Харольд Абельсон

Джеральд Джей Сассман

при участии Джули Сассман

Структура и интерпретация компьютерных программ

Добросвет, 2006

Harold Abelson

and Gerald Jay Sussman

with Julie Sussman

Structure and Interpretation of Computer Programs

Эта книга посвящается, с уважением и любовью, духу, который живет внутри компьютера.

“Мне кажется, чрезвычайно важно, чтобы мы, занимаясь информатикой, получали радость от общения с компьютером. С самого начала это было громадным удовольствием. Конечно, время от времени встревали заказчики, и через какое-то время мы стали серьезно относиться к их жалобам. Нам стало казаться, что мы вправду отвечаем за то, чтобы эти машины использовались успешно и безошибочно. Я не думаю, что это так. Я считаю, что мы отвечаем за то, чтобы их тренировать, указывать им новые направления и поддерживать уют в доме. Я надеюсь, что информатика никогда не перестанет быть радостью. Я надеюсь, что мы не превратимся в миссионеров. Не надо чувствовать себя продавцом Библий. Таких в мире и так достаточно. То, что Вы знаете о программировании, могут выучить и другие. Не думайте, что в ваших руках ключ к успешной работе с компьютерами. Что у Вас, как я думаю и надеюсь, есть — это разум: способность увидеть в машине больше, чем Вы видели, когда Вас впервые к ней подвели, увидеть, что Вы способны сделать ее бóльшим.”

Алан Дж. Перлис (1 апреля 1922 – 7 февраля 1990)

Оглавление

Предисловие . . . 9

Предисловие ко второму изданию . . . 13

Предисловие к первому изданию . . . 15

Благодарности . . . 18

1. Построение абстракций с помощью процедур . . . 22
- 1.1. Элементы программирования . . . 25
  - 1.1.1. Выражения . . . 26
  - 1.1.2. Имена и окружение . . . 27
  - 1.1.3. Вычисление комбинаций . . . 29
  - 1.1.4. Составные процедуры . . . 31
  - 1.1.5. Подстановочная модель применения процедуры . . . 33
  - 1.1.6. Условные выражения и предикаты . . . 35
  - 1.1.7. Пример: вычисление квадратного корня методом Ньютона . . . 40
  - 1.1.8. Процедуры как абстракции типа «черный ящик» . . . 43
- 1.2. Процедуры и порождаемые ими процессы . . . 47
  - 1.2.1. Линейные рекурсия и итерация . . . 48
  - 1.2.2. Древовидная рекурсия . . . 53
  - 1.2.3. Порядки роста . . . 58
  - 1.2.4. Возведение в степень . . . 59
  - 1.2.5. Нахождение наибольшего общего делителя . . . 62
  - 1.2.6. Пример: проверка на простоту . . . 64
- 1.3. Формулирование абстракций с помощью процедур высших порядков . . . 69
  - 1.3.1. Процедуры в качестве аргументов . . . 70
  - 1.3.2. Построение процедур с помощью `lambda` . . . 74
  - 1.3.3. Процедуры как обобщенные методы . . . 78
  - 1.3.4. Процедуры как возвращаемые значения . . . 83

2. Построение абстракций с помощью данных . . . 90
- 2.1. Введение в абстракцию данных . . . 93
  - 2.1.1. Пример: арифметические операции над рациональными числами . . . 93
  - 2.1.2. Барьеры абстракции . . . 97

2.1.3. Что значит слово «данные»? . . . 100
2.1.4. Расширенный пример: интервальная арифметика . . . 102
2.2. Иерархические данные и свойство замыкания . . . 106
2.2.1. Представление последовательностей . . . 107
2.2.2. Иерархические структуры . . . 115
2.2.3. Последовательности как стандартные интерфейсы . . . 120
2.2.4. Пример: язык описания изображений . . . 132
2.3. Символьные данные . . . 146
2.3.1. Кавычки . . . 146
2.3.2. Пример: символьное дифференцирование . . . 149
2.3.3. Пример: представление множеств . . . 154
2.3.4. Пример: деревья кодирования по Хаффману . . . 163
2.4. Множественные представления для абстрактных данных . . . 170
2.4.1. Представления комплексных чисел . . . 172
2.4.2. Помеченные данные . . . 175
2.4.3. Программирование, управляемое данными, и аддитивность . . . 179
2.5. Системы с обобщенными операциями . . . 186
2.5.1. Обобщенные арифметические операции . . . 187
2.5.2. Сочетание данных различных типов . . . 191
2.5.3. Пример: символьная алгебра . . . 198

3. Модульность, объекты и состояние . . . 211
3.1. Присваивание и внутреннее состояние объектов . . . 212
3.1.1. Внутренние переменные состояния . . . 213
3.1.2. Преимущества присваивания . . . 218
3.1.3. Издержки, связанные с введением присваивания . . . 221
3.2. Модель вычислений с окружениями . . . 227
3.2.1. Правила вычисления . . . 228
3.2.2. Применение простых процедур . . . 231
3.2.3. Кадры как хранилище внутреннего состояния . . . 234
3.2.4. Внутренние определения . . . 238
3.3. Моделирование при помощи изменяемых данных . . . 241
3.3.1. Изменяемая списковая структура . . . 242
3.3.2. Представление очередей . . . 250
3.3.3. Представление таблиц . . . 254
3.3.4. Имитация цифровых схем . . . 260
3.3.5. Распространение ограничений . . . 271
3.4. Параллелизм: время имеет значение . . . 281
3.4.1. Природа времени в параллельных системах . . . 284
3.4.2. Механизмы управления параллелизмом . . . 287
3.5. Потоки . . . 298
3.5.1. Потоки как задержанные списки . . . 299
3.5.2. Бесконечные потоки . . . 307
3.5.3. Использование парадигмы потоков . . . 314
3.5.4. Потоки и задержанное вычисление . . . 324

3.5.5. Модульность функциональных программ и модульность объектов . . . 330

4. Метаязыковая абстракция . . . 335
4.1. Метациклический интерпретатор . . . 338
4.1.1. Ядро интерпретатора . . . 339
4.1.2. Представление выражений . . . 343
4.1.3. Структуры данных интерпретатора . . . 350
4.1.4. Выполнение интерпретатора как программы . . . 354
4.1.5. Данные как программы . . . 357
4.1.6. Внутренние определения . . . 360
4.1.7. Отделение синтаксического анализа от выполнения . . . 365
4.2. Scheme с вариациями: ленивый интерпретатор . . . 369
4.2.1. Нормальный порядок вычислений и аппликативный порядок вычислений . . . 370
4.2.2. Интерпретатор с ленивым вычислением . . . 372
4.2.3. Потоки как ленивые списки . . . 379
4.3. Scheme с вариациями — недетерминистское вычисление . . . 381
4.3.1. `Amb` и `search` . . . 383
4.3.2. Примеры недетерминистских программ . . . 386
4.3.3. Реализация `amb`-интерпретатора . . . 394
4.4. Логическое программирование . . . 404
4.4.1. Дедуктивный поиск информации . . . 407
4.4.2. Как действует система обработки запросов . . . 417
4.4.3. Является ли логическое программирование математической логикой? 425
4.4.4. Реализация запросной системы . . . 429

5. Вычисления на регистровых машинах . . . 450
5.1. Проектирование регистровых машин . . . 451
5.1.1. Язык для описания регистровых машин . . . 454
5.1.2. Абстракция в проектировании машин . . . 457
5.1.3. Подпрограммы . . . 459
5.1.4. Реализация рекурсии с помощью стека . . . 463
5.1.5. Обзор системы команд . . . 471
5.2. Программа моделирования регистровых машин . . . 472
5.2.1. Модель машины . . . 473
5.2.2. Ассемблер . . . 476
5.2.3. Порождение исполнительных процедур для команд . . . 480
5.2.4. Отслеживание производительности машины . . . 486
5.3. Выделение памяти и сборка мусора . . . 489
5.3.1. Память как векторы . . . 489
5.3.2. Иллюзия бесконечной памяти . . . 494
5.4. Вычислитель с явным управлением . . . 500
5.4.1. Ядро вычислителя с явным управлением . . . 502
5.4.2. Вычисление последовательностей и хвостовая рекурсия . . . 507

5.4.3. Условные выражения, присваивания и определения 510
5.4.4. Запуск вычислителя 512
5.5. Компиляция 517
5.5.1. Структура компилятора 520
5.5.2. Компиляция выражений 524
5.5.3. Компиляция комбинаций 530
5.5.4. Сочетание последовательностей команд 536
5.5.5. Пример скомпилированного кода 539
5.5.6. Лексическая адресация 547
5.5.7. Связь скомпилированного кода с вычислителем 551
Литература 558
Предметный указатель 566

Предисловие

Программированием занимаются учителя, генералы, диетологи, психологи и родители. Программированию подвергаются армии, ученики и некоторые виды обществ. При решении крупных задач приходится применять последовательно множество программ, бóльшая часть которых возникает прямо в процессе решения. Эти программы изобилуют деталями, относящимися к той конкретной задаче, которую они решают. Если же Вы хотите оценить программирование как интеллектуальную деятельность особого рода, то Вам следует обратиться к программированию компьютеров; читайте и пишите компьютерные программы — много программ. Не так уж важно, что будет в них написано и как они будут применяться. Важно то, насколько хорошо они работают и как гладко стыкуются с другими программами при создании еще более крупных программ. Программист должен равно стремиться и к совершенству в деталях, и к соразмерности сложного целого. В книге, которую Вы держите в руках, словом «программирование» мы будем обозначать прежде всего создание, выполнение и изучение программ, написанных на одном из диалектов языка Лисп и предназначенных для выполнения на цифровом компьютере. Использование Лиспа не ограничивает нас в том, чтó мы можем описать в наших программах, — лишь в способе их выражения.

Продвигаясь по материалу этой книги, мы будем встречаться с тремя группами явлений: человеческий разум, совокупности компьютерных программ и компьютер. Всякая компьютерная программа — это порожденная человеческим разумом модель реального либо умозрительного процесса. Эти процессы, возникающие из нашего опыта и мысли, многочисленны, сложны в деталях, и мы всегда понимаем их лишь частично. Редко бывает так, что компьютерные программы отображают их к нашему окончательному удовлетворению. Таким образом, хотя наши программы представляют собой тщательно сработанные дискретные совокупности символов, мозаики переплетенных функций, они непрерывно развиваются: мы изменяем их по мере того, как наше восприятие модели приобретает все большую глубину, расширяется и обобщается, до тех пор, пока модель не достигнет, наконец, метастабильного состояния в рамках следующей модели, над которой нам предстоит биться. Радостное возбуждение, сопутствующее компьютерному программированию, происходит из постоянного раскрытия в голове и в компьютере все новых выраженных в виде программ механизмов и из взрыва восприятия, который они порождают. Искусство выражает наши мечты. Компьютер исполняет их под видом программ!

При всей своей мощности, компьютер требователен и придирчив. Ему нужны верные программы, и то, что мы хотим ему сказать, должно быть выражено точно в каждой мелочи. Как и при всякой другой работе с символами, мы убеждаемся в правильности

программ через доказательство. Самому Лиспу можно сопоставить семантику (между прочим, тоже модель), и если функцию программы можно выразить, скажем, в терминах исчисления предикатов, то логические методы позволят нам вывести формальное доказательство ее корректности. К сожалению, когда программы становятся большими и сложными, что с ними всегда и происходит, адекватность, непротиворечивость и корректность самих спецификаций становится предметом сомнений, так что большие программы редко сопровождаются полными формальными доказательствами корректности. Поскольку большие программы вырастают из малых, нам необходимо обзавестись арсеналом программных структур, в правильности которых мы можем быть уверены — их можно назвать идиомами — и научиться объединять их в структуры большего размера с помощью организационных методов, ценность которых также доказана. Эти методы подробно обсуждаются в книге, и их понимание существенно для участия в прометеевском предприятии под названием «программирование». Для умения создавать большие, значительные программы нет лучшего помощника, чем свободное владение мощными организационными методами. И наоборот: затраты, связанные с написанием больших программ, побуждают нас изобретать новые методы уменьшения веса функций и деталей, входящих в эти программы.

В отличие от программ, компьютеры должны повиноваться законам физики. Если мы хотим, чтобы они работали быстро — по нескольку наносекунд на смену состояния, — электроны в их цепях не должны проходить большие расстояния (более полуметра). При этом тесно сконцентрированные в пространстве приборы излучают тепло, которое нужно куда-то отводить: так развилось изысканное инженерное искусство, призванное находить равновесие между обилием функций и плотностью расположения устройств. Так или иначе, аппаратура всегда работает ниже того уровня, на котором мы бы хотели программировать. Процессы, посредством которых наши программы на Лиспе переводятся в «машинные» программы, сами являются абстрактными моделями, которые мы воплощаем в программах. Их изучение и реализация многое дают для понимания организационных методов, направленных на программирование произвольных моделей. Разумеется, так можно смоделировать и сам компьютер. Подумайте об этом: поведение мельчайшего переключателя моделируется квантовой механикой, которая описывается дифференциальными уравнениями, точное поведение которых фиксируется в численных приближениях, представленных в виде компьютерных программ, которые выполняются на компьютере, составленном из ... — и так без конца!

Раздельное выделение трех групп явлений — не просто вопрос тактического удобства. Хотя эти группы и остаются, как говорится, в голове, но, проводя это разделение, мы позволяем потоку символов между тремя группами двигаться быстрее. В человеческом опыте с этим потоком по богатству, живости и обилию возможностей сравнится разве что сама эволюция жизни. Отношения между разумом человека, программами и компьютером в лучшем случае метастабильны. Компьютерам никогда не хватает мощности и быстродействия. Каждый новый прорыв в технологии производства аппаратуры ведет к появлению более масштабных программных проектов, новых организационных принципов и к обогащению абстрактных моделей. Пусть каждый читатель время от времени спрашивает себя: «А зачем, к чему все это?» — только не слишком часто, чтобы удовольствие от программирования не сменилось горечью философского тупика.

Из тех программ, которые мы пишем, некоторые (но всегда меньше, чем хотелось бы) решают точные математические задачи, такие, как сортировка последовательности чисел или нахождение их максимума, проверка числа на простоту или вычисление квадратного корня. Такие программы называются алгоритмами, и об их оптимальном поведении известно довольно много, особенно в том, что касается двух важных параметров: времени выполнения и потребления памяти. Программист должен владеть хорошими алгоритмами и идиомами. Несмотря на то, что некоторые программы сопротивляются точной спецификации, в обязанности программиста входит оценивать их производительность и все время пытаться ее улучшить.

Лисп — ветеран, он используется уже около четверти века. Среди живых языков программирования старше него только Фортран. Эти два языка обслуживали нужды важных прикладных областей: Фортран — естественно-научных и технических вычислений, а Лисп — искусственного интеллекта. Обе эти области по-прежнему важны, а программисты, работающие в них, настолько привязаны к этим двум языкам, что Лисп и Фортран вполне могут остаться в деле еще по крайней мере на четверть столетия.

Лисп изменяется. Scheme, его диалект, используемый в этой книге, развился из первоначального Лиспа и отличается от него в некоторых важных отношениях: в частности, используются статические области связывания переменных, а функции могут возвращать в качестве значений другие функции. По семантической структуре Scheme так же близка к Алголу 60, как и к ранним вариантам Лиспа. Алгол 60, который уже никогда не будет живым языком, продолжает жить в генах Scheme и Паскаля. Пожалуй, трудно найти две более разные культуры программирования, чем те, что образовались вокруг этих двух языков и используют их в качестве единой валюты. Паскаль служит для построения пирамид — впечатляющих, захватывающих статических структур, создаваемых армиями, которые укладывают на места тяжелые плиты. При помощи Лиспа порождаются организмы — впечатляющие, захватывающие динамические структуры, создаваемые командами, которые собирают их из мерцающих мириад более простых организмов. Организующие принципы в обоих случаях остаются одни и те же, за одним существенным исключением: программист, пишущий на Лиспе, располагает на порядок большей творческой свободой в том, что касается функций, которые он создает для использования другими. Программы на Лиспе населяют библиотеки функциями, которые оказываются настолько полезными, что они переживают породившие их приложения. Таким ростом полезности мы во многом обязаны списку — исконной лисповской структуре данных. Простота структуры списков и естественность их использования отражаются в удивительной общности функций. В Паскале обилие объявляемых структур данных ведет к специализации функций, которая сдерживает и наказывает случайное взаимодействие между ними. Лучше иметь 100 функций, которые работают с одной структурой данных, чем 10 функций, работающих с 10 структурами. В результате пирамиде приходится неподвижно стоять тысячелетиями; организм же будет развиваться или погибнет.

Чтобы увидеть эту разницу, сравните подачу материала и упражнения в этой книге с тем, что Вы найдете в любом вводном тексте, авторы которого используют Паскаль. Не поддавайтесь ошибочному впечатлению, будто этот текст может усвоить лишь студент MIT — представитель специфической породы, которая только там и встречается. Нет; именно такова должна быть всякая серьезная книга, посвященная программированию на Лиспе, вне зависимости от того, где и кто по ней учится.

Учтите, что это текст о программировании, в отличие от большинства книг по Лиспу,

которые используются для подготовки работников в области искусственного интеллекта. В конце концов, основные программистские заботы вычислительной инженерии и искусственного интеллекта стремятся к взаимопроникновению по мере того, как соответствующие системы увеличиваются в объеме. Это объясняет рост интереса к Лиспу за пределами искусственного интеллекта.

Как и можно было ожидать, глядя на цели, которые ставят перед собой исследователи в области искусственного интеллекта, область эта порождает множество значительных программистских задач. В других программистских культурах такой наплыв задач рождает новые языки. В самом деле, в любой большой программной задаче один из важных принципов организации состоит в том, чтобы ограничить и изолировать потоки информации в отдельных модулях задачи, изобретая для этого язык. По мере приближения к границам системы, где мы — люди — взаимодействуем чаще всего, эти языки обычно становятся все менее примитивными. В результате такие системы содержат сложные функции по обработке языка, повторенные по многу раз. У Лиспа же синтаксис и семантика настолько просты, что синтаксический разбор можно считать элементарной задачей. Таким образом, методы синтаксического разбора не играют почти никакой роли в программах на Лиспе, и построение языковых процессоров редко служит препятствием для роста и изменения больших Лисп-систем. Наконец, именно эта простота синтаксиса и семантики возлагает бремя свободы на всех программистов на Лиспе. Никакую программу на Лиспе больше, чем в несколько строк длиной, невозможно написать, не населив ее самостоятельными функциями. Находите новое и приспосабливайте; складывайте и стройте новыми способами! Я поднимаю тост за программиста на Лиспе, укладывающего свои мысли в гнезда скобок.

Алан Дж. Перлис
Нью-Хейвен, Коннектикут

Предисловие ко второму изданию

> Возможно ли, что программы не похожи ни на что другое, что они предназначены на выброс; что вся штука состоит в том, чтобы всегда видеть в них мыльный пузырь?
>
> Алан Дж. Перлис

Материал этой книги был основой вводного курса по информатике в MIT начиная с 1980 года. К тому времени, как было выпущено первое издание, мы преподавали этот материал в течение четырех лет, и прошло еще двенадцать лет до появления второго издания. Нам приятно, что наша работа была широко признана и включена в другие тексты. Мы видели, как наши ученики черпали идеи и программы из этой книги и на их основе строили новые компьютерные системы и языки. Буквально по старому талмудическому каламбуру, наши ученики стали нашими строителями. Мы рады, что у нас такие одаренные ученики и такие превосходные строители.

Готовя это издание, мы включили в него сотни поправок, которые нам подсказали как наш собственный преподавательский опыт, так и советы коллег из MIT и других мест. Мы заново спроектировали большинство основных программных систем в этой книге, включая систему обобщенной арифметики, интерпретаторы, имитатор регистровых машин и компилятор; кроме того, мы переписали все примеры программ так, чтобы любая реализация Scheme, соответствующая стандарту Scheme IEEE (IEEE 1990), была способна выполнять этот код.

В этом издании подчеркиваются несколько новых тем. Самая важная из них состоит в том, что центральную роль в вычислительных моделях играют различные подходы ко времени: объекты, обладающие состоянием, параллельное программирование, функциональное программирование, ленивые вычисления и недетерминистское программирование. Мы включили в текст новые разделы по параллельным вычислениям и недетерминизму и постарались интегрировать эту тему в материал книги на всем ее протяжении.

Первое издание книги почти точно следовало программе нашего односеместрового курса в MIT. Рассмотреть весь материал, включая то, что добавлено во втором издании, в течение семестра будет невозможно, так что преподавателю придется выбирать. В нашей собственной практике мы иногда пропускаем раздел про логическое програм-

мирование (раздел 4.4); наши студенты используют имитатор регистровых машин, но мы не описываем его реализацию (раздел 5.2); наконец, мы даем лишь беглый обзор компилятора (раздел 5.5). Даже в таком виде курс остается интенсивным. Некоторые преподаватели предпочтут ограничиться первыми тремя или четырьмя главами, оставляя прочий материал для последующих курсов.

Сайт World Wide Web `http://mitpress.mit.edu/sicp` предоставляет поддержку пользователям этой книги. Там есть программы из книги, простые задания по программированию, сопроводительные материалы и реализации диалекта Лиспа Scheme.*

*В настоящее время (август 2005 г.) на сайте имеется также полный текст англоязычного издания. — *прим. перев.*

Предисловие к первому изданию

> Компьютер подобен скрипке. Представьте себе новичка, который сначала испытывает проигрыватель, затем скрипку. Скрипка, говорит он, звучит ужасно. Именно этот аргумент мы слышали от наших гуманитариев и специалистов по информатике. Компьютеры, говорят они, хороши для определенных целей, но они недостаточно гибки. Так же и со скрипкой, и с пишущей машинкой, пока Вы не научились их использовать.
>
> Марвин Минский.
> «Почему программирование — хороший способ выражения малопонятных и туманно сформулированных идей»

«Структура и интерпретация компьютерных программ» — это вводный курс по информатике в Массачусетском Технологическом институте (MIT). Он обязателен для всех студентов MIT на специальностях «электротехника» и «информатика», как одна из четырех частей «общей базовой программы обучения», которая включает еще два курса по электрическим схемам и линейным системам, а также курс по проектированию цифровых систем. Мы принимали участие в развитии этого курса начиная с 1978 года и преподавали этот материал в его нынешней форме начиная с осени 1980 года шестистам–семистам студентам в год. Большая часть этих студентов не имела почти или совсем никакого формального образования в области вычислительной техники, хотя у многих была возможность общения с компьютерами, а некоторые обладали значительным опытом в программировании либо проектировании аппаратуры.

Построение этого вводного курса по информатике отражает две основные задачи. Во-первых, мы хотим привить слушателям идею, что компьютерный язык — это не просто способ заставить компьютер производить вычисления, а новое формальное средство выражения методологических идей. Таким образом, программы должны писаться для того, чтобы их читали люди, и лишь во вторую очередь для выполнения машиной. Во-вторых, мы считаем, что основной материал, на который должен быть направлен курс этого уровня, — не синтаксис определенного языка программирования, не умные алгоритмы

для эффективного вычисления определенных функций, даже не математический анализ алгоритмов и оснований программирования, но методы управления интеллектуальной сложностью больших программных систем.

Наша цель — развить в студентах, проходящих этот курс, хороший вкус к элементам стиля и эстетике программирования. Они должны овладеть основными методами управления сложностью в большой системе, уметь прочитать 50-ти страничную программу, если она написана в хорошем стиле. Они должны в каждый данный момент понимать, чего сейчас не следует читать и что сейчас не нужно понимать. Они не должны испытывать страха перед модификацией программы, сохраняя при этом дух и стиль исходного автора.

Все эти умения ни в коем случае не исчерпываются компьютерным программированием. Методы, которым мы учим и из которых мы черпаем, одни и те же в любом техническом проектировании. Мы управляем сложностью с помощью построения абстракций, скрывающих, когда это нужно, детали. Мы управляем сложностью путем установления стандартных интерфейсов, которые позволяют нам строить системы из единообразных, хорошо понимаемых кусков способом «смеси и стыковки». Мы управляем сложностью с помощью построения новых языков для описания проекта, каждый из которых концентрирует внимание на определенных деталях проекта и уводит его от других.

В основе нашего подхода к предмету лежит убеждение, что «компьютерная наука» не является наукой и что ее значение мало связано с компьютерами. Компьютерная революция — это революция в том, как мы мыслим и как мы выражаем наши мысли. Сущность этих изменений состоит в появлении дисциплины, которую можно назвать *компьютерной эпистемологией*, — исследования структуры знания с императивной точки зрения, в противоположность более декларативной точке зрения классических математических дисциплин. Математика дает нам структуру, в которой мы можем точно описывать понятия типа «что такое». Вычислительная наука дает нам структуру, в которой мы можем точно описывать понятия типа «как».

В преподавании мы используем диалект языка программирования Лисп. Мы не учим формальной стороне языка, поскольку в этом не возникает нужды. Мы просто его используем, и студенты схватывают его за несколько дней. В этом состоит одно из больших преимуществ лиспоподобных языков: в них очень мало способов строить составные выражения и нет почти никакой синтаксической структуры. Все формальные детали могут быть описаны за час, как правила шахмат. Спустя некоторое время мы забываем о формальных свойствах языка (поскольку их нет) и продолжаем говорить о настоящих вопросах; определяем, что именно мы хотим вычислить, как мы будем разбивать задачу на куски разумного размера и как потом будем работать с этими кусками. Еще одно преимущество Лиспа состоит в том, что он поддерживает (но не навязывает) больше крупномасштабных стратегий разбиения программ на модули, чем любой другой известный нам язык. Можно строить абстракции процедур и данных, можно использовать функции высших порядков, чтобы охватить общие шаблоны их использования, можно моделировать локальное состояние с использованием присваивания и изменения данных, можно связывать части программы с помощью потоков и задержанных вычислений, и можно с легкостью реализовывать встроенные языки. Все это включено в диалоговое окружение с превосходной поддержкой пошагового проектирования, построения, тестирования и отладки программ. Мы благодарны всем поколениям кудесников Лиспа начиная с Джона Маккарти, которые создали замечательный инструмент непревзойденной силы и красоты.

