f 5)
S
>(s 5)
Q
>(q 2.0)
5
```

Применение внешней функции f к аргументу приводит к порождению находящейся на уровень ниже функции s, и т.д. до тех пор, пока не появится функция, находящаяся на самом внутреннем уровне. Применение этой функции к аргументу и порождает результат функции. В этом случае каррингова форма записи приводит к более сложному коду. Использовать карринга на Лиспе или Эрланге нельзя назвать традиционным методом программирования в отличие от ряда функциональных языков, в основном с ленивой семантикой: Ocaml, Haskell.

5.4. Анонимные функции в Эрланге

Функциональные объекты являются одним из основных видов термов в Эрланге являются (гл. 3). Принадлежности функциональных

объектов к термам позволяет, с одной стороны, использовать их как функции (передавать аргументы, возвращать в качестве значений функции), а с другой стороны включать в структуры данных. Функциональный объект Эрланга в определенной мере аналогичен лябда-выражениям Лиспа. Простейшая форма записи функционального объекта имеет следующим образом:

fun (p1 p2 …pn) -> F **end.**

Здесь (p1 p2 …pn) – список *формальных параметров*, называемый функционального объекта*;* F – тело функционального объекта.

Как и Лисповское лямбда-выражение, функциональный объект Эрланга может использоваться для вычисления выражения. В этом случае общая форма записи имеет вид

fun (p1 p2 …pn) -> F **end** (f1,f2,…fn)**.**

Где f1,f2,…fn – фактические параметры, которые замещают соответствующие им формальные в теле функционального объекта.

% Определение функционального объекта

```
3> fun(X,Y)->0.5*(X+Y) end.
#Fun<erl_eval.12.118843560>
```

% Вычисления с использованием функционального объекта

```
2> fun(X,Y)->0.5*(X+Y) end (6,14).
10.0000
```

% **Функциональные объекты включены в списочную структуру**

```
3> S=[fun(X)->X*X end, fun(Y)->Y end].
[#Fun<erl_eval.6.43886099>,#Fun<erl_eval.6.43886099>]
4> S.
[#Fun<erl_eval.6.43886099>,#Fun<erl_eval.6.43886099>]
5> hd(S).
#Fun<erl_eval.6.43886099>
```

Функциональный объект может быть связан с **переменной**[2], после чего он может быть использован повторно.

% Переменная представляет функциональный объект

```
3> Sred_ar=fun(X,Y)->0.5*(X+Y) end.
#Fun<erl_eval.12.118843560>
```

% Вызов функционального объекта
2> Sred_ar(6,14).
10.0000

В общем случае определение функционального объекта имеет вид, существенно отличающийся от Лисповского определения лямбда-выражения.

Fun
   (P_1,…,P_N) [**when** condition1] ->
      Body1**;**
   …**;**
   (P_1,…,P_N) [**when** conditionK] ->
      BodyK
end

Здесь condition1,… conditionК – условие выполнения (проверка) соответствующего тела функционального объекта Body1,… BodyК. В общем случае Pi является образцом, поскольку передача значений аргументов при вызове производится путем сопоставления.

```
3> Func=fun
   3> (X,Y) when X>Y -> X-Y;
   3> (X,Y) when X<Y -> Y;
   3> (X,Y) -> ostalnoe
   3> end.
#Fun<erl_eval.12.118843560>
2> Func(5,3).
2
3> Func(15,30).
30
4> Func(15,15).
Ostalnoe
```

В условиях функциональных объектов можно использовать специальные системные предикаты, аналогичные лисповским функциям проверки типов (гл. 3). Ниже приведены основные

предикаты проверок типов термов для условий функциональных объектов:

- number(X) – проверка аргумента на число;
- float (X) – проверка на вещественное число;
- integer(X) – проверка аргумента на целое число;
- atom(X) − проверка аргумента на атом;
- function(X) − проверка аргумента на функциональный объект;
- constant(X) – аргумента на отрицание списка и кортежа;
- tuple(X) − проверка аргумента на кортеж;
- list(X) – проверка аргумента на список.

```
   3> Func=fun(X)
   3> when list(X)-> tl(X);
   3> (X) when integer(X)->X*X end.
#Fun<erl_eval.6.10732646>
2> Func([1,2,3]).
[2,3]
3> Func(7).
49
```

В любой из ветвей проверок в определении функционального объекта может находиться несколько предикатов, перечисленных через запятую. В этом случае ветвь выполняется, если успешны все предикаты.

```
   3> Func_Ob=fun
   3> (X) when number(X),integer(X) -> X*X;
   3> (X) when number(X),float(X) -> X+X;
   3> (X) when atom(X) -> atom;
   3> (X) when tuple(X) -> tuple;
   3> (X) -> list end.
#Fun<erl_eval.6.10732646>
2> Func_Ob(5).
25
3> Func_Ob(5.5).
11.0000
4> Func_Ob(at).
Atom
5> Func_Ob({2,3}).
```

```
Tuple
6> Func_Ob([2,3]).
List
```

В самом общем случае на месте условия выполнения может находиться любое логическое выражение, содержащее связки «и» (,) «или» (;).

На функциональные объекты распространяются отношения равенства (п. 3.2). Два функциональных объекта считаются идентичными, если они ссылаются на одно и тоже определение.

```
1> Min2=fun(X)->X/2.0 end.
#Fun<erl_eval.6.10732646>
2> Min21=fun(X)->X/2 end.
#Fun<erl_eval.6.10732646>
% Определение, семантически эквивалентное первому
3> Min22=fun(X)->0.5*X end.
#Fun<erl_eval.6.10732646>
% Все функциональные объекты разные
4> Min2 == Min21.
False
5> Min2 == Min22.
False
% Эквивалентное Min2 с точностью до имен переменных
6> MinY =fun(Y)->Y/2.0 end.
#Fun<erl_eval.6.10732646>
% Все функциональные объекты разные
7> Min2 == MinY.
False
% Полностью идентичные объекты
8> Xmin=Min2.
#Fun<erl_eval.6.10732646>
9> Xmin == Min2.
True
```

Самая существенная особенность функциональных объектов будет продемонстрирована следующим примером:

```
3> Exp=fun
```

```
    3> (X) when X/2>0 -> X*X;
    3> (X) when list(X) -> list;
    3> (X) when X/2<0 -> -X end.
#Fun<erl_eval.6.10732646>
% Применение Exp к списку должно вызвать ошибку в первой
% ветви
2> Exp([1,2,3]).
List
% Ошибки не произошло – проверяется следующая ветвь
3> Exp(4).
16
4> Z=[1,2,3].
[1,2,3]
% Попытка деления списка на число вне функционального
% объекта приводит к ошибке
5> C=Z/2.
=ERROR REPORT==== 3-Dec-2005::06:23:48 ===
Error in process <0.30.0> with exit value:
{badarith,[{erl_eval,eval_op,3},{erl
eval,expr,5},{shell,exprs,6},{shell,eval_loop,3}]}

** exited: {badarith,[{erl_eval,eval_op,3},
                     {erl_eval,expr,5},
                     {shell,exprs,6},
                     {shell,eval_loop,3}]} **
```

Таким образом, ***отказоустойтивость***[3]**,** разрабатываемых на Эрланге приложений, закладывается уже конструкцией функциональных объектов. Ошибки, появляющиеся при выполнении функций, приводят к неудаче соответствующей ветви, функции, подпроцесса, но не вызывают аварийного прекращения работы программы (сравните запросы 2 и 5 в предыдущем примере).

5.5. Функции и модули Эрланга

Единицами компиляции в Эрланге являются модули, в которые собираются определения функций. Модуль имеет собственное пространство имен. Модуль состоит из

последовательности атрибутов и функциональных объявлений, заканчивающихся точкой (.).

```
-module (ex01).              % атрибут модуля
-export ([member/2]).        % атрибут модуля
```

% Определение функции принадлежности элемента списку

```
member([],_)->{false};
member([H|_],H)->{true};
member([H|T],L)->member(T,L).
```

Атрибут модуля определяет его конкретное свойство. Атрибут модуля определяется его именем и значением:
- имя атрибута (значение).

Предопределенные атрибуты модуля должны быть размещены перед любым определением функции. Ниже перечислены основные предопределенные атрибуты модуля:

- -module(Name_Module). Объявление имени модуля. Единственный обязательный атрибут. Должен быть самой первой фразой программы. Имя Name_Module должно совпадать с именем файла (Name_Module.erl), в котором находится модуль;
- -export(Functions). Экспорт функций. Атрибут определяет, какие из функций модуля должны быть видимы за его пределами. Указывается списком [Name1/Arity1, …, NameN/ArityN], где Name1− имя функции (атом), а Arity1− ее арность (целое). Если за пределами модуля вызывается функция, не объявленная как экспортируемая, то произойдет ошибка.

Кроме того, имеются предопределенные атрибуты, специфицирующие опции компиляции, импортируемые функции, версию модуля.

Определение функции в общем случае – последовательность предложений, разделенных **(;)** и заканчивающаяся точкой (**.**) :

```
Name(P1,…,PN) [when condition1] ->
   Body1;
…;
Name(P1,…,PN) [when conditionK] ->
   BodyK.
```

Здесь Name – имя функции (атом); (P_1,…,P_N) – список параметров, представленных образцами; when – зарезервированное слово для введения условий; condition – собственно условия (логические выражения); -> отделяет заголовок описания функции от ее тела Body.

Уникальность функции определяется совокупностью имени модуля, имени функции, ее арностью. В одном модуле могут находиться несколько функций с одинаковым именем, но разной арностью. К функциям применимы все правила (передачи параметров, проверки условий) для функциональных объектов. Программа транслируется в промежуточный байт-код[4], исполняемый виртуальной машиной.

Порядок выполнения функции следующий. Последовательно просматриваются предложения, пока не будет найдено отвечающее следующим условиям:

- образцы в заголовке предложения успешно согласовываются с фактическими параметрами;
- соответствующее условие condition также успешно.

Для такого предложения выполняется тело Body. Если предложение, отвечающее перечисленным выше требованиям не найдено то происходит ошибка.

```
% Вызов компилятора из оболочки Эрланга
   3> c(ex01).
./ex01.erl:6: Warning: variable 'H' is unused
% Сообщение о несвязанной переменной
{ok,ex01}
2> ex01:member([1,2,3,4],5).
{false}
3> ex01:member([1,2,3,4],3).
{true}
% Проверка принадлежности элементов списку
```

Примечания

1. Имеются в виду замыкания, полноценная возможность работы с которыми появилась в C# 3.0. Подробно о замыканиях на Лиспе п.8.4.
2. Хаскелл Карри (Англия) – один из авторов теории комбинаторов. Мозес Шёнфинкель (Россия) считается вторым автором, и именно ему принадлежит идея представления функции многих аргументов как последовательность функций одного аргумента.

3. В некоторых версиях Лиспа, например Franz Lisp, переменная тоже может быть связана с лямбда-выражением псевдофункцией setq, что равносильно определению функции.
4. Системы высокой готовности и отказоустойчивые системы должны обеспечивать надежное продолжительное функционирование. Для этого с программной точки зрения часто применяется технология изоляции неисправных процессов, гарантирующая локализацию ошибок в одной системе и невозможность их распространения за пределы этой системы.
5. Байт-код (*byte-code*) − машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый виртуальной машиной. Длина каждого кода операции равна одному байту, при различной длине команды. Программа на байт-коде обычно выполняется *интерпретатором байт-кода* (*виртуальной машиной*). Компиляция в байт-код занимает промежуточное положение между компиляцией в машинный код и интерпретацией. Преимущество такого подхода к выполнению программ в том, что один и тот же бинарный код может исполняться на разных *платформах* и *архитектурах*. Наиболее известные системы программирования на основе байт-кода – Java, Perl, PHP.

Резюме

• Определение функции – наиболее значимый механизм в технологии программирования. Он позволяет разбивать программу на части, отлаживать по частям.

• В Коммон Лиспе для разбиения программы на части существует механизм лямбда−выражений и механизм связывания функции с символом.

• В практическом программировании механизм лямбда-выражений применяется, когда функция должна быть сформирована в результате вычислений или когда функции необходимо в качестве аргумента передать другую функцию, но повторное вычисление с другими аргументами невозможно.

• Связывание функции с символом позволяет многократно использовать определенную ранее функцию.

• Функция определения функции Коммон Лиспа (defun) обладает механизмом ключевых слов, которые допускают различные трактовки аргументов вызова функции.

• В функциональном программировании применяются кортежная и каррингова формы записи функций от нескольких переменных.

• Кортежная форма широко применима в практическом программировании, а каррингова – в программировании высшего порядка (гл. 8) и лямбда-исчислении (гл. 9).
• Надежность приложений на Эрланге обеспечивается организацией функциональных объектов, позволяющей локализовать ошибки в рамках одного процесса, что не приводит к аварийному завершению работы всего приложения.

Упражнения

1. Запишите в карринговой и кортежной форме лямбда-выражения, для вычисления следующих выражений:
   1. (cos(x)/(x+y))-z (с аргументами x=3.15 , y=0.3, z=2)
   2. (x-y)(x+y+z) (с аргументами x=3.15 , y=0.3, z=2)
2. Определите функции для вычисления выражений из предыдущего задания. Функции должны быть записаны в карринговой и кортежной форме.
3. Определите связь между свойствами функциональных языков программирования: энергичная (ленивая) стратегия вычислений и формами (карринговой или кортежной) описания функции нескольких переменных. Какая форма записи наиболее предпочтительна для соответствующей стратегии вычислений. Помощь при ответе на этот вопрос можно найти в гл. 10.
4. Какой механизм связывания (статическое связывание или динамическое связывание) следует из порядка применения лямбда - выражения в Лиспе?
5. В задаче искусственного интеллекта требуется представить знания о студентах. Данные о наименовании группы, в которой находится студент, его стипендии хранятся как значения свойств символа, обозначающего студента. Определите следующие функции, описывающие данную предметную область:
   функцию определяющую, что два студента знают друг друга;
   функцию, определяющую успевающего студента.
   Использовать правило, согласно которому если студент не получает стипендию, то он считается неуспевающим.

Использовать правило, согласно которому студенты, обучающиеся в одной группе, знают друг друга.

ГЛАВА 6

ИМПЕРАТИВНЫЕ МЕХАНИЗМЫ И УПРАВЛЕНИЕ

Работа с контекстом в Лиспе
Организация ветвлений в Лиспе
Условные вычисления в Эрланге
Императивные механизмы Лиспа
Организация ввода-вывода в Лиспе
Организация ввода-вывода на Эрланге

В реальном программировании часто приходится выбирать между неэффективными, но выразительными конструкциями функциональных языков и эффективными, но менее выразительными конструкциями императивных языков. В таком случае функциональные языки программирования, которые предлагают в распоряжение разработчика набор императивных механизмов, являются более предпочтительными. Кроме того, следует учитывать, что обмен информацией с внешними устройствами, без которого немыслимы реальные программы, происходит за счет побочных эффектов, к которым относятся в частности и любые операции ввода-вывода. В этой главе будут разобраны основные императивные механизмы Коммон Лиспа и Эрланга [11,13].

6.1. Работа с контекстом в Лиспе

Ранее были рассмотрены следующие формы, предназначенные для работы с контекстом: блокировка вычислений (гл. 2), вызовы функций и лямбда-выражения (гл. 4). К этой группе управляющих структур относятся LET-выражения, с помощью которых создаются новые связи внутри формы.

(LET ((p1 $value_1$) (p2 $value_2$) ... (pn $value_N$)) F_1 F_2 ... F_K)

Вычисление LET-выражений происходит в несколько этапов:

- на первом этапе вычисляются значения выражений $value_1$ $value_2$ $value_N$;
- на втором этапе происходит одновременное связывание вычисленных значений с переменными p1 p2 pn;
- на третьем этапе последовательно вычисляются формы F_1 F_2 ... F_K ;
- на четвертом этапе значение последней формы F_K возвращается в качестве значения всего LET-выражения и переменным p1 p2 pn возвращаются связи, которые у них были до начала вычисления.

Такой порядок вычислений полностью соответствует порядку вычисления лямбда-вызова. Соответствующий лямбда-вызов в общем случае будет иметь вид

((Lambda (p1 p2 pn) F_1 F_2 ... F_K) $value_1$ $value_2$ $value_N$)

LET-выражение для примера из п. 4.4 будет иметь вид.

