Хендерсон П.

Х38 Функциональное программирование. Применение и реализация: Пер. с англ.—М.: Мир, 1983.—349 с, ил.

Книга английского специалиста по программированию, обобщающая опыт использования функционального программирования. Обсуждаются особенности функциональных языков и возможности их реализации на современных ЭВМ Изложение иллюстрируется многочисленными примерами. Для программистов, математиков-прикладников, для всех, кто преподает и изучает программирование.

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Функциональное программирование — это 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.

Алгол, Фортран и Ассемблер сделали свое дело, и тем нескольким сотням тысяч людей у нас в стране, которым приходится писать программы для ЭВМ, пожалуй, будет нелегко представить себе, как можно приступать к программированию, не опираясь на эти, казалось бы, незыблемые конструкции языков программирования.

Внимательное чтение этой книги будет на первых порах вызывать у них чувство протеста — они раз за разом будут ощущать себя как спортсмены, которым нужно играть в волейбол с одной рукой, привязанной к телу. Потребуется большое доверие к автору для того, чтобы при чтении книги преодолеть дебют с такой заметной интеллектуальной форой.

На основании собственного болезненного опыта упражнения в функциональном программировании мне хотелось бы предупредить читателей, что ощущение трудности и необычности составления функциональных программ вполне законно и естественно.

Мало того, они могут взять себе в союзницы историю науки. Известный американский ученый Алонсо Чёрч, автор лямбда-нотации и тезиса Чёрча, создал на подступах к теории алгоритмов лямбда-исчисление, которое сейчас можно считать не чем иным, как теоретической моделью современного функционального программирования. Он бился немало месяцев над тем, чтобы запрограммировать в лямбда-исчислении операцию вычитания единицы из натурального числа

$$\begin{cases} 0 \dot{-} 1 = 0 \\ (n+1) \dot{-} 1 = n \end{cases}$$

Чёрч так и не справился с этой задачей и уже уверился в неполноте своего исчисления, но в 1932 г. Стефен Клини, тогда молодой аспирант, предложил следующую, на первый взгляд искусственную, но на самом деле полностью отвечающую сути дела конструкцию:

$$\begin{cases} F(x,\ y,\ z) = \text{если } x \text{ то } 0 \text{ иначе если } y+1 = x \text{ то } z \text{ иначе} \\ \qquad\qquad\qquad\qquad\qquad\qquad F(x,\ y+1,\ z+1) \\ n \dot{-} 1 \equiv F(n,\ 0,\ 0) \end{cases}$$

История той же науки — теории алгоритмов и вычислимости — говорит, что лямбда-исчисление, сыграв принципиальную роль в угадывании объема понятия эффективно вычислимой функции, в дальнейшем отошла на второй план, уступив место развитию теории на основе понятия частично-рекурсивных функций и машин Тьюринга. И тем не менее почти через пятьдесят лет эта теоретическая модель снова берется на вооружение в программировании, с надеждой найти в ней ответ на ряд трудных проблем сегодняшнего развития.

Если продолжить разговор об исторических предпосылках функционального программирования, то для советского читателя будет особенно интересно узнать, что в Советском Союзе была и существует методологическая школа построения пакетов прикладных программ, которая ведет свой счет времени с 1954 г., предвосхитив и реализовав значительную часть основных понятий, относящихся к функциональному программированию. Я имею в виду крупноблочное программирование обработки составных объектов в функциональных обозначениях, предложенное и разработанное Л. В. Канторовичем и его учениками и реализованное в серии конкретных программных систем для ЭВМ Стрела, БЭСМ и М-20 в 50-е и 60-е годы.

Глубокая интуиция Л. В. Канторовича позволила ему предвидеть трудности, присущие «блок-схемному» программированию, ухватить ряд сущностей функционального программирования и предложить приемы, направленные на повышение эффективности функциональных программ (некоторые из них описаны в этой книге). В то же время начальные варианты подхода школы Канторовича, как сейчас видно, содержали ряд пробелов, а натиск общественного мнения со стороны «традиционных» программистов, идущих от вовсю критикуемой ныне, но всесильной в то время фон-неймановой архитектуры, поставил представителей школы в положение обороняющихся и сильно затормозил развитие ее идей.

Следующий выход в функциональное программирование связан с Лиспом, разработанным в 1961 г. американским ученым Дж. Маккарти. Сам язык и его судьба хорошо известны программистам, однако, несмотря на всю популярность, школа Лиспа долго представлялась «большинству» чем-то вроде сектантского учения.

Сам Маккарти хорошо сознавал фундаментальный характер функциональной модели вычислений, лежащей в основе Лиспа, и его анализ этой модели, сделанный в 1965 г. в докладе на Международном конгрессе по обработке информации, привел к построению важного раздела теоретического программирования — теории рекурсивных программ. В это же время появилась ставшая теперь классической статья английского ученого П. Лэндина об использовании лямбда-исчисления для описания семантики Алгола-60. Хотя эти два направления работ в течение более чем 10 лет были известны только теоретикам, тем не менее они создали предпосылки к выходу функционального подхода в программировании на более широкую арену.

Можно рассматривать как некий символ, что новый период функционального программирования начался с работы человека, которому мы, пожалуй, в большей степени, чем кому-либо другому, обязаны за увековечение архитектуры фон Неймана и «традиционного» программирования. Я имею в виду изобретателя Фортрана научного сотрудника ИБМ Дж. Бэкуса и его Тьюрингову лекцию 1978 г. под названием «Может ли программирование освободиться от бремени фон-неймановского стиля? Функциональный стиль и его алгебра программ». Здесь не место повторять анализ особенностей и потенциальные преимущества функционального программирования, и я скажу только, что понять и принять функциональное программирование легче, если рассматривать задачу программирования в ее полном контексте, начиная со спецификации задачи и логического анализа ее разрешимости, побочным продуктом которого является сама программа.

Возвращаясь к этой книге, следует воздать должное автору, который педагогически очень тактично вводит и дозирует столь непривычные на первый взгляд приемы функционального программирования. Более того! Чувствуя момент, когда у читателя появляются первые проблески уверенности, он честно показывает, что новые меха годятся и для старого вина, демонстрируя, как ряд приемов традиционного программирования с использованием памяти и итерационных процессов можно закодировать в функциональной нотации. С другой стороны, рассказывая о реализации функциональных программ «до конца», т. е. вплоть до аппаратной реализации, автор подталкивает активного читателя на поиски новой архитектуры ЭВМ, более непосредственно отражающей рекурсивный характер функциональных вычислений.

Академгородок *А. П. Ершов*
октябрь 1982 г.

ПРЕДИСЛОВИЕ

Основным мотивом, побудившим меня написать эту книгу, было желание собрать воедино многие важные идеи, которые возникают при написании программ в чисто функциональном, или аппликативном, стиле. Эти идеи проистекают из двух различных областей нашей науки — изучения семантики языков программирования и работ по искусственному интеллекту. Книга полностью охватывает функциональный стиль программирования, структуру функциональных языков, их применение и реализацию и такие современные концепции, как вычисления с задержкой, недетерминированные программы и функции высших порядков.

Мой подход является прагматическим и больше касается применений и реализации функционального программирования, нежели его теоретических основ. Так, книга начинается с изложения основной методики построения функциональных программ, затем описывается, как можно реализовать функциональные языки на современных машинах и, наконец, дается описание наиболее развитых черт таких языков посредством определения их смысла в терминах реализации.

Функциональные языки используются в исследованиях по семантике двояко. Один способ — это описать интерпретатор для изучаемого языка; функциональные языки идеально приспособлены для этой роли. Другой способ состоит в том, чтобы для каждой программы на изучаемом языке определить эквивалентную функцию или функциональную программу. В любом случае чрезвычайная простота и мощность функционального языка делают его весьма удобным для таких семантических спецификаций.

В работах по искусственному интеллекту приходится оперировать сложными, обычно символьными структурами данных и такими же сложными алгоритмами. Маккарти в разработанном им языке программирования Лисп, имеющем функциональную основу, преодолел многие из этих трудностей. Использование чисто функционального стиля программирования естественно для приложений, характеризующихся многократным получением- сложных структур данных из других таких же структур.

В настоящее время техника функционального программирования получает широкое распространение, и важность ее как средства, позволяющего продвинуться в развитии языков самого высокого уровня, будет возрастать.

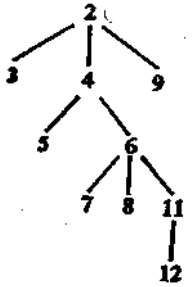
По нашему мнению, полное понимание семантики функционального языка, и вообще всякого языка программирования, достигается только тогда, когда исследована связь между языком и его реализацией. Поэтому книга включает полное описание реализации одного, чисто функционального языка, названного Лисп-инструмент (сокращенный Лисп). Эта реализация важного варианта Лиспа представлена в форме заготовок — набора программ, из которого прилежный читатель, имеющий доступ к машине, может легко составить себе единую систему. Два существенных компонента этой системы, компилятор в исходной и компилятор в объектной форме, даны в приложении. Все, что требуется для построения системы,— это написать простой имитатор для абстрактной машины и затем ввести в него компилятор. Компилятор будет компилировать себя и любую другую программу на языке Лисп-инструмент, позволяя, таким образом, читателю экспериментировать с программами и языками, описанными в книге. Конструктивные детали содержатся в гл. 11 и 12. Для читателя может оказаться полезным время от времени обращаться к этим главам, чтобы подкрепить свое понимание точного смысла некоторых, концепций функционального программирования.

В то время как Лисп-инструмент служит конкретным примером реализованного функционального языка, на протяжении всей книги для написания программ используется язык с несколько более удобным для чтения синтаксисом. Существуют простые правила транслитерации, так что любую программу из книги или нужную для упражнений можно выполнить в системе Лисп-инструмент.

На протяжении нескольких лет я использовал материал этой книги в лекциях по семантике языков программирования для студентов старших курсов. Изложение практически основано на том, что новые понятия можно легко сопоставить с понятиями, уже встречавшимися при изучении более традиционных языков. Кроме того, возхможность экспериментировать с описанными здесь интерпретатором и компилятором и вторгаться в семантику изучаемых языков позволяет воспринимать тонкости семантики языков программирования. В целом книга поможет студентам сопоставить и сравнить факты, изучаемые в таких различных областях, как теория программирования, компиляторы, языки программирования, структуры данных и машинная архитектура. Для более старших студентов, специализирующихся в программировании, языках программирования,

искусственном интеллекте или машинной архитектуре, книга послужит введением во многие существенно важные разделы использования языков как в программировании, так и для целей спецификации. Читателям, заинтересованным в практическом знакомстве с теорией вычислений или с языками весьма высокого уровня для своих экспериментов, построение этой системы также весьма полезно.

Чтобы осветить все темы, включенные в книгу, потребуется, вероятно, двухсеместровый курс. Однако, как показывает схема зависимости глав, можно опускать те или иные разделы соответственно тому, на что делается ударение. В моем собственном курсе лекций, который продолжается примерно семестр, я пытаюсь научить функциональному программированию как практическому и полезному навыку и поэтому делаю упор на примеры и упражнения, а также на экспериментирование с интерпретатором и компилятором. Для этого используются гл. 2, 4, 5 и 6,. а также избранный материал из гл. 8 и 9. Разумеется, студенты имеют доступ к реализации системы Лисп-инструмент как для экспериментов по ее использованию, так и для модификации самой системы. В более продолжительном курсе можно было бы поручить студентам самим построить для себя такую систему. Это способствовало бы изучению функционального стиля программирования и лучшему пониманию семантики языков программирования.



Многие помогали мне в подготовке этой книги. Я получил ценные советы от тех, кто прочитал те или иные разделы рукописи. Особенно значительный вклад внесли Тони Хоар и Саймон Джонс. Очень ценна эффективная реализация системы Лисп-инструмент, выполненная Фредом Кингом. Я благодарен Джулии Леннокс, которая печатала и выверяла рукопись, а также всем моим коллегам по Университету Ньюкасла, создавшим стимулирующую обстановку для работы.

Питер Хендерсон

Глава 1.

ФУНКЦИИ И ПРОГРАММЫ

В первых машинах и размеры памяти, и быстродействие были невелики по сравнению с современными. Поэтому для программиста самым важным было сделать программу как можно более эффективной. По мере того как увеличивались размеры памяти и быстродействие машин, стало возможным несколько поступиться эффективностью в угоду большей ясности и выразительности программ. Программы, становясь более легкими для понимания, становились экономичнее при построении и эксплуатации. Были разработаны такие высокоуровневые языки программирования, как Фортран, Алгол, Кобол, ПЛ/1 и Паскаль. Из-за того, что программы, написанные на этих языках, транслируются на машинные языки, пользователи теряют в скорости их выполнения от 2 до 10 раз. Эта готовность выиграть в легкости понимания программы пусть даже за счет некоторой потери эффективности является тенденцией, которая значительно усилится в ближайшем будущем по мере того, как мы будем пожинать плоды технологических усовершенствований в проектировании ЭВМ.

Современной тенденцией в технологии является разительное снижение стоимости оборудования. Кроме того, быстродействие машин и их размеры постоянно изменяются настолько, что уже в начале восьмидесятых годов можно предвидеть улучшение их характеристик на целые порядки. Таким образом, ввиду повышения эффективности оборудования все для большего числа приложений ясность и легкость понимания программ будет значить больше, чем оптимальное использование оборудования. Можно ожидать поэтому, что уровень языков и их ориентируемость на пользователя будут прогрессивно возрастать. В частности, можно надеяться, что языки будут становиться более простыми и единообразными. Для современных языков, таких как уже упомянутые, характерен компромисс между ориентированностью на машину и ориентированностью на пользователя, что приводит к большому разнообразию внутренних свойств в каждом языке. Языки, не учитывающие возможностей машины, обладают тем свойством, что их программы выполняются более медленно.

Язык, о котором пойдет речь в этой книге, меньше ориентирован на современные машины, зато доставляет пользователю для общения с машиной средства высокого уровня. Мы говорим об этом языке как о строго функциональном, так как понятие функции является первичным для программ на такого рода языках. Функциональные языки являются наиболее широко изучаемым классом среди языков самого высокого уровня. Программы, которые выглядели бы длинными и трудными на традиционных языках высокого уровня, часто бывают более короткими и ясными в функциональной форме. Таким образом, с уверенностью можно сказать, что функциональные языки имеют более высокий уровень, нежели уже упомянутые современные языки. Ценой, которую приходится за это платить, является то, что их реализация приводит к программам, менее эффективно использующим оборудование. Но, как уже говорилось, тенденции в развитии технологии таковы, что эта эффективность становится менее важной и для многих приложений в будущем совсем не будет иметь значения.

Мы начнем эту главу с рассмотрения математического понятия функции и покажем, что программы могут быть построены как композиции функций. В разд. 1.1 мы определим, что такое функциональный язык программирования, а в разд. 1.2 обсудим, чего нет в функциональном языке по сравнению с традиционными высокоуровневыми языками. В частности, мы коснемся некоторых свойств машинной ориентированности языков высокого уровня. Остальная часть книги посвящена изучению того, как именно строго функциональный язык может использоваться для широкого круга практических приложений и как он может быть с достаточной эффективностью поддержан современными машинами. Основной тезис, который мы выдвигаем, состоит в том, что функциональные программы яснее выражают свои цели, чем традиционные программы, и поэтому они легче для понимания, эксплуатации и в первую очередь их легче составить правильно.