Scheme, тот диалект Лиспа, который мы используем, пытается совместить силу и красоту Лиспа и Алгола. От Лиспа мы берем метаязыковую мощь, которой он обязан простоте своего синтаксиса, единообразное представление программ как объектов данных, выделение данных из кучи с последующей их утилизацией сборщиком мусора. От Алгола мы берем лексическую область действия и блоковую структуру, подаренные нам первопроходцами проектирования языков программирования из комитета по Алголу. Мы хотим упомянуть Джона Рейнольдса и Питера Ландина, открывших связь Чёрчева лямбда-исчисления со структурой языков программирования. Мы также отдаем дань признательности математикам, разведавшим эту область за десятилетия до появления на сцене компьютеров. Среди этих первопроходцев были Алонсо Чёрч, Беркли Россер, Стефен Клини и Хаскелл Карри.

Благодарности

Мы хотели бы поблагодарить множество людей, которые помогли нам создать эту книгу и этот курс.

Наш курс — очевидный интеллектуальный потомок «6.321», замечательного курса по компьютерной лингвистике и лямбда-исчислению, который читали в MIT в конце 60-х Джек Уозенкрафт и Артур Эванс мл.

Мы очень обязаны Роберту Фано, который реорганизовал вводную программу MIT по электротехнике и информатике, сосредоточившись на принципах технического проектирования. Он вдохновил нас на это предприятие и написал первую программу курса, из которого развилась эта книга.

Стиль и эстетика программирования, которые мы пытаемся привить читателю, во многом были разработаны совместно с Гаем Льюисом Стилом мл., который вместе с Джеральдом Джеем Сассманом участвовал в первоначальной разработке языка Scheme. В дополнение к этому Дэвид Тёрнер, Питер Хендерсон, Дэн Фридман, Дэвид Уайз и Уилл Клингер научили нас многим из приемов функционального программирования, которые излагаются в данной книге.

Джон Мозес научил нас структурировать большие системы. Благодаря его опыту с системой символьных вычислений Macsyma мы стали понимать, что необходимо избегать усложненности структур управления и в первую очередь заботиться о такой организации данных, которая отражает реальную структуру моделируемого мира

Марвин Минский и Сеймур Пэйперт сильно повлияли на формирование нашего подхода к программированию и к его месту в нашей интеллектуальной жизни. Благодаря им мы понимаем, что вычисление дает нам средство выражения и исследования мыслей, которые иначе были бы слишком сложны, чтобы с ними можно было точно работать. Они подчеркивают, что способность писать и изменять программы дает студенту мощное средство, с помощью которого исследование становится естественной деятельностью.

Кроме того, мы полностью согласны с Аланом Перлисом в том, что программирование — это огромное удовольствие и что нам нужно стараться поддерживать радость программирования. Часть этой радости приходит от наблюдения за работой великих мастеров. Нам выпало счастье быть учениками у ног Билла Госпера и Ричарда Гринблатта.

Трудно перечислить всех тех, кто принял участие в развитии программы нашего курса. Мы благодарим всех лекторов, инструкторов и тьюторов, которые работали с нами в прошедшие пятнадцать лет и потратили много часов сверхурочной работы на наш предмет, особенно Билла Сиберта, Альберта Мейера, Джо Стоя, Рэнди Дэвиса, Луи Брэйда, Эрика Гримсона, Рода Брукса, Линна Стейна и Питера Соловитца. Мы бы хотели особо отметить выдающийся педагогический вклад Франклина Турбака, который теперь пре-

подает в Уэллесли: его работа по обучению младшекурсников установила стандарт, на который мы все можем равняться. Мы благодарны Джерри Сальтцеру и Джиму Миллеру, которые помогли нам бороться с тайнами параллельных вычислений, а также Питеру Соловитцу и Дэвиду Макаллестеру за их вклад в представление недетерминистских вычислений в главе 4.

Много людей вложило немалый труд в преподавание этого материала и в других университетах. Вот некоторые из тех, с кем мы тесно общались в работе: это Джекоб Кацнельсон в Технионе, Хэрди Майер в Калифорнийском университете в Ирвине, Джо Стой в Оксфорде, Элиша Сэкс в университете Пердью и Ян Коморовский в Норвежском университете Науки и Техники. Мы гордимся коллегами, которые получили награды за адаптацию этого предмета в других университетах: это Кеннет Йип в Йеле, Брайан Харви в Калифорнийском университете в Беркли и Дон Хаттенлохер в Корнелле.

Эл Мойе дал нам возможность прочитать этот материал инженерам компании Хьюлетт-Паккард и устроил производство видеоверсии этих лекций. Мы хотели бы поблагодарить одаренных преподавателей — в особенности Джима Миллера, Билла Сиберта и Майка Айзенберга, — которые разработали курсы повышения квалификации с использованием этих видеоматериалов и преподавали по ним в различных университетах и корпорациях по всему миру.

Множество работников образования проделали значительную работу по переводу первого издания. Мишель Бриан, Пьер Шамар и Андре Пик сделали французское издание, Сюзанна Дэниелс-Хэрольд выполнила немецкий перевод, а Фумио Мотоёси — японский. Мы не знаем авторов китайского издания, однако считаем для себя честью быть выбранными в качестве объекта «неавторизованного» перевода.

Трудно перечислить всех людей, внесших технический вклад в разработку систем программирования на языке Scheme, которые мы используем в учебных целях. Кроме Гая Стила, в список важнейших волшебников входят Крис Хансон, Джо Боубир, Джим Миллер, Гильермо Росас и Стефен Адамс. Кроме них, существенное время и силы вложили Ричард Столлман, Алан Боуден, Кент Питман, Джон Тафт, Нил Мэйл, Джон Лэмпинг, Гуин Оснос, Трейси Ларраби, Джордж Карретт, Сома Чаудхури, Билл Киаркиаро, Стивен Кирш, Лей Клотц, Уэйн Носс, Тодд Кэсс, Патрик О'Доннелл, Кевин Теобальд, Дэниел Вайзе, Кеннет Синклер, Энтони Кортеманш, Генри М. Ву, Эндрю Берлин и Рут Шью.

Помимо авторов реализации MIT, мы хотели бы поблагодарить множество людей, работавших над стандартом Scheme IEEE, в том числе Уильяма Клингера и Джонатана Риса, которые редактировали R^4RS, а также Криса Хэйнса, Дэвида Бартли, Криса Хансона и Джима Миллера, которые подготовили стандарт IEEE.

Долгое время Дэн Фридман был лидером сообщества языка Scheme. Работа сообщества в более широком плане переходит границы вопросов разработки языка и включает значительные инновации в образовании, такие как курс для старшей школы, основанный на EdScheme компании Schemer's Inc. и замечательные книги Майка Айзенберга, Брайана Харви и Мэтью Райта.

Мы ценим труд тех, кто принял участие в превращении этой работы в настоящую книгу, особенно Терри Элинга, Ларри Коэна и Пола Бетджа из издательства MIT Press. Элла Мэйзел нашла замечательный рисунок для обложки. Что касается второго издания, то мы особенно благодарны Бернарду и Элле Мэйзел за помощь с оформлением книги, а также Дэвиду Джонсу, великому волшебнику TEXа. Мы также в долгу перед читателя-

ми, сделавшими проницательные замечания по новому проекту: Джекобу Кацнельсону, Харди Мейеру, Джиму Миллеру и в особенности Брайану Харви, который был для этой книги тем же, кем Джули была для его книги *Просто Scheme*.

Наконец, мы хотели бы выразить признательность организациям, которые поддерживали нашу работу в течение этих лет. Мы благодарны компании Хьюлетт-Паккард за поддержку, которая стала возможной благодаря Айре Гольдстейну и Джоэлю Бирнбауму, а также агентству DARPA за поддержку, которая стала возможной благодаря Бобу Кану.*

*Со своей стороны хотелось бы поблагодарить Константина Добкина, Андрея Комеча, Сергея Коропа, Алексея Овчинникова, Алекса Отта, Вадима Радионова, Марию Рубинштейн и особенно Бориса Смилгу. — *прим. перев.*

pgh

Глава 1

Построение абстракций с помощью процедур

> Действия, в которых ум проявляет свои способности в отношении своих простых идей, суть главным образом следующие три: 1) Соединение нескольких простых идей в одну сложную; так образовались все сложные идеи, 2) Сведение вместе двух идей, все равно, простых или сложных, и сопоставление их друг с другом так, чтобы обозревать их сразу, но не соединять в одну; так ум приобретает все свои идеи отношений, 3) Обособление идей от всех других идей, сопутствующих им в реальной действительности; это действие называется «абстрагированием», и при его помощи образованы все общие идеи в уме.
>
> Джон Локк.
> «Опыт о человеческом разуме» (1690)
> (Перевод А.Н. Савина)

Мы собираемся изучать понятие *вычислительного процесса* (computational process). Вычислительные процессы — это абстрактные существа, которые живут в компьютерах. Развиваясь, процессы манипулируют абстракциями другого типа, которые называются *данными* (data). Эволюция процесса направляется набором правил, называемым *программой* (program). В сущности, мы заколдовываем духов компьютера с помощью своих чар.

Вычислительные процессы и вправду вполне соответствуют представлениям колдуна о ду́хах. Их нельзя увидеть или потрогать. Они вообще сделаны не из вещества. В то же время они совершенно реальны. Они могут выполнять умственную работу, могут отвечать на вопросы. Они способны воздействовать на внешний мир, оплачивая счета в банке или управляя рукой робота на заводе. Программы, которыми мы пользуемся для заклинания процессов, похожи на чары колдуна. Они тщательно составляются из символических выражений на сложных и немногим известных *языках программирования* (programming languages), описывающих задачи, которые мы хотим поручить процессам.

На исправно работающем компьютере вычислительный процесс выполняет программы точно и безошибочно. Таким образом, подобно ученику чародея, программисты-но-

вички должны научиться понимать и предсказывать последствия своих заклинаний. Даже мелкие ошибки (их обычно называют *блохами* (bugs) или *глюками* (glitches)), могут привести к сложным и непредсказуемым последствиям.

К счастью, обучение программированию не так опасно, как обучение колдовству, поскольку духи, с которыми мы имеем дело, надежно связаны. В то же время программирование в реальном мире требует осторожности, профессионализма и мудрости. Например, мелкая ошибка в программе автоматизированного проектирования может привести к катастрофе самолета, прорыву плотины или самоуничтожению промышленного робота.

Специалисты по программному обеспечению умеют организовывать программы так, чтобы быть потом обоснованно уверенными: получившиеся процессы будут выполнять те задачи, для которых они предназначены. Они могут изобразить поведение системы заранее. Они знают, как построить программу так, чтобы непредвиденные проблемы не привели к катастрофическим последствиям, а когда эти проблемы возникают, программисты умеют*отлаживать* (debug) свои программы. Хорошо спроектированные вычислительные системы, подобно хорошо спроектированным автомобилям или ядерным реакторам, построены модульно, так что их части могут создаваться, заменяться и отлаживаться по отдельности.

Программирование на Лиспе

Для описания процессов нам нужен подходящий язык, и с этой целью мы используем язык программирования Лисп. Точно так же, как обычные наши мысли чаще всего выражаются на естественном языке (например, английском, французском или японском), а описания количественных явлений выражаются языком математики, наши процедурные мысли будут выражаться на Лиспе. Лисп был изобретен в конце 1950-х как формализм для рассуждений об определенном типе логических выражений, называемых *уравнения рекурсии* (recursion equations), как о модели вычислений. Язык был придуман Джоном Маккарти и основывается на его статье «Рекурсивные функции над символьными выражениями и их вычисление с помощью машины» (McCarthy 1960).

Несмотря на то, что Лисп возник как математический формализм, это практический язык программирования. *Интерпретатор* (interpreter) Лиспа представляет собой машину, которая выполняет процессы, описанные на языке Лисп. Первый интерпретатор Лиспа написал сам Маккарти с помощью коллег и студентов из Группы по Искусственному Интеллекту Исследовательской лаборатории по Электронике MIT и Вычислительного центра MIT[1]. Лисп, чье название происходит от сокращения английских слов LISt Processing (обработка списков), был создан с целью обеспечить возможность символьной обработки для решения таких программистских задач, как символьное дифференцирование и интегрирование алгебраических выражений. С этой целью он содержал новые объекты данных, известные под названием атомов и списков, что резко отличало его от других языков того времени.

Лисп не был результатом срежиссированного проекта. Он развивался неформально, экспериментальным путем, с учетом запросов пользователей и прагматических соображений реализации. Неформальная эволюция Лиспа продолжалась долгие годы, и сооб-

[1] *Руководство программиста по Лиспу 1* появилось в 1960 году, а *Руководство программиста по Лиспу 1.5* (McCarthy 1965) в 1965 году. Ранняя история Лиспа описана в McCarthy 1978.

щество пользователей Лиспа традиционно отвергало попытки провозгласить какое-либо «официальное» описание языка. Вместе с гибкостью и изяществом первоначального замысла такая эволюция позволила Лиспу, который сейчас по возрасту второй из широко используемых языков (старше только Фортран), непрерывно адаптироваться и вбирать в себя наиболее современные идеи о проектировании программ. Таким образом, сегодня Лисп представляет собой семью диалектов, которые, хотя и разделяют большую часть изначальных свойств, могут существенным образом друг от друга отличаться. Тот диалект, которым мы пользуемся в этой книге, называется Scheme (Схема)[2].

Из-за своего экспериментального характера и внимания к символьной обработке первое время Лисп был весьма неэффективен при решении вычислительных задач, по крайней мере по сравнению с Фортраном. Однако за прошедшие годы были разработаны компиляторы Лиспа, которые переводят программы в машинный код, способный производить численные вычисления с разумной эффективностью. А для специализированных приложений Лисп удавалось использовать весьма эффективно[3]. Хотя Лисп и не преодолел пока свою старую репутацию безнадежно медленного языка, в наше время он используется во многих приложениях, где эффективность не является главной заботой. Например, Лисп стал любимым языком для оболочек операционных систем, а также в качестве языка расширения для редакторов и систем автоматизированного проектирования.

Но коль скоро Лисп не похож на типичные языки, почему же мы тогда используем его как основу для нашего разговора о программировании? Потому что этот язык обладает уникальными свойствами, которые делают его замечательным средством для изучения важнейших конструкций программирования и структур данных, а также для соотнесения их с деталями языка, которые их поддерживают. Самое существенное из этих свойств — то, что лисповские описания процессов, называемые *процедурами* (procedures), сами по себе могут представляться и обрабатываться как данные Лиспа. Важность этого в том, что существуют мощные методы проектирования программ, которые опираются на возможность сгладить традиционное различение «пассивных» данных и «активных» процессов. Как мы обнаружим, способность Лиспа рассматривать процедуры в качестве данных делает его одним из самых удобных языков для исследования этих методов. Способность

[2]Большинство крупных Лисп-программ 1970х, были написаны на одном из двух диалектов: MacLisp (Moon 1978; Pitman 1983), разработанный в рамках проекта MAC в MIT, и InterLisp (Teitelman 1974), разработанный в компании «Болт, Беранек и Ньюман» и в Исследовательском центре компании Xerox в Пало Альто. Диалект Portable Standard Lisp (Переносимый Стандартный Лисп, Hearn 1969; Griss 1981) был спроектирован так, чтобы его легко было переносить на разные машины. MacLisp породил несколько поддиалектов, например Franz Lisp, разработанный в Калифорнийском университете в Беркли, и Zetalisp (Moon 1981), который основывался на специализированном процессоре, спроектированном в лаборатории Искусственного Интеллекта в MIT для наиболее эффективного выполнения программ на Лиспе. Диалект Лиспа, используемый в этой книге, называется Scheme (Steele 1975). Он был изобретен в 1975 году Гаем Льюисом Стилом мл. и Джеральдом Джеем Сассманом в лаборатории Искусственного Интеллекта MIT, а затем заново реализован для использования в учебных целях в MIT. Scheme стала стандартом IEEE в 1990 году (IEEE 1900). Диалект Common Lisp (Steele 1982; Steele 1990) был специально разработан Лисп-сообществом так, чтобы сочетать свойства более ранних диалектов Лиспа и стать промышленным стандартом Лиспа. Common Lisp стал стандартом ANSI в 1994 году (ANSI 1994).

[3]Одним из таких приложений был пионерский эксперимент, имевший научное значение — интегрирование движения Солнечной системы, которое превосходило по точности предыдущие результаты примерно на два порядка и продемонстрировало, что динамика Солнечной системы хаотична. Это вычисление стало возможным благодаря новым алгоритмам интегрирования, специализированному компилятору и специализированному компьютеру; причем все они были реализованы с помощью программных средств, написанных на Лиспе (Abelson et al. 1992; Sussman and Wisdom 1992).

представлять процедуры в качестве данных делает Лисп еще и замечательным языком для написания программ, которые должны манипулировать другими программами в качестве данных, таких как интерпретаторы и компиляторы, поддерживающие компьютерные языки. А помимо и превыше всех этих соображений, писать программы на Лиспе — громадное удовольствие.

1.1. Элементы программирования

Мощный язык программирования — это нечто большее. чем просто средство, с помощью которого можно учить компьютер решать задачи. Язык также служит средой, в которой мы организуем свое мышление о процессах. Таким образом, когда мы описываем язык, мы должны уделять особое внимание тем средствам, которые в нем имеются для того, чтобы комбинировать простые понятия и получать из них сложные. Всякий язык программирования обладает тремя предназначенными для этого механизмами:

элементарные выражения, представляющие минимальные сущности, с которыми язык имеет дело;

средства комбинирования, с помощью которых из простых объектов составляются сложные;

средства абстракции, с помощью которых сложные объекты можно называть и обращаться с ними как с единым целым.

В программировании мы имеем дело с двумя типами объектов: процедурами и данными. (Впоследствии мы обнаружим, что на самом деле большой разницы между ними нет.) Говоря неформально, данные — это «материал», который мы хотим обрабатывать, а процедуры — это описания правил обработки данных. Таким образом, от любого мощного языка программирования требуется способность описывать простые данные и элементарные процедуры, а также наличие средств комбинирования и абстракции процедур и данных.

В этой главе мы будем работать только с простыми численными данными, так что мы сможем сконцентрировать внимание на правилах построения процедур[4]. В последующих главах мы увидим, что те же самые правила позволяют нам строить процедуры для работы со сложными данными.

[4]Называть числа «простыми данными» — это бесстыдный блеф. На самом деле работа с числами является одной из самых сложных и запутанных сторон любого языка программирования. Вот некоторые из возникающих при этом вопросов: Некоторые компьютеры отличают *целые числа* (integers), вроде 2, от *вещественных* (real numbers), вроде 2.71. Отличается ли вещественное число 2.00 от целого 2? Используются ли одни и те же арифметические операции для целых и для вещественных чисел? Что получится, если 6 поделить на 2: 3 или 3.0? Насколько большие числа мы можем представить? Сколько десятичных цифр после запятой мы можем хранить? Совпадает ли диапазон целых чисел с диапазоном вещественных? И помимо этих вопросов, разумеется, существует множество проблем, связанных с ошибками округления — целая наука численного анализа. Поскольку в этой книге мы говорим о проектировании больших программ, а не о численных методах, все эти проблемы мы будем игнорировать. Численные примеры в этой главе будут демонстрировать такое поведение при округлении, какое можно наблюдать, если использовать арифметические операции, сохраняющие при работе с вещественными числами ограниченное число десятичных цифр после запятой.

1.1.1. Выражения

Самый простой способ начать обучение программированию — рассмотреть несколько типичных примеров работы с интерпретатором диалекта Лиспа Scheme. Представьте, что Вы сидите за терминалом компьютера. Вы печатаете *выражение* (expression), а интерпретатор отвечает, выводя результат *вычисления* (evaluation) этого выражения.

Один из типов элементарных выражений, которые Вы можете вводить — это числа. (Говоря точнее, выражение, которое Вы печатаете, состоит из цифр, представляющих число по основанию 10.) Если Вы дадите Лиспу число

```
486
```

интерпретатор ответит Вам, напечатав[5]

```
486
```

Выражения, представляющие числа, могут сочетаться с выражением, представляющим элементарную процедуру (скажем, + или *), так что получается составное выражение, представляющее собой применение процедуры к этим числам. Например:

```
(+ 137 349)
486

(- 1000 334)
666

(* 5 99)
495

(/ 10 5)
2

(+ 2.7 10)
12.7
```

Выражения такого рода, образуемые путем заключения списка выражений в скобки с целью обозначить применение функции к аргументам, называются *комбинациями* (combinations). Самый левый элемент в списке называется*оператором* (operator), а остальные элементы — *операндами* (operands). Значение комбинации вычисляется путем применения процедуры, задаваемой оператором, к*аргументам* (arguments), которые являются значениями операндов.

Соглашение, по которому оператор ставится слева от операндов, известно как *префиксная нотация* (prefix notation), и поначалу оно может сбивать с толку, поскольку существенно отличается от общепринятой математической записи. Однако у префиксной нотации есть несколько преимуществ. Одно из них состоит в том, что префиксная запись может распространяться на процедуры с произвольным количеством аргументов, как в следующих примерах:

[5]Здесь и далее, когда нам нужно будет подчеркнуть разницу между вводом, который набирает на терминале пользователь, и выводом, который производит компьютер, мы будем изображать последний наклонным шрифтом.

```
(+ 21 35 12 7)
75

(* 25 4 12)
1200
```

Не возникает никакой неоднозначности, поскольку оператор всегда находится слева, а вся комбинация ограничена скобками.

Второе преимущество префиксной нотации состоит в том, что она естественным образом расширяется, позволяя комбинациям *вкладываться* (nest) друг в друга, то есть допускает комбинации, элементы которых сами являются комбинациями:

```
(+ (* 3 5) (- 10 6))
19
```

Не существует (в принципе) никакого предела для глубины такого вложения и общей сложности выражений, которые может вычислять интерпретатор Лиспа. Это мы, люди, путаемся даже в довольно простых выражениях, например

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

а интерпретатор с готовностью вычисляет его и дает ответ 57. Мы можем облегчить себе задачу, записывая такие выражения в форме

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

Эти правила форматирования называются *красивая печать* (pretty printing). Согласно им, всякая длинная комбинация записывается так, чтобы ее операнды выравнивались вертикально. Получающиеся отступы ясно показывают структуру выражения[6].

Даже работая со сложными выражениями, интерпретатор всегда ведет себя одинаковым образом: он считывает выражение с терминала, вычисляет его и печатает результат. Этот способ работы иногда называют *циклом чтение-вычисление-печать* (read-eval-print loop). Обратите особое внимание на то, что не нужно специально просить интерпретатор напечатать значение выражения[7].

1.1.2. Имена и окружение

Одна из важнейших характеристик языка программирования — какие в нем существуют средства использования имен для указания на вычислительные объекты. Мы

[6]Как правило, Лисп-системы содержат средства, которые помогают пользователям форматировать выражения. Особенно удобны две возможности: сдвигать курсор на правильную позицию для красивой печати каждый раз, когда начинается новая строка и подсвечивать нужную левую скобку каждый раз, когда печатается правая.

[7]Лисп следует соглашению, что у всякого выражения есть значение. Это соглашение, вместе со старой репутацией Лиспа как неэффективного языка, послужило источником остроумного замечания Алана Перлиса (парафразы из Оскара Уайльда), что «Программисты на Лиспе знают значение всего на свете, но ничему не знают цену».

говорим, что имя обозначает *переменную* (variable), чьим *значением* (value) является объект.

В диалекте Лиспа Scheme мы даем вещам имена с помощью слова `define`. Предложение

```
(define size 2)
```

заставляет интерпретатор связать значение 2 с именем `size`[8]. После того, как имя `size` связано со значением 2, мы можем указывать на значение 2 с помощью имени:

```
size
2

(* 5 size)
10
```

Вот еще примеры использования `define`:

```
(define pi 3.14159)

(define radius 10)

(* pi (* radius radius))
314.159

(define circumference (* 2 pi radius))

circumference
62.8318
```

Слово `define` служит в нашем языке простейшим средством абстракции, поскольку оно позволяет нам использовать простые имена для обозначения результатов сложных операций, как, например, вычисленная только что длина окружности — `circumference`. Вообще говоря, вычислительные объекты могут быть весьма сложными структурами, и было бы очень неудобно, если бы нам приходилось вспоминать и повторять все их детали каждый раз, когда нам захочется их использовать. На самом деле сложные программы конструируются методом построения шаг за шагом вычислительных объектов возрастающей сложности. Интерпретатор делает такое пошаговое построение программы особенно удобным, поскольку связи между именами и объектами могут создаваться последовательно по мере взаимодействия программиста с компьютером. Это свойство интерпретаторов облегчает пошаговое написание и тестирование программ, и во многом благодаря именно ему получается так, что программы на Лиспе обычно состоят из большого количества относительно простых процедур.

Ясно, что раз интерпретатор способен ассоциировать значения с символами и затем вспоминать их, то он должен иметь некоторого рода память, сохраняющую пары имя-объект. Эта память называется *окружением* (environment) (а точнее, *глобальным*

[8] Мы не печатаем в этой книге ответы интерпретатора при вычислении определений, поскольку они зависят от конкретной реализации языка.

окружением (global environment), поскольку позже мы увидим, что вычисление может иметь дело с несколькими окружениями)[9].

1.1.3. Вычисление комбинаций

Одна из наших целей в этой главе — выделить элементы процедурного мышления. Рассуждая в этом русле, примем во внимание, что интерпретатор, вычисляя значение комбинации, тоже следует процедуре:

- Чтобы вычислить комбинацию, требуется:
  - Вычислить все подвыражения комбинации.
  - Применить процедуру, которая является значением самого левого подвыражения (оператора) к аргументам — значениям остальных подвыражений (операндов).

Даже в этом простом правиле видны несколько важных свойств процессов в целом. Прежде всего, заметим, что на первом шаге для того, чтобы провести процесс вычисления для комбинации, нужно сначала проделать процесс вычисления для каждого элемента комбинации. Таким образом, правило вычисления *рекурсивно* (recursive) по своей природе; это означает, что в качестве одного из своих шагов оно включает применение того же самого правила[10].