```
>(let ( (x 9.0) (y 3.0) (z 2.0)) (/ (-  x y) z))
3
```

В Коммон Лиспе определена также форма LET* , основное отличие которой от формы LET состоит в последовательном связывании переменных p1 p2 pn со значениями $value_1$ $value_2$ $value_N$.

(LET* ((p1 $value_1$) (p2 $value_2$) ... (pn $value_N$)) F_1 F_2 ... F_K)

Это отличие хорошо иллюстрируется следующим примером.

```
>(let ((x 5.0) (y (* x 2))) (+ x y))
error: unbound variable - X      ; несвязанная переменная X
; X и Y связываются одновременно при связывании Y y X связи еще нет
>(let* ((x 5.0) (y (* x 2))) (+ x y))
15
>(let* ( (a 1) (b (cons a nil)) ) (setq r (cons a b)) (cons a r) )
(1 1 1)
```

6.2. Организация ветвлений в Лиспе

В Коммон Лиспе введены следующие формы для организации ветвления:

- условное предложение COND;
- условное предложение IF;
- условное предложение WHEN;
- условное предложение UNLESS;
- условное предложение CASE;
- предикаты AND, OR, NOT.

Условное предложение COND является основной классической формой организации ветвления, существующей в Лиспе еще с первых интерпретаторов. В общем виде условное предложение выглядит следующим образом:

```
(COND (p1 f1) (p2 f2) ... (pn fn))
```

Порядок вычисления условного предложения следующий:

- на первом этапе последовательно вычисляются предикаты p1 p2 ... pn до момента обнаружения предиката pi, вычисленное значение которого отличается от NIL;
- на втором этапе вычисляется форма fi, соответствующая предикату pi;
- значение формы fi возвращается в качестве значения предложения COND.

Если истинный предикат так и не будет обнаружен, то предложение COND возвращает значение NIL. Одна или несколько форм fi могут отсутствовать. Тогда в качестве значения предложения COND возвращается значение соответствующего этой форме предиката pi. На месте любой из форм fi может находиться последовательность форм fij. В этом случае вычисляются последовательно слева направо все формы, и значение последней формы возвращается в качестве значения предложения COND.

```
(COND (p1 f1) (p2 f2) ...
          (pi fi1 f12 ... fik)
               (pn fn))
```

Приведем примеры функций с использованием форм организации ветвлений в программе:

```
>(defun f (x y) (cond
( (> x y) (* x x x) )
( t (* y y y) )))
>(f 5 1)
125
>(f 5 10)
1000

>(defun s (x y z)
(cond
( (> x y) (setq w (cons y nil)) (cons x w) ) ; последовательность 2 форм
( (> x z) )   ; данному предикату нет соответствующей формы
( t   (setq w (cons z nil)) (setq w (cons y w)) (cons x w))))
; последовательность 3 форм
>(s 5 2 1)
(5 2)
>(s 5 6 1)
T
>(s 5 6 7)
(5 6 7)
```

Помимо формы COND для организации ветвления в Коммон Лиспе наиболее часто применяется форма IF.

```
(IF  pred  Fthen  Felse)
```

Если значение предиката pred не NIL, то выполняется форма Fthen, иначе форма Felse. Следующий пример показывает функцию, которая относит свой аргумент к одной из двух групп.

```
>(defun s (l)
 (if (numberp l) 'number 'ostalnoe))
s
>(s 2)
NUMBER
```

```
>(s ‘(1 2 3))
OSTALNOE
```

Форма CASE является выбирающим предложением, аналогично одноименному оператору большинства императивных языков.

```
(CASE expression
        (List-case1 F11 F12 ... F1n)
        (List-case2 F21 F22 ... F21k)
        . . . . . .
        (List-case_m Fm1 Fm2 ... Fms))
```

Порядок выбора в предложении CASE следующий:

- вычисляется значение ключевой формы expression;
- значение expression сравнивается с элементами List-case1, List-case1... List-case_m;
- если какой-либо элемент списков List-case1, List-case1... List-case_m совпадет со значением выражения expression, то выполняется последовательность форм Fij соответствующего списка и возвращается значение последней из форм;
- в противном случае возвращается значение NIL.

Действие выбирающего предложения иллюстрируется следующим примером:

```
>(defun stran (gor)
            (case gor
                ((moscva piter bryansk) 'russian)
                ((berlin franc kmstad) 'ger)))
Stran

>(stran ‘bryansk)
RUSSIAN
```

6.3. Условные вычисления в Эрланге

Условные формы в Эрланге не имеют такого значения, как в других языках программирования. Структура определения функции, основана на проверках и сопоставлении с образцом и дает

возможность реализовать условные вычисления (гл. 5). Условные формы Эрланга (if и case), аналогичны предложениям Лиспа (cond и case) и записываются следующим образом:

```
if
    condition1 -> Body1;
   ...;
   conditionK ->BodyK
end
```

Здесь condition1,… conditionК – условие выполнения (проверка) соответствующего тела функционального объекта Body1,… BodyК. Основное отличие от формы cond Лиспа заключается в том, что если все условия condition будут вычислены со значением false, то произойдет ошибка. Поэтому последнее условие conditionК должно обязательно равняться true.

```
% функциональный объект нахождения максимального двух чисел
    1> Max=fun (X,Y)->
    1> if
    1> X>Y -> X;
    1> X<Y -> Y;
    % последняя ветвь должна иметь успешное условие
    1> true -> X
    % символ ; не ставится
    1> end end.
    #Fun<erl_eval.12.118843560>

    2> Max(5,3).
    5

    3> Max(5,8).
    8

    4> Max(8,8).
    8
```

Условное предложение case позволяет выбирать необходимую ветвь решения в зависимости от результата некоторого выражения

Expr. Выбирается ветвь, образец которой PatternI, сопоставляется с значением выражения Expr.

```
case Expr of
Pattern1 [when condition1] ->Body1;
...;
PatternN [when conditionK] ->BodyN
end
```

Следующий пример иллюстрирует использование условного предложения для выбора решения, подпадающего под определенный образец:

```
% функция определения страны по названию столицы
land(Town)->
case Town of
    moscow -> russia;
    london -> england;
    paris -> france;
% анонимная переменная сопоставляется с любым значением
% иначе будет ошибка
    _ -> unknown
    end.
```

Обратим внимание на одно отличие условных конструкций Эрланга и Лиспа. В Эрланге в случае неудачи всех ветвей условного предложения возникает ошибка, а в Лиспе возвращается NIL. Поэтому в Эрланге последняя ветвь условного предложения обязательно должна иметь положительный заголовок (true для конструкции if, анонимную переменную для case).

6.4. Императивные механизмы Лиспа

Для организации повторяющихся вычислений в Коммон Лиспе существую формы, позволяющих определить циклический (итерационный) процесс. К таким формам относятся предложения DO и DO*, которые в общем виде записываются следующим образом:

```
(DO  ( (p1 val1 step1) (p2 val2 step2) ... (pn valn stepn) )
       ( pred  Fp1 Fp2 ... Fpk)
        F1 F2 ... Fn
)
```

Здесь

- p1 p2 pn − внутренние переменные предложения DO и DO*;
- val1 val2 valn − начальные значения внутренних переменных;
- step1 step2 stepn − формы изменения начальных переменных;
- pred − предикат, определяющий условия выхода из цикла;
- Fp1 Fp2 ... Fpk и F1 F2 ... Fn − последовательность форм.

Предложения DO и DO* действуют следующим образом:

- на первом этапе переменным p1 p2 pn присваиваются начальные значения val1 val2 valn аналогично присваиванию значений для форм LET и LET* (одновременно для предложения DO и последовательно для DO*);
- если начальное значение для переменной отсутствует, то присваивается значение NIL;
- на втором этапе вычисляется значение предиката pred , определяющего условия прекращения цикла;
- дальнейшие вычисления будут зависить от значения предиката (если это значение не NIL, то последовательно вычисляются формы Fp1 Fp2 ... Fpk и значение последней формы возвращается в качестве значения предложения DO, что определяет прекращение цикла);
- если значение предиката pred равно NIL, то вычисляются последовательно формы F1 F2 ... Fn, после чего вычисляются формы изменения переменных цикла, и вычисления продолжаются с первого этапа.

Если формы изменения переменных цикла не заданы, то они должны быть определены в теле цикла. Следующий пример показывает вычисление факториала числа N с помощью предложения DO. Процедура вычисления основана на последовательном вычислении произведения всех чисел от 1 до числа N.

```
(defun fac (n)
        (do ((x 1 (+ x 1)) (R 1))
             ((> x n) R)
         (setq R (* R x))
```

```
        )
)

(defun st (x n)
   (do ((res 1))
      ((= n 0) res)
      (setq res (* res x))
      (setq n (- n 1))))
```

Итерационные вычисления оказываются практически единственным механизмом для решения задач, на подобие задачи нахождения суммы ряда. В данном случае изначально не известно, в какой момент следует прекратить вычисления. Будем считать, что сложение членов ряда следует прекратить в момент достижения некоторым членом определенного, заранее заданного значения (что нельзя считать соответствующим математическому определению суммы ряда). Функция, вычисляющая сумму ряда $(-1)^n*1/n$, определена ниже.

```
(defun sum  (e)
   (do ((n 2.0 (+ n 1.0)) (et 1.0) (s 1.0))
   ((< et e) s)
   (setq s1 (f n))
   (setq s (+ s s1))
   (setq et (- (f (- n 1)) s1)))
)
```

Функция для значения определения члена ряда

```
(defun f (n)
(/ 1.0 n)
)
```

Последовательные вычисления на Лиспе являются одним из самых распространенных императивных механизмов. Выражения типа LET позволяют вычислять несколько форм подряд и возвращать значение последней формы в качестве значения всего выражения. Помимо этого, существует формы для организации последовательных вычислений:

- (PROG1 F1 F2 ... Fn) вычисляет последовательно формы F1 F2 ... Fn и возвращает значение первой формы;
- (PROG2 F1 F2 ... Fn) вычисляет последовательно формы F1 F2 ... Fn и возвращает значение второй формы;
- (PROGN F1 F2 ... Fn) вычисляет последовательно формы F1 F2 ... Fn и возвращает значение последней формы

```
>(defun s () (prog1 (setq x 5) (setq y (+ x 10)) (setq z (+ x y))))
s
>(s)
5
>(defun s () (prog2 (setq x 5) (setq y (+ x 10)) (setq z (+ x y))))
s
>(s)
15
>(defun s () (progn (setq x 5) (setq y (+ x 10)) (setq z (+ x y))))
s
>(s)
20
```

6.5. Организация ввода-вывода в Лиспе

Функциональный стиль программирования предполагает, что программа состоит из набора функций, каждая из которых описывает конкретное свойство искомого решения. Процесс решения предполагает, что каждая функция возвращает результат в вызвавшую ее функцию. В практическом программировании часто возникает необходимость в управлении текущим вычислительным процессом, в обмене информации с файлами. Такие действия по отношению к чисто функциональному решению называются *побочными эффектами*. Все перечисленные действия в Лиспе выполняются с помощью псевдофункций ввода–вывода.

Псевдофункция чтения обрабатывает вводимое S-выражение как целое. Процедура организации чтения в общем виде выглядит следующим образом:

```
>(READ <stream>)
(вход)          ; вводимое пользователем выражение
(ВХОД)          ; введенное выражение возвращается как значение
```

;псевдофункции READ

>

Здесь <stream> − входной поток, который по умолчанию связан с пользовательским терминалом. Чтобы с введенным выражением можно было в дальнейшем работать, значение функции READ необходимо связать с переменной. Следующий пример показывает ввод списка:

```
>(setq w (read))
(u popa bila sobaka)          ; введенное выражение
(U POPA BILA SOBAKA) ; возвращенное значение
>w
(U POPA BILA SOBAKA) ; связанное выражение
```

Псевдофункция READ основана на определенной на системном уровне *процедуре чтения(list reader).* Процедура читает вводимое из потока S-выражение как последовательность символов. Среди последовательности символов встречаются специальные знаки, такие как, открывающиеся и закрывающая скобки, апостроф, точка и.т.п. Этим знакам поставлено в соответствие определенное действие. Соответствие между специальными знаками и определенными действиями задается определенной на системном уровне *таблицей чтения*(*readtable*). *Макрознаками (макросами чтения)* называются знаки, вызывающие специальные действия. Изменяя или дополняя таблицу чтения, программист может таким образом изменять синтаксис Лиспа.

Для вывода в XLISP существуют следующие функции вывода:

- (print <expr> [<stream>]) − вычисляет значение <expr>, выводит в выходной поток [<stream>] и переводит строку;
- (prin1 <expr> [<stream>]) − вычисляет значение <expr> и выводит его результат в выходной поток без перевода на новую строку;
- (princ <expr> [<stream>]) − вычисляет значение <expr> и выводит его результат без перевода на новую строку и без ограничивающих кавычек;
- (terpri [<stream>]) − переводит строку в текущем выходном потоке <stream>, в качестве значения возвращает NIL.

По умолчанию в качестве выходного потока принят *стандартный выходной поток* (вывод на монитор). Псевдофункции PRIN1 и PRINC позволяют выводить в выходной поток, кроме атомов и списков еще и строки. Строка в Лиспе − последовательность символов, ограниченная с обеих сторон кавычками. Так, следующая последовательность представляет строку " это строка ".

```
>(print (car ‘(a s d)))
A
; побочный эффект – вывод значения аргумента
A       ; функциональный возврат значения
>(prin1 (car ‘(a s d)))
AA
; эффект вывода и возврат значения на одной строке

>(prin1 ?a s d?)
a s d ?a s d?
; вывод без ограничивающих кавычек и возврат значения
>(prong (prin1 'stroka’)
(terpri)  ; перевод строки
(print ‘end))
stroka?
END
```

Вывод с помощью псевдофункций PRINT, PRIN1, PRINC требует для печати сложных выражений построения специальных структур, например с помощью предложений PROG, как в последнем примере. Более удобным будет использовать псевдофункцию FORMAT, обеспечивающую вывод в соответствии с образцом. В общем виде вызов псевдофункции FORMAT записывается следующим образом.

(format <stream> <fmt> &REST<arg>)

Здесь аргументы формы имеют следующий смысл:

- <stream> выходной поток. Стандартный выходной поток, связанный с экраном монитора имеет значение T.

- <fmt> образец вывода. Представлен управляющей строкой, начинающейся знаком ~ и содержащей управляющие коды;
- <arg> аргументы вывода. Становятся в соответствие управляющим кодам строки.

Если управляющая строка и соответствующие ей аргументы отсутствуют, то FORMAT выводит строку-аргумент как псевдофункция PRINC, но в качестве значения возвращает NIL.

```
>(format t ‘Output end’)
Output end
NIL
```

В качестве управляющих кодов псевдофункции FORMAT применятся следующие:

- ~A вывод следующего аргумента псевдофункцией PRINC;
- ~S вывод следующего аргумента псевдофункцией PRIN1;
- ~% вывод с новой строки;
- ~~ вывод символа тильда.

Следующий пример показывает решение задачи табулирования функции и вывод промежуточных результатов с помощью псевдофункции FORMAT.