1.1. Программирование при помощи функций

Концепция функции является одним из фундаментальных понятий математики. Интуитивно, *функция* является правилом, сопоставляющим каждому элементу некоторого класса соответствующий ему единственный элемент из другого класса. Другими словами, для заданных двух классов элементов, соответственно называемых *областью определения* и *областью значений* данной функции, каждому элементу из области определения функция ставит в соответствие в точности один элемент из области значений. Соответствие, изображенное на рис. 1.1, определяет функцию, назовем ее f, для которой областью определения является класс элементов $\{a, b, c, d\}$ и областью значений — класс $\{p, q, r\}$. Важным свойством соответствия, которое характеризует его как функцию, является то, что элементы области значений единственным образом определяются по элементам из области определения. Этот факт позволяет нам обозначать через $f(x)$ тот элемент из области значений, который

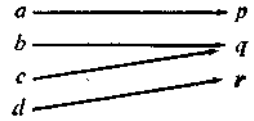


Рис. 1.1.

функция f соотносит элементу x из области определения этой функции. Говорят также, что f отображает x в $f(x)$ или что $f(x)$ является образом x относительно f. При вычислениях чаще говорят, что $f(x)$ есть результат применения f к аргументу x.

Существует много способов описания простых функций, подобных приведенной выше. Например, можно записать соответствие в виде последовательности пар

$(a, p), (b, q), (c, q), (d, r)$

или протабулировать функцию, как показано в табл. 1.1, что будет равносильно предыдущему.

Таблица 1.1

x	a	b	c	d
$f(x)$	p	q	q	r

Другим способом является запись соответствия в виде ряда равенств:

$f(a)=p, \quad f(b)=q, \quad f(c)=q, \quad f(d)=r$

Каждый из этих способов изображения страдает тем недостатком, что здесь неявно выражено и поэтому нелегко проверяется то свойство соответствия, что $f(x)$ единственным образом определяется по x. Когда область определения велика, то на первый взгляд не очевидно, обозначает ли некоторая конкретная запись действительно функцию. Когда же область определения бесконечна, мы вовсе не можем использовать такие обозначения.

Для этих функций можно использовать такие записи, только чтобы перечислить отдельные образцы соответствия, устанавливаемого функцией.

Рассмотрим более реальный пример. Функция *квадрат* имеет областью определения класс всех целых чисел (положительных, нуль и отрицательных), а областью значений — класс всех неотрицательных чисел. Элемент области значений функции *квадрат,* который соответствует элементу *x* из области определения этой функции, есть *xxx.* Таким образом, мы имеем, например,

квадрат(3)=9, квадрат(6)=36, *квадрат(*—2)=4

Невозможно, разумеется, изобразить все соответствие, так как область определения функции *квадрат* бесконечна. Однако можно определить все это соответствие полностью, написав такое правило (или определение):

квадрат (x)z=x x *x*

Здесь точно указано, как вычислить образ *x* для любого *x* из области определения функции *квадрат.* Образ элемента *x* получается просто возведением его в квадрат, т. е. умножением элемента на самого себя. При написании этого определения мы использовали переменную *x* для обозначения любого элемента из области определения и записали правило вычисления соответствующего элемента из области значений в виде выражения, использующего эту переменную. Такую переменную обычно называют параметром определения.

Определяющее правило, или определение, такое, как мы только что рассмотрели для функции *квадрат,* является тем стандартным способом, которым мы и будем представлять функции. В таком представлении область определения и область значений даны неявно и не всегда очевидны. То, что областью определения для функции *квадрат* мы предполагаем множество целых чисел, выражается только в дополнительных ограничениях, а тот факт, что областью значений является в этом случае множество неотрицательных целых чисел, хотя и выводится из вида правила, тем не менее не выражен явно. Мы использовали бы то же самое правило, если бы определяли функцию *квадрат* для всех вещественных чисел. Вообще для определяемых функций достаточно, если охарактеризовать их области определения и области значений неформально, в виде дополнительных ограничений. В действительности, мы часто будем называть областью определения и областью значений множества даже более широкие, в которых некоторые элементы из названной области определения функции не имеют соответствующего элемента в области значений и наоборот. Например, можно сказать, что функция *квадрат* отображает целые числа (как область определения) в целые (как область значений). Здесь в область значений включаются необязательные отрицательные числа. Аналогично о функции

$$\textit{обратная}(x) \equiv \frac{1}{x}$$

можно сказать, что она отображает вещественные числа в вещественные. Однако вещественное число 0 не входит на самом деле в область определения функции, так как элемент *обратная(0)* не определен приведенным выше правилом.

Говорят, что функция, для которой в качестве области определения задано множество *A,* является *частичной* над *A* (или *частично определенной* в множестве *A),* если в *A* существуют элементы, для которых образ посредством этой функции не определен. Функция, которая не является частичной над *A,* является *всюду определенной* в *A,* или *общей* функцией. Мы не будем часто пользоваться этими терминами, но важно отметить, что вообще все функции, которые описываются в книге, будут частично определенными в множествах, которые заданы как области определения.

Рассмотрим другой пример. Функция *макс,* примененная к паре чисел, выдает в качестве результата наибольшее из них. Так, например,

макс(1, 3)=3, *макс* (1,7, —2,0)=1,7

Правило для этого удобнее всего выражается через **два** параметра:

$$\textit{макс}(x,\ y) \equiv \begin{cases} x, & \text{если } x \geqslant y \\ y & \text{в противном случае} \end{cases}$$

Этого громоздкого определения можно избежать, если воспользоваться условной формой **если. . .то. . .иначе.** . ., которая знакома нам по традиционным языкам программирования:

$$\textit{макс}(x, y) \equiv \textbf{если } x \geqslant y \textbf{ то } x \textbf{ иначе } y$$

Как и в определении функции *квадрат,* справа в правиле здесь находится алгебраическое выражение, включающее параметры *x* и *y.* В данном случае форма **если. . .то. . .иначе.** . . требует сначала вычислить предикат $x \geqslant y$ и в зависимости от результата (истина или ложь) выбрать подвыражения *x* или *y* в качестве результата конкретного применения функции.

Например, *макс(1,3),* т. е. результат применения функции *макс* к аргументам (1,3), вычисляется по этому правилу так: конструкция *применение,* называемая также в программировании *вызовом функции,* гласит, что параметрам *x* и *y* должны быть сопоставлены значения 1 и 3 соответственно. Далее тре-

буется вычислить $x \geqslant y$, т. е. $1 \geqslant 3$, что дает, разумеется, значение ложь. Соответственно этому выбираем значение *y,* т. е. 3, в качестве результата применения. Так мы определили, что *макс(1,3)=3.*

Начинаем знакомиться с тем, как можно программировать при помощи функций. Мы определяем базовый набор полезных функций, таких как *квадрат* и *макс,* и, беря их с соответствующими аргументами, вычисляем их результат способом, описанным выше. Функция является по существу программой, которая воспринимает входной сигнал (ее аргументы) и вырабатывает выходной (ее результат). Чтобы иметь возможность конструировать более мощные программы, необходимо уметь определять новые функции через старые. В конечном счете эти новые функции должны, разумеется, ссылаться при их вычислении только на основные, базовые функции, и очевидно, что этого можно достичь построением иерархии определений. Рассмотрим, например, следующее определение:

$$наиб(x,\ y,\ z) \equiv макс(макс(x,\ y),\ z)$$

по имени функции легко догадаться, что *наиб(x, y, z)* вычисляет наибольший из трех ее аргументов. Она делает это, используя функцию *макс,* сначала применяя ее к первым двум аргументам, а затем к промежуточному результату и третьему аргументу.

Определение функции *наиб* иллюстрирует один из фундаментальных способов построения новых функций из старых. Это метод композиции функций. Вложением одного применения функции *макс* в другое мы добиваемся композиции функции *макс* с собой же. Вообще функции могут вкладываться одна в другую на произвольную глубину и составлять таким образом произвольно глубокие композиции. Например, наибольшее из шести вещественных чисел *a, b, c, d, e* и *f* может быть определено одной из следующих композиций:

макс(наиб(a, b, c), наиб(d, e, f))
наиб(макс(a, b), макс(c, d), макс(e, f))
макс(макс(макс(a, b), макс(c, d)), макс(e, f))

Чтобы программировать при помощи функций, нужно иметь возможность определить достаточно богатое множество основных функций и затем использовать композицию, чтобы определять новые функции в терминах исходных. Таким образом можно построить целую библиотеку функций; некоторые из них, возможно построенные над слоями других, могут представляться достаточно мощными, чтобы называться программами — строго функциональными программами. Возникает вопрос, нужно ли еще что-нибудь, кроме такого набора исходных функций и возможности составлять из них композиции, для того, чтобы определить библиотеку функциональных программ. Наш ответ гласит, что ничего больше не требуется, и целью этой книги является иллюстрация и подтверждение этой точки зрения.

Выбор совокупности основных функций, безусловно, очень важен. Для практических целей, как для достаточно эффективной реализации на современных машинах, так и для легкости изображения задачи программой, набор основных функций должен быть весьма мощным. Во второй главе мы введем множество основных функций, которые в функциональном программировании часто являются стандартными и которые адекватно определяют функциональные программы во многих важных приложениях. Возможен другой выбор совокупности основных функций, и это обсуждается далее в десятой главе.

В число основных функций необходимо включить арифметические операции. Мы привыкли в математике и программировании записывать алгебраические выражения, включающие арифметические операции, в виде

$$a + b \times c,$$
$$(2 \times x + 3 \times y)/(x - y)$$

Разумеется, операции +, —, X и / являются функциями, каждая из которых имеет областью определения упорядоченные пары вещественных чисел и областью значений — все вещественные числа. Можно явнр определить их как функции:

$$плюс(x,\ y) \equiv x + y \qquad умн(x,\ y) \equiv x \times y$$
$$минус(x,\ y) \equiv x - y \qquad дел(x,\ y) \equiv x/y$$

Используя вместо операций эти имена функций, можно переписать приведенные выше алгебраические выражения в виде

плюс(a, умн(b, c))
дел(плюс(умн(2,x), умн(3,y)), минус(x, y))

Запись выражений в этой форме показывает то, что Лэндин назвал аппликативной структурой выражений в математическом языке (и языке программирования). Каждое выражение в таком языке может быть разбито на составляющие его части, каждая из которых является либо операцией, либо операндом. Операнд обозначает значение, в то время как операция обозначает функцию. Структура выражения проста: она состоит из операции, применяемой к операндам.

Важное понятие, связанное с аппликативной структурой, состоит в том, что значение выражения единственным образом определяется по значениям составляющих его частей. Таким образом, если некоторое выражение дважды встречается в одном и том же контексте, оно имеет одно и то же значение в обоих

случаях. Язык, в котором это свойство сохраняется для всех его выражений, обычно называют *аппликативным*. В этой книге мы называем его *строго функциональным языком*. Вообще используются оба названия, подчеркивая доминантную роль, которую играют функции и применение функций. Прежде чем начать наше исследование функционального программирования на строго функциональном языке, мы дадим в следующем разделе краткий обзор структуры традиционных языков программирования и обсудим те их черты, которые не являются строго функциональными.

1.2. Программирование при помощи процедур

Во всех современных языках программирования высокого уровня имеется конструкция, называемая процедурой или подпрограммой, понятие которой в значительной мере базируется на идее функции. Действительно, чаще всего эти процедуры используются для реализации новых функций в том смысле, в каком говорилось об этом в предыдущем разделе. В программах (1.1) — (1.3) функция *max* записана соответственно на

```
      REAL FUNCTION MAX(X,Y)
      REAL X,Y
      IF(X.GE.Y) GO TO 10
      MAX=Y                                          (1.1)
      RETURN
10    MAX=X
      RETURN
      END
```

```
MAX: PROCEDURE (X,Y) RETURNS (REAL);
       DECLARE X, Y REAL;
       DECLARE Z REAL;                               (1.2)
            IF X > Y THEN Z = X; ELSE Z = Y;
            RETURN(Z);
       END;
```

```
function max(x,y:real): real;
begin
   if x≥y then max:=x else max:=y                    (1.3)
end
```

Фортране, ПЛ/1 и Паскале. Эти определения реализуют строгие функции. Чтобы определить наибольшее из трех значений, на Фортране, например, функция может быть вызвана таким оператором присваивания:

M=MAX(MAX(A, B), C)

Однако в этих языках может быть определен только очень ограниченный класс функций, и в практических программах приходится прибегать к более общему использованию процедур.

Мы начнем отходить от строгих функций, когда определим процедуры, которые не выдают результат, но присваивают его значение одному из своих параметров. Например, можно определить функцию *MAX* так, что вызов *MAX (A, B, M)* вычисляет наибольшее из чисел *A* и *B* и присваивает его значение параметру *M*. Это записано в (1.4) — (1.6) соответственно на Фортране, ПЛ/1 и Паскале.

```
      SUBROUTINE MAX(X,Y,Z)
      REAL X,Y,Z
      IF(X.GE.Y) GO TO 10
      Z=Y                                            (1.4)
      RETURN
10    Z=X
      RETURN
      END
```

```
MAX: PROCEDURE (X,Y,Z);
       DECLARE X,Y,Z REAL;                           (1.5)
            IF X≥ Y THEN Z=X; ELSE Z=Y;
END;
```

```
procedure max(x,y:real; var z:real);
begin                                                (1.6)
   if x≥y then z:=x else z:=y
end
```

Чтобы вычислить наибольшее из трех значений, теперь нужно запрограммировать два последовательных вызова процедуры: *MAX (A, B, M)* и *MAX(M, C, M)*. Можно доказать, что здесь имеют место чисто синтаксические изменения в написании вызова функции, само понятие функции не нарушается. Мы лишились только некоторых удобств при написании той композиции применений, которая вычисляет наибольшее из трех значений.

Мы отходим немного дальше от понятия строгой функции, когда определяем *MAX* как такую процедуру, которая при вызове *MAX (A, B)* вычисляет наибольшее из чисел *A* и *B* и присваивает это значение параметру *A,* как это сделано в (1.7) — (1.9). Таким образом, последовательные вызовы *MAX (A, B)* и *MAX (A, C)* определяют наибольшее из трех значений (оставляя результат в *A),* но при этом могут изменить исходное значение параметра *A.*

```
      SUBROUTINE MAX(X,Y)
      REAL X, Y
      IF (X.GE.Y) GO TO 10
      X = Y
10    RETURN
      END
```
(1.7)

```
MAX: PROCEDURE (X,Y);
        DECLARE X,Y REAL;
            IF X< Y THEN X= Y;
        END;
```
(1.8)

```
procedure max(var x:real; y:real);
begin
    if x<y then x:=y
end
```
(1.9)

Снова можно констатировать, что строгое понятие функции не нарушено (только сделано неявным), так как результат вызова *MAX (A, B)* по-прежнему единственным образом определяется по исходным значениям аргументов. Программы, составленные из чистых функций, просто определяют, как новые значения вычисляются по исходным значениям. Однако процедуры (1.4)— (1.9) непросто вычисляют значения, но они имеют также побочный эффект присваивания относительно одного из своих параметров. При составлении программ из таких процедур приходится думать в терминах нарастающих изменений значений переменных при следующих одно за другим присваиваниях. Несмотря на использование локальных присваиваний, процедуры (1.1) — (1.3) не имеют побочного эффекта и поэтому реализуют строгие функции.

Более сильный пример побочного эффекта дает нам процедура, которая изменяет значение переменной, не являющейся ее параметром. В (1.10) — (1.12) процедура *MAX* определена таким образом, что вызов *MAX(A, B)* присваивает наибольшее из двух значений переменной *M.* Переменная *M* не является параметром. Наибольшее из трех значений вычисляется теперь

```
      SUBROUTINE MAX(X,Y)
      REAL X,Y
      COMMON M
      REAL M
      IF (X.GE.Y) GO TO 10
      M= Y
      RETURN
10    M=X
      RETURN
      END
```
(1.10)

```
MAX: PROCEDURE (X,Y):
        DECLARE X,Y REAL;
            IF X >Y THEN M=X; ELSE M=Y;
END;
```
(1.11)

```
procedure max(x,y:real);
begin
    if x >y then m:=x else m:=y
end
```
(1.12)

двумя последовательными вызовами *MAX (A, B)* в *MAX(M, C),* при этом результат остается в *M.* Процедура с такими определениями, как эти, приводит к программам, которые очень трудны для понимания. Но средства, позволяющие определять такие непонятные, неясные процедуры в традиционных языках программирования, включаются в эти языки по той причине, что без них программы не могут достаточно эффективно использовать машину.