Заметьте, какую краткость понятие рекурсии придает описанию того, что в случае комбинации с глубоким вложением выглядело бы как достаточно сложный процесс. Например, чтобы вычислить

```
(* (+ 2 (* 4 6))
   (+ 3 5 7))
```

требуется применить правило вычисления к четырем различным комбинациям. Картину этого процесса можно получить, нарисовав комбинацию в виде дерева, как показано на рис. 1.1. Каждая комбинация представляется в видевершины, а ее оператор и операнды — в виде ветвей, исходящих из этой вершины. Концевые вершины (то есть те, из которых не исходит ни одной ветви) представляют операторы или числа. Рассматривая вычисление как дерево, мы можем представить себе, что значения операндов распространяются от концевых вершин вверх и затем комбинируются на все более высоких уровнях. Впоследствии мы увидим, что рекурсия — это вообще очень мощный метод обработки иерархических, древовидных объектов. На самом деле форма правила вычисления «распространить значения наверх» является примером общего типа процессов, известного как *накопление по дереву* (tree accumulation).

[9] В главе 3 мы увидим, что понятие окружения необходимо как для понимания работы интерпретаторов, так и для их реализации.

[10] Может показаться странным, что правило вычисления предписывает нам в качестве части первого шага вычислить самый левый элемент комбинации, — ведь до сих пор это мог быть только оператор вроде `+` или `*`, представляющий встроенную процедуру, например, сложение или умножение. Позже мы увидим, что полезно иметь возможность работать и с комбинациями, чьи операторы сами по себе являются составными выражениями.

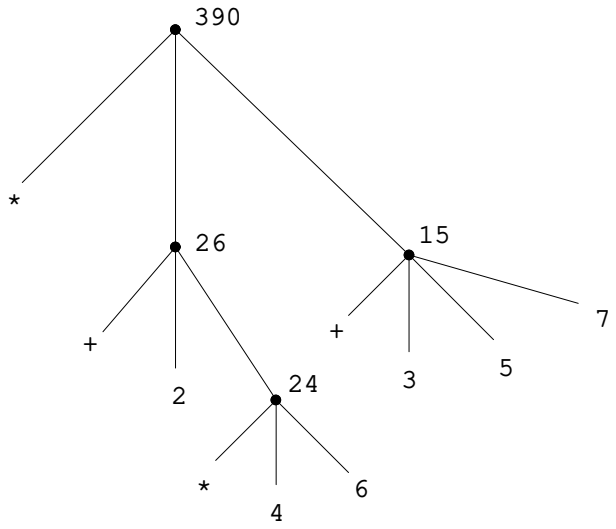


Рис. 1.1. Вычисление, представленное в виде дерева.

Далее, заметим, что многократное применение первого шага приводит нас к такой точке, где нам нужно вычислять уже не комбинации, а элементарные выражения, а именно числовые константы, встроенные операторы или другие имена. С этими случаями мы справляемся, положив, что:

- значением числовых констант являются те числа, которые они называют;
- значением встроенных операторов являются последовательности машинных команд, которые выполняют соответствующие операции; и
- значением остальных имен являются те объекты, с которыми эти имена связаны в окружении.

Мы можем рассматривать второе правило как частный случай третьего, постановив, что символы вроде `+` и `*` тоже включены в глобальное окружение и связаны с последовательностями машинных команд, которые и есть их «значения». Главное здесь — это роль окружения при определении значения символов в выражениях. В таком диалоговом языке, как Лисп, не имеет смысла говорить о значении выражения, скажем, `(+ x 1)`, не указывая никакой информации об окружении, которое дало бы значение символу `x` (и даже символу `+`). Как мы увидим в главе 3, общее понятие окружения, предоставляющего контекст, в котором происходит вычисление, будет играть важную роль в нашем понимании того, как выполняются программы.

Заметим, что рассмотренное нами правило вычисления не обрабатывает определений. Например, вычисление `(define x 3)` не означает применение `define` к двум аргументам, один из которых значение символа `x`, а другой равен 3, поскольку смысл `define` как раз и состоит в том, чтобы связать `x` со значением. (Таким образом, `(define x 3)` — не комбинация.)

Такие исключения из вышеописанного правила вычисления называются *особыми формами* (special forms). `Define` — пока что единственный встретившийся нам пример особой формы, но очень скоро мы познакомимся и с другими. У каждой особой формы свое собственное правило вычисления. Разные виды выражений (вместе со своими правилами вычисления) составляют синтаксис языка программирования. По сравнению с большинством языков программирования, у Лиспа очень простой синтаксис; а именно, правило вычисления для выражений может быть описано как очень простое общее правило плюс специальные правила для небольшого числа особых форм[11].

1.1.4. Составные процедуры

Мы нашли в Лиспе некоторые из тех элементов, которые должны присутствовать в любом мощном языке программирования:

- Числа и арифметические операции представляют собой элементарные данные и процедуры.
- Вложение комбинаций дает возможность комбинировать операции.
- Определения, которые связывают имена со значениями, дают ограниченные возможности абстракции.

Теперь мы узнаем об *определениях процедур* (procedure definitions) — значительно более мощном методе абстракции, с помощью которого составной операции можно дать имя и затем ссылаться на нее как на единое целое.

Для начала рассмотрим, как выразить понятие «возведения в квадрат». Можно сказать так: «Чтобы возвести что-нибудь в квадрат, нужно умножить его само на себя». Вот как это выражается в нашем языке:

```
(define (square x) (* x x))
```

Это можно понимать так:

`(define`	`(square`	`x)`	`*`	`x`	`x))`
↑	↑	↑	↑	↑	↑
Чтобы	возвести в квадрат	что-л.	умножь	это	само на себя

Здесь мы имеем *составную процедуру* (compound procedure), которой мы дали имя `square`. Эта процедура представляет операцию умножения чего-либо само на себя. Та вещь, которую нужно подвергнуть умножению, получает здесь имя `x`, которое играет ту

[11] Особые синтаксические формы, которые представляют собой просто удобное альтернативное поверхностное представление для того, что можно выразить более унифицированным способом, иногда называют *синтаксическим сахаром* (syntactic sugar), используя выражение Питера Ландина. По сравнению с пользователями других языков, программистов на Лиспе, как правило, мало волнует синтаксический сахар. (Для контраста возьмите руководство по Паскалю и посмотрите, сколько места там уделяется описанию синтаксиса). Такое презрение к синтаксису отчасти происходит от гибкости Лиспа, позволяющего легко изменять поверхностный синтаксис, а отчасти из наблюдения, что многие «удобные» синтаксические конструкции, которые делают язык менее последовательным, приносят в конце концов больше вреда, чем пользы, когда программы становятся большими и сложными. По словам Алана Перлиса, «Синтаксический сахар вызывает рак точки с запятой».

же роль, что в естественных языках играет местоимение. Вычисление этого определения создает составную процедуру и связывает ее с именем `square`[12].

Общая форма определения процедуры такова:

```
(define (⟨имя⟩ ⟨формальные-параметры⟩) ⟨тело⟩)
```

⟨*Имя*⟩ — это тот символ, с которым нужно связать в окружении определение процедуры[13]. ⟨*Формальные-параметры*⟩ — это имена, которые в теле процедуры используются для отсылки к соответствующим аргументам процедуры. ⟨*Тело*⟩ — это выражение, которое вычислит результат применения процедуры, когда формальные параметры будут заменены аргументами, к которым процедура будет применяться[14]. ⟨*Имя*⟩ и ⟨*формальные-параметры*⟩ заключены в скобки, как это было бы при вызове определяемой процедуры.

Теперь, когда процедура `square` определена, мы можем ее использовать:

```
(square 21)
441

(square (+ 2 5))
49

(square (square 3))
81
```

Кроме того, мы можем использовать `square` при определении других процедур. Например, $x^2 + y^2$ можно записать как

```
(+ (square x) (square y)))
```

Легко можно определить процедуру `sum-of-squares`, которая, получая в качестве аргументов два числа, дает в результате сумму их квадратов:

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
25
```

Теперь и `sum-of-squares` мы можем использовать как строительный блок при дальнейшем определении процедур:

---

[12]Заметьте, что здесь присутствуют две различные операции: мы создаем процедуру, и мы даем ей имя `square`. Возможно, и на самом деле даже важно, разделить эти два понятия: создавать процедуры, никак их не называя, и давать имена процедурам, уже созданным заранее. Мы увидим, как это делается, в разделе 1.3.2.

[13]На всем протяжении этой книги мы будем описывать обобщенныйсинтаксис выражений, используя курсив в угловых скобках — напр. ⟨*имя*⟩, чтобы обозначить «дырки» в выражении, которые нужно заполнить, когда это выражение используется в языке.

[14]В более общем случае тело процедуры может быть последовательностью выражений. В этом случае интерпретатор вычисляет по очереди все выражения в этой последовательности и возвращает в качестве значения применения процедуры значение последнего выражения.

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

Составные процедуры используются точно так же, как элементарные. В самом деле, глядя на приведенное выше определение `sum-of-squares`, невозможно выяснить, была ли `square` встроена в интерпретатор, подобно `+` и `*`, или ее определили как составную процедуру.

1.1.5. Подстановочная модель применения процедуры

Вычисляя комбинацию, оператор которой называет составную процедуру, интерпретатор осуществляет, вообще говоря, тот же процесс, что и для комбинаций, операторы которых называют элементарные процедуры — процесс, описанный в разделе 1.1.3. А именно, интерпретатор вычисляет элементы комбинации и применяет процедуру (значение оператора комбинации) к аргументам (значениям операндов комбинации).

Мы можем предположить, что механизм применения элементарных процедур к аргументам встроен в интерпретатор. Для составных процедур процесс протекает так:

- Чтобы применить составную процедуру к аргументам, требуется вычислить тело процедуры, заменив каждый формальный параметр соответствующим аргументом.

Чтобы проиллюстрировать этот процесс, вычислим комбинацию

```
(f 5)
```

где `f` — процедура, определенная в разделе 1.1.4. Начинаем мы с того, что восстанавливаем тело `f`:

```
(sum-of-squares (+ a 1) (* a 2))
```

Затем мы заменяем формальный параметр `a` на аргумент 5:

```
(sum-of-squares (+ 5 1) (* 5 2))
```

Таким образом, задача сводится к вычислению комбинации с двумя операндами и оператором `sum-of-squares`. Вычисление этой комбинации включает три подзадачи. Нам нужно вычислить оператор, чтобы получить процедуру, которую требуется применить, а также операнды, чтобы получить аргументы. При этом `(+ 5 1)` дает 6, а `(* 5 2)` дает 10, так что нам требуется применить процедуру `sum-of-squares` к 6 и 10. Эти значения подставляются на место формальных параметров `x` и `y` в теле `sum-of-squares`, приводя выражение к

```
(+ (square 6) (square 10))
```

Когда мы используем определение `square`, это приводится к

```
(+ (* 6 6) (* 10 10))
```

что при умножении сводится к

```
(+ 36 100)
```

и, наконец, к

```
136
```

Только что описанный нами процесс называется *подстановочной моделью* (substitution model) применения процедуры. Ее можно использовать как модель, которая определяет «смысл» понятия применения процедуры, пока рассматриваются процедуры из этой главы. Имеются, однако, две детали, которые необходимо подчеркнуть:

- Цель подстановочной модели — помочь нам представить, как применяются процедуры, а не дать описание того, как на самом деле работает интерпретатор. Как правило, интерпретаторы вычисляют применения процедур к аргументам без манипуляций с текстом процедуры, которые выражаются в подстановке значений для формальных параметров. На практике «подстановка» реализуется с помощью локальных окружений для формальных параметров. Более подробно мы обсудим это в главах 3 и 4, где мы детально исследуем реализацию интерпретатора.

- На протяжении этой книги мы представим последовательность усложняющихся моделей того, как работает интерпретатор, завершающуюся полным воплощением интерпретатора и компилятора в главе 5. Подстановочная модель — только первая из них, способ начать формально мыслить о моделях вычисления. Вообще, моделируя различные явления в науке и технике, мы начинаем с упрощенных, неполных моделей. Подстановочная модель в этом смысле не исключение. В частности, когда в главе 3 мы обратимся к использованию процедур с «изменяемыми данными», то мы увидим, что подстановочная модель этого не выдерживает и ее нужно заменить более сложной моделью применения процедур[15].

Аппликативный и нормальный порядки вычисления

В соответствии с описанием из раздела 1.1.3, интерпретатор сначала вычисляет оператор и операнды, а затем применяет получившуюся процедуру к получившимся аргументам. Но это не единственный способ осуществлять вычисления. Другая модель вычисления не вычисляет аргументы, пока не понадобится их значение. Вместо этого она подставляет на место параметров выражения-операнды, пока не получит выражение, в котором присутствуют только элементарные операторы, и лишь затем вычисляет его. Если бы мы использовали этот метод, вычисление

```
(f 5)
```

прошло бы последовательность подстановок

[15] Несмотря на простоту подстановочной модели, дать строгое математическое определение процессу подстановки оказывается удивительно сложно. Проблема возникает из-за возможности смешения имен, которые используются как формальные параметры процедуры, с именами (возможно, с ними совпадающими), которые используются в выражениях, к которым процедура может применяться. Имеется долгая история неверных определений *подстановки* (substitution) в литературе по логике и языкам программирования. Подробное обсуждение подстановки можно найти в Stoy 1977.

```
(sum-of-squares (+ 5 1) (* 5 2))

(+    (square (+ 5 1))      (square (* 5 2))  )

(+    (* (+ 5 1) (+ 5 1))   (* (* 5 2) (* 5 2)))
```

за которыми последуют редукции

```
(+         (* 6 6)             (* 10 10))

(+           36                   100)

                     136
```

Это дает тот же результат, что и предыдущая модель вычислений, но процесс его получения отличается. В частности, вычисление `(+ 5 1)` и `(* 5 2)` выполняется здесь по два раза, в соответствии с редукцией выражения

```
(* x x)
```

где **x** заменяется, соответственно, на `(+ 5 1)` и `(* 5 2)`.

Альтернативный метод «полная подстановка, затем редукция» известен под названием *нормальный порядок вычислений* (normal-order evaluation), в противоположность методу «вычисление аргументов, затем применение процедуры», которое называется *аппликативным порядком вычислений* (applicative-order evaluation). Можно показать, что для процедур, которые правильно моделируются с помощью подстановки (включая все процедуры из первых двух глав этой книги) и возвращают законные значения, нормальный и аппликативный порядки вычисления дают одно и то же значение. (См. упражнение 1.5, где приводится пример «незаконного» выражения, для которого нормальный и аппликативный порядки вычисления дают разные результаты.)

В Лиспе используется аппликативный порядок вычислений, отчасти из-за дополнительной эффективности, которую дает возможность не вычислять многократно выражения вроде приведенных выше `(+ 5 1)` и `(* 5 2)`, а отчасти, что важнее, потому что с нормальным порядком вычислений становится очень сложно обращаться, как только мы покидаем область процедур, которые можно смоделировать с помощью подстановки. С другой стороны, нормальный порядок вычислений может быть весьма ценным инструментом, и некоторые его применения мы рассмотрим в главах 3 и 4[16].

1.1.6. Условные выражения и предикаты

Выразительная сила того класса процедур, которые мы уже научились определять, очень ограничена, поскольку пока что у нас нет способа производить проверки и выполнять различные операции в зависимости от результата проверки. Например, мы не способны определить процедуру, вычисляющую модуль числа, проверяя, положительное

[16] В главе 3 мы описываем *обработку потоков* (stream processing), которая представляет собой способ обработки структур данных, кажущихся «бесконечными», с помощью ограниченной формы нормального порядка вычислений. В разделе 4.2 мы модифицируем интерпретатор Scheme так, что получается вариант языка с нормальным порядком вычислений.

ли это число, отрицательное или ноль, и предпринимая различные действия в соответствии с правилом

$$|x| = \begin{cases} x & \text{если} \quad x > 0 \\ 0 & \text{если} \quad x = 0 \\ -x & \text{если} \quad x < 0 \end{cases}$$

Такая конструкция называется *разбором случаев* (case analysis). В Лиспе существует особая форма для обозначения такого разбора случаев.Она называется `cond` (от английского слова *conditional*, «условный») и используется так:

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

Общая форма условного выражения такова:

```
(cond (⟨p1⟩ ⟨e1⟩)
      (⟨p2⟩ ⟨e2⟩)
      ⋮
      (⟨pn⟩ ⟨en⟩)))
```

Она состоит из символа `cond`, за которым следуют заключенные в скобки пары выражений `(`$\langle p \rangle$ $\langle e \rangle$`)`, называемых *ветвями* (clauses). В каждой из этих пар первое выражение — *предикат* (predicate), то есть выражение, значение которого интерпретируется как истина или ложь[17].

Условные выражения вычисляются так: сначала вычисляется предикат $\langle p_1 \rangle$. Если его значением является ложь, вычисляется $\langle p_2 \rangle$. Если значение $\langle p_2 \rangle$ также ложь, вычисляется $\langle p_3 \rangle$. Этот процесс продолжается до тех пор, пока не найдется предикат, значением которого будет истина, и в этом случае интерпретатор возвращает значение соответствующего *выражения-следствия* (consequent expression) в качестве значения всего условного выражения. Если ни один из $\langle p \rangle$ ни окажется истинным, значение условного выражения не определено.

Словом *предикат* называют процедуры, которые возвращают истину или ложь, а также выражения, которые имеют значением истину или ложь. Процедура вычисления модуля использует элементарные предикаты `<`,`=` и `>`[18].

Они принимают в качестве аргументов по два числа и, проверив, меньше ли первое из них второго, равно ему или больше, возвращают в зависимости от этого истину или ложь.

Можно написать процедуру вычисления модуля и так:

---

[17] «Интерпретируется как истина или ложь» означает следующее: в языке Scheme есть два выделенных значения, которые обозначаются константами `#t` и `#f`. Когда интерпретатор проверяет значение предиката, он интерпретирует `#f` как ложь. Любое другое значение считается истиной. (Таким образом, наличие `#t` логически не является необходимым, но иметь его удобно.) В этой книге мы будем использовать имена `true` и `false`, которые связаны со значениями `#t` и `#f`, соответственно.

[18] Еще она использует операцию «минус» `-`, которая, когда используется с одним операндом, как в выражении `(- x)`, обозначает смену знака.

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

что на русском языке можно было бы выразить следующим образом: «если x меньше нуля, вернуть $-x$; иначе вернуть x». `Else` — специальный символ, который в заключительной ветви `cond` можно использовать на месте ⟨*p*⟩. Это заставляет `cond` вернуть в качестве значения значение соответствующего ⟨*e*⟩ в случае, если все предыдущие ветви были пропущены. На самом деле, здесь на месте ⟨*p*⟩ можно было бы использовать любое выражение, которое всегда имеет значение истина.

Вот еще один способ написать процедуру вычисления модуля:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Здесь употребляется особая форма `if`, ограниченный вид условного выражения. Его можно использовать при разборе случаев, когда есть ровно два возможных исхода. Общая форма выражения `if` такова:

`(if` ⟨*предикат*⟩ ⟨*следствие*⟩ ⟨*альтернатива*⟩`)`

Чтобы вычислить выражение `if`, интерпретатор сначала вычисляет его ⟨*предикат*⟩. Если ⟨*предикат*⟩ дает истинное значение, интерпретатор вычисляет ⟨*следствие*⟩ и возвращает его значение. В противном случае он вычисляет ⟨*альтернативу*⟩ и возвращает ее значение[19].

В дополнение к элементарным предикатам вроде `<`, `=` и `>`, существуют операции логической композиции, которые позволяют нам конструировать составные предикаты. Из них чаще всего используются такие:

- `(and` ⟨e_1⟩ `. . .` ⟨e_n⟩`)`

Интерпретатор вычисляет выражения ⟨*e*⟩ по одному, слева направо. Если какое-нибудь из ⟨*e*⟩ дает ложное значение, значение всего выражения `and` — ложь, и остальные ⟨*e*⟩ не вычисляются. Если все ⟨*e*⟩ дают истинные значения, значением выражения `and` является значение последнего из них.

- `(or` ⟨e_1⟩ `. . .` ⟨e_n⟩`)`

Интерпретатор вычисляет выражения ⟨*e*⟩ по одному, слева направо. Если какое-нибудь из ⟨*e*⟩ дает истинное значение, это значение возвращается как результат выражения `or`, а остальные ⟨*e*⟩ не вычисляются. Если все ⟨*e*⟩ оказываются ложными, значением выражения `or` является ложь.

- `(not` ⟨*e*⟩`)`

Значение выражения `not` — истина, если значение выражения ⟨*e*⟩ ложно, и ложь в противном случае.

[19]Небольшая разница между `if` и `cond` состоит в том, что в `cond` каждое ⟨*e*⟩ может быть последовательностью выражений. Если соответствующее ⟨*p*⟩ оказывается истинным, выражения из ⟨*e*⟩ вычисляются по очереди, и в качестве значения `cond` возвращается значение последнего из них. Напротив, в `if` как ⟨*следствие*⟩, так и ⟨*альтернатива*⟩ обязаны состоять из одного выражения.

Заметим, что `and` и `or` — особые формы, а не процедуры, поскольку не обязательно вычисляются все подвыражения. `Not` — обычная процедура.

Как пример на использование этих конструкций, условие что число x находится в диапазоне $5 < x < 10$, можно выразить как

```
(and (> x 5) (< x 10))
```

Другой пример: мы можем определить предикат, который проверяет, что одно число больше или равно другому, как

```
(define (>= x y)
  (or (> x y) (= x y)))
```

или как

```
(define (>= x y)
  (not (< x y)))
```

Упражнение 1.1.
Ниже приведена последовательность выражений. Какой результат напечатает интерпретатор в ответ на каждое из них? Предполагается, что выражения вводятся в том же порядке, в каком они написаны.

```
10

(+ 5 3 4)

(- 9 1)

(/ 6 2)

(+ (* 2 4) (- 4 6))

(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))
```

```
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

Упражнение 1.2.
Переведите следующее выражение в префиксную форму:

$$\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}$$

Упражнение 1.3.
Определите процедуру, которая принимает в качестве аргументов три числа и возвращает сумму квадратов двух бóльших из них.

Упражнение 1.4.
Заметим, что наша модель вычислений разрешает существование комбинаций, операторы которых — составные выражения. С помощью этого наблюдения опишите, как работает следующая процедура:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

Упражнение 1.5.
Бен Битобор придумал тест для проверки интерпретатора на то, с каким порядком вычислений он работает, аппликативным или нормальным. Бен определяет такие две процедуры:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

Затем он вычисляет выражение

```
(test 0 (p))
```

Какое поведение увидит Бен, если интерпретатор использует аппликативный порядок вычислений? Какое поведение он увидит, если интерпретатор использует нормальный порядок? Объясните Ваш ответ. (Предполагается, что правило вычисления особой формы `if` одинаково независимо от того, какой порядок вычислений используется. Сначала вычисляется выражение-предикат, и результат определяет, нужно ли вычислять выражение-следствие или альтернативу.)

1.1.7. Пример: вычисление квадратного корня методом Ньютона

Процедуры, как они описаны выше, очень похожи на обыкновенные математические функции. Они устанавливают значение, которое определяется одним или более параметром. Но есть важное различие между математическими функциями и компьютерными процедурами. Процедуры должны быть эффективными.

В качестве примера рассмотрим задачу вычисления квадратного корня. Мы можем определить функцию «квадратный корень» так:

$$\sqrt{x} = \text{ такое } y, \text{ что } y \geq 0 \text{ и } y^2 = x$$

Это описывает совершенно нормальную математическую функцию. С помощью такого определения мы можем решать, является ли одно число квадратным корнем другого, или выводить общие свойства квадратных корней. С другой стороны, это определение не описывает процедуры. В самом деле, оно почти ничего не говорит о том, как найти квадратный корень данного числа. Не поможет и попытка перевести это определение на псевдо-Лисп:

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

Это только уход от вопроса.

Противопоставление функций и процедур отражает общее различие между описанием свойств объектов и описанием того, как что-то делать, или, как иногда говорят, различие между декларативным знанием и императивным знанием. В математике нас обычно интересуют декларативные описания (что такое), а в информатике императивные описания (как)[20].

Как вычисляются квадратные корни? Наиболее часто применяется Ньютонов метод последовательных приближений, который основан на том, что имея некоторое неточное значение y для квадратного корня из числа x, мы можем с помощью простой манипуляции получить более точное значение (более близкое к настоящему квадратному корню), если возьмем среднее между y и x/y[21]. Например, мы можем вычислить квадратный

[20] Декларативные и императивные описания тесно связаны между собой, как и математика с информатикой. Например, сказать, что ответ, получаемый программой, «верен», означает сделать об этой программе декларативное утверждение. Существует большое количество исследований, направленных на отыскание методов доказательства того, что программа корректна, и большая часть сложности этого предмета исследования связана с переходом от императивных утверждений (из которых строятся программы) к декларативным (которые можно использовать для рассуждений). Связана с этим и такая важная область современных исследований по проектированию языков программирования, как исследование так называемыхязыков сверхвысокого уровня, в которых программирование на самом деле происходит в терминах декларативных утверждений. Идея состоит в том, чтобы сделать интерпретаторы настолько умными, чтобы, получая от программиста знание типа «что такое», они были бы способны самостоятельно породить знание типа «как». В общем случае это сделать невозможно, но есть важные области, где удалось достичь прогресса. Мы вернемся к этой идее в главе 4.

[21] На самом деле алгоритм нахождения квадратного корня представляет собой частный случай метода Ньютона, который является общим методом нахождения корней уравнений. Собственно алгоритм нахождения квадратного корня был разработан Героном Александрийским в первом веке н.э. Мы увидим, как выразить общий метод Ньютона в виде процедуры на Лиспе, в разделе 1.3.4.

корень из 2 следующим образом: предположим, что начальное приближение равно 1.

Приближение	Частное x/y	Среднее
1	$\frac{2}{1} = 2$	$\frac{2+1}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{1.3333+1.5}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{1.4167+1.4118}{2} = 1.4142$
1.4142	...	...

Продолжая этот процесс, мы получаем все более точные приближения к квадратному корню.