```
(defun sum1 (a h b)
(do ((x a (+ x h))) ((> x b) (format t "_END_")) (setq y (f1 x))
(format t "~s      ~s~%" x y)))

(defun f1 (x)
(* x x)
)
```

Как отмечалось, ввод-вывод организован посредством потоков <stream>. На системном уровне существует несколько стандартных потоков, которые являются значениями глобальных переменных. Системные переменные определяют для функций ввода-вывода соответствующий файл по умолчанию, которым в начале сеанса является консоль пользователя. Чтобы пользоваться этими файлами по умолчанию в псевдофункциях ввода-вывода (кроме FORMAT), на месте аргумента <stream> не требуется ничего указывать. Для организации операций ввода-вывода в файл пользователя необходимо

перенаправить на него соответствующий поток ввода или вывода. Таким образом, порядок действий, которые необходимо выполнить для общения с нестандартным файлом, следующий:

- открыть файл для ввода или вывода;
- открытый файл связать с переменной, которая в дальнейшем будет представлять соответствующий поток ввода или вывода;
- указывать в соответствующих псевдофункциях на месте аргумента <stream> имя переменной, представляющей соответствующий поток, при осуществлении операции ввода-вывода;
- закрыть файл по окончании ввода−вывода.

Для открытия файла предназначена директива OPEN, которая в общем виде записывается следующим образом:

(open <fname> &key :direction)

Здесь аргументы имеют следующий смысл:

- <fname> имя файла, которое может быть представлено строкой или символом, например file1.lsp;
- :direction ключ, определяющий тип потока. В XLISP потоки могут быть двух типов. Входной поток (:input) является потоком по умолчанию. Выходной поток обозначается как :output;
- псевдофункция возвращает открытый поток.

Закрыть поток можно директивой

(close <stream>)

Здесь <stream> − открытый ранее поток. Псевдофункция возвращает NIL. Поток, открытый для записи, должен быть обязательно закрыт. В противном случае записанные данные не будут сохранены. Следующий пример показывает вывод результатов табулирования функции в файл.

(defun sum1 (a h b)

(setq d (open "file1.lsp" :direction :output)) ; открытие файла и связь с переменной d, которая ; в дальнейшем будет представлять выходной поток

```
(do ((x a (+ x h)))
((> x b) (format d "_END_"))
; вывод в поток сообщения о конце работы
(setq y (f1 x))
(format d "~s      ~s~%" x y))
 ; вывод в поток результатов табулирования
(close d)
; закрытие файла, с которым связана переменная d
)

 (defun f1 (x)
(* x x)
)
```

6.6. Организация ввода-вывода в Эрланге

На Эрланге организовать ввод-вывод можно следующими способами:

- как непосредственный обмен с файлами и внешними устройствами;
- работу с базами данных.

В первом случае порядок работы аналогичен рассмотренному ранее на примере Лиспа. Основные функции, обеспечивающие наиболее часто использующиеся операции ввода-вывода, определены в модулях IO, File. Кроме того, есть ряд модулей, включающих функции, обеспечивающие дополнительные возможности работы с внешними устройствами. Модуль io включает стандартные функции ввода-вывода. Функции ввода-вывода обладают побочным эффектом, который заключается в выполнении обмена с внешними устройствами. Успехом считается выполнение операции ввода-вывода (побочный эффект). В общем виде вызов функции модуля io имеют вид:

```
io:name_function([IoDevice,] Arg1, ..ArgN)
```

Здесь [IoDevice,] – необязательный параметр, который позволяет перенаправлять потоки ввода-вывода; Arg1, ..ArgN – аргументы функции.

Так, наиболее часто используемые функции вывода имеют вид:

fwrite([IoDevice,] Format, Arguments)

format([IoDevice,] Format, Arguments)

Здесь Format − управляющие параметры вывода, Arguments-список аргументов.

1> io:fwrite("вывод строки. Далее управляющие параметры:~n",[]).

вывод строки. Далее управляющие параметры:

ok

% вывод значения в десятичном коде

2> io:format("~.10B~n",[32]).

32

ok

% вывод значения в шестнадцатеричном коде

3> io:format("~.16B~n",[31]).

1F

ok

% вывод значения в двоичном коде

4> io:format("~.2B~n",[31]).

11111

ok

Функции модуля File предоставляют программисту возможность манипулирования файловой системой. Здесь собраны функции, которыми можно, в частности, выполнять следующие действия: копировать файлы, удалять файлы и директории, отрывать файлы, читать файл, содержащий термы. Значение, возвращаемое функцией, будет иметь формат {ok, Value}, в случае успеха выполняются операции или {error, Reason} при неудаче. Отметим, что возвращаемое значение может быть различным в разных операционных системах. Кроме того, управляя опциями можно добиться фонового режима выполнения операций ввода-вывода.

Другая возможность организовать обмен информацией с внешними носителями состоит в непосредственном взаимодействии с базами данных. Приложение ODBC[1] обеспечивает интерфейс с реляционными базами данных вызовом команд языка структурных

запросов SQL. При этом в распоряжении программы должны иметься драйверы базы данных MS ODBC. Так, следующее утверждение устанавливает связь с базой данных и ассоциирует ее с процессом.

connect(ConnectStr, Options)

При этом могут создаваться новые процессы, с помощью которых обрабатывается установленная ранее связь. Процесс завершается при разрыве связи.

Примечание

1. ODBC (Open DataBase Connectivity) − Открытый интерфейс взаимодействия с базами данных, разработанный корпорацией Microsoft. С точки зрения приложений ODBC-совместимый источник данных рассматривается как реляционная база данных, взаимодействие с который осуществляется с помощью языка SQL. Для работы с источником данных через ODBC прикладной программе требуется специальный ODBC−драйвер, который «скрывает» в себе особенности работы с конкретной СУБД. Практически для всех современных СУБД разработаны ODBC-драйверы, которые поставляются в комплекте с их клиентской частью.

Резюме

К императивным включениям в Лиспе относятся:

- средства организации циклических вычислений;
- последовательные вычисления;
- механизмы ввода-вывода;
- в одних случаях императивные механизмы увеличивают эффективность решения (гл. 7), в других случаях это практически единственный инструмент решения задачи (пример вычисления суммы и вывода в файл);
- обмен программы данными с файлами осуществляется перенаправлением выходного потока.

Упражнения

1. Чем отличаются структуры работы с контекстом от функций?

2. Почему в главе формы организации циклических вычислений отнесены к императивным механизмам?
3. Какие формы организации ветвлений обеспечивают многовариантность выбора (>2)?
4. Можно ли назвать форму организации ветвления в программе COND функцией выбора? Аргументируйте свой ответ, используя механизм связывания формальных параметров, необходимый для функции.
5. Почему в главе формы организации последовательных вычислений отнесены к императивным механизмам?
6. Почему функции организации ввода-вывода относятся к псевдофункциям?
7. Рассмотрите вопрос последовательного и параллельного связывания параметров в формах let и let*, а также форм do и do*, с позиций организации энергичных и ленивых вычислений.
8. Составьте программу вычисления суммы ряда с заданной точностью e. Считать решение полученным, когда значение некоторого члена ряда не станет меньше заданной точности. S=1/(1*2*3)+1/(2*3*4)+1/(3*4*5)+…
9. Составьте программу табулирования функции $Y=x^3+x+12$, на промежутке [-1,5;0]. Результаты табулирования запишите в файл.
10. Выполните задачу 8 с выводом решения в файл и чтения исходных данных из файла.
11. Определите, используя формы организации циклов функцию, добавляющую число в список, состоящий из упорядоченных чисел, без нарушения порядка.

```
>(aps ‘(2 4 6 7) 5)
(2 4 5 6 7)
```

Решение должно быть функциональным при создании новых структур.
12. Определите с помощью циклических механизмов предикат, проверяющий список, состоящий из чисел, на упорядоченность.

```
>(up ‘(1 2 4 67 78))
T
```

13. Определите функцию, выполняющую слияние списков с помощью формы Do. Решение должно быть функциональным при создании новых структур.

```
>(ap_it ‘(1 2 3)  '(4 5 6))
(1 2 3 4 5 6)
```

14. Определить функцию, выполняющую суммирование четных элементов списка с помощью формы Do. Решение должно быть функциональным при создании новых структур.

```
>(chet '(1  3 5 2))
 5
```

ГЛАВА 7

РЕКУРСИВНОЕ ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Способы решения задач
Основные положения теории рекурсивных функций
Классификация видов рекурсии
Реализация рекурсивного процесса
Хвостовая рекурсия
Рекурсивные определения стандартных функций
Эффективность и критерии выбора рекурсивного решения

В программировании, где запрещены побочные эффекты, а значит присваивания и циклические конструкции, единственным средством организации повторяющихся вычислений, является рекурсия. Основная структура функциональных языков программирования – список, по определению рекурсивна (п. 3.2.).

Практика реального практического программирования в функциональном стиле требует от студента знания следующих теоретических вопросов: основы *теории рекурсивных функций* в приложении к практическому программированию (п. 7.2.), классификация рекурсивных функций и видов *рекурсии* (п. 7.3.), порядок выполнения и эффективность рекурсивного процесса (п. 7.4., п. 7.5). Помимо этих знаний студент должен уметь разрабатывать рекурсивные функции и, самое главное, уметь четко и аргументировано выбирать итерационный или рекурсивный алгоритм решения задачи (п. 7.6., 7.7.).

7.1. Способы решения задач

Любую задачу можно решить двумя способами. Первый подход предполагает, что задачу можно свести к композиции более простых подзадач, второй способ предполагает, что задачу можно свести к ней самой, но с более простыми исходными данными.

При решении задачи по второму способу должно быть известно следующее:

- решение задачи для простейших исходных данных;
- порядок сведения задачи к более простым исходным данным во всех остальных случаях.

Полное решение задачи может быть получено, если несколько последовательных этапов упрощения приводят задачу к известному простому решению.

Способ сведения задачи к ней же самой, но с измененными исходными данными, называется рекурсией.

Алгоритм решения задачи, который содержит обращения к себе, называется *рекурсивным алгоритмом*.

В функциональном программировании отсутствуют такие механизмы императивного программирования, как присваивание, циклы и передача управления, являющиеся основными при организации повторяющихся вычислений в императивном программировании (гл. 2). Единственным действием в функциональном программировании является вызов функции. Отсюда следует, что повторные вычисления в функциональном программировании должны быть организованы посредством рекурсии, т.е. посредством вызова функциями самих себя. С другой стороны, структурный основной объект Лиспа – список − имеет рекурсивное строение (список − это структура, которая либо пуста (NIL), либо состоит из головы и хвоста, а хвост, в свою очередь, является списком). Таким образом, список как основной структурный объект максимально подходит для организации рекурсивного решения.

Рекурсия широко распространена в живой и неживой природе, технике, науке[1]**.** Логично предположить, что для таких задач рекурсивный метод решения будет оптимальным.

7.2. Основные положения теории рекурсивных функций

Теория рекурсивных функций − область математики, в которой изучаются теоретические вопросы, связанные с вычислимостью, а сами *рекурсивные алгоритмы* рассматриваются и классифицируются, исходя из того, какие функции можно определить и вычислить с использованием различных форм рекурсии[2]. При этом полагается, что функция может быть с помощью *рекуррентных соотношений*

сведена к известным начальным значениям. Чтобы отношение было рекуррентным, необходимо выполнить следующие условий:

- все переменные принимают значения из множества целых положительных чисел;
- значение f(n) описывается через значения f(m), так, что m<n;
- должны быть известны некоторые *начальные условия рекурсии* f(0) или f(1).

В общем случае рекуррентное соотношение может быть записано в виде

g(n)=f(n,g(n-1),g(n-2), ..., g(n-k)),

где f − некоторая функция k+1 аргумента. Число k называется *порядком соотношения.* Приведем примеры рекуррентных соотношений:

- Факториал числа определяется зависимостями f(0)=1; f(n)=n*f(n-1) при n>0, порядок соотношения k=1.
- Числа Фибоначчи определяются зависимостями f(1)=f(2)=1.
- f(n)=f(n-1)+f(n-2) при n>2, порядок соотношения k=2.

Все рекурсивные функции можно разделить на два основных класса. К первому классу относятся *примитивно рекурсивные функции (primitive recursive).*

К *примитивно рекурсивным* относятся функции, которые могут быть получены из определенных базовых функций применением конечного числа операций композиции и рекурсии.

В качестве исходных функций обычно применяются *нуль-функция*, *функция проецирования* и *функция последования*. Нуль-функция имеет значение ноль на всей области определения. Функция проецирования выделяет элемент последовательности, а функция последования генерирует очередной элемент последовательности. Классическим примером примитивно рекурсивных функций является функция, вычисляющая факториал числа. Рекурсивное определение функции, вычисляющей факториал числа, практически повторяет его математическое представление. Лисповское определение функции следующее.

```
>(defun fact (n)
   (cond
      ((= n 0)  1)
```

```
        (t        (* n (fact (- n 1)))))
)

FACT
>(fact 5)
120
```

Функции, относящиеся ко второму классу, называются *общерекурсивными (general recursive).* Классическим примером функций этого класса является *функция Акермана*, определенная на парах целых неотрицательных чисел.

A(0,n)=n+1; A(m+1,0)=A(m,1);
A(m+1,n+1)=A(m,A(m+1,n)).

Функция Акермана рекурсивная, но вызов функции находится на месте аргумента другого вызова этой функции. В связи этим вычислительные потребности этой функции очень быстро увеличиваются при увеличении параметра m. Так, если A(2,n)=2*n+3, то уже $A(3,n)=2^{n+3}-3$. Благодаря этому свойству функция Акермана используется для проверки способности компиляторов выполнять рекурсию.

```
>(defun akk (m n)
          (cond ((= m 0) (+ n 1))
                ((= n 0) (akk (- m 1) 1))
                (t  (akk (- m 1) (akk m (- n 1)))))
)

>(akk 2 3)
9
>(akk 3 3)  ; переполнение стека
```

Функция Акермана на языке Эрланг имеет следующее определение.

```
-module(recc).
-export([akk/2]).

akk(M,N) when M==0 -> N+1;
```

```
akk(M,N) when N==0 -> akk((M-1),1);
akk(M,N) -> akk((M-1),akk(M,(N-1))).

1> recc:akk(2,3).
9
2> recc:akk(3,3).
61
3> recc:akk(3,10).
8189
```

7.3. Классификация видов рекурсии

Исходя из способов применения операций композиции и рекурсии над вызовами функций, предлагается следующая классификация видов рекурсии: явная и взаимная рекурсии.

Явная (прямая) рекурсия определяется наличием в теле функции явных обращений к ней самой. В общем виде определение функций с прямой рекурсией

```
(defun function (...)
 ..... (function ....)
              (function ...)
                (other ... (function ...) )
)
```

Простая рекурсия содержит один рекурсивный вызов. При этом рекурсивные вызовы могут встречаться в нескольких ветвях (но по одному в каждой ветви). В общем функция для реализации простой рекурсии может иметь следующий вид.

```
(defun fsimple1 (...)
        (cond (p1 GRANICA)
                (p2 (fsimple1 ...))
                .....................
              (pn (fsimple1 ...)))
)
```

Здесь fsimple − рекурсивная функция;

GRANICA − форма, определяющая начальные условия;

p1, p2, ... pn − предикаты, определяющие выполнение соответствующей ветви. Представленный вариант определяет

функцию с рекурсией по значению, так как рекурсивные вызовы определяют результат функции. Для иллюстрации этого варианта рекурсии рассмотрим функцию выделения из списка элемента с заданным номером (аналог функции NTH).

```
>(defun nth1 (x l)      ; функция возвращает элемент списка l,
                        ; находящийся на позиции x от начала списка
  (cond  ((null l) nil)             ; если список пустой, то возвратить NIL
         ((= x 0) (car l))
; если значение позиции x=0, то возвратить голову списка
         (t (nth (- x 1) (cdr l))))
; в противном случае вызвать функцию nth1 y
                ; которой первый аргумент − значение позиции будет
                 ; на единицу меньше, а второй аргумент хвост списка
 )

>(nth1 2 '(a s d f g))  ; выделить второй элемент списка
D                       ; нумерация элементов списка начинается с 0
```

В Эрланге

```
nth1(_,L) when L==[] -> [];
nth1(X,[H|_]) when X==0 -> H;
nth1(X,[_|T]) -> nth1((X-1),T).

2> recc:nth1(2,[a,s,d,f,g]).
d
```

Возможен и другой вариант представления функции с простой рекурсией.

```
(defun fsimple2 (...)
        (cond (p1 GRANICA)
                (p2 (f1 (fsimple2 ...) ... ))
                .....................................
                (pn (fn (fsimplen ...) ...))))
```

Здесь f1 ... fn − функции, результат которых возвращается в качестве значения рекурсивной функции fsimple2, которая участвует в

вычислении аргументов функций f1 ... fn. Такой вариант рекурсии называется о *рекурсии по аргументу,* так как рекурсивные вызовы вычисляют аргументы другой функции, которая, в свою очередь, формирует результат. Следующий пример показывает функцию, вычисляющую сумму элементов списка, состоящего из чисел, и является иллюстрацией для рекурсии по аргументу.

```
>(defun sum (l)                    ; вычисляет сумму элементов списка l
  (cond ((null l) 0)
; если список пустой, то возвращается ноль
        (t (+ (car l) (sum (cdr l))))))
; в противном случае вызвать функцию сложения
; для головы списка и суммы элементов хвоста
>(sum '(1 2 3 4))
10
```

На Эрланге

```
sum(L) when L==[] ->0;
sum([H|T]) -> H+sum(T).
2> recc:sum([1,2,3,4]).
10
```

Рекурсия, определяемая как *параллельная,* предполагает, что в теле рекурсивной функции содержится вызов другой функции, как минимум от двух аргументов. Несколько аргументов этой функции являются вызовами рекурсивной функции.