Строго функциональный язык, который мы определим в следующей главе, не допускает побочных эффектов. Программист может только определять функции, которые вычисляют значения, единственным образом определяемые по значениям аргументов. Соответственно этому в строго функциональном языке отсутствуют многие понятия, знакомые по традиционным языкам. Наиболее важным является отсутствие присваивания. То же самое относится к известному в программировании понятию переменной, обладающей значением, которое время от времени изменяется присваиванием. Переменные в нашем строго функциональном языке скорее подобны математическим переменным,

которые мы использовали в предыдущем разделе как параметры в определениях функций. Они используются, только чтобы дать имена значениям аргументов функции, и связаны с этими постоянными значениями на протяжении всего вычисления алгебраического выражения, определяющего результат функции.

Зато строго функциональный язык обладает двумя весьма мощными средствами. Первое — это способность структурировать данные. Второе — возможность определять так называемые функции высших порядков. Мощность структурирования данных состоит в том, что целые структуры данных выступают как единые значения. Они могут вводиться в функцию как аргументы и выдаваться из нее как результаты. Наиболее важно, что однажды построенная структура не может изменяться, ее значение остается неизменным. Разумеется, эта структура может включаться в другие, но до тех пор пока в программе требуется ее значение, оно остается доступным. Этот специальный механизм имеет фундаментальную важность для применимости функциональных языков и требует большой осторожности при реализации, если должна быть получена достаточная эффективность. Почти то же самое можно сказать о функциях высших порядков. Функции высших порядков имеют другие функции либо своими аргументами, либо результатом. Использование таких функций может привести к более кратким и сжатым программам. Их правильная реализация составляет предмет рассмотрения последующих глав.

Будущие языки программирования явятся по необходимости компромиссом между языками, на которых программы могут быть ясно выражены, и языками, которые могут быть эффективно реализованы. Однако стоимость оборудования радикально изменяется, оборудование становится более дешевым и более мощным по всем параметрам. Эта тенденция только усиливается. Поэтому настало время вновь взглянуть на наши языки программирования и ответить на вопрос, какие средства высокого уровня можно попытаться включить в них, чтобы облегчить построение некоторых видов программ. Класс языков, на которые мы ссылаемся как на аппликативные, или строго функциональные, языки, широко изучается. Программы на этих языках могут быть на порядок короче, чем программы для решения тех же самых задач, но записанные на традиционных языках. Их, следовательно, легче написать правильно, легче понять и поэтому легче эксплуатировать. Цель этой книги — продемонстрировать эти возможности, описать технику про-, граммирования на таких языках, сопоставить эту технику с некоторыми теоретическими исследованиями в программировании и полностью описать способ реализации такого языка на современных машинах.

Глава 2.

СТРОГО ФУНКЦИОНАЛЬНЫЙ ЯЗЫК

Эта глава посвящается введению абстрактных понятий, которые используются на протяжении всей книги для представления функциональных программ. В предыдущей главе были использованы неформальные понятия и определены функции, которые отображали числовые значения в числовые. Чтобы записать интересующие нас программы, потребуются более богатые области определения функций, нежели обычные представимые в машине числа, для определенности будет использована специальная форма символьных данных, введенная Маккарти в языке Лисп. Эта форма данных, которую обычно называют S-выражением, имеет то преимущество, что является одновременно очень простой и весьма общей. Мы начнем с описания того, как можно записать символьное выражение, и затем введем основные функции манипулирования с ними. Опять-таки мы начнем точно с тех функций, которые являются основными в Лиспе, так что, когда в последующих главах будет введен наш собственный вариант Лиспа, предназначаемый для семантических спецификаций, мы сможем быть уверены, что опираемся на материал, уже знакомый читателю. Остальная часть главы посвящается методам построения еще более мощных функций. Поэтому мы обсудим различные вопросы, включая рекурсию, функции высших порядков, выбор параметров, чтобы таким образом полностью познакомить читателя с методами и понятиями функционального программирования. Понятия, введенные в этой главе, используются на протяжении всей книги.

2.1. Символьные данные

Определим класс символьных выражений, которые называются *S-выражениями* и образуют область определения для функциональных программ, которые мы будем позднее конструировать. Каждая из следующих строк содержит S-выражение:

(ДЖОН СМИТ 33 *ГОДА)*
((ДЭЙВ 17) *(МЭРИ* 24) *(ЭЛИЗАБЕТ* 6))
((МОЙ ДОМ) ИМЕЕТ (БОЛЬШИЕ (СВЕТЛЫЕ ОКНА)))

Самым очевидным общим свойством, которое можно подметить в этих примерах, является использование скобок. Действительно, использование скобок в S-выражениях является основополагающим. Если изменить положение или количество скобок, то вместе с этим изменится структура и соответственно смысл самого выражения. Этот раздел посвящается описанию правил построения S-выражений в той форме, в какой они преимущественно используются в этой книге. Небольшое расширение, которое оказывается полезным в последующих главах, отложено до разд. 2.9.

Кроме скобок в приведенных примерах S-выражения состоят из элементов, которые называются *атомами*. Вот примеры атомов:

ДЖОН —127
СМИТ *МОЙ*
33 *АВ13*

Атомы бывают двух типов — *символьные* и *числовые*. Символьный атом обычно является последовательностью букв, хотя может содержать и другие символы, включая цифры, но обязательно должен содержать по меньшей мере один символ, отличающий его от числа. Символьный атом рассматривается как одно неделимое целое. Мы никогда не расчленяем его на составляющие символы. К символьным атомам применяется только одна операция — сравнение, чтобы определить, одинаковые они или разные. Числовые атомы являются последовательностью цифр, возможно следующих за знаком, и обозначают, как обычно, десятичные числа. Вообще, S-выражение содержит смесь символьных и числовых атомов, хотя у нас будут и S-выражения с однотипными атомами.

Простейшей формой S-выражения является отдельный атом. Более сложной формой является простой список атомов. Он образуется заключением в скобки записанной последовательности атомов. Так, выражение

(ДЖОН СМИТ 33 *ГОДА)*

состоит из последовательности четырех атомов. Расположение S-выражения на странице и, в частности, промежутки не имеют значения, кроме того, что необходимо иметь по меньшей мере один пробел между соседними атомами, чтобы отличить их. Другие примеры простых списков даны в (2.1). Первый список состоит из восьми символьных атомов. Второй список состоит из пяти разнотипных атомов, третий состоит ровно из одного символьного атома *БАНАН*. Четвертый пример отличается от третьего, он состоит из пяти символьных однобуквенных атомов.

```
(ДЖОН СМИТ ОЧЕНЬ ЧАСТО ЕСТ ХЛЕБ
        С МАСЛОМ)
(14 ГРАДУСОВ И 50 МИНУТ)                (2.1)
(БАНАН)
(Б А Н А Н)
```

Можно получить S-выражения значительно более сложной структуры, если допустить, что элементами списка могут быть не только простые атомы, но сами S-выражения. Так, запись

((ДЭЙВ 17) *(МЭРИ* 24) *(ЭЛИЗАБЕТ* 6))

является S-выражением, так как представляет собой список из трех элементов, каждый из которых сам является S-выражением. Снова напомним, что расположение S-выражения на странице безразлично, однако скобки чрезвычайно важны. Как уже говорилось, выше записан трехэлементный список, каждый элемент которого сам является двухэлементным списком. В то же время запись

(ДЭЙВ 17 *МЭРИ* 24 *ЭЛИЗАБЕТ* 6)

является 6-элементным простым списком, который имеет совершенно другую структуру. Рассмотрим теперь построение более сложного примера. Поскольку S-выражениями являются элементы *БОЛЬШИЕ* и *(СВЕТЛЫЕ ОКНА),* то *(БОЛЬШИЕ (СВЕТЛЫЕ ОКНА))* тоже S-выражение. S-выражениями являются также *(МОЙ ДОМ)* и *ИМЕЕТ,* поэтому

((МОЙ ДОМ) ИМЕЕТ (БОЛЬШИЕ (СВЕТЛЫЕ ОКНА)))

тоже есть S-выражение. На практике списки могут вкладываться один в другой на большую глубину и иметь весьма сложную скобочную структуру. Трудности при чтении длинных S-выражений перекрываются легкостью их обработки.

Общие правила построения S-выражений можно просуммировать в следующем рекурсивном определении:

1) атом есть S-выражение,
2) последовательность S-выражений, заключенная в скобки, является S-выражением (списком).

В разд. 2.9 мы несколько расширим это определение, но в подавляющем большинстве примеров будем использовать S-выражения, определенные приведенными выше правилами.

Перед написанием программ сначала рассмотрим, как мы собираемся записывать наши данные в формате S-выражений. Представим себе, например, что мы хотели бы составить программу суммирования последовательности целых чисел. В какой

форме представили бы мы эти данные в программе? Вероятно, как список целых чисел. Например,

(127 462 45 781 —18 842 96)

или

(18 17 16 —16 —17 —18)

так как эта структура будет простейшей при обработке.

Пусть мы хотим выбрать представление в виде S-выражений для простых алгебраических формул. Здесь мы должны различать константы, переменные и бинарные операции. Пусть константы и переменные представляются атомами, а бинарная операция помещается на первом месте в списке и за ней следуют ее операнды. Далее, для операций мы выбираем специальные символьные имена. Мы можем подытожить описание способа представления формул *S*-выражениями, если приведем для каждой возможной формулы соответствующее ей *S*-выражение, как это сделано в табл. 2.1.

Таблица 2.1

Формула	*S-выражение*
Константа	Число
Переменная	Символ
$p+q$	*(ПЛЮС p q)*
$p-q$	*(МИНУС p q)*
$p\times q$	*(УМН p q)*
p/q	*(ДЕЛ p q)*
p^q	*(СТЕП p q)*

Здесь в столбце формул p и q заменяют части формулы, а в столбце S-выражений — соответствующие этим частям *S*-выражения. Так, из того, что формула

$2 \times x+1$

имеет вид p_1+q_1 , где $p_1=2 \times x$ и $q_x=1,$ а p_1 имеет вид $p_2 \times q_3$, где $p_2=2$ и $q_2=x,$ следует, что эта формула представляется *S*-выражением

(ПЛЮС (УМН 2 X) 1)

По аналогичным соображениям формула

$x^2+2x-31$

может быть представлена в виде

(ПЛЮС (СТЕП X 2) (МИНУС (УМН 2 X) 31))

Она может быть также представлена выражением

(МИНУС (ПЛЮС (СТЕП X 2) (УМН 2 X)) 31)

и многими другими возможными способами.

Любая программа, которую мы напишем для манипулирования с этой формулой, должна обрабатывать все эти различные формы.

В качестве третьего, совсем отличного от предыдущих примера рассмотрим программу обработки данных, связанных с игрой в шахматы. Во-первых, нужно уметь обозначать символами название и цвет каждой фигуры, например *(БЕЛЫЙ КОРОЛЬ)* или *(ЧЕРНАЯ ПЕШКА)*. Затем нужно уметь обозначать позицию каждой фигуры. Мы можем выбрать для представления координат пары целых чисел (предполагая, что строки и столбцы некоторым образом перенумерованы от 1 до 8). Так, клетки шахматной доски могут быть представлены парами (4 7), (2 1) и т. д. Теперь вся доска может быть представлена списком пар, в котором с каждой фигурой связана ее позиция. 1ак, шахматная позиция, изображенная на рис. 2.1, может быть представлена списком (2.2).

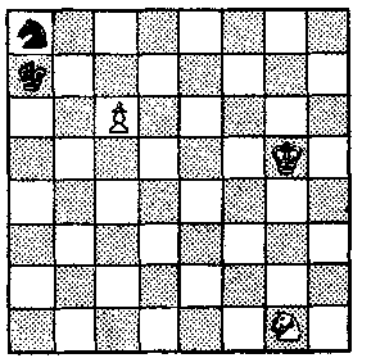


Рис. 2.1.

```
(((БЕЛЫЙ КОРОЛЬ) (4 7))
((ЧЕРНЫЙ КОРОЛЬ) (2 1))
((БЕЛАЯ ПЕШКА) (3 3))          (2.2)
((БЕЛЫЙ КОНЬ) (8 7))
((ЧЕРНЫЙ КОНЬ) (1 1)))
```

Разумеется, существует много различных способов для представления одних и тех же данных, и, какой мы в действительности выберем, почти целиком определяется тем, что именно должны мы с этими данными делать. Этот конкретный выбор шахматных данных может быть пригодным для одних целей и быть совершенно непригодным для других.

2.2. Элементарные селекторы и конструкторы

Для того чтобы оперировать с *S*-выражениями, введем некоторые примитивные функции, расчленяющие или составляющие эти выражения. Затем покажем, как можно определять новые функции в терминах этих примитивных. Например, можно будет определить функцию *длина(x),* которая, будучи применена к списку *x,* выдает целое число, равное количеству элементов в этом списке. В табл. 2.2 приводятся значения этой функции для нескольких списков.

Таблица 2.2

x	длина (x)
(A B)	2
(A B C D)	4
((A B)(C D))	2
(A)	1

Приводя примеры результатов функции, будем в дальнейшем использовать эту форму табулирования. Здесь в столбце *x* перечислены некоторые простые *S*-выражения и соответственно в столбце *длина(x)* — результаты вызова функции *длина* для этих значений *x.* Мы видим, что в списке учитываются только элементы верхнего уровня и не учитываются элементы подсписков.

Функция *длина(x)* не является примитивной. Прежде чем мы сможем определить ее, нам надо ввести некоторые примитивные функции над S-выражениями. Этим функциям даны их традиционные имена языка Лисп, хотя на первый взгляд они несколько необычны. Чем больше пользоваться ими, тем привычнее они станут. Первая функция, которая нам необходима, позволяет выбрать из списка первый элемент. Это функция *car* (табл. 2.3).

Таким образом, функция *car* может применяться к списку и ее результатом является первый (самый левый) элемент в списке. Мы видим, что выбранный элемент может быть либо атомом, либо списком. Функция *car* для атома не определена. Наряду с функцией *car* имеется связанная с ней функция *cdr.* Функция *cdr* от списка есть список, полученный из исходного отбрасыванием его первого члена. Для атома функция *cdr* не определена. Таким образом, функции *car* и *cdr* являются дополнительными. Если исходный список состоит из единственного атома, то значением функции *cdr* для этого списка является (специальный) атом *НИЛ (NIL).*

Таблица 2.3

x	car (x)
(A B)	A
(A B C)	A
((A B)(C D))	(A B)
(A)	A

При помощи двух этих примитивных функций можно выбрать из списка любой его элемент, зная, что он имеется в списке. Например, третий элемент списка *x* задается выражением

car(cdr(cdr(x)))

Так, если *x* есть список *(A B C D),* то *cdr(x)* есть *(B C D)* и *cdr(cdr(x))* есть *(C D).* Отсюда *car(cdr(cdr(x)))* есть элемент *C,* который является третьим в исходном списке.