Теперь формализуем этот процесс в терминах процедур. Начнем с подкоренного числа и какого-то значения приближения. Если приближение достаточно хорошо подходит для наших целей, то процесс закончен; если нет, мы должны повторить его с улучшенным значением приближения. Запишем эту базовую стратегию в виде процедуры:

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

Значение приближения улучшается с помощью взятия среднего между ним и частным подкоренного числа и старого значения приближения:

```
(define (improve guess x)
  (average guess (/ x guess)))
```

где

```
(define (average x y)
  (/ (+ x y) 2))
```

Нам нужно еще сказать, что такое для нас «достаточно хорошее» приближение. Следующий вариант сойдет для иллюстрации, но на самом деле это не очень хороший тест. (См. упражнение 1.7.) Идея состоит в том, чтобы улучшать приближения до тех пор, пока его квадрат не совпадет с подкоренным числом в пределах заранее заданного допуска (здесь 0.001)[22]:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

[22]Обычно мы будем давать предикатам имена, заканчивающиеся знаком вопроса, чтобы было проще запомнить, что это предикаты. Это не более чем стилистическое соглашение. С точки зрения интерпретатора, вопросительный знак — обыкновенный символ.

Наконец, нужно с чего-то начинать. Например, мы можем для начала предполагать, что квадратный корень любого числа равен 1[23]:

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

Если мы введем эти определения в интерпретатор, мы сможем использовать `sqrt` как любую другую процедуру:

```
(sqrt 9)
3.00009155413138

(sqrt (+ 100 37))
11.704699917758145

(sqrt (+ (sqrt 2) (sqrt 3)))
1.7739279023207892

(square (sqrt 1000))
1000.000369924366
```

Программа `sqrt` показывает также, что того простого процедурного языка, который мы описали до сих пор, достаточно, чтобы написать любую чисто вычислительную программу, которую можно было бы написать, скажем, на Си или Паскале. Это может показаться удивительным, поскольку в наш язык мы не включили никаких итеративных (циклических) конструкций, указывающих компьютеру, что нужно производить некое действие несколько раз. `Sqrt-iter`, с другой стороны, показывает, как можно выразить итерацию, не имея никакого специального конструкта, кроме обыкновенной способности вызвать процедуру[24].

Упражнение 1.6.
Лиза П. Хакер не понимает, почему `if` должна быть особой формой. «Почему нельзя просто определить ее как обычную процедуру с помощью `cond`?» — спрашивает она. Лизина подруга Ева Лу Атор утверждает, что, разумеется, можно, и определяет новую версию `if`:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Ева показывает Лизе новую программу:

[23]Обратите внимание, что мы записываем начальное приближение как 1.0, а не как 1.Во многих реализациях Лиспа здесь не будет никакой разницы. Однако интерпретатор MIT Scheme отличает точные целые числа от десятичных значений, и при делении двух целых получается не десятичная дробь, а рациональное число. Например, поделив 10/6, получим 5/3, а поделив 10.0/6.0, получим 1.6666666666666667. (Мы увидим, как реализовать арифметические операции над рациональными числами, в разделе 2.1.1.) Если в нашей программе квадратного корня мы начнем с начального приближения 1, а x будет точным целым числом, все последующие значения, получаемые при вычислении квадратного корня, будут не десятичными дробями, а рациональными числами. Поскольку при смешанных операциях над десятичными дробями и рациональными числами всегда получаются десятичные дроби, то начав со значения 1.0, все прочие мы получим в виде десятичных дробей.

[24]Читателям, которых заботят вопросы эффективности, связанные с использованием вызовов процедур для итерации, следует обратить внимание на замечания о «хвостовой рекурсии» в разделе 1.2.1.

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Обрадованная Лиза переписывает через `new-if` программу вычисления квадратного корня:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))
```

Что получится, когда Лиза попытается использовать эту процедуру для вычисления квадратных корней? Объясните.

Упражнение 1.7.
Проверка `good-enough?`, которую мы использовали для вычисления квадратных корней, будет довольно неэффективна для поиска квадратных корней от очень маленьких чисел. Кроме того, в настоящих компьютерах арифметические операции почти всегда вычисляются с ограниченной точностью. Поэтому наш тест оказывается неадекватным и для очень больших чисел. Альтернативный подход к реализации `good-enough?` состоит в том, чтобы следить, как от одной итерации к другой изменяется `guess`, и остановиться, когда изменение оказывается небольшой долей значения приближения. Разработайте процедуру вычисления квадратного корня, которая использует такой вариант проверки на завершение. Верно ли, что на больших и маленьких числах она работает лучше?

Упражнение 1.8.
Метод Ньютона для кубических корней основан на том, что если y является приближением к кубическому корню из x, то мы можем получить лучшее приближение по формуле

$$\frac{x/y^2 + 2y}{3}$$

С помощью этой формулы напишите процедуру вычисления кубического корня, подобную процедуре для квадратного корня. (В разделе 1.3.4 мы увидим, что можно реализовать общий метод Ньютона как абстракцию этих процедур для квадратного и кубического корня.)

1.1.8. Процедуры как абстракции типа «черный ящик»

`Sqrt` — наш первый пример процесса, определенного множеством зависимых друг от друга процедур. Заметим, что определение `sqrt-iter` *рекурсивно* (recursive); это означает, что процедура определяется в терминах самой себя. Идея, что можно определить процедуру саму через себя, возможно, кажется Вам подозрительной; неясно, как такое «циклическое» определение вообще может иметь смысл, не то что описывать хорошо определенный процесс для исполнения компьютером. Более осторожно мы подойдем к этому в разделе 1.2. Рассмотрим, однако, некоторые другие важные детали, которые иллюстрирует пример с `sqrt`.

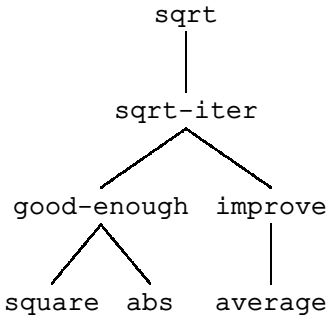


Рис. 1.2. Процедурная декомпозиция программы `sqrt`.

Заметим, что задача вычисления квадратных корней естественным образом разбивается на подзадачи: как понять, что очередное приближение нас устраивает, как улучшить очередное приближение, и так далее. Каждая из этих задач решается с помощью отдельной процедуры. Вся программа `sqrt` может рассматриваться как пучок процедур (показанный на рис. 1.1.8), отражающий декомпозицию задачи на подзадачи.

Важность декомпозиционной стратегии не просто в том, что задача разделяется на части. В конце концов, можно взять любую большую программу и поделить ее на части: первые десять строк, следующие десять строк и так далее. Существенно то, что каждая процедура выполняет точно определенную задачу, которая может быть использована при определении других процедур. Например, когда мы определяем процедуру `good-enough?` с помощью `square`, мы можем рассматривать процедуру `square` как «черный ящик». В этот момент нас не интересует, *как* она вычисляет свой результат, — важно только то, что она способна вычислить квадрат. О деталях того, как вычисляют квадраты, можно сейчас забыть и рассмотреть их потом. Действительно, пока мы рассматриваем процедуру `good-enough?`, `square` — не совсем процедура, но скорее абстракция процедуры, так называемая *процедурная абстракция* (procedural abstraction). На этом уровне абстракции все процедуры, вычисляющие квадрат, одинаково хороши.

Таким образом, если рассматривать только возвращаемые значения, то следующие две процедуры для возведения числа в квадрат будут неотличимы друг от друга. Каждая из них принимает числовой аргумент и возвращает в качестве значения квадрат этого числа[25].

```
(define (square x) (* x x))

(define (square x)
  (exp (double (log x))))

(define (double x) (+ x x))
```

Таким образом, определение процедуры должно быть способно скрывать детали. Мо-

[25]Неясно даже, которая из этих процедур более эффективна. Это зависит от того, какая имеется аппаратура. Существуют машины, на которых «очевидная» реализация будет медленней. Представьте себе машину, в которой очень эффективным способом хранятся большие таблицы логарифмов и обратных логарифмов.

жет оказаться, что пользователь процедуры не сам ее написал, а получил от другого программиста как черный ящик. От пользователя не должно требоваться знания, как работает процедура, чтобы ее использовать.

Локальные имена

Одна из деталей реализации, которая не должна заботить пользователя процедуры — это то, какие человек, писавший процедуру, выбрал имена для формальных параметров процедуры. Таким образом, следующие две процедуры должны быть неотличимы:

```
(define (square x) (* x x))

(define (square y) (* y y))
```

Этот принцип — что значение процедуры не должно зависеть от имен параметров, которые выбрал ее автор, — может сначала показаться очевидным, однако он имеет глубокие следствия. Простейшее из этих следствий состоит в том, что имена параметров должны быть локальными в теле процедуры. Например, в программе вычисления квадратного корня при определении `good-enough?` мы использовали `square`:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

Намерение автора `good-enough?` состоит в том, чтобы определить, достаточно ли близко квадрат первого аргумента лежит ко второму. Мы видим, что автор `good-enough?` обращается к первому аргументу с помощью имени `guess`, а ко второму с помощью имени `x`. Аргументом `square` является `guess`. Поскольку автор `square` использовал имя `x` (как мы видели выше), чтобы обратиться к этому аргументу, мы видим, что `x` в `good-enough?` должно отличаться от `x` в `square`. Запуск процедуры `square` не должен отразится на значении `x`, которое использует `good-enough?`, поскольку это значение `x` понадобится `good-enough?`, когда `square` будет вычислена.

Если бы параметры не были локальны по отношению к телам своих процедур, то параметр `x` в `square` смешался бы с параметром `x` из `good-enough?`, и поведение `good-enough?` зависело бы от того, какую версию `square` мы использовали. Таким образом, процедура `square` не была бы черным ящиком, как мы того хотим.

У формального параметра особая роль в определении процедуры: не имеет значения, какое у этого параметра имя. Такое имя называется *связанной переменной* (bound variable), и мы будем говорить, что определение процедуры *связывает* (binds) свои формальные параметры. Значение процедуры не изменяется, если во всем ее определении параметры последовательным образом переименованы[26]. Если переменная не связана, мы говорим, что она *свободна* (free). Множество выражений, для которых связывание определяет имя, называется *областью действия* (scope) этого имени. В определении процедуры связанные переменные, объявленные как формальные параметры процедуры, имеют своей областью действия тело процедуры.

В приведенном выше определении `good-enough?`, `guess` и `x` — связанные переменные, а `<`, `-`, `abs` и `square` — свободные. Значение `good-enough?` должно быть

[26]Понятие последовательного переименования на самом деле достаточно тонкое и трудное для определения. Знаменитым логикам случалось делать здесь ужасные ошибки.

независимо от того, какие имена мы выберем для `guess` и `x`, пока они остаются отличными друг от друга и от `<`, `-`, `abs` и `square`. (Если бы мы переименовали `guess` в `abs`, то породили бы ошибку, *захватив* (capture) переменную `abs`. Она превратилась бы из свободной в связанную.) Однако значение `good-enough?` не является независимым от ее свободных переменных. Разумеется, оно зависит от того факта (внешнего по отношению к этому определению), что символ `abs` называет процедуру вычисления модуля числа. `Good-enough?` будет вычислять совершенно другую функцию, если в ее определении мы вместо `abs` подставим `cos`.

Внутренние определения и блочная структура

До сих пор нам был доступен только один вид изоляции имен: формальные параметры процедуры локальны по отношению к телу этой процедуры. Программа вычисления квадратного корня иллюстрирует еще один вид управления использованием имен, которым мы хотели бы владеть. Существующая программа состоит из отдельных процедур:

```
(define (sqrt x)
  (sqrt-iter 1.0 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))
```

Проблема здесь состоит в том, что единственная процедура, которая важна для пользователей `sqrt` — это сама `sqrt`. Остальные процедуры (`sqrt-iter`, `good-enough?` и `improve`) только забивают им головы. Теперь пользователи не могут определять других процедур с именем `good-enough?` ни в какой другой программе, которая должна работать совместно с программой вычисления квадратного корня, поскольку `sqrt` требуется это имя. Эта проблема становится особенно тяжелой при построении больших систем, которые пишут много различных программистов. Например, при построении большой библиотеки численных процедур многие числовые функции вычисляются как последовательные приближения и могут потому иметь в качестве вспомогательных процедуры `good-enough?` и `improve`. Нам хотелось бы локализовать подпроцедуры, спрятав их внутри `sqrt`, так, чтобы `sqrt` могла сосуществовать с другими последовательными приближениями, при том что у каждой из них была бы своя собственная процедура `good-enough?`. Чтобы сделать это возможным, мы разрешаем процедуре иметь внутренние определения, локальные для этой процедуры. Например, при решении задачи вычисления квадратного корня мы можем написать

```
(define (sqrt x)
  (define (good-enough? guess x)
```

```
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

Такое вложение определений, называемое *блочной структурой* (block structure), дает правильное решение для простейшей задачи упаковки имен. Но здесь таится еще одна идея. Помимо того, что мы можем вложить определения вспомогательных процедур внутрь главной, мы можем их упростить. Поскольку переменная `x` связана в определении `sqrt`, процедуры `good-enough?`, `improve` и `sqrt-iter`, которые определены внутри `sqrt`, находятся в области действия `x`. Таким образом, нет нужды явно передавать `x` в каждую из этих процедур. Вместо этого мы можем сделать `x` свободной переменной во внутренних определениях, как это показано ниже. Тогда `x` получит свое значение от аргумента, с которым вызвана объемлющая их процедура `sqrt`. Такой порядок называется *лексической сферой действия* (lexical scoping) переменных[27].

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Мы будем часто использовать блочную структуру, чтобы разбивать большие программы на куски разумного размера[28]. Идея блочной структуры происходит из языка программирования Алгол 60. Она присутствует в большинстве современных языков программирования. Это важный инструмент, который помогает организовать построение больших программ.

1.2. Процедуры и порождаемые ими процессы

В предыдущем разделе мы рассмотрели элементы программирования. Мы использовали элементарные арифметические операции, комбинировали их и абстрагировали получившиеся составные операции путем определения составных процедур. Но всего

[27] Правило лексической сферы действия говорит, что свободные переменные в процедуре ссылаются на связывания этих переменных, сделанные в объемлющих определениях процедур; то есть они ищутся в окружении, в котором процедура была определена. Мы детально рассмотрим, как это работает, в главе 3, когда будем подробно описывать окружения и работу интерпретатора.

[28] Внутренние определения должны быть в начале тела процедуры. За последствия запуска программ, перемешивающих определения и их использование, администрация ответственности не несет.

этого еще недостаточно, чтобы сказать, что мы умеем программировать. Положение, в котором мы находимся, похоже на положение человека, выучившего шахматные правила, но ничего не знающего об основных дебютах, тактике и стратегии. Подобно шахматисту-новичку, мы пока ничего не знаем об основных схемах использования понятий в нашей области знаний. Нам недостает знаний о том, какие именно ходы следует делать (какие именно процедуры имеет смысл определять), и не хватает опыта предсказания последствий сделанного хода (выполнения процедуры).

Способность предвидеть последствия рассматриваемых действий необходима для того, чтобы стать квалифицированным программистом, — равно как и для любой другой синтетической, творческой деятельности. Например, квалифицированному фотографу нужно при взгляде на сцену понимать, насколько темным каждый ее участок покажется после печати при разном выборе экспозиции и разных условиях обработки. Только после этого можно проводить обратные рассуждения и выбирать кадр, освещение, экспозицию и условия обработки так, чтобы получить желаемый результат. Чтобы стать специалистами, нам надо научиться представлять процессы, генерируемые различными типами процедур. Только развив в себе такую способность, мы сможем научиться надежно строить программы, которые ведут себя так, как нам надо.

Процедура представляет собой шаблон *локальной эволюции* (local evolution) вычислительного процесса. Она указывает, как следующая стадия процесса строится из предыдущей. Нам хотелось бы уметь строить утверждения об общем, или *глобальном* (global) поведении процесса, локальная эволюция которого описана процедурой. В общем случае это сделать очень сложно, но по крайней мере мы можем попытаться описать некоторые типичные схемы эволюции процессов.

В этом разделе мы рассмотрим некоторые часто встречающиеся «формы» процессов, генерируемых простыми процедурами. Кроме того, мы рассмотрим, насколько сильно эти процессы расходуют такие важные вычислительные ресурсы, как время и память. Процедуры, которые мы будем рассматривать, весьма просты. Они будут играть такую же роль, как простые схемы в фотографии: это скорее упрощенные прототипические шаблоны, а не практические примеры сами по себе.

1.2.1. Линейные рекурсия и итерация

Для начала рассмотрим функцию факториал, определяемую уравнением

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

Существует множество способов вычислять факториалы. Один из них состоит в том, чтобы заметить, что $n!$ для любого положительного целого числа n равен n, умноженному на $(n-1)!$:

$$n! = n \cdot [(n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n-1)!$$

Таким образом, мы можем вычислить $n!$, вычислив сначала $(n-1)!$, а затем умножив его на n. После того, как мы добавляем условие, что $1!$ равен 1, это наблюдение можно непосредственно перевести в процедуру:

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Рис. 1.3. Линейно рекурсивный процесс для вычисления 6!.

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Можно использовать подстановочную модель из раздела 1.1.5 и увидеть эту процедуру в действии при вычислении 6!, как показано на рис. 1.3.

Теперь рассмотрим вычисление факториала с другой точки зрения. Мы можем описать правило вычисления $n!$, сказав, что мы сначала умножаем 1 на 2, затем результат умножаем на 3, затем на 4, и так пока не достигнем n. Мы можем описать это вычисление, сказав, что счетчик и произведение с каждым шагом одновременно изменяются согласно правилу

$$\text{произведение} = \text{счетчик} \cdot \text{произведение}$$
$$\text{счетчик} = \text{счетчик} + 1$$

и добавив условие, что $n!$ — это значение произведения в тот момент, когда счетчик становится больше, чем n.

Опять же, мы можем перестроить наше определение в процедуру вычисления факториала[29]:

[29] В настоящей программе мы, скорее всего, спрятали бы определение `fact-iter` с помощью блочной структуры, введенной в предыдущем разделе:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Здесь мы этого не сделали, чтобы как можно меньше думать о разных вещах одновременно.

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Рис. 1.4. Линейно итеративный процесс для вычисления 6!.

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Как и раньше, мы можем с помощью подстановочной модели изобразить процесс вычисления 6!, как показано на рис. 1.4.

Сравним эти два процесса. С одной стороны, они кажутся почти одинаковыми. Оба они вычисляют одну и ту же математическую функцию с одной и той же областью определения, и каждый из них для вычисления $n!$ требует количества шагов, пропорционального n. Действительно, два этих процесса даже производят одну и ту же последовательность умножений и получают одну и ту же последовательность частичных произведений. С другой стороны, когда мы рассмотрим «формы» этих двух процессов, мы увидим, что они ведут себя совершенно по-разному

Возьмем первый процесс. Подстановочная модель показывает сначала серию расширений, а затем сжатие, как показывает стрелка на рис. 1.3. Расширение происходит по мере того, как процесс строит цепочку *отложенных операций* (deferred operations), в данном случае цепочку умножений. Сжатие происходит тогда, когда выполняются эти отложенные операции. Такой тип процесса, который характеризуется цепочкой отложенных операций, называется *рекурсивным процессом* (recursive process). Выполнение этого процесса требует, чтобы интерпретатор запоминал, какие операции ему нужно выполнить впоследствии. При вычислении $n!$ длина цепочки отложенных умножений, а следовательно, и объем информации, который требуется, чтобы ее сохранить, растет линейно с ростом n (пропорционален n), как и число шагов. Такой процесс называется *линейно рекурсивным процессом* (linear recursive process).

Напротив, второй процесс не растет и не сжимается. На каждом шаге при любом значении n необходимо помнить лишь текущие значения переменных `product`, `counter` и `max-count`. Такой процесс мы называем *итеративным* (iterative process).

В общем случае, итеративный процесс — это такой процесс, состояние которого можно описать конечным числом *переменных состояния* (state variables) плюс заранее заданное правило, определяющее, как эти переменные состояния изменяются от шага к шагу, и плюс (возможно) тест на завершение, который определяет условия, при которых процесс должен закончить работу. При вычислении $n!$ число шагов линейно растет с ростом n. Такой процесс называется *линейно итеративным процессом* (linear iterative process).

Можно посмотреть на различие этих двух процессов и с другой точки зрения. В итеративном случае в каждый момент переменные программы дают полное описание состояния процесса. Если мы остановим процесс между шагами, для продолжения вычислений нам будет достаточно дать интерпретатору значения трех переменных программы. С рекурсивным процессом это не так. В этом случае имеется дополнительная «спрятанная» информация, которую хранит интерпретатор и которая не содержится в переменных программы. Она указывает, «где находится» процесс в терминах цепочки отложенных операций. Чем длиннее цепочка, тем больше информации нужно хранить[30].

Противопоставляя итерацию и рекурсию, нужно вести себя осторожно и не смешивать понятие рекурсивного *процесса* с понятием рекурсивной *процедуры*. Когда мы говорим, что процедура рекурсивна, мы имеем в виду факт синтаксиса: определение процедуры ссылается (прямо или косвенно) на саму эту процедуру. Когда же мы говорим о процессе, что он следует, скажем, линейно рекурсивной схеме, мы говорим о развитии процесса, а не о синтаксисе, с помощью которого написана процедура. Может показаться странным, например, высказывание «рекурсивная процедура `fact-iter` описывает итеративный процесс». Однако процесс действительно является итеративным: его состояние полностью описывается тремя переменными состояния, и чтобы выполнить этот процесс, интерпретатор должен хранить значение только трех переменных.

Различие между процессами и процедурами может запутывать отчасти потому, что большинство реализаций обычных языков (включая Аду, Паскаль и Си) построены так, что интерпретация любой рекурсивной процедуры поглощает объем памяти, линейно растущий пропорционально количеству вызовов процедуры, даже если описываемый ею процесс в принципе итеративен. Как следствие, эти языки способны описывать итеративные процессы только с помощью специальных«циклических конструкций» вроде `do`, `repeat`, `until`, `for` и `while`. Реализация Scheme, которую мы рассмотрим в главе 5, свободна от этого недостатка. Она будет выполнять итеративный процесс, используя фиксированный объем памяти, даже если он описывается рекурсивной процедурой. Такое свойство реализации языка называется поддержкой *хвостовой рекурсии* (tail recursion)*. Если реализация языка поддерживает хвостовую рекурсию, то итерацию можно выразить с помощью обыкновенного механизма вызова функций, так что специальные циклические конструкции имеют смысл только как синтаксический сахар[31].

---

[30] Когда в главе 5 мы будем обсуждать реализацию процедур с помощью регистровых машин, мы увидим, что итеративный процесс можно реализовать «в аппаратуре» как машину, у которой есть только конечный набор регистров и нет никакой дополнительной памяти. Напротив, для реализации рекурсивного процесса требуется машина со вспомогательной структурой данных, называемой*стек* (stack).

*Словарь `multitran.ru` дает перевод «концевая рекурсия». Наш вариант, как кажется, изящнее и сохраняет метафору, содержащуюся в англоязычном термине. — *прим. перев.*

[31] Довольно долго считалось, что хвостовая рекурсия — особый трюк в оптимизирующих компиляторах. Ясное семантическое основание хвостовой рекурсии было найдено Карлом Хьюиттом (Hewitt 1977), который выразил ее в терминах модели вычислений с помощью «передачи сообщений» (мы рассмотрим эту модель в

Упражнение 1.9.
Каждая из следующих двух процедур определяет способ сложения двух положительных целых чисел с помощью процедур `inc`, которая добавляет к своему аргументу 1, и `dec`, которая отнимает от своего аргумента 1.

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Используя подстановочную модель, проиллюстрируйте процесс, порождаемый каждой из этих процедур, вычислив `(+ 4 5)`. Являются ли эти процессы итеративными или рекурсивными?

Упражнение 1.10.
Следующая процедура вычисляет математическую функцию, называемую функцией Аккермана.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

Каковы значения следующих выражений?

```
(A 1 10)

(A 2 4)

(A 3 3)
```

Рассмотрим следующие процедуры, где `A` — процедура, определенная выше:

```
(define (f n) (A 0 n))

(define (g n) (A 1 n))

(define (h n) (A 2 n))

(define (k n) (* 5 n n))
```

Дайте краткие математические определения функций, вычисляемых процедурами `f`, `g` и `h` для положительных целых значений n. Например, `(k n)` вычисляет $5n^2$.

главе 3). Вдохновленные этим, Джеральд Джей Сассман и Гай Льюис Стил мл. (см. Steele 1975) построили интерпретатор Scheme с поддержкой хвостовой рекурсии. Позднее Стил показал, что хвостовая рекурсия является следствием естественного способа компиляции вызовов процедур (Steele 1977). Стандарт Scheme IEEE требует, чтобы все реализации Scheme поддерживали хвостовую рекурсию.

1.2.2. Древовидная рекурсия

Существует еще одна часто встречающаяся схема вычислений, называемая *древовидная рекурсия* (tree recursion). В качестве примера рассмотрим вычисление последовательности чисел Фибоначчи, в которой каждое число является суммой двух предыдущих:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Общее правило для чисел Фибоначчи можно сформулировать так:

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{если } n = 0 \\ 1 & \text{если } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{в остальных случаях} \end{cases}$$

Можно немедленно преобразовать это определение в процедуру:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Рассмотрим схему этого вычисления. Чтобы вычислить `(fib 5)`, мы сначала вычисляем `(fib 4)` и `(fib 3)`. Чтобы вычислить `(fib 4)`, мы вычисляем `(fib 3)` и `(fib 2)`. В общем, получающийся процесс похож на дерево, как показано на рис. 1.5. Заметьте, что на каждом уровне (кроме дна) ветви разделяются надвое; это отражает тот факт, что процедура `fib` при каждом вызове обращается к самой себе дважды.

Эта процедура полезна как пример прототипической древовидной рекурсии, но как метод получения чисел Фибоначчи она ужасна, поскольку производит массу излишних вычислений. Обратите внимание на рис. 1.5: все вычисление `(fib 3)` — почти половина общей работы, — повторяется дважды. В сущности, нетрудно показать, что общее число раз, которые эта процедура вызовет `(fib 1)` или `(fib 0)` (в общем, число листьев) в точности равняется $\mathrm{Fib}(n+1)$. Чтобы понять, насколько это плохо, отметим, что значение $\mathrm{Fib}(n)$ растет экспоненциально при увеличении n. Более точно (см. упражнение 1.13), $\mathrm{Fib}(n)$ — это целое число, ближайшее к $\phi^n/\sqrt{5}$, где

$$\phi = (1 + \sqrt{5})/2 \approx 1.6180$$

то есть *золотое сечение* (golden ratio), которое удовлетворяет уравнению

$$\phi^2 = \phi + 1$$

Таким образом, число шагов нашего процесса растет экспоненциально при увеличении аргумента. С другой стороны, требования к памяти растут при увеличении аргумента всего лишь линейно, поскольку в каждой точке вычисления нам требуется запоминать только те вершины, которые находятся выше нас по дереву. В общем случае число шагов, требуемых древовидно-рекурсивным процессом, будет пропорционально числу вершин дерева, а требуемый объем памяти будет пропорционален максимальной глубине дерева.