```
(defun fparallel (....)
      (fother ... (fparallel ...) (fparallel ...) )
. . . )
```

Параллельная рекурсия будет одновременно рекурсией по значению, так как вызовы рекурсивной функции вычисляют аргументы другой функции. Функция вычисления *чисел Фибоначчи* является примером воплощения параллельной рекурсии. Причем, так же как и с определением факториала числа, определение функции на Лиспе полностью повторяет ее математическое определение.

```
( defun fib (n)
        (cond ((= n 0) 0)                              ; начальные условия
              ((= n 1) 1)
              (t         (+ (fib (- n 1)) (fib (- n 2)))))
; рекурсивное соотношение
)
```

На языке Эрланг следующее определение

```
fib(N) when N==0 ->0;
fib(N) when N==1 ->1;
fib(N) -> fib(N-1)+fib(N-2).
2> recc:fib(5).
5
3> recc:fib(2).
1
4> recc:fib(1).
1
5> recc:fib(6).
8
```

Параллельную рекурсию еще называют *рекурсией по дереву*. Она используется для обработки древовидной структуры, т.е. списка, элементы которого могут быть подсписками. Ниже показан пример обработки древовидной структуры – развертывание многоуровневого списка в линейный одноуровневый список.

```
(defun level1 (list)
(cond
; пустой список не требуется развертывать
((null list) nil)
; если аргумент атом – конструировать список из аргумента
((atom list) (cons list nil))
; в противном случае выполнит слияние
(t (append
; развернутой в один уровень головы
(level1 (car list))
; и развернутого в один уровень хвоста
(level1 (cdr list))
```

))))
Пример древовидной структуры
(setq list '(1 (((2))) (3 4 5) (((6)))))

> (level1 list)
(1 2 3 4 5 6)

При *взаимной (неявной или косвенной)* рекурсии функция содержит обращение к себе через цепочку вызовов других процедур. Определение такой рекурсивной функции состоит из определений как минимум двух функций: определения самой рекурсивной функции, содержащей вызов другой (вспомогательной) функции и определения вспомогательной функции, содержащей вызов определяемой (рекурсивной) функции. Отметим, что эта вспомогательная функция является также взаимно рекурсивной. Взаимная рекурсия в общем виде может быть представлена следующим образом;

```
(defun function (...)
        (other1 ....) )
(defun other1 (...)
        (other2 ...) )
. . . . . .
(defun other_i (...)
        (other_n ...))
(defun other_n (...)
       (function ...) )
```

Рассмотрим в качестве примера следующую задачу. Определить предикат, проверяющий, содержит ли список четное число элементов на верхнем уровне, и предикат, проверяющий список на нечетность числа элементов. Декларативное описание задачи следующее:

- пустой список четный;
- у четного списка нечетный хвост;
- у нечетного списка четный хвост

```
>(defun even1 (l)
      (cond
        ((null l) t)                    ; пустой список четный
         ( t     (number1 (cdr l))))) ; у четного списка нечетный хвост
```

```
EVEN1
>(defun number1 (l)
    (cond
       ((null  l) nil)          ; пустой список − не нечетный
        (t (even1 (cdr l))))) ; у нечетного списка четный хвост
NUMBER1
>(even1 '(1 2 3 4))
T
>(even1 '(1 2 3))
NIL
>(number1 '(1 2 3 4))
NIL
>(number1 '(1 2 3))
T
```

На Эрланге следующее определение:

```
even1(L) when L==[] -> true;
even1([_|T]) -> number1(T).

number1(L) when L==[] -> false;
number1([_|T]) -> even1(T).
1> recc:even1([1,2,3]).
false
2> recc:even1([1,2,3,4]).
true
3> recc:number1([1,2,3,4]).
false
```

В случае *рекурсии высокого порядка* аргументом рекурсивного вызова является рекурсивный вызов.

```
(defun function (...)
    (  ...  (function  ...  (function  ...)  ...)  )
)
```

Отметим, что функции с рекурсией высокого порядка относятся к классу общерекурсивных функций, но не примитивно-рекурсивных (п. 9.2.2.). На практике программирование таких функций

встречается редко. С одной стороны, применением сложных рекурсивных определений можно добиться большей выразительности (лаконичными выражениями можно записывать довольно серьезные вычисления). С другой стороны, более затруднено понимание программы, и, как показано в примере с функцией Аккермана, возможно лавинообразное увеличение времени счета и требований к объему оперативной памяти.

В большинстве случаев если в задаче явно присутствуют элементы рекурсии, то соответствующее решение будет состоять в тривиальном переписывании условий проблемы в нотациях конкретного языка программирования.

7.4. Реализация рекурсивного процесса

При выборе способа решения задачи следует обратить внимание на специфику реализации рекурсивного процесса. При его выполнении каждое обращение к функции вызывает независимую активацию этой функции. Совокупность всех данных, необходимых для одной *активации функции*, называется *фреймом активации*, который содержит независимые копии всех локальных переменных и формальных параметров функции. Если при выполнении рекурсивного процесса функция обращается к себе несколько раз, то образуется несколько одновременно существующих активаций. Каждая активация характеризуется своим фреймом активации. Таким образом, при выполнении рекурсивного процесса выделяется несколько стадий:

- *стадия (этап) порождения рекурсивных* вызовов, когда процесс выполнения функции P_i доходит до рекурсивной ветви, запоминается существующий на данный момент фрейм активации F_i, и запускается процесс выполнения этой функции P_{i+1}, но с новым фреймом активации F_{i+1};
- *стадия (этап) возврата* начинается, когда в процессе P_k будут выполнены начальные условия соответствующего фрейма F_k. Функция возвращает значение в предшествующий процесс P_{k-1} и так до тех пор, пока не будут выполнены все отложенные процессы P_i. Значение первого отложенного процесса P_0 будет значением функции.

За выполнением рекурсивного процесса можно проследить введением трассировки рекурсивной функции. Для управления трассировкой в Лиспе введены функции TRACE и UNTRACE.

```
>(TRACE function)
FUNCTION
```

С этого момента при каждом обращении к функции будет выдаваться информация о текущем фрейме активации. Для каждого входа в функцию − значения фактических параметров, для каждого возврата значения функцией − возвращаемые значения

```
>(UNTRACE function) ; отключение трассировки
```

Следующий пример показывает трассировку функции, вычисляющей сумму элементов списка, состоящего из чисел. Сама функция определена в предыдущем параграфе.

```
>(trace sum)                ; включение трассировки функции
SUM
>(sum '(1 2 3))
Entering: SUM, Argument list: ((1 2 3))
;стадия порождения рекурсивных процессов
 Entering: SUM, Argument list: ((2 3))
  Entering: SUM, Argument list: ((3))
   Entering: SUM, Argument list: (NIL)
; выполнены начальные условия
   Exiting: SUM, Value 0
; стадия возврата и первое возвращенное
; значение соответствует начальному условию
 Exiting: SUM, Value 3
Exiting: SUM, Value 5
Exiting: SUM, Value 6                ; возврат значения процессом P1
6                                    ; возврат значения функцией
>(untrace sum)              ; отмена трассировки
```

Наиболее приемлемой структурой данных для организации рекурсивного процесса является стек[3]. При выполнении рекурсии

образуется стек прерванных процессов, состоящий из фреймов активации.

7.5. Хвостовая рекурсия

В достаточно большом числе случаев существует возможность предложить более эффективное решение, воспользовавшись возможностями современных трансляторов преобразовывать рекурсию в итерацию. Ниже представлены два рекурсивных варианта решения одной задачи. Требуется определить число элементов одноуровнего списка.

```
; первый вариант функция определения длины списка
(defun leng (list)
(cond
((null list) 0)
(t (+ 1 (leng (cdr list))))
))
; второй вариант с использованием вспомогательной функции
; ln – основная, нерекурсивная с обращением к функции с накапливающим параметром
(defun ln (list) (ln_l list 0))
; вспомогательная рекурсивная функция определения длины списка с накапливающим параметром
(defun ln_l (list x)
(cond
((null list) x)
(t (ln_l (cdr list) (+ x 1)))))
```

Во втором варианте параметр функции X является аккумулирующей переменной, в которой формируется результат. Значение этой переменной возвращается по окончании рекурсии. Такой вариант реализации называется *хвостовой рекурсией*. Память при этом расходуется только на хранение адресов возврата значения функции. Хвостовая рекурсия представляет собой специальный вид рекурсии, в которой имеется единственный вызов рекурсивной функции, выполняющийся после всех вычислений.

Вычисления хвостовой рекурсии могут быть сведены к итерациям и выполняться в постоянном объеме памяти. Большинство современных трансляторов языков программирования способны распознавать хвостовую рекурсию и реализовывать вычисления такой функции в виде цикла. В программных руководствах в таких случаях внимание программиста может быть специально обращено на использование возможности сведения функции к хвостовой рекурсии.

Приведем основные приемы построения функций с накапливающим параметром:

1. Вводится новая функция с дополнительным параметром (X), в котором формируется результат.

```
(defun ln_l (list x)
(cond
((null list) x)
(t (ln_l (cdr list) (+ x 1)))))
```

2. Выполнение граничных условий во вспомогательной функции соответствует возращению значения накапливающего параметра.

```
((null list) x)
```

3. В рекурсивных определениях накапливающий параметр вспомогательной функции ln_l формирует результат.

```
(t (ln_l (cdr list) (+ x 1)))))
```

4. Основная функция содержит вызов вспомогательной рекурсивной с начальными значениями для накапливающего параметра.

```
(defun ln (list) (ln_l list 0))
```

Построение функций с накапливающим параметром не всегда гарантирует получение хвостовой рекурсии.

7.6. Рекурсивные определения стандартных функций

Базовые функции (примитивы) определяют в функциональном программировании, по сути, всю архитектуру языка (гл. 3). В системах программирования разработчик интерпретатора (компилятора) определяет набор функции для выполнения этих стандартных (часто встречающихся) действий и включает их в систему. Применительно к среде XLISP это означает, что вызовы этих функций можно записывать в программе пользователя, не заботясь о загрузке этих функций в систему. Для задач обработки списков стандартный набор действий предполагает определение следующих функций:

- слияния списков;
- определение принадлежности элемента списку;
- удаление элемента списка;
- обращение списка;
- замена вхождений элемента в список.

Функция слияния списков имеет имя APPEND. *Декларативное описание* задачи слияния списков состоит из следующих условий:

- условие первое: если первый список пустой, то результат слияния − второй список;
- условие второе: если второй список пустой, то результат слияния − первый список;
- условие третье: во всех остальных случаях результирующий список состоит из головы первого списка и результата слияния хвоста первого со вторым.

```
>(defun append (l s)
        (cond ((null l) s)     ; условие первое
                    ((null s) l)    ; условие второе
                    (t (cons (car l) (append (cdr l) s))))) ; условие третье
APPEND
>(append ‘(1 2) ‘(3 4 5))
(1 2 3 4 5)
>(append nil ‘(1 2 3))
(1 2 3)
```

В декларативном описании задачи и функции содержатся два условия, которые можно отнести к начальным условиям. Оба этих условия описывают случаи, когда какой-либо из списков будет пустым. Однако только первое условие на самом деле является начальным для рекурсии. При работе функции рекурсивно откладываются вызовы функции CONS до тех пор, пока не будет исчерпан первый список и в качестве значения функции не будет возвращен указатель на второй список. Это соответствует первому начальному условию. Таким образом, структура функции APPEND была предопределена примитивом CONS и тем, что он строит список слева направо. Таким образом, второе условие в описании функции, в работе рекурсивного процесса вообще не участвует. Второе условие позволит увеличить эффективность решения только в одном случае − когда второй список пустой. Тогда при работе функции для получения решения не придется проходить весь первый список, откладывая вызовы функции CONS. Однако во всех остальных случаях на проверку второго условия будет затрачиваться определенное время. Соответственно решение задачи слияния списков без определения второго условия будет в большинстве случаев более эффективным. Кроме того, функцию слияния списков можно использовать для получения логической копии списка.

```
>(defun append_First (l s) ; определение функции с одним условием
        (cond ((null l) s)      ; условие первое
                    (t (cons (car l) (append_First (cdr l) s)))))
APPEND_FIRST
>(setq w '(1 2 3))
(1 2 3)
>(setq copi (append_first w nil))
(1 2 3)
>(equal copi w)
T
>(defun cop (list) (append list nil))
COP
```

Рассмотренные способы определения функции слияния по классификации относятся к рекурсии по аргументам. Возможно слияние списков и в виде рекурсии по значению. В данном случае первое условие остается без изменений, а второе условие

необходимо сформулировать иначе: если первый список не пустой, то результат слияния состоит из хвоста первого списка и списка, составленного из головы первого списка и второго списка. Особенность такого определения состоит в том, что в результирующем списке первый исходный список будет обращен.

```
>(defun dubap (l s)
        (cond ((null l) s)
                (t (dubap (cdr l) (cons (car l) s)))))
DUBAP
>(dubap '(1 2 3) '(4 5 6))
(3 2 1 4 5 6)
```

Функция DUBAP формирует результат на стадии появления рекурсивных вызовов, и в момент выполнения начальных условий результат сформирован. На всей стадии возврата рекурсии результатами всех рекурсивных вызовов будет одно и то же значение − возвращаемый функцией результат.

Принадлежность элемента списку определяется в XLISP предикатом member. Для декларативного описания задачи принадлежности элемента списку введем следующие правила:

- Правило первое: пустому списку элемент не принадлежит.
- Правило второе: если элемент совпадает с головой списка, то он принадлежит списку.
- Правило третье: если элемент не совпадает с головой списка, то необходимо проверить хвост списка.

```
>(defun memberp (x list)
        (cond ((null list) nil)                  ; первое условие
                ((eql (car list) x) T)           ; второе условие
                (t (memberp x (cdr list)))))  ; третье условие
MEMBERP
>(memberp 6 '(1 2 3 4))
NIL
>(memberp 2 '(1 2 3 4))
T
```

Отметим, что встроенный предикат member определен таким образом, что при обнаружении присутствия элемента в списке он возвращает хвост списка, начинающийся с этого элемента.

```
>(defun member (x list)
      (cond ((null list) nil)                    ; первое условие
            ((eql (car list) x) list)            ; второе условие
            (t (member x (cdr list)))))
MEMBER
>(member 2 '(1 2 3 4))
(2 3 4)
```

Удаление элемента из списка может быть описано следующими декларативными правилами:

- Правило первое: если список пустой, то удалять нечего.
- Правило второе: если удаляемый элемент находится в голове списка, то удалить элемент из хвоста списка.
- Правило третье: если удаляемый элемент отсутствует в голове списка, то составить список из головы и хвоста, в котором элемент, также необходимо убрать.

```
>(defun remov (x list)
  (cond  ((null list)        nil)                      ; правило первое
         ((eql x (car list)) (remov x (cdr list)))     ; правило второе
         (t (cons (car list) (remov x (cdr list))))))  ; правило третье
REMOV
>(remov 3 '(1 2 3 5 3 6 3 4))
(1 2 5 6 4)
>(remov 3 '(1 2 (3 4) 3 5))
(1 2 (3 4) 5)
>(remov '(1 2) '(1 (1 2) 3 4))
(1 (1 2) 3 4)
```

Идея процедурного подхода к определению функции удаления элемента из списка заключается в следующем. Строится новый список путем копирования в него всех списочных ячеек, кроме тех, которые указывают на удаляемый элемент. При этом удаляемый элемент оказывается не включенным в список. Построение нового списка осуществляется рекурсивной ветвью, содержащей функцию

CONS. Ветвь, содержащая условие совпадения элемента с головой списка, обеспечивает при выполнении условия переход к следующему вызову рекурсивной функции без включения совпавшего элемента в новый список. Как видно из приведенного примера, на функцию удаления элемента из списка распространяются те же ограничения, что и на предикат принадлежности элемента списку.

Функция обращения списка изменяет порядок следования элементов в списке. При определении функции обращения целесообразно использовать введенную ранее функцию слияния списков. В этом случае обращенный список можно рассматривать как результат слияния двух списков, первый из которых − обращенный хвост исходного списка. Второй список должен быть составлен из головы первого списка.

```
>(defun revers (list)
    (cond ((null list) nil)
          (t (append (revers (cdr list)) (cons (car list) nil)))))
REVERS
>(revers ‘(1 2 3 4))
(4 3 2 1)
>(revers ‘(1 2 (3 4) 5 6))
(6 5 (3 4 ) 2 1)
```

Более лаконичное определение получится, если воспользоваться вторым определением функции слияния списков. Тогда функция обращения списка будет рассматриваться как результат слияния исходного списка с пустым списком.