Таблица 2.4

x	cdr (x)
(A B)	(B)
(A B C)	(B C)
((A B)(C D))	((C D))
(A)	НИЛ

Мы будем допускать введение новых непримитивных функций при помощи функциональных определений. Например, соотношение

третий(x)=car(cdr(cdr(x)))

определяет новую функцию *третий(x),* рассмотренную выше. Определяемая функция стоит слева от знака тождества, определение — справа. В определение могут входить примитивные функции, другие определенные функции и параметры определяемой функции. Рассмотрим, например, следующие определения функций, предназначенных для обработки *S*-выражений, являющихся списками ровно из двух элементов, которые в свою очередь являются такими же списками. Рассмотрим пример

((A B)
(C D))

начертание которого подсказывает нам имена, выбранные для наших функций, перечисленных в (2.3). Разумеется, эти функции могут быть не определены, если применяются к S-выражениям неподходящей формы.

$$\begin{aligned} &\textit{первый}(x) \equiv \textit{car}(x)\\ &\textit{второй}(x) \equiv \textit{car}(\textit{cdr}(x))\\ &\textit{верхнийлевый}(x) \equiv \textit{первый}(\textit{первый}(x))\\ &\textit{верхнийправый}(x) \equiv \textit{второй}(\textit{первый}(x))\\ &\textit{нижнийлевый}(x) \equiv \textit{первый}(\textit{второй}(x))\\ &\textit{нижнийправый}(x) \equiv \textit{второй}(\textit{второй}(x)) \end{aligned} \qquad (2.3$$

Теперь перейдем к примитивному конструктору — функции *cons.* Эта функция берет в качестве аргументов два S-выражения и соединяет их в единое S-выражение таким образом, чтобы исходные компоненты получались применением к результату функций *car* и *cdr* (табл. 2.5).

Таблица 2.5

x	*y*	*cons (x, y)*
A	*(B C)*	*(A B C)*
(A B)	*((C D))*	*((A B)(C D))*
(A B)	*(C D)*	*((A B) C D)*
A	*НИЛ*	*(A)*

Вообще, второй аргумент функции *cons* должен быть списком или специальным атомом *НИЛ.* Результатом *cons* тогда будет список, полученный включением первого аргумента в качестве первого элемента этого списка. Отсюда если *y* есть список длины *n,* то *cons(x, y)* будет списком длины *n+1,* независимо от значения *x.* В связи с этим удобно и принято рассматривать *НИЛ* как список нулевой длины, или пустой список. Это символьный атом *(НИЛ),* но он играет также специальную роль, обозначая пустой список.

В качестве примера использования функции cons рассмотрим определения:

$$\textit{двучлен}(x,y) \equiv \textit{cons}(x,\textit{cons}(y,\textit{НИЛ}))$$
$$\textit{квадрат}(a,b,c,d) \equiv \textit{двучлен}(\textit{двучлен}(a,b),\textit{двучлен}(c,d))$$

Функция *квадрат* строит структуру, которая может быть расчленена использованием функций *верхнийлевый, верхнийправый, нижнийлевый* и *нижнийправый.* Эти функции даны просто как пример использования примитивов *car, cdr* и *cons:* они не используются далее в книге. Для дальнейшего чтения, однако, важно иметь точное представление о функциях *car, cdr* и *cons.*

2.3. Элементарные предикаты и арифметика

Для того чтобы обрабатывать списки различной структуры, необходимо: а) уметь распознавать, является ли значение S-выражения атомом или списком, и б) устанавливать равенство атомов. Функция *атом(x)* называется предикатом, так как она выдает значение истина или ложь. Мы будем обозначать истину атомом *И* и ложь атомом *Л.* Функция *атом(x)* принимает значение *И* только в том случае, когда *x* является символьным или числовым атомом. Некоторые примеры даны в табл. 2.6.

Таблица 2.6

x	*атом (x)*
A	*И*
(A)	*Л*
(A B C)	*Л*
СЛОВАРЬ	*И*
НИЛ	*И*
127	*И*
(127)	*Л*

Мы используем такие предикаты, чтобы проверить символьное значение, прежде чем применить к нему функцию, которая в ряде случаев может быть не определена. Рассмотрим, например, функцию

$$f(x) \equiv \textbf{если } \textit{атом}(x) \textbf{ то } \textit{НИЛ} \textbf{ иначе } \textit{car}(x)$$

которая выдает первый элемент списка, если применяется к списку, но выдает *НИЛ,* если применяется к атому. В то время как функция *car(x)* не определена для некоторых S-выражений (частичная функция), *f(x)* определена для всех выражений (всюду определенная функция). Отметим, что функция *атом(x)* не различает символьные и числовые атомы.

Можно также сравнить два атома и определить, равны они или нет. Это делает предикат *равно(x, y),* но нужна осторожность при его использовании. Либо одно, либо другое из сравниваемых значений должно быть атомом. Если оба значения являются' списками, результат не определен. Кроме того, ре-

Таблица 2.7

x	*y*	*равно (x, y)*
A	*A*	*И*
A	*B*	*Л*
127	127	*И*
127	128	*Л*
127	*A*	*Л*
(*A B*)	*A*	*Л*
(*A*)	*A*	*Л*
A	(*A B*)	*Л*

зультат *равно(x, y)* принимает значение *И* только в том случае, когда *x* и *y* являются одним и тем же атомом, и значение *Л,* если либо это различные атомы, либо один из аргументов не является атомом (см. табл. 2.7). Обычный способ использования этого предиката иллюстрирует следующее определение:

размер(*x*) ≡ **если** *равно*(*x*, *НИЛ*) **то** 0 **иначе**
если *равно*(*cdr* (*x*), *НИЛ*) **то** 1 **иначе** 2

Эта функция выдает значение 0, 1 или 2 соответственно тому, что ее аргумент является списком из 0, 1 или более атомов. Теперь мы вплотную подошли к определению функции *длина,* которая была введена в начале предыдущего раздела. Сначала, однако, перечислим арифметические функции, которые мы допускаем.

Будут использоваться арифметические операции +, —, *X, /* (или дел) и **ост.** Они вычисляют соответственно сумму, разность, произведение, частное и остаток двух операндов, которые являются целыми числами. Операция **ост** определяется соотношением

$$x \text{ ост } y = x - (x \div y) \times y$$

т. е. она дает остаток при делении нацело двух целых чисел.

Если мы хотим определить функцию, которая выдает двучлен из частного от деления нацело и остатка, то записываем

частост (*x*, *y*) ≡ *двучлен* (*x* ÷ *y*, *x* **ост** *y*)

Примеры значений этой функции приведены в табл. 2.8.

Мы видим, что частное *q* от деления нацело и остаток *r* удовлетворяют соотношению $x = q \times y + r$.

Будем также использовать предикат $x \leqslant y$, чтобы определять является ли *x* меньше или равен *y* (только для числовых атомов).

Таблица 2.8

x	*y*	*частост (x, y)*
17	3	(5 2)
17	—3	(—5 2)
—17	3	(—5 —2)
—17	—3	(5 —2)

Предикат не определен для символьных атомов или списков. В качестве примера использования этого предиката рассмотрим функцию

расстояние (*x*, *y*) ≡ **если** $x \leqslant y$ **то** $y - x$ **иначе** $x - y$

значением которой является модуль разности между *x* и *y.* Аналогичные предикаты можно построить для других обычных отношений между числами <, > и $\geqslant$. Однако в последующих главах (4 и 6), где вводится наш вариант Лиспа и будет построен для него интерпретатор и компилятор, мы будем ограничиваться в языке только отношением $\leqslant$, чтобы сократить размеры реализующих программ. Поэтому в книге используются примеры только с отношениями $\leqslant$ и =.

2.4. Рекурсивные функции

Рассмотрим теперь, как можно определить функцию *длина(x),* применяемую к спискам и выдающую в качестве результата количество элементов в этом списке. Нам требуется функция, которая, в частности, выдает значения, приведенные в табл. *2.9.*

Таблица 2.9

x	*длина (x)*
(*A B C D*)	4
(*B C D*)	3
(*C D*)	2
(*D*)	1
НИЛ	0

Эта последовательность значений подсказывает следующий алгоритм: последовательно берем *cdr* от *x* и считаем, сколько раз это может быть сделано, пока не дойдем до *НИЛ.* Однако это

еще не позволяет нам немедленно программировать алгоритм как рекурсивную функцию. Вообще при оперировании со списками отдельно рассматриваем два случая: *x=НИЛ* и *x=/=НИЛ.* Во втором случае, так как можно взять *cdr* от *x* и это будет списком, можно применять функцию, определяемую рекурсивно. Рекурсия обязательно завершится, поскольку с каждым рекурсивным вызовом список становится на один элемент короче.

Этот прием можно применить для построения функции длина *(x)*, как показано в (2.4).

$$\begin{array}{lll} \text{случай (1)} & x = НИЛ & длина(x) = 0 \\ \text{случай (2)} & x \neq НИЛ & \text{пусть } длина(cdr(x)) = n \\ & & \text{тогда } длина(x) = n + 1 \end{array} \qquad (2.4)$$

Случай *x=НИЛ* тривиален. В случае*x=/=НИЛ*мы предполагаем наличие результата рекурсивного вызова, примененного к *cdr(x).* Здесь предполагается, что *длина(cdr(x))=n,* и в этом случае *длина(x)* должна быть на единицу больше. Это приводит нас к записи определения функции *длина (x)* в следующем виде:

длина(x) на **если** *равно(x,НИЛ)* **то** 0 **иначе** *длина(cdr(x))+1*

Можно применить этот метод построения к другой подобной же функции *сумма(x),* которая берет в качестве аргумента список целых чисел и выдает как результат сумму этих чисел (табл. 2.10). Так как *НИЛ* представляет пустой список, т. е. список с нулевым количеством чисел в нем, его сумма есть 0.

Таблица 2.10

x	*сумма (x)*
(1 2 3)	6
(2 —2)	0
НИЛ	0

Рассмотрение возможных случаев приводит к анализу, заданному соотношениями (2.5).

$$\begin{array}{lll} \text{случай (1)} & x = НИЛ & сумма(x) = 0 \\ \text{случай (2)} & x \neq НИЛ & \text{пусть } сумма(cdr(x)) = n \\ & & \text{тогда } сумма(x) = car(x) + n \end{array} \qquad (2.5)$$

Это очень похоже на функцию длинв. Случай *x=НИЛ* тривиален.

Случай *x =/= НИЛ* приводит к определению *суммы от* списка *x* в виде *car* от *x* плюс *сумма* от *cdr* этого списка. Записываем определение функции в следующем виде:

сумма(**x**)=**если***равно(x, НИЛ)* то 0 **иначе** *car(x)+сумма{cdr(x))*

Третьим примером будет функция *соединить (x, y),* которая берет в качестве аргументов два списка *x* и *y* и выдает как результат единый список, содержащий последовательно все элементы списка *x* и все элементы списка *y* (табл. 2.11).

Таблица 2.11

x	*y*	*соединить (x, y)*
(A B C)	(D E)	(A B C D E)
(A B)	(C D E)	(A B C D E)
НИЛ	(A B C)	(A B C)
(A B C)	*НИЛ*	(A B C)
НИЛ	*НИЛ*	*НИЛ*

Так как у функции *соединить* два аргумента и оба — списки, мы должны сделать анализ случаев (2.6) для каждого из них.

$$\begin{array}{lll} \text{случай (1) } x = НИЛ & & \\ \quad \text{подслучай (1.1) } y = НИЛ & & соединить(x,y) = НИЛ \\ \quad \text{подслучай (1.2) } y \neq НИЛ & & соединить(x,y) = y \\ \text{случай (2) } x \neq НИЛ & & \\ \quad \text{подслучай (2.1) } y = НИЛ & & соединить(x,y) = x \\ \quad \text{подслучай (2.2) } y \neq НИЛ & & \\ \quad\quad \text{пусть } соединить(cdr(x),y) = z & & \\ \quad\quad \text{тогда } соединить(x,y) = cons(car(x),z) & & \end{array} \qquad (2.6)$$

Здесь во всех случаях, кроме (2.2), можно сразу выписать результат, а в случае (2.2) используется рекурсия. Хотя можно было бы взять *cdr* как от *x,* так и от *y,* на самом деле нужно только *cdr* от *x.* Здесь завершение рекурсии гарантируется конечностью *x.* По ходу этого анализа мы на каждом шаге исследуем, чем располагаем, и смотрим, не можем ли мы прямо написать ответ. Если известно, что некоторый список не *НИЛ,* то смотрим, не можем ли мы написать неявный ответ в терминах *cdr* от этого списка. В случае (2.2) имеются три кандидата на вычисление значений рекурсивным вызовом:

соединить(cdr(x), y)
соединить(x, cdr(y))
соедишить(cdr(x), cdr(y))

записать так:

соединить(*x*, *y*) ≡ **если** *равно*(*x*, *НИЛ*) **то**
если *равно*(*y*, *НИЛ*) **то** *НИЛ* **иначе** *y* **иначе**
если *равно*(*y*, *НИЛ*) **то** *x* **иначе**
cons(*car*(*x*), *соединить* (*cdr*(*x*), *y*))

Но в таком виде обычно эту функцию не записывают, потому что многие замечают, что при *x=НИЛ* функция *соединить (x, y)=y* независимо от того, будет ли *y* равен *НИЛ* или нет. То есть подвыражение

если *равно (y, НИЛ)* то *НИЛ* иначе *y*

можно заменить на *y,* откуда получаем

соединить(*x*, *y*) ≡ **если** *равно*(*x*, *НИЛ*) **то** *y* **иначе**
если *равно*(*y*, *НИЛ*) **то** *x* **иначе**
cons(*car*(*x*), *соединить*(*cdr*(*x*), *y*)

Далее, некоторые предпочитают использовать тот факт, что при *x=/=НИЛ*

соединить (x, y)=cons(car(x), соединить (cdr(x), y))

независимо от того, будет ли *y* равен *НИЛ* или нет. Это означает, что проверка *y* может быть опущена и определение записывается в виде

соединить(*x*, *y*) ≡ **если** *равно*(*x*, *НИЛ*) **то** *y* **иначе**
cons(*car*(*x*), *соединить*(*cdr*(*x*), *y*))

Все три версии определяют эквивалентные функции. Все они дают один и тот же результат. Одна может представляться легче для понимания, чем другая. Первую версию можно предпочесть за то, что она явно перечисляет все случаи, а третью — за ее краткость. Вторая и третья версии имеют, однако, существенно разную эффективность. Вторая версия избегает вычислений в случае *y=НИЛ* ценой излишних проверок, когда *y=/=НИЛ.* Третья версия избегает повторных проверок *y,* однако рекурсивно перестраивает *x* даже при *y=НИЛ.*

2.5. Другие рекурсивные функции

Часто бывает удобно при определении функции вводить вспомогательные функции для облегчения нашей задачи. Например, при определении функции *соединить (x, y)* мы могли бы использовать дополнительную промежуточную функцию *соед(x, y),* которая соединяет *x* и *y* в предположении *y=/=НИЛ.* Тогда мы имели бы определение

соединить(*x*, *y*) ≡ **если** *равно*(*y*, *НИЛ*) **то** *x* **иначе** *соед*(*x*, *y*)
соед(*x*, *y*) ≡ **если** *равно*(*x*, *НИЛ*) **то** *y* **иначе**
cons(*car*(*x*), *соед*(*cdr*(*x*), *y*))

Отметим, что здесь объединяются достоинства второй и третьей версии функции *соединить (x, y)* из предыдущего раздела. В функциональном программировании на пути определения главной функции принято определять новые функции в терминах старых, устанавливая таким образом множество функций, определенных одна через другую. Таким образом, функциональная программа состоит из множества функций, одна из которых рассматривается как первопричина других. Это функция, которую программист рассчитывает вычислить, и она как бы является главной программой, в то время как остальные функции служат для нее подпрограммами.