Для получения чисел Фибоначчи мы можем сформулировать итеративный процесс. Идея состоит в том, чтобы использовать пару целых a и b, которым в начале даются

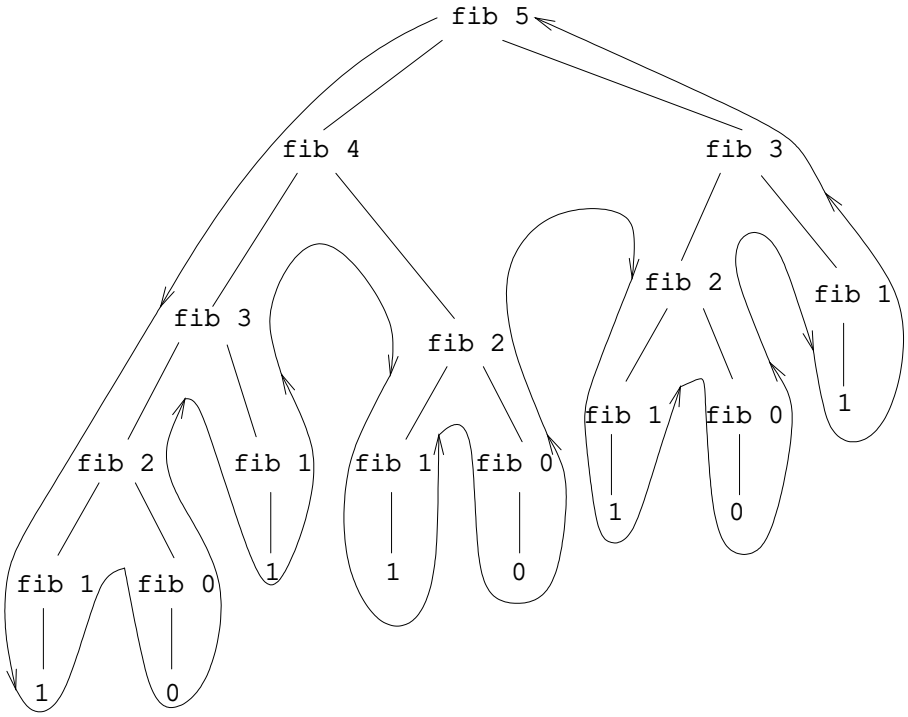


Рис. 1.5. Древовидно-рекурсивный процесс, порождаемый при вычислении `(fib 5)`.

значения $\mathrm{Fib}(1) = 1$ и $\mathrm{Fib}(0) = 0$, и на каждом шаге применять одновременную трансформацию

$$a \leftarrow a + b$$
$$b \leftarrow a$$

Нетрудно показать, что после того, как мы проделаем эту трансформацию n раз, a и b будут соответственно равны $\mathrm{Fib}(n+1)$ и $\mathrm{Fib}(n)$. Таким образом, мы можем итеративно вычислять числа Фибоначчи при помощи процедуры

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

Второй метод вычисления чисел Фибоначчи представляет собой линейную итерацию. Разница в числе шагов, требуемых двумя этими методами — один пропорционален n, другой растет так же быстро, как и само $\mathrm{Fib}(n)$, — огромна, даже для небольших значений аргумента.

Не нужно из этого делать вывод, что древовидно-рекурсивные процессы бесполезны. Когда мы будем рассматривать процессы, работающие не с числами, а с иерархически структурированными данными, мы увидим, что древовидная рекурсия является естественным и мощным инструментом[32]. Но даже при работе с числами древовидно-рекурсивные процессы могут быть полезны — они помогают нам понимать и проектировать программы. Например, хотя первая процедура `fib` и намного менее эффективна, чем вторая, зато она проще, поскольку это немногим более, чем перевод определения последовательности чисел Фибоначчи на Лисп. Чтобы сформулировать итеративный алгоритм, нам пришлось заметить, что вычисление можно перестроить в виде итерации с тремя переменными состояния.

Размен денег

Чтобы сочинить итеративный алгоритм для чисел Фибоначчи, нужно совсем немного смекалки. Теперь для контраста рассмотрим следующую задачу: сколькими способами можно разменять сумму в 1 доллар, если имеются монеты по 50, 25, 10, 5 и 1 цент? В более общем случае, можно ли написать процедуру подсчета способов размена для произвольной суммы денег?

У этой задачи есть простое решение в виде рекурсивной процедуры. Предположим, мы как-то упорядочили типы монет, которые у нас есть. В таком случае верно будет следующее уравнение:

Число способов разменять сумму a с помощью n типов монет равняется

- числу способов разменять сумму a с помощью всех типов монет, кроме первого, плюс

[32] Пример этого был упомянут в разделе 1.1.3: сам интерпретатор вычисляет выражения с помощью древовидно-рекурсивного процесса.

• число способов разменять сумму $a - d$ с использованием всех n типов монет, где d — достоинство монет первого типа.

Чтобы увидеть, что это именно так, заметим, что способы размена могут быть поделены на две группы: те, которые не используют первый тип монеты, и те, которые его используют. Следовательно, общее число способов размена какой-либо суммы равно числу способов разменять эту сумму без привлечения монет первого типа плюс число способов размена в предположении, что мы этот тип используем. Но последнее число равно числу способов размена для суммы, которая остается после того, как мы один раз употребили первый тип монеты.

Таким образом, мы можем рекурсивно свести задачу размена данной суммы к задаче размена меньших сумм с помощью меньшего количества типов монет. Внимательно рассмотрите это правило редукции и убедите себя, что мы можем использовать его для описания алгоритма, если укажем следующие вырожденные случаи[33]:

- Если a в точности равно 0, мы считаем, что имеем 1 способ размена.
- Если a меньше 0, мы считаем, что имеем 0 способов размена.
- Если n равно 0, мы считаем, что имеем 0 способов размена.

Это описание легко перевести в рекурсивную процедуру:

```
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

(Процедура `first-denomination` принимает в качестве входа число доступных типов монет и возвращает достоинство первого типа. Здесь мы упорядочили монеты от самой крупной к более мелким, но годился бы и любой другой порядок.) Теперь мы можем ответить на исходный вопрос о размене доллара:

```
(count-change 100)
292
```

[33] Рассмотрите для примера в деталях, как применяется правило редукции, если нужно разменять 10 центов на монеты в 1 и 5 центов.

`Count-change` порождает древовидно-рекурсивный процесс с избыточностью, похожей на ту, которая возникает в нашей первой реализации `fib`. (На то, чтобы получить ответ 292, уйдет заметное время.) С другой стороны, неочевидно, как построить более эффективный алгоритм для получения этого результата, и мы оставляем это в качестве задачи для желающих. Наблюдение, что древовидная рекурсия может быть весьма неэффективна, но зато ее часто легко сформулировать и понять, привело исследователей к мысли, что можно получить лучшее из двух миров, если спроектировать «умный компилятор», который мог бы трансформировать древовидно-рекурсивные процедуры в более эффективные, но вычисляющие тот же результат[34].

Упражнение 1.11.
Функция f определяется правилом: $f(n) = n$, если $n < 3$, и $f(n) = f(n-1) + f(n-2) + f(n-3)$, если $n \geq 3$. Напишите процедуру, вычисляющую f с помощью рекурсивного процесса. Напишите процедуру, вычисляющую f с помощью итеративного процесса.

Упражнение 1.12.
Приведенная ниже таблица называется *треугольником Паскаля* (Pascal's triangle).

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1 \\
 & & & & \ldots & & & &
\end{array}
$$

Все числа по краям треугольника равны 1, а каждое число внутри треугольника равно сумме двух чисел над ним[35]. Напишите процедуру, вычисляющую элементы треугольника Паскаля с помощью рекурсивного процесса.

Упражнение 1.13.
Докажите, что $\mathrm{Fib}(n)$ есть целое число, ближайшее к $\phi^n/\sqrt{5}$, где $\phi = (1+\sqrt{5})/2$. Указание: пусть $\psi = (1-\sqrt{5})/2$. С помощью определения чисел Фибоначчи (см. раздел 1.2.2) и индукции докажите, что $\mathrm{Fib}(n) = (\phi^n - \psi^n)/\sqrt{5}$.

---

[34] Один из способов избежать избыточных вычислений состоит в том, чтобы автоматически строить таблицу значений по мере того, как они вычисляются. Каждый раз, когда нужно применить процедуру к какому-нибудь аргументу, мы могли бы сначала обращаться к таблице, смотреть, не хранится ли в ней уже значение, и в этом случае мы избежали бы избыточного вычисления. Такая стратегия, называемая *табуляризацией* (tabulation) или *мемоизацией* (memoization), легко реализуется. Иногда с помощью табуляризации можно преобразовать процессы, требующие экспоненциального числа шагов (вроде `count-change`), в процессы, требования которых к времени и памяти линейно растут по мере роста ввода. См. упражнение 3.27.

[35] Элементы треугольника Паскаля называются *биномиальными коэффициентами* (binomial coefficients), поскольку n-й ряд состоит из коэффициентов термов при разложении $(x+y)^n$. Эта схема вычисления коэффициентов появилась в передовой работе Блеза Паскаля 1653 года по теории вероятностей *Traité du triangle arithmétique*. Согласно Knuth 1973, та же схема встречается в труде *Цзу-юань Юй-чэнь* («Драгоценное зеркало четырех элементов»), опубликованном китайским математиком Цзю Ши-Цзе в 1303 году, в трудах персидского поэта и математика двенадцатого века Омара Хайяма и в работах индийского математика двенадцатого века Бхаскары Ачарьи.

1.2.3. Порядки роста

Предшествующие примеры показывают, что процессы могут значительно различаться по количеству вычислительных ресурсов, которые они потребляют. Удобным способом описания этих различий является понятие *порядка роста* (order of growth), которое дает общую оценку ресурсов, необходимых процессу при увеличении его входных данных.

Пусть n — параметр, измеряющий размер задачи, и пусть $R(n)$ — количество ресурсов, необходимых процессу для решения задачи размера n. В предыдущих примерах n было числом, для которого требовалось вычислить некоторую функцию, но возможны и другие варианты. Например, если требуется вычислить приближение к квадратному корню числа, то n может быть числом цифр после запятой, которые нужно получить. В задаче умножения матриц n может быть количеством рядов в матрицах. Вообще говоря, может иметься несколько характеристик задачи, относительно которых желательно проанализировать данный процесс. Подобным образом, $R(n)$ может измерять количество используемых целочисленных регистров памяти, количество исполняемых элементарных машинных операций, и так далее. В компьютерах, которые выполняют определенное число операций за данный отрезок времени, требуемое время будет пропорционально необходимому числу элементарных машинных операций.

Мы говорим, что $R(n)$ имеет порядок роста $\Theta(f(n))$, что записывается $R(n) = \Theta(f(n))$ и произносится «тета от $f(n)$», если существуют положительные постоянные k_1 и k_2, независимые от n, такие, что

$$k_1 f(n) \leq R(n) \leq k_2 f(n)$$

для всякого достаточно большого n. (Другими словами, значение $R(n)$ заключено между $k_1 f(n)$ и $k_2 f(n)$.)

Например, для линейно рекурсивного процесса вычисления факториала, описанного в разделе 1.2.1, число шагов растет пропорционально входному значению n. Таким образом, число шагов, необходимых этому процессу, растет как $\Theta(n)$. Мы видели также, чтотребуемый объем памятирастет как $\Theta(n)$. Для итеративного факториала число шагов по-прежнему $\Theta(n)$, но объем памяти $\Theta(1)$ — то есть константа[36]. Древовидно-рекурсивное вычисление чисел Фибоначчи требует $\Theta(\phi^n)$ шагов и $\Theta(n)$ памяти, где ϕ — золотое сечение, описанное в разделе 1.2.2.

Порядки роста дают всего лишь грубое описание поведения процесса. Например, процесс, которому требуется n^2 шагов, процесс, которому требуется $1000n^2$ шагов и процесс, которому требуется $3n^2 + 10n + 17$ шагов — все имеют порядок роста $\Theta(n^2)$. С другой стороны, порядок роста показывает, какого изменения можно ожидать в поведении процесса, когда мы меняем размер задачи. Для процесса с порядком роста $\Theta(n)$ (линейного) удвоение размера задачи примерно удвоит количество используемых ресурсов. Для экспоненциального процесса каждое увеличение размера задачи на единицу будет умножать количество ресурсов на постоянный коэффициент. В оставшейся части раздела 1.2 мы рассмотрим два алгоритма, которые имеют логарифмический порядок ро-

[36] В этих утверждениях скрывается важное упрощение. Например, если мы считаем шаги процесса как «машинные операции», мы предполагаем, что число машинных операций, нужных, скажем, для вычисления произведения, не зависит от размера умножаемых чисел, а это становится неверным при достаточно больших числах. Те же замечания относятся и к оценке требуемой памяти. Подобно проектированию и описанию процесса, анализ процесса может происходить на различных уровнях абстракции.

ста, так что удвоение размера задачи увеличивает требования к ресурсам на постоянную величину.

Упражнение 1.14.
Нарисуйте дерево, иллюстрирующее процесс, который порождается процедурой `count-change` из раздела 1.2.2 при размене 11 центов. Каковы порядки роста памяти и числа шагов, используемых этим процессом при увеличении суммы, которую требуется разменять?

Упражнение 1.15.
Синус угла (заданного в радианах) можно вычислить, если воспользоваться приближением $\sin x \approx x$ при малых x и употребить тригонометрическое тождество

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

для уменьшения значения аргумента sin. (В этом упражнении мы будем считать, что угол «достаточно мал», если он не больше 0.1 радиана.) Эта идея используется в следующих процедурах:

```
(define (cube x) (* x x x))

(define (p x) (- (* 3 x) (* 4 (cube x))))

(define (sine angle)
   (if (not (> (abs angle) 0.1))
       angle
       (p (sine (/ angle 3.0)))))
```

а. Сколько раз вызывается процедура `p` при вычислении `(sine 12.15)`?

б. Каковы порядки роста в терминах количества шагов и используемой памяти (как функция a) для процесса, порождаемого процедурой `sine` при вычислении `(sine a)`?

1.2.4. Возведение в степень

Рассмотрим задачу возведения числа в степень. Нам нужна процедура, которая, приняв в качестве аргумента основание b и положительное целое значение степени n, возвращает b^n. Один из способов получить желаемое — через рекурсивное определение

$$\begin{aligned} b^n &= b \cdot b^{n-1} \\ b^0 &= 1 \end{aligned}$$

которое прямо переводится в процедуру

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

Это линейно рекурсивный процесс, требующий $\Theta(n)$ шагов и $\Theta(n)$ памяти. Подобно факториалу, мы можем немедленно сформулировать эквивалентную линейную итерацию:

```
(define (expt b n)
  (expt-iter b n 1))

(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                 (- counter 1)
                 (* b product))))
```

Эта версия требует $\Theta(n)$ шагов и $\Theta(1)$ памяти.

Можно вычислять степени за меньшее число шагов, если использовать последовательное возведение в квадрат. Например, вместо того, чтобы вычислять b^8 в виде

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

мы можем вычислить его за три умножения:

$$\begin{aligned} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{aligned}$$

Этот метод хорошо работает для степеней, которые сами являются степенями двойки. В общем случае при вычислении степеней мы можем получить преимущество от последовательного возведения в квадрат, если воспользуемся правилом

$$\begin{aligned} b^n &= (b^{n/2})^2 \quad \text{если } n \text{ четно} \\ b^n &= b \cdot b^{n-1} \quad \text{если } n \text{ нечетно} \end{aligned}$$

Этот метод можно выразить в виде процедуры

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

где предикат, проверяющий целое число на четность, определен через элементарную процедуру `remainder`:

```
(define (even? n)
  (= (remainder n 2) 0))
```

Процесс, вычисляющий `fast-expt`, растет логарифмически как по используемой памяти, так и по количеству шагов. Чтобы увидеть это, заметим, что вычисление b^{2n} с помощью этого алгоритма требует всего на одно умножение больше, чем вычисление b^n. Следовательно, размер степени, которую мы можем вычислять, возрастает примерно вдвое с каждым следующим умножением, которое нам разрешено делать. Таким образом, число умножений, требуемых для вычисления степени n, растет приблизительно так же быстро, как логарифм n по основанию 2. Процесс имеет степень роста $\Theta(\log(n))$[37].

[37] Точнее, количество требуемых умножений равно логарифму n по основанию 2 минус 1 и плюс количество единиц в двоичном представлении n. Это число всегда меньше, чем удвоенный логарифм n по основанию 2. Произвольные константы k_1 и k_2 в определении порядка роста означают, что для логарифмического процесса основание, по которому берется логарифм, не имеет значения, так что все такие процессы описываются как $\Theta(\log(n))$.

Если n велико, разница между порядком роста $\Theta(\log(n))$ и $\Theta(n)$ оказывается очень заметной. Например, `fast-expt` при $n = 1000$ требует всего 14 умножений[38]. С помощью идеи последовательного возведения в квадрат можно построить также итеративный алгоритм, который вычисляет степени за логарифмическое число шагов (см. упражнение 1.16), хотя, как это часто бывает с итеративными алгоритмами, его нельзя записать так же просто, как рекурсивный алгоритм[39].

Упражнение 1.16.
Напишите процедуру, которая развивается в виде итеративного процесса и реализует возведение в степень за логарифмическое число шагов, как `fast-expt`. (Указание: используя наблюдение, что $(b^{n/2})^2 = (b^2)^{n/2}$, храните, помимо значения степени n и основания b, дополнительную переменную состояния a, и определите переход между состояниями так, чтобы произведение ab^n от шага к шагу не менялось. Вначале значение a берется равным 1, а ответ получается как значение a в момент окончания процесса. В общем случае метод определения *инварианта* (invariant quantity), который не изменяется при переходе между шагами, является мощным способом размышления о построении итеративных алгоритмов.)

Упражнение 1.17.
Алгоритмы возведения в степень из этого раздела основаны на повторяющемся умножении. Подобным же образом можно производить умножение с помощью повторяющегося сложения. Следующая процедура умножения (в которой предполагается, что наш язык способен только складывать, но не умножать) аналогична процедуре `expt`:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

Этот алгоритм затрачивает количество шагов, линейно пропорциональное `b`. Предположим теперь, что, наряду со сложением, у нас есть операции `double`, которая удваивает целое число, и `halve`, которая делит (четное) число на 2. Используя их, напишите процедуру, аналогичную `fast-expt`, которая затрачивает логарифмическое число шагов.

Упражнение 1.18.
Используя результаты упражнений 1.16 и 1.17, разработайте процедуру, которая порождает итеративный процесс для умножения двух чисел с помощью сложения, удвоения и деления пополам, и затрачивает логарифмическое число шагов[40].

Упражнение 1.19.
Существует хитрый алгоритм получения чисел Фибоначчи за логарифмическое число шагов. Вспомните трансформацию переменных состояния a и b процесса `fib-iter` из раздела 1.2.2:

---

[38] Если Вас интересует, зачем это кому-нибудь может понадобиться возводить числа в 1000-ю степень, смотрите раздел 1.2.6.

[39] Итеративный алгоритм очень стар. Он встречается в *Чанда-сутре* Ачарьи Пингалы, написанной до 200 года до н.э. В Knuth 1981, раздел 4.6.3, содержится полное обсуждение и анализ этого и других методов возведения в степень.

[40] Этот алгоритм, который иногда называют «методом русского крестьянина», очень стар. Примеры его использования найдены в Риндском папирусе, одном из двух самых древних существующих математических документов, который был записан (и при этом скопирован с еще более древнего документа) египетским писцом по имени А'х-мосе около 1700 г. до н.э.

$a \leftarrow a + b$ и $b \leftarrow a$. Назовем эту трансформацию T и заметим, что n-кратное применение T, начиная с 1 и 0, дает нам пару $\mathrm{Fib}(n+1)$ и $\mathrm{Fib}(n)$. Другими словами, числа Фибоначчи получаются путем применения T^n, n-ой степени трансформации T, к паре (1,0). Теперь рассмотрим T как частный случай $p = 0, q = 1$ в семействе трансформаций T_{pq}, где T_{pq} преобразует пару (a, b) по правилу $a \leftarrow bq + aq + ap, b \leftarrow bp + aq$. Покажите, что двукратное применение трансформации T_{pq} равносильно однократному применению трансформации $T_{p'q'}$ того же типа, и вычислите p' и q' через p и q. Это дает нам прямой способ возводить такие трансформации в квадрат, и таким образом, мы можем вычислить T^n с помощью последовательного возведения в квадрат, как в процедуре `fast-expt`. Используя все эти идеи, завершите следующую процедуру, которая дает результат за логарифмическое число шагов[41]:

```
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   ⟨??⟩  ; вычислить p'
                   ⟨??⟩  ; вычислить q'
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.2.5. Нахождение наибольшего общего делителя

По определению, наибольший общий делитель (НОД) двух целых чисел a и b — это наибольшее целое число, на которое и a, и b делятся без остатка. Например, НОД 16 и 28 равен 4. В главе 2, когда мы будем исследовать реализацию арифметики на рациональных числах, нам потребуется вычислять НОДы, чтобы сокращать дроби. (Чтобы сократить дробь, нужно поделить ее числитель и знаменатель на их НОД. Например, 16/28 сокращается до 4/7.) Один из способов найти НОД двух чисел состоит в том, чтобы разбить каждое из них на простые множители и найти среди них общие, однако существует знаменитый и значительно более эффективный алгоритм.

Этот алгоритм основан на том, что если r есть остаток от деления a на b, то общие делители a и b в точности те же, что и общие делители b и r. Таким образом, можно воспользоваться уравнением

$$\text{НОД}(a, b) = \text{НОД}(b, r)$$

чтобы последовательно свести задачу нахождения НОД к задаче нахождения НОД все

[41] Это упражнение нам предложил Джо Стойна основе примера из Kaldewaij 1990.

меньших и меньших пар целых чисел. Например,

$$\begin{aligned}\text{НОД}(206,40) &= \text{НОД}(40,6)\\ &= \text{НОД}(6,4)\\ &= \text{НОД}(4,2)\\ &= \text{НОД}(2,0)\\ &= 2\end{aligned}$$

сводит НОД$(206,40)$ к НОД$(2,0)$, что равняется двум. Можно показать, что если начать с произвольных двух целых чисел и производить последовательные редукции, в конце концов всегда получится пара, где вторым элементом будет 0. Этот способ нахождения НОД известен как *алгоритм Евклида* (Euclid's Algorithm)[42].

Алгоритм Евклида легко выразить в виде процедуры:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Она порождает итеративный процесс, число шагов которого растет пропорционально логарифму чисел-аргументов.

Тот факт, что число шагов, затрачиваемых алгоритмом Евклида, растет логарифмически, интересным образом связан с числами Фибоначчи:

> **Теорема Ламэ:**
>
> Если алгоритму Евклида требуется k шагов для вычисления НОД некоторой пары чисел, то меньший из членов этой пары больше или равен k-тому числу Фибоначчи[43].

С помощью этой теоремы можно оценить порядок роста алгоритма Евклида. Пусть n будет меньшим из двух аргументов процедуры. Если процесс завершается за k шагов,

[42] Алгоритм Евклида называется так потому, что он встречается в *Началах* Евклида (книга 7, ок. 300 г. до н.э.). По утверждению Кнута (Knuth 1973), его можно считать самым старым из известных нетривиальных алгоритмов. Древнеегипетский метод умножения (упражнение 1.18), разумеется, древнее, но, как объясняет Кнут, алгоритм Евклида — самый старый алгоритм, представленный в виде общей процедуры, а не через набор иллюстрирующих примеров.

[43] Эту теорему доказал в 1845 году Габриэль Ламэ, французский математик и инженер, который больше всего известен своим вкладом в математическую физику. Чтобы доказать теорему, рассмотрим пары (a_k, b_k), где $a_k \geq b_k$ и алгоритм Евклида завершается за k шагов. Доказательство основывается на утверждении, что если $(a_{k+1}, b_{k+1}) \to (a_k, b_k) \to (a_{k-1}, b_{k-1})$ — три последовательные пары в процессе редукции, то $b_{k+1} \geq b_k + b_{k-1}$. Чтобы доказать это утверждение, вспомним, что шаг редукции определяется применением трансформации $a_{k-1} = b_k, b_{k-1} =$ остаток от деления a_k на b_k. Второе из этих уравнений означает, что $a_k = qb_k + b_{k-1}$ для некоторого положительного числа q. Поскольку q должно быть не меньше 1, имеем $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$. Но из предыдущего шага редукции мы имеем $b_{k+1} = a_k$. Таким образом, $b_{k+1} = a_k \geq b_k + b_{k-1}$. Промежуточное утверждение доказано. Теперь можно доказать теорему индукцией по k, то есть числу шагов, которые требуются алгоритму для завершения. Утверждение теоремы верно при $k = 1$, поскольку при этом требуется всего лишь чтобы b было не меньше, чем $\text{Fib}(1) = 1$. Теперь предположим, что утверждение верно для всех чисел, меньших или равных k, и докажем его для $k+1$. Пусть $(a_{k+1}, b_{k+1}) \to (a_k, b_k) \to (a_{k-1}, b_{k-1})$ — последовательные пары в процессе редукции. Согласно гипотезе индукции, $b_{k-1} \geq \text{Fib}(k-1), b_k \geq \text{Fib}(k)$. Таким образом, применение промежуточного утверждения совместно с определением чисел Фибоначчи дает $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k-1) = \text{Fib}(k+1)$, что и доказывает теорему Ламэ.

должно выполняться $n \geq \mathrm{Fib}(k) \approx \phi^k/\sqrt{5}$. Следовательно, число шагов k растет как логарифм n (по основанию ϕ). Следовательно, порядок роста равен $\Theta(\log n)$.