```
>(defun dubrev (list)
      (dubap list nil))
DUBREV
>(dubrev ‘(1 2 3 4))
(4 3 2 1)
```

Функция замены старого элемента old на новый new в списке list можно определить с использованием принципов, положенных в основу определения функции удаления элемента из списка.

Построение примитивом CONS копии исходного списка list, в которой вместо элемента old будет находиться новый элемент new.

```
>(defun subs (new old list)
   (cond ((null list) nil)
         ((eql old (car list))
; если старый элемент совпал с головой списка
         (cons new (subs new old (cdr list))))
 ; то составить список, в котором элемент new будет головой,
 ; а хвостом будет хвост исходного списка,
 ; с выполненными заменами
         (t (cons (car list) (subs new old (cdr list))))))
; если старый элемент не совпал с головой списка,
 ; то составить список, в котором голова исходного
 ; списка будет головой результирующего, а хвостом будет хвост
 ; исходного с выполненными заменами
SUBS
>(subs 77 2 '(1 2 3 2 4 2 5))
(1 77 3 77 4 77 5)
```

7.7. Эффективность и критерии выбора рекурсивного решения

Рассмотрим два варианта решения задачи определения факториала числа. В первом варианте предполагает используются форма DO и организации итераций, а второй вариант чисто рекурсивный.

```
(defun fac (n)                          ; итеративный вариант функции
(do ((x 1 (+ x 1)) (R 1))     ; цикл с переменной X
((> x n) R)                  ; условие прекращения итерационного процесса
(setf R (* R x))          ; тело цикла
)
)

(defun fact (n)      ; рекурсивная функция
   (cond ((= n 0)  1)     ; начальные условия
(t         (* n (fact (- n 1))))))  ; тело рекурсивной функции
)
```

В следующей таблице сведены признаки, сопутствующие обоим вариантам решения.

Признаки итерационного варианта функции	Признаки рекурсивного варианта функции
1. Наличие условия прекращения итераций	1. Наличие условие. Прекращение этапа порождения рекурсивных вызовов
2. Наличие тела цикла	2. Наличие тела рекурсии
3. Наличие переменной цикла	3. Аналог отсутствует
4. Присутствие присваивания	4. Аналог отсутствует

Аналогия в определении первых двух признаков показывает эквивалентность данных определений функций. При этом можно считать, что каждая активация рекурсивной функции fact эквивалентна одному из повторений цикла DO. Однако различие в следующих двух признаках указывает на различную природу вычислительных процессов, проходящих при выполнении этих функций. Работа функции в итерационном варианте основана на использовании структуроразрушающего присваивания для изменения значения переменной цикла (счетчика) и локальной переменной R, в которой формируется результат. При этом для формирования результата достаточно двух ячеек памяти на протяжении всего процесса выполнения функции.

В итерационном варианте объем потребляемой вычислительным процессом памяти остается постоянным на протяжении всего вычислительного процесса.

Совершенно иная ситуация возникает при выполнении рекурсивной функции. В рекурсивном процессе при каждой активации образуется новый и независимый фрейм активации со своими значениями локальных переменных и формальных параметров и других вспомогательных данных.

В рекурсивном варианте объем потребляемой вычислительным процессом памяти увеличивается на протяжении всего этапа порождения рекурсивных вызовов.

При использовании рекурсивного варианта функции необходимо учитывать возможность переполнения стека рекурсивных вызовов. Соответственно различие по третьему и четвертому признаку указывают на меньшую эффективность рекурсивного вычислительного процесса по критерию затрат памяти.

Реализация запроса на выделение памяти для фрейма активации, кроме расхода собственно памяти, требует и дополнительного времени процессора. Это время расходуется на действия, связанные с захватом памяти фреймов (на этапе порождения рекурсивных вызовов) и освобождением памяти фреймов (на стадии возврата рекурсии). Рекурсивный процесс будет также менее эффективным и по критерию затрат времени процессора.

Приведенные соображения эффективности рекурсивного и итерационного процессов (по критериям затрат памяти и процессорного времени) будут общими и для императивного, и для функционального программирования. Однако в функциональном программировании на критерий затрат памяти следует обратить большее внимание: при работе функциональной программы генерируются новые структуры и неизбежно появляется мусор (гл. 4).

Однозначных критериев выбора между рекурсивным и нерекурсивным способами решения задачи не существует. Несмотря на отсутствие однозначных критериев выбора, можно предложить ряд эвристических правил:

- при одинаковой сложности рекурсивной и нерекурсивной версий алгоритма предпочтение следует отдавать нерекурсивной версии, как более эффективной по затратам памяти и процессорного времени (пример с функциями обращения списка);
- если применение нерекурсивной версии алгоритма значительно усложняет его логику или требует применения более сложных структур данных, то предпочтение следует отдать рекурсивному алгоритму.

Примечания

1. Фрактал − это бесконечно самоповторяющаяся геометрическая фигура, самоподобное множество нецелой размерности. Различают геометрические (кривая Коха, снежинка Коха, кривая Гильберта, ломаная Дракона), алгебраические (множество Мандельброта, множество Жюлиа, Бассейны Ньютона, биоморфы), стохастические фракталы.
2. Слова «рекуррентный» и «рекурсия» произошли от латинского *recurro*, что в переводе означает «бежать назад», «возвращаться».
3. *Стек* – линейный список, все записи в котором выбираются, вставляются и удаляются с одного конца. Это предполагает обеспечение доступа к записям по принципу «последним вошел − первым вышел».

Резюме

Для рекурсивного решения необходимо:

- наличие известного решения задачи (начальные условия);
- реккурентных зависимостей, позволяющих свести все остальные случай к известному решению;
- выполнение рекурсивной функции проходит в два этапа (этап порождения отложенных вызовов и этап возврата);
- рекурсивное решение при прочих равных условиях менее эффективно, но более выразительно;
- метод накапливающего параметра не всегда приводит к хвостовой рекурсии, однако он однозначно помогает уменьшить общий объём памяти

Упражнения

1. Что необходимо для формирования рекурсивного решения?
2. Какие функции относятся к примитивно рекурсивным функциям?
3. Какие функции относятся к общерекурсивным функциям?
4. Как будут меняться потребности в памяти при выполнении общерекурсивной функции?
5. Какие базовые функции Лиспа можно отнести к нуль-функциям, функциям проецирования и функциям последования?
6. Решить следующие задачи двумя способами: рекурсивно и с помощью механизмов организации циклов Лиспа. Сравнить оба способа по следующим критериям: вычислительной эффективности и выразительности. Выполнить трассировку рекурсивной функции и проследить за выполнением рекурсивного процесса. Все программы выполнять в инструментальной среде Лиспа и Эрланга.

Действия, которые должна выполнять функция « name»	Примеры применения (для Лиспа)
Переводить число из десятичной системы счисления в двоичную	>(name ‘(2 7)) (1 1 0 1 1)
Возвращать «n» элемент списка от начала и от конца	>(name 3 ‘(a d g h j)) g >(namec 2 ‘(a d g h j)) h
Подсчитывать сумму всех нечётных элементов	>(name ‘(1 2 3 2))

списка (по месту нахождения)	4
Подсчитывать произведение всех четных элементов списка (по месту нахождения)	>(name ‘(1 2 3 2)) 4
Выдавать отсортированный в порядке возрастания список, состоящий из чисел, исключая повторы Действия, которые должна выполнять функция “ name”	>(name ‘(1 4 3 3 6 2)) (1 2 3 4 6) Примеры применения (для Лиспа)
Выдавать отсортированный в порядке убывания список, состоящий из чисел	>(name ‘(1 4 3 5 6)) (6 5 4 3 1)
Проверять, является ли отсортированным в порядке возрастания список, состоящий из чисел	>(name ‘(1 4 3 5 6)) nil >(name ‘(1 2 3)) t
Проверять, является ли отсортированным в порядке убывания список, состоящий из чисел	>(name ‘(1 4 3 5 6)) nil >(name ‘(3 2 1)) t
Добавлять число в упорядоченный в порядке возрастания список без нарушения порядка	>(name 7 ‘(1 4 6 10 11)) (1 4 6 7 10 11)
Объединять два упорядоченных списка, состоящих из чисел, в упорядоченный список	>(name ‘(2 4 7 9 13 16) ‘(2 5 6 11 12)) (2 2 4 5 6 7 9 11 12 13 16)
Подсчитывать количество разных элементов списка	>(name ‘(a1 a2 a3 a1 a2 a4)) 4
Подсчитывать сумму разных элементов списка	>(name ‘(1 2 3 1 2 3)) 6
Формировать список, в котором каждый элемент является подсписком уровня, соответствующего его месту.	>(name ‘(a w q e)) (a (w) ((q)) (((e))))
Формировать список, в котором каждый элемент является подсписком уровня, соответствующего его месту, и в котором каждый элемент встречается не более одного раза	>(name ‘(a w a q q e e)) (a (w) ((q)) (((e))))
Формировать список, являющийся объединением двух списков, за исключением элементов, встречающихся в обоих списках	>(name ‘(a s d f g) ‘(q w a s)) (d w f g)
Формировать список состоящий из элементов, которые встречаются в обоих исходных списках	>(name ‘(a s d f g h) ‘(q a w s e d r)) (a s d)

При сравнении рекурсивного и нерекурсивного методов решения задачи можно использовать оценки по таким параметрам

решения, как затраты памяти или процессорного времени (для эффективности), а также размер исходного кода (для выразительности). Предполагается, что рекурсивное решение должно быть функциональным.

ГЛАВА 8

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИЙ ВЫСШЕГО ПОРЯДКА

Понятие функции высшего порядка
Применяющие функционалы Лиспа
Отображающие функционалы
Функций высшего порядка в Эрланге
Лексические замыкания

В предыдущих главах учебного пособия каждая функция рассматривалась как статическая часть кода для преобразования входных данных в выходные (концепция программирования первого порядка). В этой главе будут рассмотрены основные принципы программирования с использованием функций высшего порядка (функционалов), а также показано, каким образом такое программирование может быть использовано для представления общих образцов рекурсии и для выражения рекурсивных функций через нерекурсивное применение функций высшего порядка [11,12,13].

8.1. Понятие функции высшего порядка

В функциональных языках программирования предполагается, что функции должны иметь тот же статус, что и любой объект данных и быть как входными, так и выходными данными других функций (программирование высшего порядка). В предыдущих главах каждая функция рассматривалась как статическая часть кода для преобразования входных величин в выходные. При этом предполагалось, что и входные величины, и выходные представляемые в виде выражений, являются данными. Функции, получающие в качестве аргумента или возвращающие в качестве значения выражения, относящиеся к данным, называются *функциями первого порядка.* Соответственно программирование с помощью таких функций называется *программированием первого порядка.* Как

отмечалось, одной из основных идей функционального программирования является отсутствие существенных различий между программами и данными (см. введение). На этом свойстве функциональных языков базируется *концепция функций высшего порядка*. Исходя из этой концепции, принимают, что функции должны иметь тот же статус, что и любой объект данных. Следовательно, функции сами могут быть входными и выходными данными других функций (значениями функций и аргументами функций). Аргумент, значением которого является функция, называется *функциональным аргументом*. Функция, которая в качестве результата возвращает также функцию, называется *функцией с функциональным значением.*

Функции, которые в качестве параметров используют другие функции или возвращают их как результат своей работы, называются функциями высшего порядка (*функционалами*).

При этом программирование с использованием функций высшего порядка можно рассматривать как один из способов композиции функций (наряду с обыкновенным вызовом в теле функции, рекурсивным вызовом, вложенным рекурсивным вызовом).

8.2. Применяющие функционалы Лиспа

В Лиспе одно и то же S-выражение может выступать и как обыкновенный аргумент функции, и как функциональный аргумент (гл. 5). Конкретное назначение аргумента будет определяться его синтаксической позицией. В Коммон Лиспе переданный функции в качестве параметра функциональный объект можно использовать только через явный вызов специальных функций, называемых *применяющими функционалами* [13]. Применяющие функционалы дают возможность преобразовывать данные в программу и применять ее в вычислениях. В Коммон Лиспе к применяющим функционалам относятся функции APPLY и FUNCALL.

Функция APPLY в общем виде представляется выражением следующего вида:

```
(APPLY Funct List)
```

Здесь Funct − функция, которая является первым аргументом APPLY; List − список, к элементам которого применяется функция Function

Если List == (t1 t2 ... tn) то (APPLY Function List) == (Function ‘t1 ‘t2 ... ‘tn)

```
>(apply ‘member ‘(1 (2 1 3)))
(1 3)
>(apply ‘append ‘((1 2 3) (4 5 6)))
(1 2 3 4 5 6)
>(apply ‘+ ‘(1 2))
3
>(apply ‘reverse ‘((1 2 3 4)))
(4 3 2 1)
>(setq s ‘car)
CAR
>(apply s ‘((1 2 3)))
1
>(setq d ‘cdr)
CDR
>(apply d ‘((1 2 3 )))
(2 3)
```

Как видно из примеров, использование функционала APPLY позволяет в зависимости от значения функционального аргумента Function выполнять различные вычисления. При этом вызываться будет только одна функция APPLY. Однако этим значение функции APPLY не ограничивается.

Функция APPLY, наряду с функцией EVAL, составляет основу так называемого энергичного интерпретатора функциональных языков программирования (APPLY/EVAL интерпретатор). Функция EVAL вычисляет значение произвольного выражения, а функционал APPLY вычисляет значение вызова функции, являющейся его аргументом.

Другой системно определенный функционал Лиспа − функционал FUNCALL, действует аналогично APPLY, но аргументы для вызываемой функции принимает по отдельности. В общем виде использование функционала FUNCALL записывается следующим образом.

(FUNCALL function t1 t2 ... tn) ,

что равносильно следующему применению функции funct:

(funct t1 t2 ... tn)

```
>(funcall ‘member 2 ‘(1 2 3 4))
(2 3 4)
>(funcall ‘append ‘(1 2 3) ‘(4 5 6))
(1 2 3 4 5 6)
>(funcall ‘reverse ‘(1 2 3 4))
(4 3 2 1)
```

Функционал FUNCALL можно использовать для обобщения вычислений. Так, с помощью определенной ниже функции S можно выполнить вычисления с любой функцией от двух аргументов. В качестве параметров вызова для функционала S будет функция, определяющая вычисления и ее аргументы.

```
>(defun s (fun x l)
; fun − функциональный аргумент
; x и l аргументы функции fun
      (funcall fun x l))
S                                    ; S − функционал
>(s ‘member 2 ‘(1 2 3 4))
(2 3 4)
>(s ‘append ‘(1 2 3) ‘(4 5 6))
(1 2 3 4 5 6)
```

Отметим, что первый аргумент функционалов − выражение, представляющее собой здесь функциональный объект, но вычисляется этот аргумент по обычным правилам. Следовательно, если имя объекта указано без блокировки вычислений, то в качестве значения объекта берется значение, определенное формой SET. Поэтому во всех предыдущих примерах имя функции указывается в кавычках.

```
>(funcall car ‘(1 2 3 4))
error: unbound variable – CAR
```

Этим свойством можно воспользоваться, чтобы применять имена функций как обыкновенные переменные. Тогда имя одной функции может хранить другую функцию, и в зависимости от условий применения будут доступны оба значения.

```
>(setq car ‘cdr)
CDR
>(funcall car ‘(1 2 3 4))
(2 3 4)
>(car ‘(1 2 3 4))
1
>(setq car ‘car)
CAR
>(funcall car ‘(1 2 3 4))
1
```

Применение функционалов увеличивает выразительность программы, а в некоторых случаях и эффективность, по сравнению с программами при рекурсивном решении. Для этого существуют отображающие функционалы.

8.3. Отображающие функционалы Лиспа

Большой класс функций высшего порядка составляют функции, которые часто называются map, поскольку они определенным образом отображают конкретный элемент списка. Они еще называются *отображающими функционалами*. Список состоит из головы и хвоста, поэтому среди map-функций, входящих в множество базовых функций Лиспа, есть функции, которые применяют свой функциональный аргумент последовательно либо к головам списка, либо к его хвостам. Следующая функция применяет свой первый (функциональный) аргумент к элементам списка, являющегося вторым аргументом.

(mapcar <fn> List)

здесь

- <fn> функция или имя функции;
- list⇔ ‘(e1 e2 … en) список, представляющий второй аргумент.