Проблема выбора подфункций при разработке главной функции является извечной проблемой хорошего структурирования программы. Иногда стандартные подфункции выявляются сами собой, но более часты случаи, когда удачный подбор подфункций специального назначения может значительно упростить структуру множества функций в целом. Если функции, выбранные в этой книге, как кажется, обладают этим качеством, то это потому, что были отвергнуты многие предварительные варианты. И на самом деле, чтобы сделать хорошо структурированную функциональную программу, можно дать только один совет: пытаться непрерывно улучшать то, что имеется. В этом разделе мы рассмотрим построение наиболее разработанных функций.

Интересной функцией, с которой мы и далее будем встречаться, является функция *обратить (x),* которая выдает список с элементами, перечисленными в обратном порядке (табл. 2.12).

x	*обратить (x)*
(A B C)	*(C B A)*
((A B)(C D))	*((C D)(A B))*
НИЛ	*НИЛ*

Заметим, что если элементы списка в свою очередь являются списками, то элементы этих последних не обращаются. По-

скольку *x* список, мы должны сделать анализ случаев. Если *x* есть *НИЛ,* то *обратить(x)* тоже *НИЛ.* Если *x=/=НИЛ,* то можно использовать *обратить(cdr(x)).* Итак, для начала мы имеем *обратить (cdr(x)),* и, чтобы завершить задачу, нужно иметь функцию, которая могла бы поместить *car(x)* в конец *обратить (cdr(x)).* С этой точки зрения в разработке полезна функция

добавить (x, y)

которая получает список *x* и элемент *y* и делает *y* новым последним членом списка *x.* Несколько примеров таких действий дано в табл. 2.13.

Таблица 2.13

x	*y*	*добавить (x,y)*
(A B C)	*D*	*(A B C D)*
((A B)(C D))	*(E F)*	*((A B)(C D)(E F))*
НИЛ	*A*	*(A)*

Если предположить, что мы можем построить такую функцию, но сделаем это позже, то можно выписать законченное построение (2.7) функции *обратить(x).*

$$\begin{array}{l}\text{случай (1) } x = НИЛ \quad обратить(x) = НИЛ\\ \text{случай (2) } x \neq НИЛ\\ \quad \text{пусть } обратить(cdr(x)) = z\\ \quad\quad \text{тогда } обратить(x) = добавить(z, car(x))\end{array} \qquad (2.7)$$

Это можно записать как функциональное определение

$$обратить(x) \equiv \textbf{если } равно(x, НИЛ) \textbf{ то } НИЛ \textbf{ иначе } добавить(обратить(cdr(x)),\ car(x))$$

Теперь можно построить функцию *добавить.*

Эта функция имеет два аргумента, первый из которых список и второй — некоторое S-выражение. Поэтому наш анализ случаев касается только первого аргумента. На самом деле это так похоже на нашу предыдущую разработку, что мы можем немедленно выписать построение (2.8):

$$\begin{array}{l}\text{случай (1) } x = НИЛ \quad добавить(x, y) = cons(y, НИЛ)\\ \text{случай (2) } x \neq НИЛ\\ \quad \text{пусть } добавить(cdr(x), y) = z\\ \quad\quad \text{тогда } добавить(x, y) = cons(car(x), z)\end{array} \qquad (2.8)$$

Это построение приводит к функциональному определению

$$добавить(x,\ y) \equiv \textbf{если } равно(x,\ НИЛ) \textbf{ то } cons(y, НИЛ) \textbf{ иначе } cons(car(x),\ добавить(cdr(x),\ y))$$

Теперь определение функции *обратить* использует *добавить* как подфункцию, и поэтому эти две функции вместе образуют программу обращения списка. Между прочим, мы могли бы определить функцию *добавить* через функцию *соединить* следующим образом:

$$добавить\,(x,\ y) \equiv соединить\,(x,\ cons\,(y,\ НИЛ))$$

В этом случае программа для обращения списка состояла бы из трех функциональных определений. Однако привычнее включить вызов *соединить* непосредственно в функцию *обратить* и избавиться от функции *добавить,* что приводит к программе (2.9).

$$\begin{array}{l}обратить(x) \equiv \textbf{если } равно(x, НИЛ) \textbf{ то } НИЛ \textbf{ иначе}\\ \qquad соединить(обратить(cdr(x)),\ cons(car(x), НИЛ))\\ соединить(x, y) \equiv \textbf{если } равно(x, НИЛ) \textbf{ то } y \textbf{ иначе}\\ \qquad cons(car(x),\ соединить(cdr(x), y))\end{array} \qquad (2.9)$$

Для этой последней программы можно провести интересные подсчеты. Функция *соединить (x, y)* вызывает себя рекурсивно *n* раз, если длина списка *x* равна *n.* Отсюда обнаруживаем, что всякий раз, как мы используем *соединить (x, y),* будет *n* раз вызываться *cons.* Теперь рассмотрим функцию *обратить (x).* Из аналогичных соображений эта функция вызывает себя рекурсивно *n* раз, если длина списка *x* равна *n.* Отсюда функция *обратить* делает также *n* вызовов *cons,* но она же делает и *n* вызовов функции *соединить.* Для каждого из этих *n* вызовов *соединить* длина первого ее аргумента равна соответственно О, 1, 2, . . ., *n*—1. Тогда количество вызовов *cons* функцией *соединить* от имени *обратить* есть

$$0+1+2+\ldots+(n-1)=n(n-1)/2$$

Общее количество вызовов *cons* в функции *обратить* будет поэтому

$$n+n(n-1)/2 = n(n+1)/2$$

что является очень большим числом (табл. 2.14).

При обращении списка длины *n* можно было бы рассчитывать на *n* вызовов *cons.* Число излишних вызовов, сделанных функцией *обратить,* представляется чрезмерным. В гл. 12 мы увидим, что вызовы *cons* дорого обходятся, но еще прежде в разд. 2.6 фактически будет показано, как можно перепрограммировать

Таблица 2.14

длина (x)	число вызовов *cons* функцией *обратить (x)*
10	50
100	5050
1000	500500

функцию *обратить,* чтобы избежать лишних вызовов *cons,* что сделает программу более быстрой, даже если не знать, как дорогостоящи вызовы *cons.*

Обратимся теперь к более типичным применениям S-выражений. Разработаем функцию *набор (t),* которая получает как аргумент список *t* атомов и выдает в качестве результата список, в котором каждый атом из *t* имеет ровно одно вхождение (табл. 2.15).

Таблица 2.15

t	*набор (t)*
(*А В А В С А*)	(*А В С*)
(*И В ПОЛЕ И В ЛЕСУ*)	(*И В ПОЛЕ ЛЕСУ*)

Такое использование списка, который выдает эта функция, является обычным применением S-выражений. Результатом функции *набор(t)* является множество, представленное в виде списка без повторений. Теперь построим функцию *набор(t)* и проведем анализ случаев относительно *t.*

$$\begin{array}{l}\text{случай (1) } t = НИЛ \quad набор(t) = НИЛ\\ \text{случай (2) } t \neq НИЛ\\ \quad \text{пусть } набор(cdr(t)) = s\\ \quad\quad \text{тогда } набор(t) = включить(s, car(t))\end{array} \qquad (2.10)$$

Здесь случай *t=НИЛl* тривиален, как обычно. В случае *i=/=НИЛ,* где мы предположили, что список атомов, использованных в *cdr(t),* есть s, для того чтобы вычислить список атомов, использованных в *t,* мы должны добавить к *s car(t),* если этот элемент еще не встречался. Предполагаем, что мы можем определить соответствующую функцию *включить (s,x),* которая делает это. Тогда можно построить функцию

набор (t) ≡ **если** *равно (t, НИЛ)* **то** *НИЛ* **иначе**
включить (набор (cdr (t)), car (t))

функция *включить (s, x)* тривиальна, если предположить, что имеется функция, проверяющая, является ли *x* элементом множества s. Имеем

включить (s, x)=если *элемент (x, s)* то s иначе *cons(x,* s)

Остается построить функцию *элемент(x, s).* Вспомним, что, хотя *s* логически является множеством, оно все еще остается и списком, и поэтому можно провести обычный анализ случаев (2.11).

$$\begin{array}{l}\text{случай (1) } s = НИЛ \quad элемент(x,s) = Л\\ \text{случай (2) } s \neq НИЛ\\ \quad \text{пусть } элемент(x,cdr(s)) = b\\ \quad \text{тогда } элемент(x,s) = \text{если } b \text{ то } И \text{ иначе}\\ \quad \text{если } x = car(s) \text{ то } И \text{ иначе } Л\end{array} \qquad (2.11)$$

Мы написали все правильно, но это описание приводит к несколько неэффективной функции

элемент(x, s) ≡ **если** *равно(s, НИЛ)* **то** *Л* **иначе**
если *элемент(x, cdr(s))* **то** *И* **иначе**
если *равно(x, car(s))* **то** *И* **иначе** *Л*

Простое переупорядочение проверок позволит избежать поиска ло всему списку в случае *x=car(s).* Можно записать

элемент(x, s) ≡ **если** *равно(s, НИЛ)* **то** *Л* **иначе**
если *равно(x, car(s))* **то** *И* **иначе**
если *элемент(x, cdr(s))* **то** *И* **иначе** *Л*

В действительности функция может быть еще несколько подправлена и приведена к виду, в котором она обычно известна, если принять к сведению, что выражение

если *b* то *И* **иначе** *Л*

имеет всегда то же самое значение, что и *b* (если *b* принимает логические значения).

элемент(x, s) ≡ **если** *равно(s, НИЛ)* **то** *Л* **иначе**
если *равно(x, car(s))* **то** *И* **иначе**
элемент(x, cdr(s))

Мы закончили программу для функции *набор (t),* которая состоит из трех функций: *набор, включить* и *элемент.* Кратко рассмотрим обобщение. Расширим функцию *набор(t)* так, чтобы

она могла действовать и тогда, когда элементами *t* являются снова списки и вообще списки, вложенные на произвольную глубину. Другими словами, *t* является общей списочной структурой и *набор(t)* есть список атомов, которые по меньшей мере однажды входят в *t*. Отсюда результат снова является множеством, представленным в виде списка атомов (табл. 2.16).

Таблица 2.16

t	*набор (t)*
(*ПЛЮС* (*УМН X* 2) (*УМН X X*))	(*ПЛЮС УМН X* 2)
(((*A B*)*A*)(*A*(*B A*)))	(*A B*)

Перестроим функцию *набору),* как это показано в (2.12).

```
случай (1) t = НИЛ     набор(t) = НИЛ
случай (2) t ≠ НИЛ
  пусть набор(cdr(t)) = s
  тогда
подслучай (2.1) car(t) есть атом                          (2.12)
                набор(t) = включить(s, car(t))
подслучай (2.2) car(t) есть список
  пусть набор(car(t)) = s'
  тогда набор(t) = объединить(s', s)
```

Теперь при t=/=*НИЛ* имеем два подслучая. Когда *car(t)* — атом, имеем тот же результат, что и прежде. Если *car(t)* — список, то мы должны вызвать *набор(car (t))*, чтобы определить множество атомов, входящих в *car(t)*. Теперь у нас два множества: s и *s'*, и мы должны вычислить результат путем объединения этих множеств, т. е. построить множество, которое содержит все элементы как *s,* так и *s'*. Следовательно, мы построили функцию

набор(t) ≡ **если** *равно(t, НИЛ)* **то** *НИЛ* **иначе**
если *атом(car(t))* **то** *включить* (*набор* (*cdr(t)*), *car(t)*) **иначе**
объединить(*набор*(*car*(*t*)), *набор*(*cdr*(*t*)))

Чтобы завершить это расширение, необходимо построить функцию *объединить(u, v),* где *u* и *v*—множества. Это делается непосредственно, аналогично тому, как строились многие функции до сих пор. Имеем анализ случаев (2.13).

```
случай (1) u = НИЛ     объединить(u, v) = v
случай (2) u ≠ НИЛ
  пусть объединить(cdr(u), v) = w                          (2.13)
    тогда объединить(u, v) = включить(w, car(u))
```

Таким образом, если *u=/=НИЛ,* вычисляем объединение *cdr(u)* и *v* и включаем *car(u)* в это множество. Это дает нам функцию

объединить(u, v) ≡ **если** *равно* (*u*, *НИЛ*) **то** *v* **иначе**
включить(*объединить*(*cdr*(*u*), *v*), *car*(*u*))

Расширенная программа для функции *набор(t)* использует четыре функции, определения которых повторно выписаны в (2.14—2.17).

```
набор(t) ≡ если равно (t, НИЛ) то НИЛ иначе
           если атом(car(t)) то включить (набор(cdr(t)), car(t))   (2.14)
           иначе объединить (набор(car(t)), набор(cdr(t)))

объединить(u, v) ≡ если равно(u, НИЛ) то v иначе
                   включить(объединить(cdr(u), v), car(u))         (2.15)

включить(s, x)   ≡ если элемент(x, s) то s иначе cons(x, s)        (2.16)

элемент(x, s)    ≡ если равно(s, НИЛ) то Л иначе
                   если равно(x, car(s)) то И иначе                (2.17)
                   элемент(x, cdr(s))
```

2.6. Накапливающие параметры

В этом разделе вводится один интересный метод, часто используемый в функциональном программировании. Начнем иллюстрацию этого метода, заново программируя функцию *обратить(x)*. Основная идея состоит в том, чтобы определить вспомогательную функцию с лишним параметром, который используется для накапливания требуемого результата, В случае обращения мы определяем функцию *обр(x, y)* с тем свойством, что *x* есть обращаемый список, а *y* — параметр, накапливающий обращенный список. Имеем определение функции

обр(x, y) ≡ **если** *равно(x, НИЛ)* **то** *y* **иначе**
обр(*cdr*(*x*), *cons*(*car*(*x*), *y*))

через которую можно определить функцию *обратить(x):*

обратить(x)= обр(x, НИЛ)

Пожалуй, легче увидеть, как действует эта функция, чем описывать это. Когда вызвана функция *обр(x, y),* список *y* уже накопил в обратном порядке все рассмотренные к этому моменту элементы обращаемого списка. Следовательно, когда *x* есть

НИЛ, то *y* накопил уже весь результат в целом, а если *x=/=НИЛ,* то мы можем накопить в *y car* от *x* и вызвать рекурсивно функцию *обр* для обработки *cdr* от *x.* В табл. 2.17 даны значения *x* и *y,* полученные при последовательных вызовах функции *обр,* если первоначально функция *обратить* применяется к списку *(A B C D).*

Таблица 2.17

x	*y*
(A B C D)	*НИЛ*
(B C D)	*(A)*
(C D)	*(B A)*
(D)	*(C B A)*
НИЛ	*(D C B A)*

В случае функции *обр(x, y)* второй параметр *y* является накапливающим. Проблема построения таких функций является проблемой подбора соответствующей подфункции с накапливающим параметром, который вычисляет требуемый результат. Здесь мы не применяли нашей обычной методики с анализом случаев, но, скорее, взяли *обр(x, y)* наугад и затем описали, как она работает.

Чтобы применить наш анализ случаев, мы должны решить, каким будет общий вид результата *обр(x, y)* при произвольном выборе значения *y.* Пусть результатом *обр(x, y)* в случае, когда *x* и *y* оба являются списками (возможно, пустыми), будет список всех элементов *x,* взятых в обратном порядке, дополненный всеми элементами *y* (в их первоначальном порядке). Формально можно написать

обр{x, y)=соединить(обратить(x), y)

хотя это не является подходящим определением, так как нам нужно определить функцию *обратить.* Теперь можно вывести наш проект (2.18) из анализа случаев по *x.*

$$
\begin{array}{l}
\text{случай (1) } x = НИЛ \quad обр(x, y) = y \\
\text{случай (2) } x \neq НИЛ \\
\quad \text{пусть } обр(cdr(x), z) = соединить\ (обратить(cdr(x)), z) \\
\quad\quad \text{тогда } обр(x, y) = соединить\ (обратить(x), y) = \\
\qquad\qquad = соединить\ (обратить(cdr(x)), cons(car(x), y)) \\
\qquad\qquad = обр(cdr(x), cons(car(x), y))
\end{array}
\qquad (2.18)
$$

Это требует некоторых пояснений. Это скорее доказательство правильности функции, данной в начале раздела, чем ее построение, поскольку преобразования в случае (2) весьма сложны. Рассмотрим случаи по порядку.