Упражнение 1.20.
Процесс, порождаемый процедурой, разумеется, зависит от того, по каким правилам работает интерпретатор. В качестве примера рассмотрим итеративную процедуру `gcd`, приведенную выше. Предположим, что мы вычисляем эту процедуру с помощью нормального порядка, описанного в разделе 1.1.5. (Правило нормального порядка вычислений для `if` описано в упражнении 1.5.) Используя подстановочную модель для нормального порядка, проиллюстрируйте процесс, порождаемый при вычислении `(gcd 206 40)` и укажите, какие операции вычисления остатка действительно выполняются. Сколько операций `remainder` выполняется на самом деле при вычислении `(gcd 206 40)` в нормальном порядке? При вычислении в аппликативном порядке?

1.2.6. Пример: проверка на простоту

В этом разделе описываются два метода проверки числа n на простоту, один с порядком роста $\Theta(\sqrt{n})$, и другой, «вероятностный», алгоритм с порядком роста $\Theta(\log n)$. В упражнениях, приводимых в конце раздела, предлагаются программные проекты на основе этих алгоритмов.

Поиск делителей

С древних времен математиков завораживали проблемы, связанные с простыми числами, и многие люди занимались поисками способов выяснить, является ли число простым. Один из способов проверки числа на простоту состоит в том, чтобы найти делители числа. Следующая программа находит наименьший целый делитель (больший 1) числа n. Она проделывает это «в лоб», путем проверки делимости n на все последовательные числа, начиная с 2.

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

Мы можем проверить, является ли число простым, следующим образом: n простое тогда и только тогда, когда n само является своим наименьшим делителем.

```
(define (prime? n)
  (= n (smallest-divisor n)))
```

Тест на завершение основан на том, что если число n не простое, у него должен быть делитель, меньше или равный $\sqrt{n}$[44]. Это означает, что алгоритм может проверять

[44] Если d — делитель n, то n/d тоже. Но d и n/d не могут оба быть больше $\sqrt{n}$.

делители только от 1 до $\sqrt{n}$. Следовательно, число шагов, которые требуются, чтобы определить, что n простое, будет иметь порядок роста $\Theta(\sqrt{n})$.

Тест Ферма

Тест на простоту с порядком роста $\Theta(\log n)$ основан на утверждении из теории чисел, известном как Малая теорема Ферма[45].

> **Малая теорема Ферма:**
>
> Если n — простое число, а a — произвольное целое число меньше, чем n, то a, возведенное в n-ю степень, равно a по модулю n.

(Говорят, что два числа *равны по модулю* n (congruent modulo n), если они дают одинаковый остаток при делении на n. Остаток от деления числа a на n называется также *остатком* a *по модулю* n (remainder of a modulo n) или просто a *по модулю* n.)

Если n не является простым, то, вообще говоря, большинство чисел $a < n$ не будут удовлетворять этому условию. Это приводит к следующему алгоритму проверки на простоту:имея число n, случайным образом выбрать число $a < n$ и вычислить остаток от a^n по модулю n. Если этот остаток не равен a, то n определенно не является простым. Если он равен a, то мы имеем хорошие шансы, что n простое. Тогда нужно взять еще одно случайное a и проверить его тем же способом. Если и оно удовлетворяет уравнению, мы можем быть еще более уверены, что n простое. Испытывая все большее количество a, мы можем увеличивать нашу уверенность в результате. Этот алгоритм называется тестом Ферма.

Для реализации теста Ферма нам нужна процедура, которая вычисляет степень числа по модулю другого числа:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))
```

Эта процедура очень похожа на `fast-expt` из раздела 1.2.4. Она использует последовательное возведение в квадрат, так что число шагов логарифмически растет с увеличением степени[46].

---

[45] Пьер де Ферма (1601-1665) считается основателем современной теории чисел. Он доказал множество важных теорем, однако, как правило, он объявлял только результаты, не публикуя своих доказательств. Малая теорема Ферма была сформулирована в письме, которое он написал в 1640-м году. Первое опубликованное доказательство было даноЭйлером в 1736 г. (более раннее, идентичное доказательство было найдено в неопубликованных рукописях Лейбница). Самый знаменитый результат Ферма, известный как Большая теорема Ферма, был записан в 1637 году в его экземпляре книги *Арифметика* (греческого математика третьего века Диофанта) с пометкой «я нашел подлинно удивительное доказательство, но эти поля слишком малы, чтобы вместить его». Доказательство Большой теоремы Ферма стало одним из самых известных вопросов теории чисел. Полное решение было найдено в 1995 году Эндрю Уайлсом из Принстонского университета.

[46] Шаги редукции для случаев, когда степень больше 1, основаны на том, что для любых целых чисел x, y и m мы можем найти остаток от деления произведения x и y на m путем отдельного вычисления остатков

Тест Ферма производится путем случайного выбора числа a между 1 и $n-1$ включительно и проверки, равен ли a остаток по модулю n от n-ой степени a. Случайное число a выбирается с помощью процедуры `random`, про которую мы предполагаем, что она встроена в Scheme в качестве элементарной процедуры. `Random` возвращает неотрицательное число, меньшее, чем ее целый аргумент. Следовательно, чтобы получить случайное число между 1 и $n-1$, мы вызываем `random` с аргументом $n-1$ и добавляем к результату 1:

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

Следующая процедура прогоняет тест заданное число раз, как указано ее параметром. Ее значение истинно, если тест всегда проходит, и ложно в противном случае.

```
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

Вероятностные методы

Тест Ферма отличается по своему характеру от большинства известных алгоритмов, где вычисляется результат, истинность которого гарантирована. Здесь полученный результат верен лишь с какой-то вероятностью. Более точно, если n не проходит тест Ферма, мы можем точно сказать, что оно не простое. Но то, что n проходит тест, хотя и является очень сильным показателем, все же не гарантирует, что n простое. Нам хотелось бы сказать, что для любого числа n, если мы проведем тест достаточное количество раз и n каждый раз его пройдет, то вероятность ошибки в нашем тесте на простоту может быть сделана настолько малой, насколько мы того пожелаем.

К сожалению, это утверждение неверно. Существуют числа, которые «обманывают» тест Ферма: числа, которые не являются простыми и тем не менее обладают свойством, что для всех целых чисел $a < n$ a^n равно a по модулю n. Такие числа очень редки, так что на практике тест Ферма вполне надежен[47]. Существуют варианты теста Ферма, которые обмануть невозможно. В таких тестах, подобно методу Ферма, проверка числа n на простоту ведется путем выбора случайного числа $a < n$ и проверки некоторого условия, зависящего от n и a. (Пример такого теста см. в упражнении 1.28.) С другой

x по модулю m, y по модулю m, перемножения их, и взятия остатка по модулю m от результата. Например, в случае, когда e четно, мы можем вычислить остаток $b^{e/2}$ по модулю m, возвести его в квадрат и взять остаток по модулю m. Такой метод полезен потому, что с его помощью мы можем производить вычисления, не используя чисел, намного больших, чем m. (Сравните с упражнением 1.25.)

[47] Числа, «обманывающие» тест Ферма, называются *числами Кармайкла* (Carmichael numbers), и про них почти ничего неизвестно, кроме того, что они очень редки. Существует 255 чисел Кармайкла, меньших 100 000 000. Несколько первых — 561, 1105, 1729, 2465, 2821 и 6601. При проверке на простоту больших чисел, выбранных случайным образом, шанс наткнуться на число, «обманывающее» тест Ферма, меньше, чем шанс, что космическое излучение заставит компьютер сделать ошибку при вычислении «правильного» алгоритма. То, что по первой из этих причин алгоритм считается неадекватным, а по второй нет, показывает разницу между математикой и техникой.

стороны, в отличие от теста Ферма, можно доказать, что для любого n условие не выполняется для большинства чисел $a < n$, если n не простое. Таким образом, если n проходит тест для какого-то случайного a, шансы, что n простое, уже больше половины. Если n проходит тест для двух случайных a, шансы, что n простое, больше, чем 3 из 4. Проводя тест с большим количеством случайных чисел, мы можем сделать вероятность ошибки сколь угодно малой.

Существование тестов, для которых можно доказать, что вероятность ошибки можно сделать сколь угодно малой, вызвало большой интерес к алгоритмам такого типа. Их стали называть *вероятностными алгоритмами* (probabilistic alorithms). В этой области ведутся активные исследования, и вероятностные алгоритмы удалось с успехом применить во многих областях[48].

Упражнение 1.21.
С помощью процедуры `smallest-divisor` найдите наименьший делитель следующих чисел: 199, 1999, 19999.

Упражнение 1.22.
Бóльшая часть реализаций Лиспа содержат элементарную процедуру `runtime`, которая возвращает целое число, показывающее, как долго работала система (например, в миллисекундах). Следующая процедура `timed-prime-test`, будучи вызвана с целым числом n, печатает n и проверяет, простое ли оно. Если n простое, процедура печатает три звездочки и количество времени, затраченное на проверку.

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))

(define (report-prime elapsed-time)
  (display " *** ")
  (display elapsed-time))
```

Используя эту процедуру, напишите процедуру `search-for-primes`, которая проверяет на простоту все нечетные числа в заданном диапазоне. С помощью этой процедуры найдите наименьшие три простых числа после 1000; после 10 000; после 100 000; после 1 000 000. Посмотрите, сколько времени затрачивается на каждое простое число. Поскольку алгоритм проверки имеет порядок роста $\Theta(\sqrt{n})$, Вам следовало бы ожидать, что проверка на простоту чисел, близких к 10 000,

[48]Одно из наиболее впечатляющих применений вероятностные алгоритмы получили в области криптографии. Хотя в настоящее время вычислительных ресурсов недостаточно, чтобы разложить на множители произвольное число из 200 цифр, с помощью теста Ферма проверить, является ли оно простым, можно за несколько секунд. Этот факт служит основой предложенного в Rivest, Shamir, and Adleman 1977 метода построения шифров, которые «невозможно» взломать. Полученный *алгоритм RSA* (RSA algorithm) стал широко используемым методом повышения секретности электронных средств связи. В результате этого и других связанных событий исследование простых чисел, которое раньше считалось образцом «чистой» математики, изучаемым исключительно ради самого себя, теперь получило важные практические приложения в таких областях, как криптография, электронная передача денежных сумм и хранение информации.

занимает в $\sqrt{10}$ раз больше времени, чем для чисел, близких к 1000. Подтверждают ли это Ваши замеры времени? Хорошо ли поддерживают предсказание $\sqrt{n}$ данные для 100 000 и 1 000 000? Совместим ли Ваш результат с предположением, что программы на Вашей машине затрачивают на выполнение задач время, пропорциональное числу шагов?

Упражнение 1.23.
Процедура `smallest-divisor` в начале этого раздела проводит множество лишних проверок: после того, как она проверяет, делится ли число на 2, нет никакого смысла проверять делимость на другие четные числа. Таким образом, вместо последовательности 2, 3, 4, 5, 6 . . . , используемой для `test-divisor`, было бы лучше использовать 2, 3, 5, 7, 9 . . . . Чтобы реализовать такое улучшение, напишите процедуру `next`, которая имеет результатом 3, если получает 2 как аргумент, а иначе возвращает свой аргумент плюс 2. Используйте `(next test-divisor)` вместо `(+ test-divisor 1)` в `smallest-divisor`. Используя процедуру `timed-prime-test` с модифицированной версией `smallest-divisor`, запустите тест для каждого из 12 простых чисел, найденных в упражнении 1.22. Поскольку эта модификация снижает количество шагов проверки вдвое, Вы должны ожидать двукратного ускорения проверки. Подтверждаются ли эти ожидания? Если нет, то каково наблюдаемое соотношение скоростей двух алгоритмов, и как Вы объясните то, что оно отличается от 2?

Упражнение 1.24.
Модифицируйте процедуру `timed-prime-test` из упражнения 1.22 так, чтобы она использовала `fast-prime?` (метод Ферма) и проверьте каждое из 12 простых чисел, найденных в этом упражнении. Исходя из того, что у теста Ферма порядок роста $\Theta(\log n)$, то какого соотношения времени Вы бы ожидали между проверкой на простоту поблизости от 1 000 000 и поблизости от 1000? Подтверждают ли это Ваши данные? Можете ли Вы объяснить наблюдаемое несоответствие, если оно есть?

Упражнение 1.25.
Лиза П. Хакер жалуется, что при написании `expmod` мы делаем много лишней работы. В конце концов, говорит она, раз мы уже знаем, как вычислять степени, можно просто написать

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Права ли она? Стала бы эта процедура столь же хорошо работать при проверке простых чисел? Объясните.

Упражнение 1.26.
У Хьюго Дума большие трудности в упражнении 1.24. Процедура `fast-prime?` у него работает медленнее, чем `prime?`. Хьюго просит помощи у своей знакомой Евы Лу Атор. Вместе изучая код Хьюго, они обнаруживают, что тот переписал процедуру `expmod` с явным использованием умножения вместо того, чтобы вызывать `square`:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                       (expmod base (/ exp 2) m))
                    m))
```

```
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))
```

Хьюго говорит: «Я не вижу здесь никакой разницы». «Зато я вижу, — отвечает Ева. — Переписав процедуру таким образом, ты превратил процесс порядка $\Theta(\log n)$ в процесс порядка $\Theta(n)$». Объясните.

Упражнение 1.27.
Покажите, что числа Кармайкла, перечисленные в сноске 47, действительно «обманывают» тест Ферма: напишите процедуру, которая берет целое число n и проверяет, правда ли a^n равняется a по модулю n для всех $a < n$, и проверьте эту процедуру на этих числах Кармайкла.

Упражнение 1.28.
Один из вариантов теста Ферма, который невозможно обмануть, называется *тест Миллера–Рабина* (Miller-Rabin test) (Miller 1976; Rabin 1980). Он основан наальтернативной формулировке Малой теоремы Ферма, которая состоит в том, что если n — простое число, а a — произвольное положительное целое число, меньшее n, то a в $n-1$-ой степени равняется 1 по модулю n. Проверяя простоту числа n методом Миллера–Рабина, мы берем случайное число $a < n$ и возводим его в $(n-1)$-ю степень по модулю n с помощью процедуры `expmod`. Однако когда в процедуре `expmod` мы проводим возведение в квадрат, мы проверяем, не нашли ли мы «нетривиальный квадратный корень из 1 по модулю n», то есть число, не равное 1 или $n-1$, квадрат которого по модулю n равен 1. Можно доказать, что если такой нетривиальный квадратный корень из 1 существует, то n не простое число. Можно, кроме того, доказать, что если n — нечетное число, не являющееся простым, то по крайней мере для половины чисел $a < n$ вычисление a^{n-1} с помощью такой процедуры обнаружит нетривиальный квадратный корень из 1 по модулю n (вот почему тест Миллера–Рабина невозможно обмануть). Модифицируйте процедуру `expmod` так, чтобы она сигнализировала обнаружение нетривиального квадратного корня из 1, и используйте ее для реализации теста Миллера–Рабина с помощью процедуры, аналогичной `fermat-test`. Проверьте свою процедуру на нескольких известных Вам простых и составных числах. Подсказка: удобный способ заставить `expmod` подавать особый сигнал — заставить ее возвращать 0.

1.3. Формулирование абстракций с помощью процедур высших порядков

Мы видели, что процедуры, в сущности, являются абстракциями, которые описывают составные операции над числами безотносительно к конкретным числам. Например, когда мы определяем

```
(define (cube x) (* x x x))
```

мы говорим не о кубе какого-то конкретного числа, а о способе получить куб любого числа. Разумеется, мы могли бы обойтись без определения этой процедуры, каждый раз писать выражения вроде

```
(* 3 3 3)
(* x x x)
(* y y y)
```

и никогда явно не упоминать понятие куба. Это поставило бы нас перед серьезным затруднением и заставило бы работать только в терминах тех операций, которые оказались примитивами языка (в данном случае, в терминах умножения), а не в терминах операций более высокого уровня. Наши программы были бы способны вычислять кубы, однако в нашем языке не было бы возможности выразить идею возведения в куб. Одна из тех вещей, которых мы должны требовать от мощного языка программирования — это возможность строить абстракции путем присвоения имен общим схемам, а затем прямо работать с этими абстракциями. Процедуры дают нам такую возможность. Вот почему все языки программирования, кроме самых примитивных, обладают механизмами определения процедур.

Но даже при обработке численных данных наши возможности создавать абстракции окажутся сильно ограниченными, если мы сможем определять только процедуры, параметры которых должны быть числами. Часто одна и та же схема программы используется с различными процедурами. Для того чтобы выразить эти схемы как понятия, нам нужно строить процедуры, которые принимают другие процедуры как аргументы либо возвращают их как значения. Процедура, манипулирующая другими процедурами, называется *процедурой высшего порядка* (higher-order procedure). В этом разделе показывается, как процедуры высших порядков могут служить в качестве мощного механизма абстракции, резко повышая выразительную силу нашего языка.

1.3.1. Процедуры в качестве аргументов

Рассмотрим следующие три процедуры. Первая из них вычисляет сумму целых чисел от `a` до `b`:

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

Вторая вычисляет сумму кубов целых чисел в заданном диапазоне:

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

Третья вычисляет сумму последовательности термов в ряде

$$\frac{1}{1\cdot 3}+\frac{1}{5\cdot 7}+\frac{1}{9\cdot 11}+\ldots$$

который (очень медленно) сходится к $\pi/8$[49].

[49] Этим рядом, который обычно записывают в эквивалентной форме $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots$, мы обязаны Лейбницу. В разделе 3.5.3 мы увидим, как использовать его как основу для некоторых изощренных вычислительных трюков.

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

Ясно, что за этими процедурами стоит одна общая схема. Большей частью они идентичны и различаются только именем процедуры, функцией, которая вычисляет терм, подлежащий добавлению, и функцией, вычисляющей следующее значение `a`. Все эти процедуры можно породить, заполнив дырки в одном шаблоне:

```
(define (⟨имя⟩ a b)
  (if (> a b)
      0
      (+ (⟨терм⟩ a)
         (⟨имя⟩ (⟨следующий⟩ a) b))))
```

Присутствие такого общего шаблона является веским доводом в пользу того, что здесь скрыта полезная абстракция, которую только надо вытащить на поверхность. Действительно, математики давно выделили абстракцию *суммирования последовательности* (summation of a series) и изобрели «сигма-запись», например

$$\sum_{n=a}^{b} f(n) = f(a) + \ldots + f(b)$$

чтобы выразить это понятие. Сила сигма-записи состоит в том, что она позволяет математикам работать с самим понятием суммы, а не просто с конкретными суммами — например, формулировать общие утверждения о суммах, независимые от конкретных суммируемых последовательностей.

Подобным образом, мы как проектировщики программ хотели бы, чтобы наш язык был достаточно мощным и позволял написать процедуру, которая выражала бы само понятие суммы, а не только процедуры, вычисляющие конкретные суммы. В нашем процедурном языке мы можем без труда это сделать, взяв приведенный выше шаблон и преобразовав «дырки» в формальные параметры:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

Заметьте, что `sum` принимает в качестве аргументов как нижнюю и верхнюю границы `a` и `b`, так и процедуры `term` и `next`. `Sum` можно использовать так, как мы использовали бы любую другую процедуру. Например, с ее помощью (вместе с процедурой `inc`, которая увеличивает свой аргумент на 1), мы можем определить `sum-cubes`:

```
(define (inc n) (+ n 1))

(define (sum-cubes a b)
  (sum cube a inc b))
```

Воспользовавшись этим определением, мы можем вычислить сумму кубов чисел от 1 до 10:

```
(sum-cubes 1 10)
3025
```

С помощью процедуры идентичности (которая просто возвращает свой аргумент) для вычисления терма, мы можем определить `sum-integers` через `sum`:

```
(define (identity x) x)

(define (sum-integers a b)
  (sum identity a inc b))
```

Теперь можно сложить целые числа от 1 до 10:

```
(sum-integers 1 10)
55
```

Таким же образом определяется `pi-sum`[50]:

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

С помощью этих процедур мы можем вычислить приближение к π:

```
(* 8 (pi-sum 1 1000))
3.139592655589783
```

Теперь, когда у нас есть `sum`, ее можно использовать в качестве строительного блока при формулировании других понятий. Например, определенный интеграл функции f между пределами a и b можно численно оценить с помощью формулы

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \ldots\right] dx$$

для малых значений dx. Мы можем прямо выразить это в виде процедуры:

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2)) add-dx b)
     dx))
```

[50]Обратите внимание, что мы использовали блочную структуру (раздел 1.1.8), чтобы спрятать определения `pi-next` и `pi-term` внутри `pi-sum`, поскольку вряд ли эти процедуры понадобятся зачем-либо еще. В разделе 1.3.2 мы совсем от них избавимся.

```
(integral cube 0 1 0.01)
.24998750000000042

(integral cube 0 1 0.001)
.249999875000001
```

(Точное значение интеграла `cube` от 0 до 1 равно 1/4.)

Упражнение 1.29.
Правило Симпсона — более точный метод численного интегрирования, чем представленный выше. С помощью правила Симпсона интеграл функции f между a и b приближенно вычисляется в виде

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \ldots + 2y_{n-2} + 4y_{n-1} + y_n]$$

где $h = (b - a)/n$, для какого-то четного целого числа n, а $y_k = f(a + kh)$. (Увеличение n повышает точность приближенного вычисления.) Определите процедуру, которая принимает в качестве аргументов f, a, b и n, и возвращает значение интеграла, вычисленное по правилу Симпсона. С помощью этой процедуры проинтегрируйте `cube` между 0 и 1 (с $n = 100$ и $n = 1000$) и сравните результаты с процедурой `integral`, приведенной выше.

Упражнение 1.30.
Процедура `sum` порождает линейную рекурсию. Ее можно переписать так, чтобы суммирование выполнялось итеративно. Покажите, как сделать это, заполнив пропущенные выражения в следующем определении:

```
(define (sum term a next b)
  (define (iter a result)
     (if ⟨??⟩
         ⟨??⟩
         (iter ⟨??⟩ ⟨??⟩))))
  (iter ⟨??⟩ ⟨??⟩))
```

Упражнение 1.31.
а. Процедура `sum` — всего лишь простейшая из обширного множества подобных абстракций, которые можно выразить через процедуры высших порядков.[51]. Напишите аналогичную процедуру под названием `product`, которая вычисляет произведение значений функции в точках на указанном интервале. Покажите, как с помощью этой процедуры определить `factorial`. Кроме того, при помощи `product` вычислите приближенное значение π по формуле[52]

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

[51]Смысл упражнений 1.31–1.33 состоит в том, чтобы продемонстрировать выразительную мощь, получаемую, когда с помощью подходящей абстракции обобщается множество операций, казалось бы, не связанных между собой. Однако, хотя накопление и фильтрация — изящные приемы, при их использовании руки у нас пока что несколько связаны, поскольку пока что у нас нет структур данных, которые дают подходящие к этим абстракциям средства комбинирования. В разделе 2.2.3 мы вернемся к этим приемам и покажем, как использовать *последовательности* (sequences) в качестве интерфейсов для комбинирования фильтров и накопителей, так что получаются еще более мощные абстракции. Мы увидим, как эти методы сами по себе становятся мощным и изящным подходом к проектированию программ.

[52]Эту формулу открыл английский математик семнадцатого века Джон Уоллис.

б. Если Ваша процедура `product` порождает рекурсивный процесс, перепишите ее так, чтобы она порождала итеративный. Если она порождает итеративный процесс, перепишите ее так, чтобы она порождала рекурсивный.

Упражнение 1.32.
а. Покажите, что `sum` и `product` (упражнение 1.31) являются частными случаями еще более общего понятия, называемого *накопление* (accumulation), которое комбинирует множество термов с помощью некоторой общей функции накопления

```
(accumulate combiner null-value term a next b)
```

`Accumulate` принимает в качестве аргументов те же описания термов и диапазона, что и `sum` с `product`, а еще процедуру `combiner` (двух аргументов), которая указывает, как нужно присоединить текущий терм к результату накопления предыдущих, и `null-value`, базовое значение, которое нужно использовать, когда термы закончатся. Напишите `accumulate` и покажите, как и `sum`, и `product` можно определить в виде простых вызовов `accumulate`.

б. Если Ваша процедура `accumulate` порождает рекурсивный процесс, перепишите ее так, чтобы она порождала итеративный. Если она порождает итеративный процесс, перепишите ее так, чтобы она порождала рекурсивный.

Упражнение 1.33.
Можно получить еще более общую версию `accumulate` (упражнение 1.32), если ввести понятие *фильтра* (filter) на комбинируемые термы. То есть комбинировать только те термы, порожденные из значений диапазона, которые удовлетворяют указанному условию. Получающаяся абстракция `filtered-accumulate` получает те же аргументы, что и `accumulate`, плюс дополнительный одноаргументный предикат, который определяет фильтр. Запишите `filtered-accumulate` в виде процедуры. Покажите, как с помощью `filtered-accumulate` выразить следующее:
а. сумму квадратов простых чисел в интервале от `a` до `b` (в предположении, что процедура `prime?` уже написана);

б. произведение всех положительных целых чисел меньше n, которые просты по отношению к n (то есть всех таких положительных целых чисел $i < n$, что НОД$(i, n) = 1$).

1.3.2. Построение процедур с помощью `lambda`

Когда в разделе 1.3.1 мы использовали `sum`, очень неудобно было определять тривиальные процедуры вроде `pi-term` и `pi-next` только ради того, чтобы передать их как аргументы в процедуры высшего порядка. Было бы проще вместо того, чтобы вводить имена `pi-next` и `pi-term`, прямо определить «процедуру, которая возвращает свой аргумент плюс 4» и «процедуру, которая вычисляет число, обратное произведению аргумента и аргумента плюс 2». Это можно сделать, введя особую форму `lambda`, которая создает процедуры. С использованием `lambda` мы можем записать требуемое в таком виде:

```
(lambda (x) (+ x 4))
```

и

```
(lambda (x) (/ 1.0 (* x (+ x 2))))
```

Тогда нашу процедуру `pi-sum` можно выразить безо всяких вспомогательных процедур:

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
```

Еще с помощью `lambda` мы можем записать процедуру `integral`, не определяя вспомогательную процедуру `add-dx`:

```
(define (integral f a b dx)
  (* (sum f
          (+ a (/ dx 2.0))
          (lambda (x) (+ x dx))
          b)
     dx))
```

В общем случае, `lambda` используется для создания процедур точно так же, как `define`, только никакого имени для процедуры не указывается:

```
(lambda (⟨формальные-параметры⟩) ⟨тело⟩)
```

Получается столь же полноценная процедура, как и с помощью `define`. Единственная разница состоит в том, что она не связана ни с каким именем в окружении. На самом деле

```
(define (plus4 x) (+ x 4))
```

эквивалентно

```
(define plus4 (lambda (x) (+ x 4)))
```

Можно читать выражение `lambda` так:

`(lambda`	`(x)`		`(+`	`x`		`4))`
↑	↑		↑	↑		↑
Процедура	от аргумента x,	которая	складывает	x	и	4

Подобно любому выражению, значением которого является процедура, выражение с `lambda` можно использовать как оператор в комбинации, например

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

Или, в более общем случае, в любом контексте, где обычно используется имя процедуры[53].