```
>(mapcar ‘(lambda (x) (+ 1 x)) ‘(1 2 3))
(2 3 4)
>(mapcar ‘reverse ‘((1 2) (3 4) (5 6))
((2 1) (4 3) (6 5))
; определяя функцию вычисления среднего значения
>(defun sr (x y)
(/ (+ x y) 2.0))
SR
>(mapcar ‘sr ‘(1 2 3) ‘(5 6 7))
(3 4 5)
```

Функция выполняет действия над элементами списков, которых может быть любое число (в соответствии с числом аргументов функции). Функцию, выполняющую действия, аналогичные функции mapcar, можно определить с использованием применяющего функционала и механизма рекурсии.

```
>(defun mapcaw (fun l)
(cond ((null l) nil)
(t (cons (funcall fun (car l)) (mapcaw fun (cdr l))))))
>(mapcaw ‘(lambda (x) (sqrt x)) ‘(1.0 2.0 4.0))
(1 1.414214 2)
```

Однако в этом случае вычислительная эффективность полученного решения будет меньше.

Вместе с функционалом mapcar, в Коммон Лиспе существует базовая функция maplist, которая выполняет действия, предписанные ее первым (функциональным) аргументом, последовательно к «хвостам» списка, являющегося ее вторым аргументом.

(maplist <fcn> <list1> <list>...)

```
>(defun su (l)
(cond ((null l) 0)
(t (+ (car l) (su (cdr l))))))
SU
; SU определяет сумму элементов списка
>(maplist ‘su ‘(1 2 3 4))
```

```
(10 9 7 4)
>(maplist ‘cdr ‘(1 2 3 4))
((2 3 4) (3 4) (4) NIL)
```

Рекурсивный аналог функции maplist можно определить следующим образом:

```
>(defun mapca (fun l)
(cond ((null l) nil)
(t (cons (funcall fun l) (mapca fun (cdr l))))))
MAPCA
>(mapca ‘reverse ‘(1 2 3 4))
((4 3 2 1) (4 3 2) (4 3) (4))
```

Применением функций maplist и mapcar можно увеличить выразительность программных конструкций при записи повторяющихся вычислений.

8.4. Функции высшего порядка в Эрланге

Разница в реализации динамической типизации на Лиспе и Эрланге, наиболее явно проявляется при обращении к функциям высокого порядка. В Лиспе существует возможность передать функционалу уже определенную, имеющую имя (атом) функцию. С помощью средств управления контекстом удается указать, что в данном случае под атомом (именем функции) понимается функциональный объект.

```
>(mapca ‘reverse ‘(1 2 3 4))
; блокирована связь атома reverse со значением, определяемым фор;мой SET  и предполагается связь с функциональным объектом
((4 3 2 1) (4 3 2) (4 3) (4))
```

В Эрланге аргументы передаются образцами, и атом в таком качестве не может представлять имя другой функции. В таком случае приходиться использовать функциональные объекты, которые передаются в качестве аргументов (гл. 5). Ниже показаны определения функций высокого порядка, аналогичных функциям mapcar и maplist Лиспа.

```
- module(higs).
- export([maplist/2,mapcar/2]).
% Fun – функциональный аргумент
mapcar(Fun, [First|Rest]) ->
   [Fun(First)|mapcar(Fun,Rest)];
mapcar(Fun, []) ->
   [].

maplist(Fun,[])->[];
maplist(Fun,[Head|Tail]) ->
   [Tail|maplist(Fun,Tail)].

1> c(higs).
./higs.erl:7: Warning: variable 'Fun' is unused
./higs.erl:10: Warning: variable 'Fun' is unused
./higs.erl:11: Warning: variable 'Head' is unused
{ok,higs}
2> Inc=fun(X)->X+1 end.
#Fun<erl_eval.6.43886099>
3> Cdr=fun([_|Tail])->Tail end.
#Fun<erl_eval.6.43886099>
4> higs:mapcar(Inc,[1,2,3,4]).
[2,3,4,5]
5> higs:maplist(Cdr,[1,2,3,4]).
[[2,3,4],[3,4],[4],[]]
6> Inc(3).
4
7> Cdr([1,2,3,4]).
[2,3,4]
```

8.5. Лексические замыкания

Основу энергичного интерпретатора Лиспа составляют две функции: EVAL и APPLY. С позиций λ-исчисления (гл. 9) назначение функций, составляющих основу энергичного интерпретатора, можно сформулировать следующим образом. EVAL вычисляет значение аргумента, преобразуя его к нормальной форме, а

APPLY вычисляет значение применения функции с помощью преобразования этого применения к нормальной форме. Построенный таким образом интерпретатор называется EVAL/APPLY. Впервые такой интерпретатор был описан создателем языка Лисп Дж. Маккарти.

EVAL/APPLY реализует применение функции (β-правило редукции), оставляя тело функции в неприкосновенности и запоминая выражение, которое должно заменять связанную переменную в специальной структуре данных, называемой *контекстом* [11]. Контекст[1] обеспечивает связь между именами (идентификаторами связанных переменных) и выражениями. Сформированный на время вычисления функции контекст после окончания вычисления пропадает. Однако существуют задачи, в которых необходимо сослаться на более ранний вычислительный контекст. К таким задачам относится, например, программирование генераторов. *Генератором* называется функциональный объект, каждый вызов которого порождает очередное значение некоторого ряда. Генератор порождают значения только при необходимости.

В основе программирования генераторов лежит понятие «лексического замыкания»[2]. *Лексическим замыканием* (closure) называется функциональный объект, состоящий из функции и контекста. В Коммон Лиспе замыкание создается формой FUNCTION. Отметим, что в замыкании из всего контекста определения функции включаются только связи свободных переменных. Если в функции нет свободных переменных, то форма FUNCTION эквивалентна форме QUOTE.

```
>(setq y 10)
10
>(setq clo1 (function (lambda (x) (cons x y))))
 #<Closure:  #3d4f0220>
 ; замыкание организовано - организован
 ; частично вычисленный контекст
>(funcall clo1 6)
(6 . 10)
>(setq y 3)
; изменение значения свободной переменной
3
>(funcall clo1 6)
```

```
(6 . 3)
; в XLISP
>(funcall clo1 6)
;  в Коммон Лиспе связи свободных переменных заморожены
; на момент организации замыкания
(6 . 10)
```

В этом примере видно, что замыкания позволяют осуществить частичные вычисления. Для продолжения вычислений следует передать значения недостающих параметров, в предшествующем примере − применение (funcall clo1 6). Следующий пример показывает определение генератора, порождающего список из своего аргумента.

```
>(defun Slist (x)
  (function (lambda () (setq x (cons x nil)))))
>(setq s (slist 5))
#<Closure:  #3d4f0220>
>(setq w (slist 10))
#<Closure:  #3d4f0162>
>(funcall s)
(5)
>(funcall s)
((5))
>(funcall w)
(10)
>(funcall w)
((10))
```

Обратим внимание на отличия лексических переменные замыкания от глобальных переменных программы и переменных в объектах: лексические переменные не занимают глобальное пространство имён, они привязаны к функциям, а не объектам.

Возможности функций высокого порядка с функциональным значением определенным образом связаны с возможностями *карринга* и выполнения отложенных вычислений. Карринг позволяет связывать аргументы с функцией и ожидать, пока остальные аргументы не появятся позже. Ниже показано, как на Лиспе можно имитировать ленивые вычисления.

```
(defun f (x)
        (defun s (y)
(if  (= x y) 17
; если первые два аргумента равны, то результат равен 17
                (defun q (z)
 (/ (+ x y) z)))))
> (f 10)
S
> (s 10)
17
; получен результат
; передача третьего параметра и вычисление (/ (+ x y) z)
; не производится
> (q 2)
error: unbound function – Q
; Q в данном применении не понадобилась и не определялась
> (f 5)
S
> (q 2)
error: unbound function – Q
; аргументы связываются заново
> (f 5)
S
> (s 1)
Q
> (q 2)
3
```

Примечания

1. В общем, под вычислительным контекстом понимается совокупность действующих связей переменных с их значениями.
2. В определенном смысле замыкания можно назвать понятием противоположным понятию объекта в объектно-ориентированном программировании. Объект – данные с присоединенными методами (функциями обработки), а замыкания – функции с присоединенными данными.

Резюме

- Применение функций высшего порядка повышает выразительность решения.
- В Липе функции высшего порядка получаются применением системно определенных функционалов Apply и Funcall.
- Map − функции отображают каждый элемент списка в соответствии с действиями, предписанными функциональным аргументом.
- Замыкание − функция, определенная в теле другой функции. Вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции
- Замыкание создаётся каждый раз во время выполнения.
- Замыкания позволяют в определенной мере имитировать отложенные вычисления на Лиспе.

Упражнения

1. В примере создания замыкания есть следующий фрагмент