При *x=НИЛ,* так как мы требуем, чтобы

обр(x, y) — соединить(обратить(x), y)

то заключаем просто, что *обр(x, y)=y.* Однако, когда *x=/=НИЛ,* так как по-прежнему требуется выполнение приведенного выше равенства, следует предположить, что

обр(cdr(x), z)=соединить(обратить(cdr(x)), z)

для любого *z.* Это предположение аналогично тому, которое делается в математических доказательствах по индукции, и оно справедливо для таких доказательств. Мы делаем следующее. Предполагая, что можно определить функцию *обр* для любого первого аргумента, более короткого чем *x,* показываем затем, что можно определить эту функцию и для *x.* Теперь, имея соотношение

соединить(рбратить(x),y) = соединить(обратить(cdr (x)), cons(car(x), y))

Заключаем, что выбор

z=cons(car(x), y)

является подходящим для определения функции *обр(x, y)* через *обр(cdr(x), z).*

Чтобы завершить это определение функции *обратить,* использующее накапливающий параметр, подсчитаем количество вызовов функции *cons* в этом варианте. Ясно, что функция *обратить(x)* делает все свои вызовы *cons* в *обр(x, НИЛ).* Однако *обр(x, y)* вызывает себя рекурсивно *n* раз, если *n* — длина списка *x.* Таким образом, *обр(x, y)* вызывает *cons* ровно л раз, а, следовательно, столько же раз вызывает *cons* и функция *обратить.* Сравним это с числом вызовов *n(n+1)/2* в случае первого определения функции (2.9). Имеем огромную экономию.

Теперь применим метод накапливающих параметров в более типичном случае. Это случай, когда надо определить функцию, которая накапливает более чем один результат. В одном из предыдущих разделов была определена функция *сумма(x),* которая вычисляла сумму списка целых чисел. Аналогичную функцию можно определить для вычисления произведения (см. упр. 2.1). Рассмотрим определение функции *сумпроизв(x),* которая выдает в качестве результата двучлен, у которого первый элемент является суммой, а второй — произведением элементов *x.* То есть мы хотим определить функцию так:

сумпроизв(x) = двучлен (сумма (x), произв(x))

Чтобы сделать это, используя накапливающие параметры, мы введем вспомогательную функцию *сп(x, s, p)* с двумя дополнительными параметрами, которые будут накапливать сумму и произведение соответственно. Иначе говоря, попытаемся определить функцию *сп* в виде

сп(x, s, p) = двучлен(s + сумма(x), p x *произв(x))*

и тогда можем определить

сумпроизв(x) = сп(x, 0, 1)

Функцию *сп(x,* s, *p)* строим путем анализа случаев по *x* (2.19).

$$
\begin{aligned}
&\textit{случай (1) } x = \textit{НИЛ} \quad \textit{сп}(x, s, p) = \textit{двучлен}(s, p)\\
&\textit{случай (2) } x \neq \textit{НИЛ}\\
&\quad \text{пусть } \textit{сп}(\textit{cdr}(x), s', p') = \textit{двучлен}\ (s' + \textit{сумма}(\textit{cdr}(x)),\\
&\qquad p' \times \textit{произв}(\textit{cdr}(x)))\\
&\quad \text{тогда } \textit{сп}(x, s, p) = \textit{двучлен}(s + \textit{сумма}(x),\ p \times \textit{произв}(x))\\
&\qquad = \textit{двучлен}(s + \textit{car}(x) + \textit{сумма}(\textit{cdr}(x)),\\
&\qquad\quad p \times \textit{car}(x) \times \textit{произв}(\textit{cdr}(x))) =\\
&\qquad = \textit{сп}(\textit{cdr}(x),\ s + \textit{car}(x),\ p \times \textit{car}(x))
\end{aligned}
\tag{2.19}
$$

Как и прежде, мы провели весьма длинный вывод. При этом нам потребовалось не только выбрать подфункции, но и использовать простые равенства при *x=/=НИЛ*

$$\textit{сумма}(x) = \textit{car}(x) + \textit{сумма}(\textit{cdr}(x))$$
$$\textit{произв}(x) = \textit{car}(x) \times \textit{произв}(\textit{cdr}(x))$$

Это приводит нас к определению

$$\textit{сп}(x,\ s,\ p) \equiv \textbf{если } \textit{равно}(x,\ \textit{НИЛ}) \textbf{ то } \textit{двучлен}(s,\ p) \textbf{ иначе } \textit{сп}(\textit{cdr}(x),\ s + \textit{car}(x),\ p \times \textit{car}(x))$$

Снова исключительно простая и интересная функция. Здесь, избежав двух переборов списка *x,* необходимых при вычислении функции

двучлен(сумма(x), произв(x))

мы избавились также от массы ненужных расчленений и соединений двучленов, которые были бы необходимы, если бы мы попытались программировать без накапливающих параметров. То есть без этих параметров определение выглядело бы примерно так:

$$\textit{сумпроизв}(x) \equiv \textbf{ если } \textit{равно}(x,\ \textit{НИЛ}) \textbf{ то } \textit{двучлен}(0,1) \textbf{ иначе } \textit{накопить}(\textit{car}\ (x),\ \textit{сумпроизв}(\textit{cdr}(x)))$$
$$\textit{накопить}(n, z) \equiv \textit{двучлен}(n + \textit{car}(z),\ n \times \textit{car}(\textit{cdr}(z)))$$

Версия функции с накапливающими параметрами строит только один двучлен, конечный результат, в то время как последняя версия строит *n*+1 двучленов, где *n* — длина исходного списка *x.*

2.7. Локальные определения

Часто бывает полезно, вычисляя значения выражений, давать им имена с тем, чтобы эти значения можно было использовать повторно. Такой пример был у нас в конце предыдущего раздела. Когда мы программировали функцию *сумпроизв(x) без* накапливающих параметров, мы сочли необходимым определить вспомогательную функцию *накопить(n, z),* чтобы избежать неэффективности такого определения:

$$\textit{сумпроизв}(x) \equiv \textbf{ если } \textit{равно}(x,\ \textit{НИЛ}) \textbf{ то } \textit{двучлен}(0,1) \textbf{ иначе } \textit{двучлен}(\textit{car}(\textit{сумпроизв}(\textit{cdr}(x))) + \textit{car}(x),\ \textit{car}(\textit{cdr}(\textit{сумпроизв}(\textit{cdr}(x)))) \times \textit{car}(x))$$

При таком определении выражение *сумппоизв(cdr(x))* вычисляется дважды. Чтобы распознать это общее подвыражение, компилятор должен быть весьма изощренным. Это дублирование рекурсивных вызовов функции *сумпроизв* дает экспоненциальный эффект, увеличивая число рекурсивных вызовов с *n* (для списка *x* длины *n)* до 2^n для того же самого списка. Чтобы для решения подобных проблем эффективности дать более удобное средство, нежели введение вспомогательных функций, мы допускаем локальные определения при помощи форм **пусть** и **где.** Переписываем приведенное выше определение в форме (2.20) и (2.21).

$$
\begin{aligned}
&\textit{сумпроизв}(x) \equiv \textbf{ если } \textit{равно}(x,\ \textit{НИЛ}) \textbf{ то } \textit{двучлен}(0,1) \textbf{ иначе}\\
&\quad \{\textbf{пусть } z = \textit{сумпроизв}(\textit{cdr}(x))\\
&\qquad \textit{двучлен}(\textit{car}(z) + \textit{car}(x),\ \textit{car}(\textit{cdr}(z)) \times \textit{car}(x))\}
\end{aligned}
\tag{2.20}
$$

$$
\begin{aligned}
&\textit{сумпроизв}(x) \equiv \textbf{ если } \textit{равно}(x,\ \textit{НИЛ}) \textbf{ то } \textit{двучлен}(0,1) \textbf{ иначе}\\
&\quad \{\textit{двучлен}(\textit{car}(z) + \textit{car}(x),\ \textit{car}(\textit{cdr}(z)) \times \textit{car}(x))\\
&\qquad \textbf{где } z = \textit{сумпроизв}(\textit{cdr}(x))\}
\end{aligned}
\tag{2.21}
$$

Выбор между этими вариантами — дело вкуса: оба они имеют тот же смысл. Используя переменную *z,* вводим локальное определение. Областью действия этого определения является выражение, заключенное в скобки. Форма **пусть** позволяет записать определение прежде, чем оно используется, форма где вводит определение после его использования. Все, что говорится относительно синтаксиса одной формы, применимо и к другой. Здесь *z* определено как значение *сумпроизв(cdr(x)),* в предположении, что оно вычисляется только однажды, и затем две части этого результата, частичная сумма *car(z)* и частичное произведение *car(cdr(z)),* используются в области действия *z.* Использование форм **пусть** и где дает средства, аналогичные блочной структуре в Алголе. Мы дадим более строгое определение правил относительно области действия, но отложим это до рассмотрения других примеров.

Можно сделать определение функции *сумпроизв* (вариант с одной функцией) больше похожим на версию с двумя функциями из предыдущего параграфа за счет введения локального определения для числа *n,* выбираемого из списка *x,* как это сделано в (2.22).

```
сумпроизв(x) ≡ если равно(x, НИЛ) то двучлен(0,1) иначе
                {пусть n = car(x)
                и z = сумпроизв(cdr(x))
                двучлен(n + car(z), n × car(cdr(z)))}
```
(2.22)

Определения внутри блока отделены разделителем и. Иногда локальные определения используются, просто чтобы сделать функцию более читабельной, наглядной. Можно сделать это для функции *сумпроизв,* вводя новые локальные переменные *s* и *p* для частичной суммы и частичного произведения, содержащихся в *z*.

```
сумпроизв(x) ≡ если равно(x, НИЛ) то двучлен(0,1) иначе
                {пусть n = car(x)
                и z = сумпроизв(cdr(x))
                {пусть s = car(z)
                и p = car(cdr(z))
                двучлен(n + s, n × p)}}
```
(2.23)

Здесь необходимо взять в скобки определения s и *p,* так как они ссылаются на значение *z*. Объяснение состоит в том, что, когда в блоке пусть имеется ряд определений, определяемые переменные доступны только в определяемом выражении, но не внутри самих определений.

```
{пусть    x1 = e1
   и  ... x2 = e2
   и      xk = ek
e}
```
(2.24)

В (2.24) используются обозначения $x_1, x_2, \ldots, x_k$ для локально определяемых переменных, $e_1, e_2, \ldots, e_k$—для выражений, которые их определяют, и *e* — для определяемого выражения. Переменные $x_1, \ldots, x_k$ могут использоваться только в *в,* где они принимают значения $e_1, \ldots, e_k$ соответственно.

Эти средства позволяют построить очень простую функцию для символьного дифференцирования. Будем оперировать только с формулами, в которые входят переменные, константы и операции + и X. Таким образом, требуется реализовать правила дифференцирования (*Dx* означает производную по *x):*

$$Dx(x) = 1$$
$$Dx(y) = 0 \qquad y \neq x \ (y — константа\ или\ переменная)$$
$$Dx(e_1 + e_2) = Dx(e_1) + Dx(e_2)$$
$$Dx(e_1 \times e_2) = e_1 \times Dx(e_2) + e_2 \times Dx(e_1)$$

и можно построить нашу программу так, чтобы эти правила вошли в нее почти непосредственно. Сначала нужно решить вопрос с представлением данных, но здесь все ясно после обсуждений в разд. 2.1. Итак, будем использовать представления

$$константа \longrightarrow число$$
$$переменная \longrightarrow символ$$
$$e_1 + e_2 \longrightarrow (ПЛЮС\ e_1\ e_2)$$
$$e_1 \times e_2 \longrightarrow (УМН\ e_1\ e_2)$$

Сначала запишем две вспомогательные функции, которые используются для формирования сумм и произведений соответственно.

$$сумма(u, v) \equiv cons(ПЛЮС, cons(u, cons(v, НИЛ)))$$
$$произв(u, v) \equiv cons(УМН, cons(u, cons(v, НИЛ)))$$

Теперь можем записать функцию *дифф(e),* которая вычисляет производную e по X.

```
дифф(e) ≡ если атом(e) то если e = X то 1 иначе 0
  иначе если равно(car(e), ПЛЮС) то
         {сумма(дифф(e1), дифф(e2))
       где e1 = car(cdr(e))
         и e2 = car(cdr(cdr(e)))}
  иначе если равно(car(e), УМН) то
         {сумма (произв(e1, дифф(e2)),
              произв(дифф(e1), e2))
       где e1 = car(cdr(e))
         и e2 = car (cdr(cdr(e)))}
  иначе ОШИБКА
```
(2.25)

Мы избежали детализации и очевидного анализа случаев при проектировании этой функции. Уместно, однако, сделать некоторые замечания. Функция *дифф(e)* организована так, что выдает атом *ОШИБКА* при поступлении неправильного аргумента. На самом деле этот атом может быть включен в результат, если неправильный аргумент затрагивает только вложенные подкомпоненты. Функция построена таким образом, что при использовании локальных определений правила дифференцирования представлены очень кратко.

2.8. Функции высших порядков и λ-выражения

Рассмотрим функцию

увелич(*x*) ≡ **если** *равно*(*x*, *НИЛ*) **то** *НИЛ* **иначе**
cons(*car*(*x*) + 1, *увелич*(*cdr*(*x*)))

Если ее применить к списку целых чисел, то она выдает список такой же длины, но с элементами, на единицу увеличенными по сравнению с исходным списком. Действие функции иллюстрирует табл. 2.18.

Таблица 2.18

x	*увелич* (*x*)
(1 2 3)	(2 3 4)
(0 −1 −2)	(1 0 −1)
(127)	(128)

Аналогично функция

ост(*x*) ≡ **если** *равно*(*x*, *НИЛ*) **то** *НИЛ* **иначе**
cons(*car*(*x*) **ост** 2, *ост*(*cdr*(*x*)))

вычисляет по исходному списку *x* список остатков от деления каждого его элемента на 2 (табл. 2.19).

Таблица 2.19

x	*ост* (*x*)
(0 2 3)	(1 0 1)
(0 −1 −2)	(0 −1 0)
(127)	(1)

Сходство этих двух функций и наша способность представить себе многие другие подобные функции приводят к рассмотрению некоторой функции общего назначения, где та конкретная операция, которая выполняется над каждым элементом списка, является параметром этой функции. Здесь эта функция общего назначения называется *отобразить* (или короче *отобр)* и определяется соотношением

отобр(*x*, *f*) ≡ **если** *равно*(*x*, *НИЛ*) **то** *НИЛ* **иначе**
cons(*f*(*car*(*x*)), *отобр*(*cdr*(*x*), *f*))

Это в точности та же самая форма, что и в определении функции *увелич* и *ост,* но функция, применяемая к каждому элементу списка *x,* не конкретна, она является параметром *f.* Мы можем переопределить функции *увелич* и *ост* через *отобр* следующим образом:

*ув*1(*z*) ≡ *z* + 1
увелич(*x*) ≡ *отобр*(*x*, *ув* 1)
*ост*2(*z*) ≡ *z* **ост** 2
ост(*x*) ≡ *отобр*(*x*, *ост* 2)

Функции, которые используют другие функции в качестве аргументов, являются примерами того, что мы будем называть функциями высших порядков. В функциональном программировании разумное использование функций высших порядков может привести к чрезвычайно простым и мощным программам.