[53] Было бы более понятно и менее страшно для изучающих Лисп, если бы здесь использовалось более ясное имя, чем `lambda`, например `make-procedure`. Однако традиция уже прочно укоренилась. Эта нотация заимствована из λ-исчисления, формализма, изобретенного математическим логиком Алонсо Чёрчем (Church 1941). Чёрч разработал λ-исчисление, чтобы найти строгое основание для понятий функции и применения функции. λ-исчисление стало основным инструментом математических исследований по семантике языков программирования.

Создание локальных переменных с помощью `let`

Еще одно применение `lambda` состоит во введении локальных переменных. Часто нам в процедуре бывают нужны локальные переменные помимо тех, что связаны формальными параметрами. Допустим, например, что нам надо вычислить функцию

$$f(x,y) = x(1+xy)^3 + y(1-y) + (1+xy)(1-y)$$

которую мы также могли бы выразить как

$$\begin{aligned} a &= 1+xy \\ b &= 1-y \\ f(x,y) &= xa^2 + yb + ab \end{aligned}$$

Когда мы пишем процедуру для вычисления f, хотелось бы иметь как локальные переменные не только x и y, но и имена для промежуточных результатов вроде a и b. Можно сделать это с помощью вспомогательной процедуры, которая связывает локальные переменные:

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

Разумеется, безымянную процедуру для связывания локальных переменных мы можем записать через `lambda`-выражение. При этом тело `f` оказывается просто вызовом этой процедуры.

```
(define (f x y)
  ((lambda (a b)
     (+ (* x (square a))
        (* y b)
        (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

Такая конструкция настолько полезна, что есть особая форма под названием `let`, которая делает ее более удобной. С использованием `let` процедуру `f` можно записать так:

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

Общая форма выражения с `let` такова:

```
(let ((⟨пер1⟩ ⟨выр1⟩)
      (⟨пер2⟩ ⟨выр2⟩)
      ...
      (⟨перn⟩ ⟨вырn⟩)))
  ⟨тело⟩)
```

Это можно понимать как

> Пусть ⟨*пер*$_1$⟩ имеет значение ⟨*выр*$_1$⟩
> и ⟨*пер*$_2$⟩ имеет значение ⟨*выр*$_2$⟩
> ...
> и ⟨*пер*$_n$⟩ имеет значение ⟨*выр*$_n$⟩
> в ⟨*теле*⟩

Первая часть `let`-выражения представляет собой список пар вида имя–значение. Когда `let` вычисляется, каждое имя связывается со значением соответствующего выражения. Затем вычисляется тело `let`, причем эти имена связаны как локальные переменные. Происходит это так: выражение `let` интерпретируется как альтернативная форма для

```
((lambda (⟨пер1⟩ ... ⟨перn⟩)
   ⟨тело⟩)
  ⟨выр1⟩ ... ⟨вырn⟩)
```

От интерпретатора не требуется никакого нового механизма связывания переменных. Выражение с `let` — это всего лишь синтаксический сахар для вызова `lambda`.

Из этой эквивалентности мы видим, что область определения переменной, введенной в `let`-выражении — тело `let`. Отсюда следует, что:

• `Let` позволяет связывать переменные сколь угодно близко к тому месту, где они используются. Например, если значение `x` равно 5, значение выражения

```
(+ (let ((x 3))
     (+ x (* x 10)))
   x)
```

равно 38. Значение `x` в теле `let` равно 3, так что значение `let`-выражения равно 33. С другой стороны, `x` как второй аргумент к внешнему `+` по-прежнему равен 5.

• Значения переменных вычисляются за пределами `let`. Это существенно, когда выражения, дающие значения локальным переменным, зависят от переменных, которые имеют те же имена, что и сами локальные переменные. Например, если значение `x` равно 2, выражение

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

будет иметь значение 12, поскольку внутри тела `let` `x` будет равно 3, а `y` 4 (что равняется внешнему `x` плюс 2).

Иногда с тем же успехом, что и `let`, можно использовать внутренние определения. Например, вышеописанную процедуру `f` мы могли бы определить как

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x (square a))
     (* y b)
     (* a b)))
```

В таких ситуациях, однако, мы предпочитаем использовать `let`, а `define` писать только при определении локальных процедур[54].

Упражнение 1.34.
Допустим, мы определили процедуру

```
(define (f g)
  (g 2))
```

Тогда мы имеем

```
(f square)
4

(f (lambda (z) (* z (+ z 1))))
6
```

Что случится, если мы (извращенно) попросим интерпретатор вычислить комбинацию `(f f)`? Объясните.

1.3.3. Процедуры как обобщенные методы

Мы ввели составные процедуры в разделе 1.1.4 в качестве механизма для абстракции схем числовых операций, так, чтобы они были независимы от конкретных используемых чисел. С процедурами высших порядков, такими, как процедура `integral` из раздела 1.3.1, мы начали исследовать более мощный тип абстракции: процедуры, которые используются для выражения обобщенных методов вычисления, независимо от конкретных используемых функций. В этом разделе мы рассмотрим два более подробных примера — общие методы нахождения нулей и неподвижных точек функций, — и покажем, как эти методы могут быть прямо выражены в виде процедур.

Нахождение корней уравнений методом половинного деления

Метод половинного деления (half-interval method) — это простой, но мощный способ нахождения корней уравнения $f(x) = 0$, где f — непрерывная функция. Идея состоит в том, что если нам даны такие точки a и b, что $f(a) < 0 < f(b)$, то функция f должна

[54] Если мы хотим понимать внутренние определения настолько, чтобы быть уверенными, что программа действительно соответствует нашим намерениям, то нам требуется более сложная модель процесса вычислений, чем приведенная в этой главе. Однако с внутренними определениями процедур эти тонкости не возникают. К этому вопросу мы вернемся в разделе 4.1.6, после того, как больше узнаем о вычислении.

иметь по крайней мере один ноль на отрезке между a и b. Чтобы найти его, возьмем x, равное среднему между a и b, и вычислим $f(x)$. Если $f(x) > 0$, то f должна иметь ноль на отрезке между a и x. Если $f(x) < 0$, то f должна иметь ноль на отрезке между x и b. Продолжая таким образом, мы сможем находить все более узкие интервалы, на которых f должна иметь ноль. Когда мы дойдем до точки, где этот интервал достаточно мал, процесс останавливается. Поскольку интервал неопределенности уменьшается вдвое на каждом шаге процесса, число требуемых шагов растет как $\Theta(\log(L/T))$, где L есть длина исходного интервала, а T есть допуск ошибки (то есть размер интервала, который мы считаем «достаточно малым»). Вот процедура, которая реализует эту стратегию:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

Мы предполагаем, что вначале нам дается функция f и две точки, в одной из которых значение функции отрицательно, в другой положительно. Сначала мы вычисляем среднее между двумя краями интервала. Затем мы проверяем, не является ли интервал уже достаточно малым, и если да, сразу возвращаем среднюю точку как ответ. Если нет, мы вычисляем значение f в средней точке. Если это значение положительно, мы продолжаем процесс с интервалом от исходной отрицательной точки до средней точки. Если значение в средней точке отрицательно, мы продолжаем процесс с интервалом от средней точки до исходной положительной точки. Наконец, существует возможность, что значение в средней точке в точности равно 0, и тогда средняя точка и есть тот корень, который мы ищем.

Чтобы проверить, достаточно ли близки концы интервала, мы можем взять процедуру, подобную той, которая используется в разделе 1.1.7 при вычислении квадратных корней[55]:

```
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

Использовать процедуру `search` непосредственно ужасно неудобно, поскольку случайно мы можем дать ей точки, в которых значения f не имеют нужных знаков, и в этом случае мы получим неправильный ответ. Вместо этого мы будем использовать `search` посредством следующей процедуры, которая проверяет, который конец интервала имеет положительное, а который отрицательное значение, и соответствующим образом зовет

[55] Мы использовали 0.001 как достаточно «малое» число, чтобы указать допустимую ошибку вычисления. Подходящий допуск в настоящих вычислениях зависит от решаемой задачи, ограничений компьютера и алгоритма. Часто это весьма тонкий вопрос, в котором требуется помощь специалиста по численному анализу или волшебника какого-нибудь другого рода.

search. Если на обоих концах интервала функция имеет одинаковый знак, метод половинного деления использовать нельзя, и тогда процедура сообщает об ошибке[56]:

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "У аргументов не разные знаки " a b)))))
```

В следующем примере метод половинного деления используется, чтобы вычислить π как корень уравнения $\sin x = 0$, лежащий между 2 и 4.

```
(half-interval-method sin 2.0 4.0)
3.14111328125
```

Во втором примере через метод половинного деления ищется корень уравнения $x^3 - 2x - 3 = 0$, расположенный между 1 и 2:

```
(half-interval-method (lambda (x) (- (* x x x) (* 2 x) 3))
                      1.0
                      2.0)
1.89306640625
```

Нахождение неподвижных точек функций

Число x называется *неподвижной точкой* (fixed point) функции f, если оно удовлетворяет уравнению $f(x) = x$. Для некоторых функций f можно найти неподвижную точку, начав с какого-то значения и применяя f многократно:

$$f(x), f(f(x)), f(f(f(x))), \ldots$$

— пока значение не перестанет сильно изменяться. С помощью этой идеи мы можем составить процедуру fixed-point, которая в качестве аргументов принимает функцию и начальное значение и производит приближение к неподвижной точке функции. Мы многократно применяем функцию, пока не найдем два последовательных значения, разница между которыми меньше некоторой заданной чувствительности:

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
```

[56] Этого можно добиться с помощью процедуры error, которая в качестве аргументов принимает несколько значений и печатает их как сообщение об ошибке.

```
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Например, с помощью этого метода мы можем приближенно вычислить неподвижную точку функции косинус, начиная с 1 как стартового приближения[57]:

```
(fixed-point cos 1.0)
.7390822985224023
```

Подобным образом можно найти решение уравнения $y = siny + cosy$:

```
(fixed-point (lambda (y) (+ (sin y) (cos y)))
             1.0)
.2587315962971173
```

Процесс поиска неподвижной точки похож на процесс, с помощью которого мы искали квадратный корень в разделе 1.1.7. И тот, и другой основаны на идее последовательного улучшения приближений, пока результат не удовлетворит какому-то критерию. На самом деле мы без труда можем сформулироватьвычисление квадратного корня как поиск неподвижной точки. Вычислить квадратного корня из произвольного числа x означает найти такое y, что $y^2 = x$. Переведя это уравнение в эквивалентную форму $y = x/y$, мы обнаруживаем, что должны найти неподвижную точку функции[58] $y \mapsto x/y$, и, следовательно, мы можем попытаться вычислять квадратные корни так:

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

К сожалению, этот поиск неподвижной точки не сходится. Рассмотрим исходное значение y_1. Следующее значение равно $y_2 = x/y_1$, а следующее за ним $y_3 = x/y_2 = x/(x/y_1) = y_1$. В результате выходит бесконечный цикл, в котором два значения y_1 и y_2 повторяются снова и снова, прыгая вокруг правильного ответа.

Один из способов управлять такими прыжками состоит в том, чтобы заставить значения изменяться не так сильно. Поскольку ответ всегда находится между текущим значением y и x/y, мы можем взять новое значение, не настолько далекое от y, как x/y, взяв среднее между ними, так что следующее значение будет не x/y, а $\frac{1}{2}(y + x/y)$. Процесс получения такой последовательности есть всего лишь процесс поиска неподвижной точки $y \mapsto \frac{1}{2}(y + x/y)$.

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

[57] Попробуйте во время скучной лекции установить калькулятор в режим радиан и нажимать кнопку `cos`, пока не найдете неподвижную точку.

[58] $\mapsto$ (произносится «отображается в») — это математический способ написать `lambda`. $y \mapsto x/y$ означает `(lambda (y) (/ x y))`, то есть функцию, значение которой в точке y есть x/y.

(Заметим, что $y = \frac{1}{2}(y + x/y)$ всего лишь простая трансформация уравнения $y = x/y$; чтобы ее получить, добавьте y к обоим частям уравнения и поделите пополам.)

После такой модификации процедура поиска квадратного корня начинает работать. В сущности, если мы рассмотрим определения, мы увидим, что последовательность приближений к квадратному корню, порождаемая здесь, в точности та же, что порождается нашей исходной процедурой поиска квадратного корня из раздела 1.1.7. Этот подход с усреднением последовательных приближений к решению, метод, который мы называем *торможение усреднением* (average damping), часто помогает достичь сходимости при поисках неподвижной точки.

Упражнение 1.35.
Покажите, что золотое сечение ϕ (раздел 1.2.2) есть неподвижная точка трансформации $x \mapsto 1 + 1/x$, и используйте этот факт для вычисления ϕ с помощью процедуры `fixed-point`.

Упражнение 1.36.
Измените процедуру `fixed-point` так, чтобы она печатала последовательность приближений, которые порождает, с помощью примитивов `newline` и `display`, показанных в упражнении 1.22. Затем найдите решение уравнения $x^x = 1000$ путем поиска неподвижной точки $x \mapsto \log(1000)/\log(x)$. (Используйте встроенную процедуру Scheme `log`, которая вычисляет натуральные логарифмы.) Посчитайте, сколько шагов это занимает при использовании торможения усреднением и без него. (Учтите, что нельзя начинать `fixed-point` со значения 1, поскольку это вызовет деление на $\log(1) = 0$.)

Упражнение 1.37.
a. Бесконечная*цепная дробь* (continued fraction) есть выражение вида

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \ldots}}}$$

В качестве примера можно показать, что расширение бесконечной цепной дроби при всех N_i и D_i, равных 1, дает $1/\phi$, где ϕ — золотое сечение (описанное в разделе 1.2.2). Один из способов вычислить цепную дробь состоит в том, чтобы после заданного количества термов оборвать вычисление. Такой обрыв — так называемая *конечная цепная дробь* (finite continued fraction) из k элементов, — имеет вид

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \ldots \cfrac{N_k}{D_k}}}$$

Предположим, что `n` и `d` — процедуры одного аргумента (номера элемента i), возвращающие N_i и D_i элементов цепной дроби. Определите процедуру `cont-frac` так, чтобы вычисление `(cont-frac n d k)` давало значение k-элементной конечной цепной дроби. Проверьте свою процедуру, вычисляя приближения к $1/\phi$ с помощью

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
```

для последовательных значений `k`. Насколько большим пришлось сделать `k`, чтобы получить приближение, верное с точностью 4 цифры после запятой?

б. Если Ваша процедура `cont-frac` порождает рекурсивный процесс, напишите вариант, который порождает итеративный процесс. Если она порождает итеративный процесс, напишите вариант, порождающий рекурсивный процесс.

Упражнение 1.38.
В 1737 году швейцарский математик Леонард Эйлер опубликовал статью *De functionibus Continuis*, которая содержала расширение цепной дроби для $e-2$, где e — основание натуральных логарифмов. В этой дроби все N_i равны 1, а D_i последовательно равны $1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots$.Напишите программу, использующую Вашу процедуру `cont-frac` из упражнения 1.37 для вычисления e на основании формулы Эйлера.

Упражнение 1.39.
Представление тангенса в виде цепной дроби было опубликовано в 1770 году немецким математиком Й.Х. Ламбертом:

$$\operatorname{tg} x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \ldots}}}$$

где x дан в радианах. Определите процедуру `(tan-cf x k)`, которая вычисляет приближение к тангенсу на основе формулы Ламберта. `K` указывает количество термов, которые требуется вычислить, как в упражнении 1.37.

1.3.4. Процедуры как возвращаемые значения

Предыдущие примеры показывают, что возможность передавать процедуры в качестве аргументов значительно увеличивает выразительную силу нашего языка программирования. Мы можем добиться еще большей выразительной силы, создавая процедуры, возвращаемые значения которых сами являются процедурами.

Эту идею можно проиллюстрировать примером с поиском неподвижной точки, обсуждаемым в конце раздела 1.3.3. Мы сформулировали новую версию процедуры вычисления квадратного корня как поиск неподвижной точки, начав с наблюдения, что $\sqrt{x}$ есть неподвижная точка функции $y \mapsto x/y$. Затем мы использовали торможение усреднением, чтобы заставить приближения сходиться. Торможение усреднением само по себе является полезным приемом. А именно, получив функцию f, мы возвращаем функцию, значение которой в точке x есть среднее арифметическое между x и $f(x)$.

Идею торможения усреднением мы можем выразить при помощи следующей процедуры:

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

`Average-damp` — это процедура, принимающая в качестве аргумента процедуру `f` и возвращающая в качестве значения процедуру (полученную с помощью `lambda`), которая, будучи применена к числу `x`, возвращает среднее между `x` и `(f x)`. Например,

применение `average-damp` к процедуре `square` получает процедуру, значением которой для некоторого числа x будет среднее между x и x^2. Применение этой процедуры к числу 10 возвращает среднее между 10 и 100, то есть 55[59]:

```
((average-damp square) 10)
55
```

Используя `average-damp`, мы можем переформулировать процедуру вычисления квадратного корня следующим образом:

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

Обратите внимание, как такая формулировка делает явными три идеи нашего метода: поиск неподвижной точки, торможение усреднением и функцию $y \mapsto x/y$. Полезно сравнить такую формулировку метода поиска квадратного корня с исходной версией, представленной в разделе 1.1.7. Вспомните, что обе процедуры выражают один и тот же процесс, и посмотрите, насколько яснее становится его идея, когда мы выражаем процесс в терминах этих абстракций. В общем случае существует много способов сформулировать процесс в виде процедуры. Опытные программисты знают, как выбрать те формулировки процедур, которые наиболее ясно выражают их мысли, и где полезные элементы процесса показаны в виде отдельных сущностей, которые можно использовать в других приложениях. Чтобы привести простой пример такого нового использования, заметим, что кубический корень x является неподвижной точкой функции $y \mapsto x/y^2$, так что мы можем немедленно обобщить нашу процедуру поиска квадратного корня так, чтобы она извлекала кубические корни[60]:

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

Метод Ньютона

Когда в разделе 1.1.7 мы впервые представили процедуру извлечения квадратного корня, мы упомянули, что это лишь частный случай*метода Ньютона* (Newton's method). Если $x \mapsto g(x)$ есть дифференцируемая функция, то решение уравнения $g(x) = 0$ есть неподвижная точка функции $x \mapsto f(x)$, где

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

а $Dg(x)$ есть производная g, вычисленная в точке x. Метод Ньютона состоит в том, чтобы применить описанный способ поиска неподвижной точки и аппроксимировать решение

[59]Заметьте, что здесь мы имеем комбинацию, оператор которой сам по себе комбинация. В упражнении 1.4 уже была продемонстрирована возможность таких комбинаций, но то был всего лишь игрушечный пример. Здесь мы начинаем чувствовать настоящую потребность в выражениях такого рода — когда нам нужно применить процедуру, полученную в качестве значения из процедуры высшего порядка.

[60]См. дальнейшее обобщение в упражнении 1.45

уравнения путем поиска неподвижной точки функции f.[61] Для многих функций g при достаточно хорошем начальном значении x метод Ньютона очень быстро приводит к решению уравнения $g(x) = 0$[62].

Чтобы реализовать метод Ньютона в виде процедуры, сначала нужно выразить понятие производной. Заметим, что «взятие производной», подобно торможению усреднением, трансформирует одну функцию в другую. Например, производная функции $x \mapsto x^3$ есть функция $x \mapsto 3x^2$. В общем случае, если g есть функция, а dx — маленькое число, то производная Dg функции g есть функция, значение которой в каждой точке x описывается формулой (при dx, стремящемся к нулю)

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

Таким образом, мы можем выразить понятие производной (взяв dx равным, например, 0.00001) в виде процедуры

```
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
```

дополненной определением

```
(define dx 0.00001)
```

Подобно `average-damp`, `deriv` является процедурой, которая берет процедуру в качестве аргумента и возвращает процедуру как значение. Например, чтобы найти приближенное значение производной $x \mapsto x^3$ в точке 5 (точное значение производной равно 75), можно вычислить

```
(define (cube x) (* x x x))

((deriv cube) 5)
75.00014999664018
```

С помощью `deriv` мы можем выразить метод Ньютона как процесс поиска неподвижной точки:

```
(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

[61] Вводные курсы анализа обычно описывают метод Ньютона через последовательность приближений $x_{n+1} = x_n - g(x_n)/Dg(x_n)$. Наличие языка, на котором мы можем говорить о процессах, а также использование идеи неподвижных точек, упрощают описание этого метода.

[62] Метод Ньютона не всегда приводит к решению, но можно показать, что в удачных случаях каждая итерация удваивает точность приближения в терминах количества цифр после запятой. Для таких случаев метод Ньютона сходится гораздо быстрее, чем метод половинного деления.

Процедура `newton-transform` выражает формулу, приведенную в начале этого раздела, а `newtons-method` легко определяется с ее помощью. В качестве аргументов она принимает процедуру, вычисляющую функцию, чей ноль мы хотим найти, а также начальное значение приближения. Например, чтобы найти квадратный корень x, мы можем с помощью метода Ньютона найти ноль функции $y \mapsto y^2 - x$, начиная со значения 1[63]. Это дает нам еще одну форму процедуры вычисления квадратного корня:

```
(define (sqrt x)
  (newtons-method (lambda (y) (- (square y) x))
                  1.0))
```

Абстракции и процедуры как полноправные объекты

Мы видели два способа представить вычисление квадратного корня как частный случай более общего метода; один раз это был поиск неподвижной точки, другой — метод Ньютона. Поскольку сам метод Ньютона был выражен как процесс поиска неподвижной точки, на самом деле мы увидели два способа вычислить квадратный корень как неподвижную точку. Каждый из этих методов получает некоторую функцию и находит неподвижную точку для некоторой трансформации этой функции. Эту общую идею мы можем выразить как процедуру:

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

Эта очень общая процедура принимает в качестве аргументов процедуру `g`, которая вычисляет некоторую функцию, процедуру, которая трансформирует `g`, и начальное приближение. Возвращаемое значение есть неподвижная точка трансформированной функции.