```
>(funcall clo1 6)
(6 . 3)
; в XLISP
>(funcall clo1 6)
; в Коммон Лиспе связи свободных переменных заморожены
; на момент организации замыкания
(6 . 10)
```

Что можно сказать про характер связывания в Коммон Лиспе и XLISP на основании данного примера?

2. Составить с помощью применяющих или отображающих функционалов функции для выполнения следующих действий:

Подсчитывать сумму всех нечётных элементов списка (по месту нахождения)	>(name ‘(1 2 3 2)) 4
Подсчитывать произведение всех четных элементов списка (по месту нахождения)	>(name ‘(1 2 3 2)) 4
Предикат, проверяющий является ли отсортированным в порядке возрастания список, состоящий из чисел	>(name ‘(1 4 3 5 6)) nil >(name ‘(1 2 3)) t
Объединяет два упорядоченных списка, состоящих из чисел, в упорядоченный список.	>(name ‘(2 4 7 9 13 16) ‘(2 5 6 11 12))

	(2 2 4 5 6 7 9 11 12 13 16)
Подсчитывает сумму разных элементов списка	>(name ‘(1 2 3 1 2 3)) 6
Список, в котором каждый элемент является подсписком уровня, соответствующего его месту.	>(name ‘(a w q e)) (a (w) ((q)) (((e))))

3. Сравнить полученные решения с рекурсивными функциями по эффективности и выразительности. При сравнении использовать результаты трассирования функций.

ГЛАВА 9

МАТЕМАТИЧЕСКИЕ ОСНОВЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

Теория λ-исчисления
Теория рекурсивных функций

Функциональное программирование основано на математических теориях – теории λ-исчисления Черча и теории рекурсивных функций. Обе эти теории представляют собой разные подходы к формализации алгоритмов [1,5,9,10,11]. В этой главе представлены основы, понятия и определения этих теорий, рассмотрена значимость математической логики для теории алгоритмов и практического программирования.

9.1. Теория λ-исчисления

Основой для функционального программирования являются родственные алгоритмические системы (гл. 2). Хронологически первой была система, предложенная советским математиком М. И. Шейнфинкелем в 1924 г. В этой системе функция рассматривалась как композиции базовых *«комбинаторов»* и была сделана попытка найти минимальный набор таких комбинаторов, пригодный для определения любой функции [1,4]. Исчисление комбинаторов было независимо разработано и глубоко исследовано Хаскелем Карри и получило наименование «*комбинаторной логики*». Однако более известно $\lambda-$*исчисление* А. Черча (1937 г.). Каждую функциональную программу можно преобразовать в эквивалентные λ-выражения. Соответственно теоретические достижения λ-исчисления можно экстраполировать на функциональное программирование в целом [11].

В параграфе 9.1 вводятся основные положения λ-исчисления и показывается аналогия между предопределенными константами $\lambda-$исчисления с набором примитивов функциональных языков программирования.

Разбор правил преобразования λ-выражений, в котором будут введены основные правила преобразования, описывается в параграфе 9.2. Здесь же будет показано, что основное правило λ-исчисления эквивалентно подстановке аргумента вызова в тело функции.

Наиболее значимые для функционального программирования результаты, связанные с введением понятия «нормальная форма» (аналог завершения вычислений в программировании), относятся к вопросам единственности решения и выбора способов получения результата (основные теоремы λ−исчисления), будут показаны в п. 9.3.

В параграфе 9.4 будет показано представление всех объектов (в том числе и предопределенных констант) в чистом λ-исчислении. Тем самым будет представлена математическая основа декларируемого в функциональном программировании принципа равноправия программ и данных.

Отметим, что λ-исчисление является не единственной алгоритмической основой функционального программирования. Основой языка программирования Рефал являются *нормальные алгоритмы* Маркова.

9.1.1. Основные понятия и синтаксис λ-исчисления

Лямбда–выражение в Лиспе представляет собой определение анонимной функции (гл. 5). Лямбда-выражение, определяющее функцию для вычисления квадрата числа в Лисповской нотации, представлено следующим S-выражением:

(lambda (x) (* x x)) (1)

Эквивалентное функции Лиспа (1) выражении в нотации λ-исчисления Черча выглядит следующим образом:

λ**x. (* x x)** (2)

выражения в форме (2) называются λ*-абстракциями.* Соответственно такое выражение может быть прочитано следующим образом: функция от x, которая возвращает (* x x). Символ x после λ в выражении (2) называется *связанной переменной абстракции,* а выражение справа от точки называется *телом абстракции.* Понятие

«связанная переменная» λ-абстракции полностью соответствует понятию «формальный параметр» функции в программировании. Каррингова форма представления функций является основной в нотации λ-исчисления, поэтому все абстракции включают только один символ λ и одну связанную переменную. Тогда в общем виде λ-выражение представляется следующим образом:

$$(\ldots(((\lambda t_1.\ \lambda t_2.\ \ldots\ \lambda t_n.S)\ t_1)\ t_2)\ \ldots\ t_n) \qquad (3)$$

По существующему в карринге соглашению функция применяется всегда к самому левому аргументу и скобки вокруг каждого вложенного применения опускают, поэтому общепринятая форма имеет следующий вид:

$$(\lambda t_1.\ \lambda t_2.\ \ldots\ \lambda t_n.S)\ t_1 t_2\ \ldots\ t_n \qquad (4)$$

Обратим внимание на то, что выражения (3) и (4) определяют лямбда-вызовы, т.е. применение λ-выражения $(\lambda t_1.\ \lambda t_2.\ \ldots\ \lambda t_n.S)$ к фактическим параметрам $t_1 t_2\ \ldots\ t_n$.

Используемое в функциональном программировании λ-исчисление, снабжается набором предопределенных констант, по аналогии с системой примитивов языков программирования. В принципе набор констант может быть произвольным.

Снабженное набором предопределенных констант λ-исчисление, в дальнейшем будем называть расширенным λ-исчислением. Соответственно λ-исчисление без набора предопределенных констант будет называться чистым λ-исчислением.

Первоначально появилось именно чистое λ-исчисление. В следующей таблице приведен примерный набор констант для расширенного λ-исчисления. Для сохранения преемственности с синтаксисом Лиспа имена констант соответствуют именам аналогичных примитивов Лиспа.

Константы	Значение
0, 1, -1, 2, ...	Множество целых чисел
T, NIL	Булевы константы
+,-,*,/	Арифметические операции
<,>,=	Предикаты, определенные на множестве целых чисел

ATOM	Функция проверки типа аргумента
NIL	Пустой список
CONS	Конструктор списков
CAR	Функция, возвращающая голову списка
CDR	Функция, возвращающая хвост списка
NULL	Функция, возвращающая T, если ее аргумент пустой список
COND	Условная функция

9.1.2. Правила вывода в λ-исчислении

Правила вывода описывают порядок вычисления λ-выражения, а следовательно, порядок получения его конечного значения из первоначального вида. λ-выражение вычисляется посредством редукций. Редукцией называется процесс упрощения λ-выражения. Существуют следующие типы редукций.

Процесс применения константной функции называется δ-правилами редукции. Для записи применения функций в λ-исчислении используется префиксная нотация, что соответствует нотации языка Лисп (без записи выражения в скобках). Далее следует примеры редукции с использованием δ-правил.

+ 2 3 $\rightarrow_\delta$ 5 применение константной функции + к числам 2 и 3;

/ (+ 5 5) (- 6 4)
$\rightarrow_\delta$ / (+ 5 5) 2
$\rightarrow_\delta$ / 10 2
$\rightarrow_\delta$ 5

COND (= 2 (- 6 4)) $\rightarrow_\delta$
$\rightarrow_\delta$ COND (= 2 2)
$\rightarrow_\delta$ T

Следующее правило редукции описывает порядок применения λ-абстракции. В программировании аналогичная ситуация соответствует вызову функции с фактическими параметрами, замещающими формальные параметры. В λ-исчислении физически заменяются все вхождения связанной переменной в теле λ-абстракции на выражения, представляющие аргумент.

Процесс копирования тела λ-абстракции с заменой всех вхождений связанной переменной на выражения аргумента называется β-редукцией.

Редукция заключается в том, что чисто текстуально убирают символ λ, связанную переменную и выражение аргумента. При этом получается измененная форма тела λ-абстракции.

(λ **x.+ x x) 5** $\rightarrow \beta$; замена связанной переменной **x** на значение аргумента + 5 5 $\rightarrow \delta$ 10.

Следующий пример преобразования λ-выражения аналогичен примеру выполнения каррингговой функции главе 5.

(((λ **x.** λ **y.** λ **z. (/ (+ x y) z)) 5) 5) 2)** $\rightarrow \beta$
((λ **y.** λ **z. (/ (+ 5 y) z)) 5) 2)** $\rightarrow \beta$
(λ **z. (/ (+ 5 5) z) 2)** $\rightarrow \beta$
(/ (+ 5 5) 2) $\rightarrow \delta$
(/ 10 2) $\rightarrow \delta$
5

Следует учитывать, что в выражении (3) на месте любого из аргументов $t_1 t_2 \ldots t_n$ также может находиться λ-выражение. Тогда при проведении β-редукции полученное выражение может быть чисто текстуально сложнее, хотя математически оно будет проще.

Выражение, которое может быть подвергнуто редукции называется редексом (от английского reducible expression ⇔ redex).

В соответствии с типом правила, которое может быть применено при редукции, различают β-редекс и δ-редекс.

При проведении β-редукции возможна так называемая ситуация *конфликта имен.* Она возникает, если связанная переменная абстракции имеет такое же имя, как заменяемая при редукции переменная. Так, в следующем выражении символ x в теле внутренней абстракции λx.x и тот же символ в выражении аргумента представляют разные переменные.

λx.(λx.x) (- 10 x)

В этом случае перед проведением β-редукции следует переименовать одну из переменных − выполнить α-преобразование.

Выражения, подвергнутые α-преобразованию, называются *алфавитно-эквивалентными.* Приведем пример преобразования λ-выражения.

(λ f. λ s. λ q. f Sq) (λ x. λ y.* x y) 5 6 $\rightarrow \beta$
(λ s. λ q.(λ x. λ y.* x y) s q) 5 6 $\rightarrow \beta$
(λ q.(λ x. λ y. * x y) 5 q) 6 $\rightarrow \beta$

Последнее выражение содержит два редекса. Первый редекс – **(λ x. λ y. * x y) 5, а** второй редекс – все выражение **(λ q.(λ x. λ y. * x y) 5 q) 6.** Соответственно возможны и два варианта дальнейшего преобразования:

Первый вариант преобразования

(λ q.(λ y. * 5 y) q) 6 $\rightarrow \beta$
(λ **y.* 5 y) 6** $\rightarrow \beta$
*** 6** $\rightarrow \delta$
30

Второй вариант преобразования

(λ x. λ y. * x y) 5 6 $\rightarrow \beta$
(λ y. * 5 y) $\rightarrow \beta$
*** 6** $\rightarrow \delta$
30

9.1.3. Теоремы λ-исчисления

Если к λ-выражению нельзя применить никакое правило редукции, то оно находится в *нормальной форме.* Нормальная форма соответствует понятию конца вычислений в программировании. Примеры нормальных форм:

55

λ x.x

Трудности возникают, если λ-выражение содержит несколько редексов и соответственно появляется несколько разных путей преобразования. При этом необходимо ответить на два вопроса:

1) какой редекс преобразовывать в первую очередь?

2) будут ли эквивалентными нормальные формы, полученные при редукции по каждому из путей преобразования?

Ответ на эти вопросы − одна из основных задач λ-исчисления.

В λ-исчислении доказаны теоремы дающие нужный ответ. Прежде чем перейти к изложению теоремы, отвечающей на первый вопрос, необходимо ввести ряд определений:

Самым левым редексом λ-выражения называется редекс, символ λ которого текстуально расположен левее всех остальных редексов.

Самым внешним редексом называется редекс, который не содержится внутри никакого другого редекса.

Самым внутренним редексом называется редекс, не содержащий других редексов.

Порядок редукций, при котором преобразовывается самый левый из самых внутренних редексов, называется аппликативным порядком редукций (АПР).

Порядок редукций, при котором преобразовывается самый левый из самых внешних редексов, называется нормальным порядком редукций (НПР).

Актуальность выбора порядка редукций иллюстрируется следующим примером.

(λx. λy.y)((λz.z z) (λz.z z))

Это λ-выражение содержит несколько редексов:

- редекс 1 - −**λx. λy.y)((λz.z z) (λz.z z))**
- редекс 2 - −**λz.z z) (λz.z z)**

Преобразование первого редекса будет соответствовать выбору нормального порядка редукции, и преобразования будут завершены за один шаг.

(λx. λy.y)((λz.z z) (λz.z z)) $\rightarrow \beta$ **λy.y** − нормальная форма.

При выборе преобразования второго редекса, что равносильно аппликативному порядку редукций, нормальная форма не будет достигнута никогда:

(λz.z z) (λz.z z) $\rightarrow \beta$

(λz.z z) (λz.z z) $\rightarrow \beta$ **. . .**

В первом случае НПР откладывает вычисление редексов внутри выражения аргумента до тех пор, пока это выражение не станет необходимым для дальнейшего продолжения вычислений и

выполняет подстановку невычисленного выражения в тело λ-выражения, в расчете на то, что выражение аргумента не понадобиться в дальнейшем. В результате за один шаг получилась нормальная форма λу.у. Применение аппликативного порядка редукций приводит к зацикливанию.

Теорема стандартизации. Если λ-выражение имеет нормальную форму, то редукция самого левого из самых внешних редексов на каждом этапе вычисления гарантирует достижение этой (с точностью до алфавитной эквивалентности) нормальной формы.

Из теоремы стандартизации следует, что необходимо выбирать именно НПР. Однако реализация аппликативного порядка редукций оказывается более эффективной на обычных (фон-неймановских) компьютерах. Большинство языков программирования (в том числе и Лисп) также предполагает вычислять значение аргумента до передачи его в тело функции. Среди функциональных языков программирования, допускающих НПР и «ленивую» семантику, отметим ML (Meta Language) и Miranda Д. Тернера, а также Haskell и Clean (гл. 2).

Следует учитывать, что АПР и НПР представляют два важных, но не единственных порядка редукций. По сути, сколько в λ- выражении есть редексов, столько и может быть порядков редукций. Поэтому решение второго вопроса, поставленного в начале параграфа, относится к проблеме единственности решения.

Теорема Черча-Россера. Если λ-выражение S может быть редуцировано к двум выражениям S1 и S2, то существует выражение W, к которому может быть приведено как S1, так и S2.

Теорема Черча-Россера формулирует так называемое ромбическое свойство отношения редукций. Для функционального программирования это означает, что вызовы функций можно вычислять одновременно, а это предоставляет возможность автоматического распараллеливания программ.

Не менее значимым для функционального программирования будет следствие из теоремы Черча - Россера, с которым связано решение вопроса о единственности решения.

Если λ-выражение может быть приведено двумя разными способами к двум нормальным формам, то эти формы являются алфавитно-эквивалентными.

Доказательство: предположим, что S1 и S2 − две различные нормальные формы одного и того же λ-выражения S. Следовательно,

$S \to S1$ и $S \to S2$.

Согласно теореме Черча-Россера существует такая форма W, к которой будут сведены λ-выражения S1 и S2. Но предположению S1 и S2 - –то нормальные формы. Следовательно, возможен только один вариант:

$S1 \cong S2 \cong W$

т.е. нормальная форма λ-выражения S − единственная. Здесь символ «$\cong$» означает алфавитную эквивалентность.

9.1.4. Чистое λ-исчисление

Расширенное λ-исчисление, применяемое в функциональном программировании, отличается от так называемого чистого λ-исчисления набором предопределенных констант и δ-правил. В чистом λ-исчисления булевы константы и базовые функции представляются λ-выражениями. Булевы константы представляются следующим образом.

(COND P x y) $\to$ (P x y)
P = TRUE (TRUE x y) $\to$ x
P = FALSE (FALSE x y) $\to$ y
COND $\Leftrightarrow$ λp. λq. λr. P q r
TRUE $\Leftrightarrow$ λx. λy. X
FALSE $\Leftrightarrow$ λx. λy. Y
AND $\Leftrightarrow$ λx. λy. X y FALSE
OR $\Leftrightarrow$ λx. λy.(X true) y
TRUE AND FALSE

Списки в чистом λ-исчислении строятся с помощью двух функций конструкторов. Первая функция – CONS, вторая − NIL. Определить списки в чистом λ-исчислении означает определить эти две функции как λ-выражения.

CONS h t

Предшествующее выражение представляется как функция, берущая в качестве одного из своих аргументов функцию селектор и применяющую ее к выражениям для головы и хвоста списка.

CONS h t $\Leftrightarrow \lambda$h. λt. λs. s h t
h – голова
t – хвост
s – функция селектор
CAR λF.F true
CDR λF.F. false
NULL $-\lambda$E.E(λh. λt. False)
NIL $-\lambda$x. TRUE

9.2. Теория рекурсивных функций

Теория рекурсивных функций представляет собой одной из направлений теории алгоритмов и применяется в математической логике. Рекурсивные функции в теории алгоритмов рассматриваются с позиций алгоритма, допустимые исходные данные которого представляют собой системы натуральных чисел, а возможные результаты применения являются натуральными числами. Основные работы в области теории рекурсивных функций принадлежат математику С. К. *Клини*, который, в свою очередь основывался на результатах исследований К. *Гёделя*, Ж. Эрбрана [9,10,11].

9.2.1. Подходы к формализации алгоритмов

В математике предложено несколько подходов к формализации алгоритмов:

- теория рекурсивных функций;
- λ-исчисление Черча;
- абстрактная машина Тьюринга;
- абстрактная машина Поста;
- нормальные алгоритмы Маркова.

Два первых подхода нашли отражение в функциональном программировании, последний является основой языка программирования Рефал.

В математике большое значение имеют *вычислимые функции*. Под вычислимыми функциями понимаются *арифметические функции*, для вычисления которых имеются алгоритмы. В постановке А.П.Ершова понятие «вычислимость» вводится как *функциональное определение.* А. Чёрч предложил первое уточнение понятия «вычислимая функция», которое в теории

рекурсивных функций называется *тезисом Чёрча*. В тезисе Чёрча отождествляется понятие «всюду определённая вычислимая функция», имеющей натуральные аргументы и значения, с понятием «общерекурсивная функция». В развитие тезиса Черч привёл первый пример функции, не являющейся вычислимой. Тезиса Чёрча позволил придать понятию «вычислимая арифметическая функция» точный математический смысл, что явилось основой для изучения этого понятия с помощью точных методов. Впоследствии А. Тьюринг и Э. Пост дали первые уточнения понятия «алгоритм» в терминах идеализированных вычислительных машин. Следующее уточнение понятие «алгоритм» принадлежит А. А. Маркову, который предложил подход к алгоритму с помощью введённого им понятия «*нормальный алгоритм».*

Теория рекурсивных функций является разделом математической теории алгоритмов и имеет приложения в ряде других направлений математики. Примером может быть математическая логика. Понятие «примитивно рекурсивная функция» используется для первоначального доказательства знаменитой теоремы Гёделя о неполноте формальной арифметики, а понятие «рекурсивная функция» в его полном объёме было использовано С. К. Клини для интерпретации интуиционистской арифметики. Клини доказал теорему о нормальной форме для общерекурсивных функций, что упростило первоначальные определения для общерекурсивных и частично-рекурсивных функций. Аппарат теории рекурсивных функций используется в теории вычислительных машин и программирования.

9.2.2. Определения и теоремы теории рекурсивных функций

В теории рекурсивных функций вводятся следующие понятия.

Суперпозиция − подстановка функции в функцию x*y+x/z-y, содержащая фиксированное число операций.

Рекурсия − Определение очередного значения функции f_{i+1} через ранее вычисленное f_i. Примитивно-рекурсивное определение функции предполагает задание значения функции для начального значения аргумента и правила определения f_{i+1} с помощью f_i.

Рекурсивной называется функция, которую можно построить из целых чисел и арифметических операций с помощью суперпозиции и рекурсии.

Простейшие числовые функции:
1) одноместная функция непосредственного следования: $S^1(x)=x+1$;
2) n-местная функция тождественного равенства 0: $0^n(x_1,x_2, \dots , x_n)=0$;
3) n-местная функция тождественного повторения значения одного из своих аргументов: $I^n_m (x_1, \dots ,x_m, \dots , x_n)=x_m$, где m − целое, в интервале от 1 до n.

Из этих элементарных операций можно построить операторы:

1. Суперпозиция частичных функций $S(g,f_1, \dots , f_n)$. Имеются некоторое число n m-местных функции вида $f_1(x_1, \dots ,x_m)$, $f_2(x_1, \dots ,x_m)$, ... , $f_n(x_1, \dots ,x_m)$. Эти функции подставляют в n-местную функцию $g(x_1, x_2, \dots , x_n)$. В результате подстановки получается n-местная функция:
$h(x_1, x_2, \dots , x_n)= g(f_1(x_1, \dots ,x_m), f_2(x_1, \dots ,x_m), \dots , f_n(x_1, \dots ,x_m))$.

2. Примитивная рекурсия. Функция f образуется из частичных функций $g(x_1, \dots , x_n)$ и $h(x_1, \dots ,x_n,k,m)$ посредством примитивной рекурсии, если для всех натуральных $x_1, \dots ,x_n$ справедливо:
$f(x_1, \dots ,x_n,0)=g(x_1, \dots , x_n)$,
$f(x_1, \dots ,x_n,m+1)=h(x_1, \dots , x_n,m+1,f(x_1, \dots ,x_n,m))$.
Эти условия задают последовательность нахождения функции f:
$f(x_1, \dots ,x_n,0)=g(x_1, \dots , x_n)$,
$f(x_1, \dots ,x_n,1)=h(x_1, \dots , x_n,1,f(x_1, \dots ,x_n,0))$,
$f(x_1, \dots ,x_n,2)=h(x_1, \dots , x_n,2,f(x_1, \dots ,x_n,1))$, …

Примитивно рекурсивной называется такая функция, которая может быть получена конечным числом операций суперпозиции и примитивной рекурсии, исходя из элементарных функций S^1, 0^n, I^n_m.

Тезис Черча. Любая вычислимая функция является рекурсивной, может быть построена из целых чисел и арифметических операций с помощью суперпозиции и рекурсии. (Множество алгоритмически вычислимых функций совпадает с классом всех рекурсивных функций).

Теоремы:

1. Любая рекурсивная функция может быть вычислена на соответствующей машине Тьюринга.

2. Любая задача, решаемая на машине Тьюринга может быть решена с помощью нормального алгоритма Маркова.

В теории рекурсивных функций рассматриваются следующие классы (подмножества) функций:

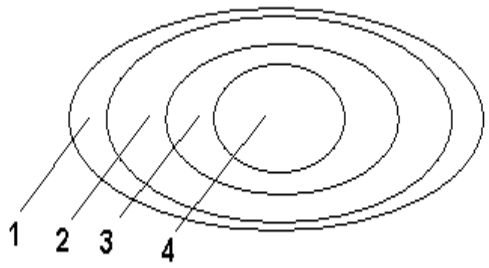
- частично-рекурсивные функции;
- общерекурсивные функции;

- примитивно-рекурсивные функции.

Общерекурсивными называются функции, определенные для любых значений своих аргументов из множества $N_0=\{0, 1, 2, \ldots\}$.

Частично-рекурсивными считаются функции не являющиеся определенными для любых значений своих аргументов из множества $N_0=\{0, 1, 2, \ldots\}$, то есть не являющиеся общерекурсивными.

Реальное программирование оперирует в основном с подмножеством примитивно-рекурсивных функций. Значительно реже встречаются функции, которые являются общерекурсивными, но не принадлежат к примитивно-рекурсивным.



Соотношение между множествами этих функций представлено на следующем рисунке. Примером этого может быть функция Аккермана.

Рис. Подмножества рекурсивных функций:
1 − множество вычислимых функций;
2 − множество частично-рекурсивных функций;
3− общерекурсивные функции;
4 − примитивно-рекурсивные функции.

Примечания

1. Клини (Kleene) Стивен Коул (р. 1909 г., Хартфорд, штат Коннектикут), американский логик и математик. В 1934 г. получил степень доктора философии в Принстонском университете. Профессор Висконсинского университета (Мадисон) с 1948 г. Основные работы посвящены теории алгоритмов и рекурсивных функций, а также интуиционистской логике и математике. В частности, им доказана эквивалентность введённого А. Чёрчем понятия λ-определимости функций с общерекурсивностью. Введённое Клини понятие (рекурсивной) реализуемости формул лежит в основе интуиционистской интерпретации арифметических суждений. Клини является

автором широко известных монографий по математической логике и теории рекурсивных функций.

2. *Исчислением* называют систему, использующую связанные переменные. λ-исчисление использует связанные переменные и единственным оператором, связывающим переменную, является оператор функциональной абстракции λ. Он превращает переменную в формальный параметр.
3. ТЬЮРИНГ (TURING) АЛАН. (1912−1954 гг.)Английский математик. Основные труды по математической логике, вычислительной математике. В 1936-37 гг. ввел математическое понятие абстрактного эквивалента алгоритма, или вычислимой функции, получившее затем название «машины Тьюринга».
4. Нормальные алгоритмы Маркова. Алгоритм в алфавите − функция, преобразующая входную последовательность символов из алфавита A={a_1, a_2, ... , a_n} в выходную. Нормальный алгоритм Маркова задается алфавитом A={a_1, a_2, ... , a_n}, содержащим конечное непустое множество элементов, нормальной схемой подстановок и порядком их применения. Нормальная схема подстановок это набор правил (формул) вида P_k -> P_n, согласно которым левое подслово P_k исходного слова P заменяется на слово P_n.

Резюме

- В λ-исчислении рассматриваются α-преобразование, соответствующее переименованию, δ-преобразование, аналогичное применению функции.
- Нормальная форма λ-выражения не содержит редексов и соответствует окончанию вычислений.
- В λ-исчислении рассматриваются нормальный и аппликативный порядки редукции, которые могут быть ассоциированы с ленивым и энергичным вычислениями.
- Теорема стандартизации утверждает, что нормальный параметр редукции гарантирует завершение преобразований (окончание вычислений), когда решение существует.
- Для функционального программирования теорема Черча-Россера означает эквивалентность (не с точки зрения эффективности)

альтернативных путей достижения результата и возможность распараллеливания вычислений.

Упражнения

1. Найдите λ-выражение определяющее селектор s.
2. Выполните преобразование λ-выражений и проверьте правильность определения примитивов.
3. Найдите все редексы следующего λ - выражения:
4. $(\lambda x.\lambda z.x(\lambda y.zy))(((\lambda x.\lambda z.z)17)(\lambda x.(\lambda z.z)x))$
5. Найдите нормальную форму предыдущего выражения

ГЛАВА 10

ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ

Примеры программ на Лиспе
Примеры программ на Эрланге
Определение функций высшего порядка

Декларативное программирование близко к образу мышления человека (мыслить в категориях, что должно быть). Цель данной главы – рассмотреть несколько характерных примеров решения конкретных задач, в которых показан перевод декларативного описания в описание на языке программирования. Расширение возможностей функционального программирования стали доступны большинству программистов с появлением языка программирования Эрланг, что иллюстрируется примерами этой главы. Приведенные ниже решения задач будут ответами на некоторые контрольные задания к главам 7,8.

10.1. Примеры программ на Лиспе

В рассматриваемых примерах показывается рекурсивное решение конкретной задачи. В каждом из них формулируется вербальное описание задачи, приводится пример применения функции, описываются все функции, в тексте программы выделяются начальные условия. В некоторых задачах для сравнения приводится императивное решение.

10.1.1. Задача о списке уровней

Определить функцию, возвращающую список, в котором каждый элемент является подсписком уровня, соответствующего месту его вхождения в исходный список. Пример применения:

>(name ‘(1 2 3 4 5))

(1 (2) ((3)) (((4))) ((((5)))))

Составляется вспомогательная функция, формирующая из элемента и числа n список, с уровнем вложенности n. Вспомогательная функция «уровень». n – номер уровня списка, l – элемент.