В качестве другого примера функции высшего порядка рассмотрим операцию редукции (идея и название фактически взяты из языка АПЛ). Мы определяем функцию *редукция(x, g, a),* где *x* — список, *g* — бинарная функция, *a* — константа, так что список x— $(x_1 \ldots x_k)$ приводится (редуцируется) к значению

$g(x_1, g(x_2 \ldots g(x_k, a)\ldots))$

Таким образом, если, в частности, мы определим

плюс (*y*, *z*) ≡ *y* + *z*

то будем иметь

редукция (*x*, *плюс*, 0) = *сумма* (*x*)

Функция *редукция* строится непосредственным образом:

случай (1) $x = НИЛ$ *редукция*(*x*, *g*, *a*) = *a*
случай (2) $x \neq НИЛ$
пусть *редукция*(*cdr*(*x*), *g*, *a*) = *z*
тогда *редукция*(*x*, *g*, *a*) = *g*(*car*(*x*), *z*)

Это приводит к определению

редукция(*x*, *g*, *a*) == **если** *равно*(*x*, *НИЛ*) **то** *a* **иначе**
g (*car* (*x*), *редукция*(*car*(*x*), *g*, *a*))

Имея эту функцию высшего порядка, можно тривиально определить функции *сумма* и *произв:*

плюс (*y*, *z*) ≡ *y* + *z*
сумма(*x*) ≡ *редукция*(*x*, *плюс*, 0)
умн(*y*, *z*) ≡ *y* × *z*
произв(*x*) ≡ *редукция*(*x*, *умн*, 1)

Довольно любопытно, что так же обстоит дело и с функцией *сумпроизв(x)*. Мы должны представлять себе, что в этом случае результатом функции *редукция* является двучлен и поэтому аргументами функции *g* будут целое число и двучлен, но у нас уже была такая функция в конце разд. 2.6.

$$\textit{накопить}(n,\ z) \equiv \textit{двучлен}(n + \textit{car}(z),\ n \times \textit{car}(\textit{cdr}(z)))$$
$$\textit{сумпроизв}(x) \equiv \textit{редукция}(x,\ \textit{накопить},\ \textit{двучлен}(0,1))$$

Некоторым недостатком при использовании функций высших порядков является необходимость давать имена функциям, используемым как фактические параметры. Но этот недостаток преодолевается использованием λ-выражений, к описанию которых мы теперь переходим. До сих пор мы вводили функции при помощи определений вида

$$f(x_1, \ldots, x_k) \equiv e$$

где f — определяемая функция, $x_1, \ldots, x_k$ ее параметры и e— определяющее функцию выражение. Имя f рассматривается затем как глобальное (его можно использовать повсюду), в то время как имена $x_1, \ldots, x_k$ являются локальными (они могут использоваться только в выражении e). Если бы мы хотели определить функцию, которая не была бы глобальной, у нас не нашлось бы средств сделать это.

λ-выражение является таким выражением, значение которого есть функция. Чтобы уяснить это, мы должны понять, что определение функции *f*, приведенное выше, присваивает *f* (позднее будем говорить «связывает с *f*») значение, которое является функцией. Значение, присвоенное приведенным выше определением, является значением, представленным λ-выражением

$$\lambda(x_1, \ldots, x_k)e$$

Функция является правилом для вычисления значения по некоторым аргументам. λ-выражение заключает в себе это правило, выделяя имена параметров (здесь это $x_1, \ldots, x_k$) и выражение заданное в терминах этих параметров. λ-выражение

$$\lambda(z)z+1$$

вычисляет функцию, которая, будучи вызвана при конкретном значении аргумента, выдает это значение, увеличенное на 1. То есть это функция, присвоенная *ув1* в определении, данном в начале этого раздела. Отсюда находим немедленное применение для λ-выражений — используем их в качестве фактических параметров функций высших порядков. Мы можем переопределить функцию *увелич(x):*

$$\textit{увелич}(x) \equiv \textit{отобр}(x,\ \lambda(z)z+1)$$

Аналогично можем написать

$$\textit{ост}(x) \equiv \textit{отобр}(x,\ \lambda(z)\ z\ \textbf{ост}\ 2)$$
$$\textit{сумма}(x) \equiv \textit{редукция}(x, \lambda(y,\ z)\ y+z,\ 0)$$

λ-выражение обозначает то же самое функциональное значение, независимо от имен, выбранных для его параметров, лишь бы они были разными для различных параметров. Так, выражения

$$\lambda(x)\ x+1$$
$$\lambda(z)\ z+1$$

обозначают одну и ту же функцию и являются взаимозаменяемыми, где бы они ни встречались. В частности, можно использовать *λ(x)x*+1 как фактический параметр функции *отобр* в определении *увелич,* хотя это и создает незначительную, но все же нежелательную путаницу из-за двух «иксов».

$$\textit{увелич}(x) \equiv \textit{отобр}(x,\ \lambda(x)\ x+1)$$

Будем разрешать использовать λ-выражения повсюду, где требуется функциональное значение. В частности, можно использовать их в комбинации с формами **пусть** и **где** в локальных определениях функций. Рассмотрим новый вариант определения функции *увелич:*

$$\textit{увелич}(x) \equiv \{\textit{отобр}(x,\ \textit{ув}1)\ \textbf{где}\ \textit{ув}1 \equiv \lambda(z)z+1\}$$

Здесь мы не избежали введения имени *ув*1 для функционального аргумента функции *отобр,* но сузили область действия этого имени, сделав его локальным. Действительно, имя *ув*1 доступно только в самом выражении *отобр(x, ув*1*)*.

Когда локальное определение вводит рекурсивную функцию, следует использовать модифицированные формы **пусть** и **где.** Будем указывать эту модификацию ключевыми словами **пустьрек** и гдерек.

Тогда можно переписать определение функции *сумпроизв(x),* данное в конце разд. 2.6, чтобы сделать функцию *сп(x, s, p),* имеющую ограниченное применение, локальной:

$$\begin{aligned}\textit{сумпроизв}(x) \equiv \{&\textit{сп}(x,\ 0,\ 1)\\ &\textbf{гдерек}\ \textit{сп} \equiv \lambda(x,\ s,\ p)\\ &\textbf{если}\ \textit{равно}(x,\ \textit{НИЛ})\ \textbf{то}\ \textit{двучлен}(s,\ p)\\ &\textbf{иначе}\ \textit{сп}(\textit{cdr}(x), s+\textit{car}(x),\ p \times \textit{car}(x))\}\end{aligned}$$

Разница между формами **где** и **гдерек** (соответственно **пусть** и **пустьрек)** заключается в разнице областей действия имен. Если имеем формы где и пусть, как это показано в (2.26), имена $x_1, \ldots, x_k$ доступны для использования со значениями

```
{e                          {пусть x1 = e1
где x1 = e1                     и x2 = e2
  и x2 = e2   или               ......
  ......                        и xk = ek
  и xk = ek}                e}
```
(2.26)

$e_1, \ldots, e_k$ только в определяемом выражении e. Однако в случае форм **гдерек** и пустырек (2.27) имена $x_1 \ldots x_k$

```
{e                          {пустырек x1 = e1
гдерек x1 = e1                  и x2 = e2
     и x2 = e2   или            ......
     ......                     и xk = ek
     и xk = ek}             e}
```
(2.27)

используются для значений e_1, e_2, . . ., e_k и в выражении e, и в выражениях e_1, . . , e_k. Если каждое из выражений e_1, . . , e_k является λ-выражением и, таким образом, каждое из x_1, . . ., x_k есть функция, то любое из выражений e_1, . . , e_k может вызвать любую из функций x_1, . . ., x_k. Именно таким способом постоянно будем использовать формы гдерек и **пустырек.**

Сокращение **рек** указывает на рекурсию и призвано подчеркнуть, что локальные определения взаимно рекурсивны. Мы увидим, что это ведет к усложнению интерпретации функциональных программ. Это может также ввести в заблуждение пользователей. Когда следует использовать рек и когда нет? Существует хорошее правило использовать рек, только когда это необходимо, т. е. когда локально определяются рекурсивные функции и когда есть уверенность, что все параллельные определения тоже являются определениями функций. То есть для надежности все локальные определения в блоках пустырек и гдерек должны быть определениями функций. Взглянем на странное определение

$$\{\{x \text{ гдерек } x=x+1\} \text{ где } x=0\}$$

Так как используется форма гдерек, оба x в выражении $x=x+1$ должны быть одинаковыми. Ясно, что это выражение бессмысленно. С другой стороны, выражение

$$\{\{x \text{ где } \boldsymbol{x=x+1)} \text{ где } x=0)$$

совершенно правильное и имеет значение 1. Здесь различные вхождения x в выражении $x=x+1$ имеют разное значение, x в правой части определяется во внешнем блоке.

Как мы видели, функциональная программа обычно состоит из множества взаимно рекурсивных функций, одна из которых отмечается как главная определяемая. Можно использовать формы гдерек и пустырек, чтобы сформировать выражение, значением которого является главная определяемая функция и где вспомогательные функции спрятаны внутри выражения. Если множество функций, которые определяются, есть $f_1, f_2, \ldots, f_k$ и f_1, является главной, то определение записывается в виде (2.28).

```
{f1
гдерек f1 = ...
     и f2 = ...
     ......
     и fk = ...}
```
(2.28)

Последняя версия функции *набор (x),* составленная в конце разд. 2.5, может быть представлена в форме (2.29).

```
{набор гдерек
    набор = λ (t) если равно(t, НИЛ) то НИЛ
                  иначе если атом(car(t)) то включить (набор(cdr(t)), car(t))
                  иначе объединить (набор(car(t)), набор(cdr(t)))
  и объединить = λ (u, v) если равно(u, НИЛ) то v
                          иначе включить(объединить(cdr(u), v), car(u))
  и включить = λ(s, x) если элемент(x, s) то s иначе cons(x, s)
  и элемент = λ(x, s) если равно(x, НИЛ) то Л
                      иначе если равно(x, car(s)) то И иначе
                      элемент(x, cdr(s))}
```
(2.29)

Наконец, в этом разделе мы хотим ввести другого рода функцию высшего порядка — функцию, которая выдает функцию, т. е. функцию, результатом которой является функция. Рассмотрим определение

$$увел\,(n) \equiv \lambda\,(z)\ z+n$$

Здесь *увел* является, безусловно, функцией, она требует в качестве аргумента целое число *n.* Однако ее результат тоже функция. Например, *увел(3)* есть функция, которая при вызове выдает ее аргумент, увеличенный на 3. Таким образом, значение выражения

$$\{f(2) \text{ где } f=увел\ (3)\}$$

есть 5. Аналогично мы можем переопределить функцию *увелич:*

$$увелич\,(x) \equiv отобр\,(x,\ увел\,(1))$$

Заметим, что если бы мы хотели иметь локальное определение функции *увел* в определении функции *увелич* (это не очень желательно и делается с целью иллюстрации), то мы бы записали

$$\textit{увелич}(x) \equiv \{\textit{отобр}(x,\ \textit{увел}(1))$$
$$\text{где } \textit{увел} = \lambda(n)\{\lambda(z)z + n\}\}$$

Здесь ясно показано, что тело функции, присваиваемое *увел,* само является λ-выражением. Мы использовали лишние скобки, чтобы яснее показать, где начинается и заканчивается каждое выражение.

В связи с используемыми здесь обозначениями возникла любопытная проблема. Мы использовали λ-выражение *λ(z) z+n,* не обращая внимания на тот факт, что функция определяется через переменную, которая не является ее параметром, в данном случае это *n.* Такие переменные называются глобальными (или свободными) переменными данной функции. Здесь эта переменная не очень глобальна, если посмотреть на контекст, в котором используется *λ(z) z+n,* но тем не менее проблема существует. Эту проблему наглядно иллюстрирует выражение

$$\{\textbf{пусть } n = 1$$
$$\{\textbf{пусть } f = \lambda(z)z + n$$
$$\{\textbf{пусть } n = 2$$
$$f(3)\}\}\}$$

Значение *f(3),* а отсюда и значение всего выражения есть либо 4, либо 5 в зависимости от того, какое *n* берется в момент вызова *f.* Мы представляем себе λ-выражение как выражение, которое вычисляет функциональное значение, и здесь это значение присваивается *f.* Будем предполагать, что функциональное значение, вычисленное таким способом, фиксирует все глобальные переменные. То есть функция такова, какой она получается при замене всех глобальных переменных их значениями в момент вычисления λ-выражения. Следовательно, при вычислении приведенного выше выражения мы сначала выполняем локальное определение *n*+1, затем вычисляем λ-выражение, присваивая *f* функциональное значение *λ(z)z*+1. После этого переопределяется *n, n=2,* и затем вызывается *f* с фактическим параметром 3. Так как *f* присвоено значение *λ(z)z*+1, этот вызов выдает значение 4. Такая форма вычисления λ-выражений, когда глобальные переменные определяются в месте определения, но не в месте вызова функции, называется *простым,* или *статическим,* связыванием (переменных). Эта семантическая особенность функционального программирования оказывает значительное воздействие на общий проект его реализации, и поэтому мы будем неоднократно возвращаться к этому на протяжении всей книги.

Мы заключим этот раздел определением функции высшего порядка, которая имеет функции и в качестве аргументов, и в качестве результата. Определим

$$\textit{комп}(f, g) \equiv \lambda(x)f(g(x))$$

Это так называемая *композиция* (или произведение) функций. Она берет в качестве аргументов две функции и выдает как результат тоже функцию, которая применяет последовательно обе исходные функции. Таким образом, она применяет *f* к результату *g.* Например,

увобр = комп (увелич, обратить)

является функцией, которая сначала обращает список и затем увеличивает каждый его элемент на 1.

Таблица 2.20

x	*увобр (x)*
(1 2 3)	(4 3 2)
(127 —127)	(—126 128)

Аналогично, если определить, подобно *увел(n),* функцию

ост(n)=λ(z)z ост *n*

то

комп(увел(1),*ост(2))*

является функцией, которая дает тот же эффект, что и

λ (z) (z ост 2)+1

так что функция

отобр(x,комп(увел(I),ост(2)))

берет каждый элемент списка *x* и добавляет 1 к остатку от его деления на 2 (табл. 2.21).

Таблица 2.21

x	*отобр(x,комп(увел(1), ост(2)))*
(1 2 3)	(2 1 2)
(127 128 129 130)	(2 1 2 1)

Функции высших порядков являются очень важной темой, они весьма трудны при реализации строго функционального языка на машине. Вот почему мы их здесь обсуждали и еще неоднократно будем обращаться к ним на протяжении книги. В частности, гл. 9 посвящена приложениям функций высших порядков.

2.9. Точечная запись выражений

Мы подошли теперь к полезному расширению понятия S-выражения, которое в гл. 4 часто будет использоваться при записи S-выражений. Здесь этот раздел включен для полноты. Его изучение можно отложить до тех пор, пока в последующих главах не потребуется точечная запись. Необходимость в этом расширении возникает из-за того, что не существует способа представить значение *cons(x, y),* когда *y* есть атом (исключая случай *y* есть *НИЛ).* Будем представлять значение

cons(A, B)

посредством S-выражения

(A.B)

с точкой между атомами. Отметим, что написанное отличается от значения

(A B)

которое является результатом выражения

cons(A, cons(B, НИЛ))

Теперь мы столкнулись с явлением, не встречавшимся нам до сих пор,— имеются два разных способа представить одно и то же значение. Значение

cons(A, НИЛ)

может быть представлено либо как *(A),* либо как *(A.НИЛ).* Иными словами, имеются различные способы записи одного и того же значения. Аналогичная ситуация встречается в десятичном представлении целых, когда мы считаем, что

127 и 0127

представляют то же самое значение, игнорируя 0 перед значащими цифрами.