```
(defun levels(n l)
  (cond
    ((<= n 0) l) ; начальные условия
    (t (cons (levels (1- n)  l) nil))
  )
)
> (levels 3 3)
(((3)))
```

Функция выполняет преобразование исходного списка в «список уровней». n − начальный уровень счета.

```
 (defun convert(l &optional (n 0))
  (cond
    ((null l) nil) ; ; начальные условия
    (t (cons (levels n (car l)) (convert (cdr l) (1+ n))   ))
  )
)

> (convert '(1 2 3) 1)
((1) ((2)) (((3))))
> (convert '(1 2 3) 0)
(1 (2) ((3)))
```

Искомая функция вызывает функцию списка уровней с начальным значением отсчета уровней 0.

```
(defun name(l)
  (convert l)
)

> (name '(1 2 3))
```

(1 (2) ((3)))

10.1.2. Объединение упорядоченных списков

Сформировать функцию, объединяющую два упорядоченных в порядке возрастания чисел списка в один упорядоченный.Пример применения функции виден ниже.

```
> (con_rec '(1 3 3 5 7 8) '(1 2 3 4 5 9))
(1 1 2 3 3 3 4 5 5 7 8 9)
```

Декларативное описание для рекурсивного решения приведено в теле определения функции.

```
(defun con_rec (list1 list2)
(cond ( (null list1)  list2)
; если первый список пустой, то результат-второй список
( (< (car list2) (car list1)) ( cons (car list2) (con_rec (cdr list2) list1)))
( t  ( cons (car list1) (con_rec (cdr list1) list2)))
; иначе искомый список состоит из головы первого списка и
;результата слияния хвоста первого со вторым списком
))
```

Итерационный вариант функции приведен для сравнения решений по критерию выразительности решения.

```
(defun con(list1 list2)
(do ((list3 nil))
( (and (null list1) (null list2)) list3)   ;условие выхода
( cond ( (null list1)
(progn (setq list3 (append list3 list2))
(setq list2 nil)))
( (null list2)
(progn (setq list3 (append list3 list1))
(setq list1 nil)))
( (<= (car list1) (car list2))
(progn (setq list3 ( append list3 (cons (car list1) nil) ))
(setq list1 (cdr list1))))
(t (progn (setq list3 ( append list3 (cons (car list2) nil) ))
(setq list2 (cdr list2))
```

```
)))
))
```

Здесь следует учесть, что итерационный вариант функции слияния использует стандартную функцию слияния append, а рекурсивный − только примитивы работы со списками.

10.1.3. Фильтрующие объединения

Сформировать функцию, объединяющую списки за исключением элементов, встречающихся в обоих списках.

```
>(not_add ‘(1 3 5 7) ‘(2 3 4 5 8))
(1 2 4 8)

 (defun nrep(l1 l2)
  (cond
    ((null l1) nil)
    ( (member (car l1) l2) (nrep(cdr l1) l2) )
    (t (cons (car l1)(nrep (cdr l1) l2)))
  )
)

; объединение двух списков, за исключением элементов,
; встречающихся в обоих списках
(defun not_add (l1 l2)
  (append (nrep l1 l2)
            (nrep l2 l1)
))
```

10.1.4. Упорядоченные внедрения

Необходимо определить функцию для добавления элемента в упорядоченный список без нарушения порядка.

Пример применения функции:

```
> (ins 5 '(1 2 3 3 7 8))
(1 2 3 3 5 7 8)
```

Необходимые пояснения даны в виде комментариев в теле программы.

```
(defun ins(elem lis)
  (cond
   ((< elem(car lis))(cons elem lis))
;критерий окончания: если элемент elem меньше
;очередного элемента списка, то он вставляется в список
;если-нет, то-рекурсия
   (t (cons (car lis)(ins elem (cdr lis)) ) )
    ))
```

10.2. Примеры программ на Эрланге

В параграфе рассматриваются примеры определения функций в следующем порядке: формулируется условие задачи, приводится текст определяемых для данного решения функций.

Каждый пример соответствует модулю и может быть помещен в файл с именем, определяемым первым атрибутом. Для некоторых задач приводятся решения, повторяющие лисповские (использующие функции селекторы и конструкторы) и с применением образцов.

10.2.1. Упорядоченное объединение

Определить функцию, выполняющую слияние двух упорядоченных функций с сохранением порядка. В модуле находятся две функции name(L1, L2) и name1(L1, L2). Первая максимально повторяет решение на Лиспе: использует hd(L2) (аналог функции car) и tl(L2) (аналог функции cdr). Вторая для выделения головы и хвоста использует механизм сопоставления с образцом.

```
-module(erl1).
-export([name/2,name1/2]).

name([], L2) -> L2;
name(L1, []) -> L1;
```

```
name(L1, L2) when (hd(L2) < hd(L1)) -> [hd(L2) | name(tl(L2),L1)];
name(L1, L2) -> [hd(L1) | name(tl(L1),L2)].
% Запись на основании нотации сопоставления с образцом
name1([], L2) -> L2;
name1(L1, []) -> L1;
name1([L1|L12], [L2|L21]) when (L2 < L1) -> [L2 | name1(L21, [L1|L12])];
name1([L1|L12], [L2|L21]) -> [L1 | name1(L12, [L2|L21])].
```

Решение предполагает поэлементное сравнение списков (when (hd(L2) < hd(L1)) или when (L2 < L1)), по результатам которого в результирующий список помещается соответствующий элемент.

10.2.2. Внедрение в упорядоченный список

Определить функцию, внедряющую числовой элемент в упорядоченный список без нарушения порядка. Решение предполагает нахождение сооответсвующего места для внедряемого элемента и размещение его в результирующем списке.

```
-module (erl2).
-export([name/2]).

name(N,[])->[N];
name(N,[H|T]) when H>N->[N|[H|T]];
name(N,[H|T])->[H|name(N,T)].
```

Решение использует образцы и предполагает нахождение места элемента в списке name(N,[H|T]) when H>N->[N|[H|T]].

10.2.3. Подсчет числа элементов

Определить функции для подсчета числа элементов стоящих на четных местах в списке и числа элементов, стоящих на нечетных позициях.

```
-module(erl3).
-export([name/1, namec/1, ch/2, nech/2]).
```

```
namec([]) ->false;
namec([H|T])->H*nech(T,0).

name([]) ->false;
name([_|T])->nech(T,1).

nech([],_)->1;
nech([H|T],0)->H*ch(T,0);
nech([_|T],1)->ch(T,1).

ch([],_)->1;
ch([H|T],1)->H*nech(T,1);
ch([_|T],0)->nech(T,0).
```

Решение предполагает использование взаимной рекурсии: функции поочередно вызывают себя. Во многом это решение аналогично показанному в параграфе 7.3. примеру.

10.2.4. Подсчет числа разных элементов списка

Определить функцию для подсчета числа разных элементов списка. Решение включает функции определения вхождения элемента головы в хвост и подсчета числа элементов списка.

```
-module(erl4).
-export([mem/2,mem1/2,name/1,name1/1]).

mem(_,[])->1;
mem(V,L)when V==hd(L)->0;
mem(V,L)->mem(V,tl(L)).

name([])->0;
name([H|T])->mem(H,T)+name(T).

mem1(_,[])->1;
mem1(V,[L|_]) when V==L->0;
mem1(V,[_|L])->mem1(V,L).
```

```
name1([])->0;
name1([H|T])->mem1(H,T)+name1(T).
```

10.3. Определение функций высшего порядка

Задача предыдущего параграфа может быть решена определением функции высшего порядка, которая в зависимости от передаваемого аргумента выполняет разные действия.

```
> (defun name (fun list)
(cond
((null list)0)
((mem (funcall fun list) (cdr list))(name fun (cdr list)))
(t (+ 1 (name fun (cdr list))))))
NAME
> (defun mem(v list)
(cond
((null list)nil)
((eql v (car list))t)
(t(mem v (cdr list)))))
MEM
> (name 'car '(1 2 3 2 4 3 5 4 2 4 3))
 ; подсчет разных элементов списка
5
> (name 'cdr '(1 2 3 2 4 3 5 4 2 4 3))
; подсчет всех элементов
11
```

Приведенные в главе решения, кроме итерационных, показанных для сравнения, являются функциональными (не используют разрушающих действий). Переменные в определениях функций не связывались с конкретным типом данных, память программистом явно не выделяется и не освобождается. Это соответствует принципам высокоуровневого функционального программирования.

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Барендрегт, Х. Лямбда-исчисление. Его синтаксис и семантика/ Х. Барендрегт. − М.: Мир, 1895.− 257 с.
2. Бизли, Д. Язык программирования PYTHON/ Д. Бизли– Киев, ДиаСофт, 2000. − 336 с.
3. Братко, И. Алгоритмы искусственного интеллекта на языке Prolog/ И. Братко. – М.: Изд. дом «Вильямс», 2004. – 640 с.
4. Братко, И. Программирование на языке Пролог для искусственного интеллекта/ И. Братко. – М.: Мир, 1990. – 560 с.
5. Вольфгаген, В.Э. Комбинаторная логика в программировании/ В.Э. Вольфгаген.– М.: МИФИ, 1994. – 209 с.
6. Душкин, Р.В. Функциональное программирование на языке Haskell/ Р.В. Душкин. – М.: ДМК Пресс, 2007. – 608 с.
7. Клоксин, У. Программирование на языке Пролог/ У. Клоксин, Д. Меллиш. – М.: Мир, 1987. – 336 с.
8. Клоксин, У. Программирование на языке Пролог/ У. Клоксин, Д. Меллиш. – М.: Мир, 1987. – 336 с.
9. Клини, С. К. Введение в метаматематику/ С.К. Клини. – М.: Либроком, 2008. − 526 с.
10. Клини, С. К. Математическая логика/ С.К. Клини. − М.: ЕДИТОРИАЛ УРСС, 2005. − 480 С
11. Клини, С. Основания интуиционистской математики с точки зрения теории рекурсивных функций : пер. с англ. / С. Клини, Р Весли. − М., Наука, 1978. − 272 с.
12. Лавров, С.С. Автоматическая обработка данных. Язык ЛИСП и его реализация / С.С. Лавров, Г.С. Силагадзе − М.: Наука, 1978.
13. Люгер, Ф. Искусственный интеллект: стратегии и методы решения сложных проблем : пер. с англ. / Ф. Люгер.− М.: Изд. дом «Вильямс», 2003. – 864 с.
14. Малпас, Дж. Реляционный язык Пролог и его применение : пер. с англ. / Дж. Малпас. − М.: Наука, 1990. – 463 с.
15. Малпас, Дж. Реляционный язык Пролог и его применение : пер с англ. / Дж. Малпас. – М.: Наука, 1990. – 463 с.
16. Стерлинг, Л. Искусство программирования на языке Пролог : пер. с англ. / Л. Стерлинг, Э. Шапиро. – М.: Мир, 1990. – 235 С.

17. Филд, А. Функциональное программирование : пер с англ. / А. Филд, П. Харрисон– М.: Мир, 1993. – 637 с.
18. Хендерсон, П. Функциональное программирование: применение и реализация : пер с англ. / П. Хендерсон . − М.: Мир, 1983. – 349 с.
19. Хювенен, Э. Мир Лиспа : перевод. с финск. В 2 т. / Э. Хювенен, Й. Сеппянен. – М.: Мир, 1990. – 447 с.
20. Шалимов П.Ю. Функциональное программирование/ П.Ю. Шалимов. – Брянск: БГТУ, 2003. – 160 с.
21. Уотермен, Д. Руководство по экспертным системам: пер. с англ. / Д. Уотермен. – М.: Мир, 1989. − 478 с.
22. Лавров, С.С. Автоматическая обработка данных. Язык ЛИСП и его реализация/ С.С. Лавров, Г.С. Силагадзе. − М.: Наука, 1978.
23. Мешалкин, В.П. Экспертные системы в химической технологии. Основы теории, опыт разработки и применения/ В.П. Мешалкин. – М.: Химия, 1995. – 386 с.
24. Erlang is documented in the book «Concurrent Programming in Erlang» [Electronic resource]. − Mode access: www.erlang.se/doc/

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

ОГЛАВЛЕНИЕ

Учебное издание

Шалимов Петр Юрьевич

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ ЛИСП, ЭРЛАНГ

Редактор издательства Л.Н. Мажугина
Компьютерный набор П.Ю. Шалимов

Темплан 2010г., п.47

Подписано в печать 09.11.06. Формат 60x84 1/16. Бумага офсетная. Офсетная печать. Усл. печ.л. 11,04. Уч.-изд.л. 11,04. Тираж 60 экз. Заказ .

Брянский государственный технический университет
241035, г. Брянск, бульвар им. 50-летия Октября, 7, тел. 58-82-49
Лаборатория оперативной полиграфии БГТУ, ул. Институтская, 16