Расширение понятия, которое мы делаем, в действительности имеет более общий характер. Например, представим значение

cons(Л, *cons(B, cons(C, D)))*

в виде

(A B CD)

Снова имеем, что

(A B C.НИЛ) и *(A BC)*

суть разные представления одного и того же значения. Вообще значение выражения

consfa, e_2)

представляется точечной парой

(Pl.Vt)

где *Vi* и v_2 суть S-выражения для значений e_e и e_a, независимо от того какого рода эти значения.

Таблица 2.22

x	*y*	*cons (x, y)*
A	*B*	*(A.B)*
A	*НИЛ*	*(A.НИЛ)*
A	*(B.НИЛ)*	*(A.(B.НИЛ))*
A	*(B.C)*	*(A.(B.C))*
(A.B)	*C*	*((A.B).C)*

Мы применяем следующие сокращения: S-выражение

(Mt>$_2$. *(Щ- • • -(fft-fft+i). • •)))*

записывается в виде

(V! 0„ . . . V_h.V_{k+1})

Отмечаем единственную точку. Далее, S-выражение

(о, и, . . . $u_к$.*НИЛ)*

записывается в виде

(fi v_2 ... v_n)

Эти правила можно суммировать так. Точка, за которой непосредственно следует открывающая скобка, может быть опущена, так же как опускаются открывающая и соответствующая закрывающая скобки. Точка, за которой непосредственно следует атом *НИЛ,* также может быть опущена, это относится и к атому *НИЛ.* Из этого следует, что значение выражения

cons(A, cons (B, cons (C, НИЛ)))

представляется структурой

(A.(B.(C.НИЛ)))

которая может быть последовательно сведена к структурам

(АВ.(С.НИЛ))
(А В С.НИЛ)
(А В С)

На самом деле все S-выражения, изображенные на рис. 2.2, представляют одно и то же значение, т. е. значение приведенного выше выражения. Стрелки показывают, какая форма из какой

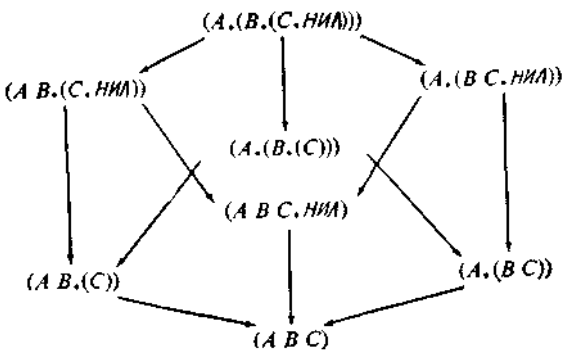


Рис. 2.2.

может быть получена применением одного из правил. Вообще будем всегда использовать кратчайшую форму, т. е. будем исключать все точки, где это возможно.

Будем использовать точечную запись в таких случаях, когда, например, хотим сослаться на первые два элемента списка произвольной длины. Запишем

$(x_1\ x_2.\ y)$

вместо

$(x_1x_2x_3...x_k)$

так что x_1 обозначает первый элемент, x_2— второй, а y обозначает весь остаток списка (здесь это $(x_3\ .\ .\ .\ x_k)$). Иногда удобнее строжить пары, используя функцию *cons,* а не *двучлен.* В значительной степени это зависит от того, должны ли эти пары печататься или нет. Например, если мы хотим составить список, состоящий из пар символьных атомов и целых чисел, то можем выбрать либо список двучленов

((*X* 1) (*У* 17) (*Z* -127))

либо список точечных пар

((X.1) (У. 17) (Z. -127))

Выбор является делом вкуса. Точечные пары требуют меньше памяти в большинстве реализаций, но если символы образуют пары с общими S-выражениями, а не с атомами, то версия с двучленами может оказаться более четкой. Сравним

((X A) (У *(B C))* *(Z НИЛ))*

с выражением

((X.A) (У *B C)* *(Z))*

которое является сокращенной формой списка, помеченного точками. Вообще при реализации для печати выбирается кратчайшая возможная форма S-выражения.

Так как теперь мы можем представить результат операции *cons* в общем случае, можно записывать функции от общих S-выражений. S-выражением является либо 1) атом, либо 2) точечная пара двух S-выражений (результат операции *cons).*

Следовательно, при построении функций от S-выражений (в противоположность просто спискам) мы имеем другой анализ случаев. Построим функцию (2.30), которая вычисляет количество атомов в S-выражении,— назовем ее *размер* (s).

$$\begin{array}{ll} \text{случай (1)} & s\text{ — атом} \qquad \textit{размер}(s) = 1 \\ \text{случай (2)} & s\text{ — не атом} \\ \quad \text{пусть} & \textit{размер}(car(s)) = m \\ \quad \text{и} & \textit{размер}(cdr(s)) = n \\ \quad \text{тогда} & \textit{размер}(s) = m + n \end{array} \qquad (2.30)$$

Итак, здесь следует рассмотреть случаи, когда аргумент является атомом и не атомом. В случае если он не является атомом, мы вызываем функцию, которая рекурсивно определена через *car* и *cdr* аргумента. Мы построили функцию

$$\textit{размер}(s) \equiv \textbf{если}\ \textit{атом}(s)\ \textbf{то}\ 1\ \textbf{иначе}\ \textit{размер}(car(s)) + \textit{размер}(cdr(s))$$

Отметим, что если применять эту функцию к списку или к списку списков и т. д., то элементы *НИЛ,* которые заключают каждый список, будут также учитываться.

Ранее мы отмечали, что предикат *равно (x, y)* может применяться только в том случае, когда по крайней мере один из его аргументов является атомом. То есть он не определен, если оба его аргумента являются S-выражениями. Однако можно построить функцию *тожд(x, y),* которая выдает значение И, если *x* и *y* являются тождественными S-выражениями в том смысле, что если мы их сравним, то в них в точности те же самые атомы будут находиться на тех же самых местах. Применим наш анализ случаев.

```
случай (1)   атом(x) = И     тожд(x, y) = равно(x, y)
случай (2)   атом(x) = Л
  подслучай (2.1) атом(y) = И     тожд(x, y) = равно(x, y)
  подслучай (2.2) атом(y) = Л                                   (2.31)
    пусть тожд(car(x), car(y)) = t1
        и тожд(cdr(x), cdr(y)) = t2
    тогда тожд(x, y) = если t1 то t2 иначе Л
```

В случаях (1) и (2.1) мы знаем, что один из аргументов — атом, и поэтому можем просто применить предикат *равно (x, y)*. В случае (2.2), когда ни *x,* ни *y* не являются атомами, мы рекурсивно используем предикат *тожд* для сравнения *car* и *cdr* наших аргументов. Теперь *x* и *y* равны, если равны как их *car,* так и *cdr.* То есть хотелось бы написать

$$тожд(x, y) = t_1 \wedge t_2$$

Вместо этого мы пишем

тожд(x, y)=**если** t_x **то** t_t **иначе** *Л*

что дает, как это легко увидеть, те же самые значения. Это больше согласуется с окончательным построением функции (и отражает тот факт, что мы несколько забегаем вперед с операцией ^ — см. гл. 5). Запишем построенную функцию (2.32).

```
тожд(x, y) = если атом(x) то равно(x, y) иначе
             если атом(y) то равно(x, y) иначе
             если тожд(car(x), car(y)) то
                  тожд(cdr(x), cdr(y)) иначе Л                  (2.32)
```

Мы могли бы записать эту функцию, используя локальные определения для t_1 и t_2 (см. 2.33), но лучше не делать этого. Эта форма предполагает, что сравниваются как *car,* так и *cdr,*

```
тожд(x, y) = если атом(x) то равно(x, y) иначе
             если атом(y) то равно(x, y) иначе
             {пусть t1 = тожд(car(x), car(y))                   (2.33)
                  и t2 = тожд(cdr(x), cdr(y))
             если t1 то t2 иначе Л}
```

в то время как функция (2.32) в случае, если *car* различны, не производит ненужного и трудоемкого сравнения *cdr.*

Отметим, что в обеих версиях функции второе применение предиката *равно* всегда ложно и может быть заменено значением *Л.*

Упражнения

2.1. Опишите рекурсивную функцию *произв(x),* которая вычисляет произведение списка целых чисел.

2.2. Опишите функцию *уменьш(x),* которая преобразует список целых чисел *x* в список, каждый из элементов которого на единицу меньше соответствующего элемента (табл. 2.23).

Таблица 2.23

x	*уменьш (x)*
НИЛ	*НИЛ*
(1 2)	(0 1)
(4 5 6)	(3 4 5)

Обобщите эту операцию таким образом, чтобы величина, на которую уменьшаются элементы, задавалась как дополнительный параметр функции.

2.3. Опишите функцию *позиция(x, y),* аргументами которой являются атом *x* и список атомов *y.* Ее результатом будет положение атома *x* в списке *y,* считая с первого по порядку элемента. Предполагается, что *x* встречается в *y* (табл. 2.24).

Таблица 2.24

x	*y*	*позиция (x, y)*
A	*(A B C)*	1
B	*(A B C)*	2
F	*(A B C D E F)*	6

2.4. Видоизмените описанную функцию так, чтобы она выдавала значение 0 в случае, если *x* не встречается в списке *y.*

2.5. Опишите функцию *индекс(i, y),* которая воспринимает в качестве аргументов целое число *i* и список *y* и выдает элемент списка, имеющей номер г. Предполагается, что *y* имеет длину не меньше *i.*

Таблица 2.25

x	*набор (x)*
(A (B C)D)	*(A B C D)*
(((A B)C)(D E))	*(A B C D E)*
((((((A))))))	*(A)*

2.6. Опишите функцию *набор (x),* аргументом которой является список *x* с подсписками любой глубины, а результатом — список атомов, обладающий тем свойством, что все атомы, появляющиеся в *x,* появляются и в *набор (x)* в том же самом порядке.

2.7. Опишите функцию *частоты((),* которая берет в качестве аргумента список атомов *t* и выдает список всех атомов, встречающихся в *t,* вместе с частотой их появления (табл. 2.26). Так как результат представляет собой множество пар, несущественно, в каком порядке эти пары расположены.

Таблица 2.26

t	*частоты(t)*
(A B A B A C A)	((A 4) (B 2) (C 1))
(НИ ТО НИ ДРУГОЕ)	((НИ 2) (ТО 1) (ДРУГОЕ 1))

2.8. Опишите функции, задающие обычные операции над множествами, на списках атомов, используемых для представления множеств таким же образом, как в разд. 2.5, т. е. без повторения. *Объединение* множеств уже рассматривалось. *Пересечение* множеств выделяет из множеств *u* и *v* множество, состоящее из всех атомов, одновременно принадлежащих и *u,* и *V. Разность* множеств строит из множеств *u* и *v* множество, состоящее из атомов и, которые не лежат в *v.* Функция *симмразность* по множествам *u* и *v* строит множество, состоящее из всех атомов, которые лежат в *u* или в *V,* но не в обоих одновременно, т. е.

суммразность(u, v)=разность(объединение(u, a), *пересечение(u, v))*

2.9. Опишите функцию *индивидуумы (i),* которая берет в качестве аргумента список *t* и выдает в результате список всех атомов, которые встречаются в *t* ровно один раз. Следует отметить, что, хотя и можно для этой цели использовать функцию *частоты (f)* и затем выбрать только атомы с единичной частотой, но действуя непосредственно, можно получить лучшую программу.

2.10. Перепрограммируйте функцию *индивидуумы (f),* определенную в упр. 2.9, с использованием накапливающих параметров. На этот раз потребуются два таких параметра: для накопления атомов, встречающихся ровно один раз и более одного раза. Сравните число вызовов функции *cons* в программах с применением и без применения накапливающих параметров.

Таблица 2.27

x	*отобрмнож(x, λ (z) z÷10)*
(1 57 84 61 53 6)	(0 5 8 6)
(101 102 103 104 105)	(10)
(78 87 79 —97)	(7 8 —9)

2.11. Опишите функцию *отобрмнож(x, /)*, аргументом которой является список *x,* рассматриваемый как множество, а результатом применения — множество, представленное опять-таки в виде списка, который можно получить применением *f* к каждому элементу *x* (табл. 2.27). Следует отметить, что порядок элементов в списке, рассматриваемом как множество, не имеет значения.

2.12. Опишите функцию *редущия2 (x,* g, a), которая, будучи применена к списку $x=(x_1 \ldots x_n)$,.выдает значение $g(\ldots g(g(a, x_1), x_2) \ldots, x_n)$.

Рассмотрите функцию *редущия2(x,лямбда(y,z)10Xy+z,*0) в случае, когда *x* — список одноразрядных положительных целых чисел. Какое значение вычисляется? Что получится, если использовать функцию *редукция* вместо *редукция??*

2.13. Сократите следующие S-выражения:

((A.(B.НИЛ)).((C.(O.НИЛ)).НИЛ))
(((A.НИЛ).НИЛ).B)
(A B C.(D E P.НИЛ))
((A.B).(C.D))

2.14. Перепроектируйте функцию *набор(x)* из упр. 2.6 так, чтобы она составляла наборы элементов общего S-выражения (табл. 2.28). Следует отметить, что при взятии набора от списка в нем появляются элементы *НИЛ.*

Таблица 2.28

x	*набор(x)*
(A.(B.C))	(A B C)
(A B)	(A B НИЛ)
((A.B)(C.D))	(A B C D НИЛ)

Если *x* — общее S-выражение, *набор (x)* представляет собой список всех атомов *x* в порядке, в котором они расположены в списке *x.*

2.15. Опишите функцию *кронтожд(x.y),* которая выдает *И,* если одинаковые атомы расположены в списках *x* и *y* в одном и том же порядке независимо от внутренней структуры *x* и *y,* и выдает *Л* в противном случае; *x* и *y* — произвольные S-выражения. Вполне корректное, но все же неудовлетворительное определение можно дать таким образом:

кронтожд(x, y)=тожд(набор(x), набор(y))

где функция *набор (x)* описана в предыдущем упражнении. Общее S-выражение можно изобразить в виде двоичного дерева. Двоичное дерево, эквивалентное (a, *b),* выглядит так:

Если a и *b* — атомы, они являются листьями дерева, если же это точечные пары — они являются поддеревьями. На рис. 2.3 двоичные деревья изображены рядом с соответствующими S-выражениями. Все эти деревья имеют одинаковые кроны. Если мы переименуем или переупорядочим листья, кроны

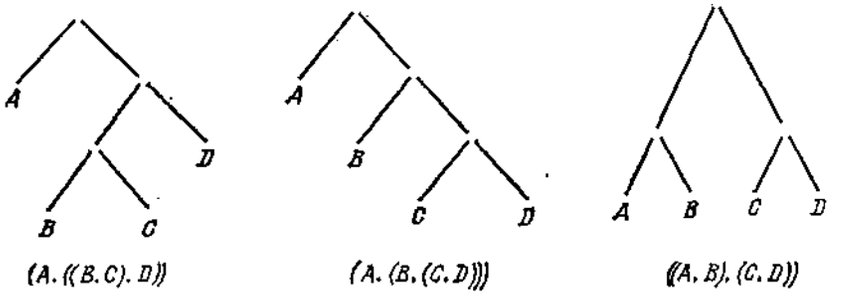


Рис. 2.3.

будут различны. Вычисление функции *набор* для больших деревьев является очень трудоемкой операцией, нежелательной при сравнении деревьев с различными кронами. Однако избежать этого довольно трудно, в чем и заключается основная сложность данного упражнения.