С помощью такой абстракции можно переформулировать процедуру вычисления квадратного корня из этого раздела (ту, где мы ищем неподвижную точку версии $y \mapsto x/y$, заторможенной усреднением) как частный случай общего метода:

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (/ x y))
                            average-damp
                            1.0))
```

Подобным образом, вторую процедуру нахождения квадратного корня из этого раздела (пример применения метода Ньютона, который находит неподвижную точку Ньютонова преобразования $y \mapsto y^2 - x$) можно представить так:

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

Мы начали раздел 1.3 с наблюдения, что составные процедуры являются важным механизмом абстракции, поскольку они позволяют выражать общие методы вычисления в виде явных элементов нашего языка программирования. Теперь мы увидели, как

[63] При поиске квадратных корней метод Ньютона быстро сходится к правильному решению, начиная с любой точки.

процедуры высших порядков позволяют нам манипулировать этими общими методами и создавать еще более глубокие абстракции.

Как программисты, мы должны быть готовы распознавать возможности поиска абстракций, лежащих в основе наших программ, строить нашу работу на таких абстракциях и обобщать их, создавая еще более мощные абстракции. Это не значит, что программы всегда нужно писать на возможно более глубоком уровне абстракции: опытные программисты умеют выбирать тот уровень, который лучше всего подходит к их задаче. Однако важно быть готовыми мыслить в терминах этих абстракций и быть готовым применить их в новых контекстах. Важность процедур высшего порядка состоит в том, что они позволяют нам явно представлять эти абстракции в качестве элементов нашего языка программирования, так что мы можем обращаться с ними так же, как и с другими элементами вычисления.

В общем случае языки программирования накладывают ограничения на способы, с помощью которых можно манипулировать элементами вычисления. Говорят, что элементы, на которые накладывается наименьшее число ограничений, имеют статус элементов вычисления *первого класса* (first-class) или *полноправных*. Вот некоторые из их «прав и привилегий»[64]:

- Их можно называть с помощью переменных.
- Их можно передавать в процедуры в качестве аргументов.
- Их можно возвращать из процедур в виде результата.
- Их можно включать в структуры данных[65].

Лисп, в отличие от других распространенных языков программирования, дает процедурам полноправный статус. Это может быть проблемой для эффективной реализации, но зато получаемый выигрыш в выразительной силе огромен[66].

Упражнение 1.40.
Определите процедуру `cubic`, которую можно было бы использовать совместно с процедурой `newtons-method` в выражениях вида

```
(newtons-method (cubic a b c) 1)
```

для приближенного вычисления нулей кубических уравнений $x^3 + ax^2 + bx + c$.

Упражнение 1.41.
Определите процедуру `double`, которая принимает как аргумент процедуру с одним аргументом и возвращает процедуру, которая применяет исходную процедуру дважды. Например, если процедура `inc` добавляет к своему аргументу 1, то `(double inc)` должна быть процедурой, которая добавляет 2. Скажите, какое значение возвращает

```
(((double (double double)) inc) 5)
```

[64] Понятием полноправного статуса элементов языка программирования мы обязаны британскому специалисту по информатике Кристоферу Стрейчи (1916-1975).

[65] Примеры этого мы увидим после того, как введем понятие структур данных в главе 2.

[66] Основная цена, которую реализации приходится платить за придание процедурам статуса полноправных объектов, состоит в том, что, поскольку мы разрешаем возвращать процедуры как значения, нам нужно оставлять память для хранения свободных переменных процедуры даже тогда, когда она не выполняется. В реализации Scheme, которую мы рассмотрим в разделе 4.1, эти переменные хранятся в окружении процедуры.

Упражнение 1.42.
Пусть f и g — две одноаргументные функции. По определению, *композиция* (composition) f и g есть функция $x \mapsto f(g(x))$. Определите процедуру `compose` которая реализует композицию. Например, если `inc` — процедура, добавляющая к своему аргументу 1,

```
((compose square inc) 6)
49
```

Упражнение 1.43.
Если f есть численная функция, а n — положительное целое число, то мы можем построить n-кратное применение f, которое определяется как функция, значение которой в точке x равно $f(f(\ldots(f(x))\ldots))$. Например, если f есть функция $x \mapsto x+1$, то n-кратным применением f будет функция $x \mapsto x+n$. Если f есть операция возведения в квадрат, то n-кратное применение f есть функция, которая возводит свой аргумент в 2^n-ю степень. Напишите процедуру, которая принимает в качестве ввода процедуру, вычисляющую f, и положительное целое n, и возвращает процедуру, вычисляющую n-кратное применение f. Требуется, чтобы Вашу процедуру можно было использовать в таких контекстах:

```
((repeated square 2) 5)
625
```

Подсказка: может оказаться удобно использовать `compose` из упражнения 1.42.

Упражнение 1.44.
Идея *сглаживания* (smoothing a function) играет важную роль в обработке сигналов. Если f — функция, а dx — некоторое малое число, то сглаженная версия f есть функция, значение которой в точке x есть среднее между $f(x-dx)$, $f(x)$ и $f(x+dx)$. Напишите процедуру `smooth`, которая в качестве ввода принимает процедуру, вычисляющую f, и возвращает процедуру, вычисляющую сглаженную версию f. Иногда бывает удобно проводить повторное сглаживание (то есть сглаживать сглаженную функцию и т.д.), получая *n-кратно сглаженную функцию* (n-fold smoothed function). Покажите, как породить n-кратно сглаженную функцию с помощью `smooth` и `repeated` из упражнения 1.43.

Упражнение 1.45.
В разделе 1.3.3 мы видели, что попытка вычисления квадратных корней путем наивного поиска неподвижной точки $y \mapsto x/y$ не сходится, и что это можно исправить путем торможения усреднением. Тот же самый метод работает для нахождения кубического корня как неподвижной точки $y \mapsto x/y^2$, заторможенной усреднением. К сожалению, этот процесс не работает для корней четвертой степени — однажды примененного торможения усреднением недостаточно, чтобы заставить сходиться процесс поиска неподвижной точки $y \mapsto x/y^3$. С другой стороны, если мы применим торможение усреднением дважды (т.е. применим торможение усреднением к результату торможения усреднением от $y \mapsto x/y^3$), то поиск неподвижной точки начнет сходиться. Проделайте эксперименты, чтобы понять, сколько торможений усреднением нужно, чтобы вычислить корень n-ой степени как неподвижную точку на основе многократного торможения усреднением функции $y \mapsto x/y^{n-1}$. Используя свои результаты для того, напишите простую процедуру вычисленияякорней n-ой степени с помощью процедур `fixed-point`, `average-damp` и `repeated` из упражнения 1.43. Считайте, что все арифметические операции, какие Вам понадобятся, присутствуют в языке как примитивы.

Упражнение 1.46.
Некоторые из вычислительных методов, описанных в этой главе, являются примерами чрезвычайно общей вычислительной стратегии, называемой *пошаговое улучшение* (iterative improvement). Пошаговое улучшение состоит в следующем: чтобы что-то вычислить, нужно взять какое-то начальное значение, проверить, достаточно ли оно хорошо, чтобы служить ответом, и если нет, то улучшить это значение и продолжить процесс с новым значением. Напишите процедуру `iterative-improve`, которая принимает в качестве аргументов две процедуры: проверку, достаточно ли хорошо значение, и метод улучшения значения. `Iterative-improve` должна возвращать процедуру, которая принимает начальное значение в качестве аргумента и улучшает его, пока оно не станет достаточно хорошим. Перепишите процедуру `sqrt` из раздела 1.1.7 и процедуру `fixed-point` из раздела 1.3.3 в терминах `iterative-improve`.

Глава 2

Построение абстракций с помощью данных

> Теперь мы подходим к решающему шагу в математической абстракции: мы забываем, что обозначают наши символы. ...[Математик] не должен стоять на месте: есть много операций, которые он может производить с этими символами, не обращая внимания на те вещи, которые они обозначают.
>
> Герман Вейль.
> «Математический способ мышления»

В главе 1 мы сконцентрировали внимание на вычислительных процессах и роли процедур в проектировании программ. Мы рассмотрели, как использовать простейшие данные (числа) и простейшие операции (арифметические), как сочетать процедуры и получать составные процедуры с помощью композиции, условных выражений и использования параметров, а также как строить абстрактные процедуры при помощи `define`. Мы убедились, что процедуру можно рассматривать как схему локального развития процесса; мы классифицировали некоторые общие схемы процессов, воплощенные в процедурах, строили о них умозаключения и производили их простейший алгоритмический анализ. Кроме того, мы увидели, что процедуры высших порядков увеличивают выразительную силу нашего языка, позволяя оперировать общими методами вычисления, а следовательно, и проводить рассуждения в их терминах. Это во многом и составляет сущность программирования.

В этой главе мы будем рассматривать более сложные данные. Все процедуры главы 1 работают с простыми численными данными, а простых численных данных часто бывает недостаточно для тех задач, которые мы хотим решать с помощью вычислений. Программы, как правило, пишут, чтобы моделировать сложные явления, и чаще всего приходится строить вычислительные объекты, состоящие из нескольких частей, чтобы смоделировать многосторонние явления реального мира. Таким образом, в отличие от главы 1, где наше внимание было в основном направлено на создание абстракций с помощью сочетания процедур и построения составных процедур, в этой главе мы обращаемся к другой важной характеристике всякого языка программирования: тем средствам, которые он предоставляет для создания абстракций с помощью сочетания объектов данных и построения *составных данных* (compound data).

Для чего в языке программирования нужны составные данные? По тем же причинам, по которым нужны составные процедуры: мы хотим повысить понятийный уровень, на котором мы проектируем программы, хотим сделать наши проекты более модульными и

увеличить выразительную силу языка. Точно так же, как способность определять процедуры дает возможность работать с процессами на более высоком содержательном уровне, чем уровень элементарных операций языка, способность конструировать составные объекты данных позволяет работать с данными на более высоком понятийном уровне, чем уровень элементарных данных нашего языка.

Рассмотрим задачу проектирования системы для арифметических вычислений с рациональными числами. Мы можем представить себе операцию `add-rat`, которая принимает два рациональных числа и вычисляет их сумму. В терминах простейших данных, рациональное число можно рассматривать как два целых числа: числитель и знаменатель. Таким образом, мы могли бы сконструировать программу, в которой всякое рациональное число представлялось бы как пара целых (числитель и знаменатель) и `add-rat` была бы реализована как две процедуры (одна из которых вычисляла бы числитель суммы, а другая знаменатель). Однако это было бы крайне неудобно, поскольку нам приходилось бы следить, какие числители каким знаменателям соответствуют. Если бы системе требовалось производить большое количество операций над большим количеством рациональных чисел, такие служебные детали сильно затемняли бы наши программы, не говоря уже о наших мозгах. Было бы намного проще, если бы мы могли «склеить» числитель со знаменателем и получить пару — *составной объект данных* (compound data object), — с которой наши программы могли бы обращаться способом, соответствующим нашему представлению о рациональном числе как о едином понятии.

Кроме того, использование составных данных позволяет увеличить модульность программ. Если бы мы могли обрабатывать рациональные числа непосредственно как объекты, то можно было бы отделить ту часть программы, которая работает собственно с рациональными числами, от деталей представления рационального числа в виде пары целых. Общий метод отделения частей программы, которые имеют дело с представлением объектов данных, от тех частей, где эти объекты данных используются, — это мощная методология проектирования, называемая *абстракция данных* (data abstraction). Мы увидим, как с помощью абстракции данных программы становится легче проектировать, поддерживать и изменять.

Использование составных данных ведет к настоящему увеличению выразительной силы нашего языка программирования. Рассмотрим идею порождения «линейной комбинации» $ax+by$. Нам может потребоваться процедура, которая принимала бы как аргументы a, b, x и y и возвращала бы значение $ax+by$. Если аргументы являются числами, это не представляет никакой трудности, поскольку мы сразу можем определить процедуру

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

Предположим, однако, что нас интересуют не только числа. Предположим, что нам хотелось бы выразить в процедурных терминах идею о том, что можно строить линейные комбинации всюду, где определены сложение и умножение — для рациональных и комплексных чисел, многочленов и многого другого. Мы могли бы выразить это как процедуру в следующей форме:

```
(define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

где `add` и `mul` — не элементарные процедуры `+` и `*`, а более сложные устройства, которые проделывают соответствующие операции, какие бы типы данных мы ни передавали как аргументы `a`, `b`, `x` и `y`. Здесь важнейшая деталь состоит в том, что единственное, что требуется знать процедуре `linear-combination` об `a`, `b`, `x` и `y` — это то, что процедуры `add` и `mul` проделывают соответствующие действия. С точки зрения процедуры `linear-combination` несущественно, что такое `a`, `b`, `x` и `y`, и еще менее существенно, как они могут быть представлены через более простые данные. Этот же пример показывает, почему так важно, чтобы в нашем языке программирования была возможность прямо работать с составными объектами: иначе у процедур, подобных `linear-combination`, не было бы способа передать аргументы в `add` и `mul`, не зная деталей их устройства[1].

Мы начинаем эту главу с реализации описанной выше системы арифметики рациональных чисел. Это послужит основанием для обсуждения составных данных и абстракции данных. Как и в случае с составными процедурами, основная мысль состоит в том, что абстракция является методом ограничения сложности, и мы увидим, как абстракция данных позволяет нам возводить полезные *барьеры абстракции* (abstraction barriers) между разными частями программы.

Мы увидим, что главное в работе с составными данными — то, что язык программирования должен предоставлять нечто вроде «клея», так, чтобы объекты данных могли сочетаться, образуя более сложные объекты данных. Существует множество возможных типов клея. На самом деле мы обнаружим, что составные данные можно порождать вообще без использования каких-либо специальных операций, относящихся к «данным» — только с помощью процедур. Это еще больше размоет границу между «процедурами» и «данными», которая уже к концу главы 1 оказалась весьма тонкой. Мы также исследуем некоторые общепринятые методы представления последовательностей и деревьев. Важная идея в работе с составными данными — понятие *замыкания* (closure): клей для сочетания объектов данных должен позволять нам склеивать не только элементарные объекты данных, но и составные. Еще одна важная идея состоит в том, что составные объекты данных могут служить *стандартными интерфейсами* (conventional interfaces), так, чтобы модули программы могли сочетаться методом подстановки. Некоторые из этих идей мы продемонстрируем с помощью простого графического языка, использующего замыкание.

Затем мы увеличим выразительную мощность нашего языка путем введения *символьных выражений* (symbolic expressions) — данных, элементарные части которых могут быть произвольными символами, а не только числами. Мы рассмотрим различные варианты представления множеств объектов. Мы обнаружим, что, подобно тому, как одна и та же числовая функция может вычисляться различными вычислительными процессами, существует множество способов представить некоторую структуру данных через элементарные объекты, и выбор представления может существенно влиять на запросы манипулирующих этими данными процессов к памяти и ко времени. Мы исследуем эти

[1]Способность прямо оперировать процедурами увеличивает выразительную силу нашего языка программирования подобным же образом. Например, в разделе 1.3.1 мы ввели процедуру `sum`, которая принимает в качестве аргумента процедуру `term` и вычисляет сумму значений `term` на некотором заданном интервале. Чтобы определить `sum`, нам необходимо иметь возможность говорить о процедуре типа `term` как о едином целом, независимо от того, как она выражена через более простые операции. Вообще говоря, не имей мы понятия «процедуры», вряд ли мы и думать могли бы о возможности определения такой операции, как `sum`. Более того, пока мы размышляем о суммировании, детали того, как `term` может быть составлен из более простых операций, несущественны.

идеи в контексте символьного дифференцирования, представления множеств и кодирования информации.

После этого мы обратимся к задаче работы с данными, которые по-разному могут быть представлены в различных частях программы. Это ведет к необходимости ввести *обобщенные операции* (generic operations), которые обрабатывают много различных типов данных. Поддержка модульности в присутствии обобщенных операций требует более мощных барьеров абстракции, чем тех, что получаются с помощью простой абстракции данных. А именно, мы вводим *программирование, управляемое данными* (data-directed programming) как метод, который позволяет проектировать представления данных отдельно, а затем сочетать их *аддитивно* (additively) (т. е., без модификации). Чтобы проиллюстрировать силу этого подхода к проектированию систем, в завершение главы мы применим то, чему в ней научились, к реализации пакета символьной арифметики многочленов, коэффициенты которых могут быть целыми, рациональными числами, комплексными числами и даже другими многочленами.

2.1. Введение в абстракцию данных

В разделе 1.1.8 мы заметили, что процедура, которую мы используем как элемент при создании более сложной процедуры, может рассматриваться не только как последовательность определенных операций, но и как процедурная абстракция: детали того, как процедура реализована, могут быть скрыты, и сама процедура может быть заменена на другую с подобным поведением. Другими словами, мы можем использовать абстракцию для отделения способа использования процедуры от того, как эта процедура реализована в терминах более простых процедур. Для составных данных подобное понятие называется *абстракция данных* (data abstraction). Абстракция данных — это методология, которая позволяет отделить способ использования составного объекта данных от деталей того, как он составлен из элементарных данных.

Основная идея абстракции данных состоит в том, чтобы строить программы, работающие с составными данными, так, чтобы иметь дело с «абстрактными данными». То есть, используя данные, наши программы не должны делать о них никаких предположений, кроме абсолютно необходимых для выполнения поставленной задачи. В то же время «конкретное» представление данных определяется независимо от программ, которые эти данные используют. Интерфейсом между двумя этими частями системы служит набор процедур, называемых *селекторами* (selectors) и *конструкторами* (constructors), реализующих абстрактные данные в терминах конкретного представления. Чтобы проиллюстрировать этот метод, мы рассмотрим, как построить набор процедур для работы с рациональными числами.

2.1.1. Пример: арифметические операции над рациональными числами

Допустим, нам нужно работать с рациональной арифметикой. Нам требуется складывать, вычитать, умножать и делить рациональные числа, а также проверять, равны ли два рациональных числа друг другу.

Для начала предположим, что у нас уже есть способ построить рациональное число

из его числителя и знаменателя. Кроме того, мы предполагаем, что имея рациональное число, мы можем получить его числитель и знаменатель. Допустим также, что эти конструктор и два селектора доступны нам в виде процедур:

- `(make-rat` $\langle n \rangle$ $\langle d \rangle$`)` возвращает рациональное число, числитель которого целое $\langle n \rangle$, а знаменатель — целое $\langle d \rangle$.
- `(numer` $\langle x \rangle$`)` возвращает числитель рационального числа $\langle x \rangle$.
- `(denom` $\langle x \rangle$`)` возвращает знаменатель рационального числа $\langle x \rangle$.

Здесь мы используем мощную стратегию синтеза: *мечтать не вредно* (wishful thinking). Пока что мы не сказали, как представляется рациональное число и как должны реализовываться процедуры `numer`, `denom` и `make-rat`. Тем не менее, если бы эти процедуры у нас были, мы могли бы складывать, вычитать, умножать, делить и проверять на равенство с помощью следующих отношений:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \text{ тогда и только тогда, когда } n_1 d_2 = n_2 d_1$$

Мы можем выразить эти правила в процедурах:

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

```
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Теперь у нас есть операции над рациональными числами, определенные в терминах процедур — селекторов и конструкторов `numer`, `denom` и `make-rat`. Однако сами эти процедуры мы еще не написали. Нам нужен какой-нибудь способ склеить вместе числитель и знаменатель, чтобы получить рациональное число.

Пары

Для реализации конкретного уровня абстракции данных в нашем языке имеется составная структура, называемая *парой* (pair), и она создается с помощью элементарной процедуры `cons`. Эта процедура принимает два аргумента и возвращает объект данных, который содержит эти два аргумента в качестве частей. Имея пару, мы можем получить ее части с помощью элементарных процедур `car` и `cdr`[2]. Таким образом, использовать `cons`, `car` и `cdr` можно так:

```
(define x (cons 1 2))

(car x)
1

(cdr x)
2
```

Заметим, что пара является объектом, которому можно дать имя и работать с ним, подобно элементарному объекту данных. Более того, можно использовать `cons` для создания пар, элементы которых сами пары, и так далее:

```
(define x (cons 1 2))

(define y (cons 3 4))

(define z (cons x y))

(car (car z))
1

(car (cdr z))
3
```

В разделе 2.2 мы увидим, что из этой возможности сочетать пары следует возможность их использовать как строительные блоки общего назначения при создании любых

[2] `Cons` означает *construct* (построить, сконструировать, собрать). Имена `car` и `cdr` происходят из исходной реализации Лиспа на IBM 704. Схема адресации этой машины позволяла обращаться к «адресной» и «декрементной» частям ячейки памяти. `Car` означает *Contents of Address Part of Register* (содержимое адресной части регистра), а `cdr` (произносится «куддер») означает *Contents of Decrement Part of Register* (содержимое декрементной части регистра).

сложных структур данных. Один-единственный примитив составных данных *пара*, реализуемый процедурами `cons`, `car` и `cdr`, — вот и весь клей, который нам нужен. Объекты данных, составленные из пар, называются *данные со списковой структурой* (list-structured data).

Представление рациональных чисел

Пары позволяют нам естественным образом завершить построение системы рациональных чисел. Будем просто представлять рациональное число в виде пары двух целых чисел: числителя и знаменателя. Тогда `make-rat`, `numer` и `denom` немедленно реализуются следующим образом[3].

```
(define (make-rat n d) (cons n d))

(define (numer x) (car x))

(define (denom x) (cdr x))
```

Кроме того, когда нам требуется выводить результаты вычислений, мы печатаем рациональное число, сначала выводя его числитель, затем косую черту и затем знаменатель[4]:

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

Теперь мы можем опробовать процедуры работы с рациональными числами:

```
(define one-half (make-rat 1 2))

(print-rat one-half)
1/2
```

---

[3] Другой способ определить селекторы и конструктор был бы

```
(define make-rat cons)
(define numer car)
(define denom cdr)
```

Первое определение связывает имя `make-rat` со значением выражения `cons`, то есть элементарной процедурой, которая строит пары. Таким образом, `make-rat` и `cons` становятся именами для одного и того же элементарного конструктора.

Такое определение конструкторов и селекторов эффективно: вместо того, чтобы заставлять `make-rat` *вызывать* `cons`, мы делаем `make-rat` и `cons` *одной и той же процедурой*, так что когда вызывается `make-rat`, происходит вызов только одной процедуры, а не двух. С другой стороны, это не дает работать отладочным средствам, которые отслеживают вызовы процедур или устанавливают на них контрольные точки: Вам может потребоваться следить за вызовами `make-rat`, но Вы уж точно никогда не захотите отслеживать каждый вызов `cons`.

В этой книге мы решили не использовать такой стиль определений.

[4] `Display` — элементарная процедура языка Scheme для печати данных. Другая элементарная процедура, `newline`, переводит строку при печати. Эти процедуры не возвращают никакого полезного значения, так что в примерах использования `print-rat` ниже, мы показываем только то, что печатает `print-rat`, а не то, что интерпретатор выводит как значение `print-rat`.

```
(define one-third (make-rat 1 3))

(print-rat (add-rat one-half one-third))
5/6

(print-rat (mul-rat one-half one-third))
1/6

(print-rat (add-rat one-third one-third))
6/9
```

Как показывает последний пример, наша реализация рациональных чисел не приводит их к наименьшему знаменателю. Мы можем исправить это упущение, изменив `make-rat`. Если у нас есть процедура `gcd`, вычисляющая наибольший общий делитель двух целых чисел, вроде той, которая описана в разделе 1.2.5, мы можем с помощью `gcd` сокращать числитель и знаменатель, прежде, чем построить пару:

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

Теперь мы имеем

```
(print-rat (add-rat one-third one-third))
2/3
```

как нам того и хотелось. Эта модификация была произведена путем изменения конструктора `make-rat`, и мы не тронули ни одну из процедур (скажем, `add-rat` или `mul-rat`), которые реализуют сами операции.

Упражнение 2.1.
Определите улучшенную версию `mul-rat`, которая принимала бы как положительные, так и отрицательные аргументы. `Make-rat` должна нормализовывать знак так, чтобы в случае, если рациональное число положительно, то и его числитель, и знаменатель были бы положительны, а если оно отрицательно, то чтобы только его числитель был отрицателен.

2.1.2. Барьеры абстракции

Прежде чем мы перейдем к другим примерам работы с составными данными и абстракцией данных, рассмотрим несколько вопросов, относящихся к примеру с рациональными числами. Мы определили операции над рациональными числами через конструктор `make-rat` и селекторы `numer` и `denom`. В общем случае основная идея абстракции данных состоит в том, чтобы определить для каждого типа объектов данных набор базовых операций, через которые будут выражаться все действия с объектами этого типа, и затем при работе с данными использовать только этот набор операций.

Мы можем представить себе структуру системы работы с рациональными числами так, как это показано на рис. 2.1. Горизонтальные линии обозначают *барьеры абстракции* (abstraction barriers), которые отделяют различные «уровни» системы друг от друга.

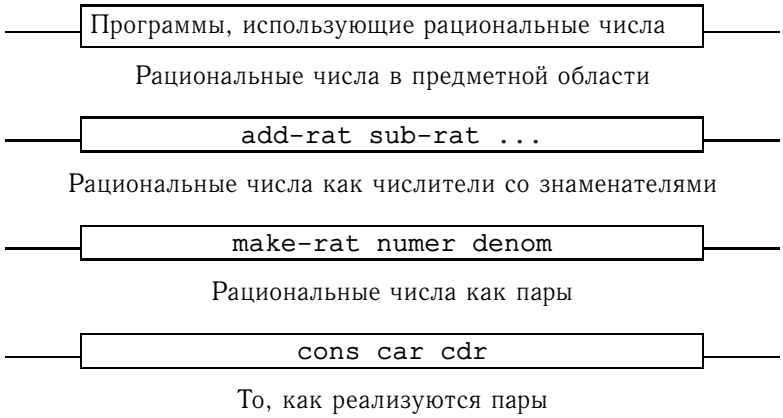


Рис. 2.1. Барьеры абстракции данных в пакете для работы с рациональными числами.

На каждом из этих уровней барьер отделяет программы, которые используют абстрактные данные (сверху) от программ, которые реализуют эту абстракцию данных (внизу). Программы, использующие рациональные числа, работают с ними исключительно в терминах процедур, которые пакет работы с рациональными числами предоставляет «для общего пользования»: `add-rat`, `sub-rat`, `mul-rat`, `div-rat` и `equal-rat?`. В свою очередь, эти процедуры используют только конструктор и селекторы `make-rat`, `numer` и `denom`, которые сами реализованы при помощи пар. Детали реализации пар не имеют значения для остальной части пакета работы с рациональными числами; существенно только, что с парами можно работать при помощи `cons`, `car` и `cdr`. По существу, процедуры на каждом уровне являются интерфейсами, которые определяют барьеры абстракции и связывают различные уровни.

У этой простой идеи много преимуществ. Одно из них состоит в том, что программы становится намного проще поддерживать и изменять. Любая сложная структура может быть представлена через элементарные структуры данных языка программирования многими способами. Разумеется, выбор представления влияет на программы, работающие с этим представлением; так что, если когда-нибудь позднее его нужно будет изменить, соответственно придется изменить и все эти программы. В случае больших программ эта задача может быть весьма трудоемкой и дорогой, если зависимость от представления не будет при проектировании ограничена несколькими программными модулями.

Например, другим способом решения задачи приведения рациональных чисел к наименьшему знаменателю было бы производить сокращение не тогда, когда мы конструируем число, а каждый раз, как мы к нему обращаемся. При этом потребуются другие конструктор и селекторы:

```
(define (make-rat n d)
  (cons n d))

(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
```

```
    (/ (car x) g)))

(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

Разница между этой реализацией и предыдущей состоит в том, когда мы вычисляем НОД с помощью `gcd`. Если при типичном использовании рациональных чисел к числителю и знаменателю одного и того же рационального числа мы обращаемся по многу раз, вычислять НОД лучше тогда, когда рациональное число конструируется. Если нет, нам может быть выгодно подождать с его вычислением до времени обращения. В любом случае, когда мы переходим от одной реализации к другой, нам ничего не нужно менять в процедурах `add-rat`, `sub-rat` и прочих.

То, что мы ограничиваем зависимость от представления несколькими интерфейсными процедурами, помогает нам и проектировать программы, и изменять их, поскольку таким образом мы сохраняем гибкость и получаем возможность рассматривать другие реализации. Продолжая наш простой пример, представим себе, что мы строим пакет работы с рациональными числами и не можем сразу решить, вычислять ли НОД при построении числа или при обращении к нему. Методология абстракции данных позволяет нам отложить это решение, не теряя возможности продолжать разработку остальных частей системы.

Упражнение 2.2.
Рассмотрим задачу представления отрезков прямой на плоскости. Каждый отрезок представляется как пара точек: начало и конец. Определите конструктор `make-segment` и селекторы `start-segment` и `end-segment`, которые определяют представление отрезков в терминах точек. Далее, точку можно представить как пару чисел: координата x и координата y. Соответственно, напишите конструктор `make-point` и селекторы `x-point` и `y-point`, которые определяют такое представление. Наконец, используя свои селекторы и конструктор, напишите процедуру `midpoint-segment`, которая принимает отрезок в качестве аргумента и возвращает его середину (точку, координаты которой являются средним координат концов отрезка). Чтобы опробовать эти процедуры, Вам потребуется способ печатать координаты точек:

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

Упражнение 2.3.
Реализуйте представление прямоугольников на плоскости. (Подсказка: Вам могут потребоваться результаты упражнения 2.2.) Определите в терминах своих конструкторов и селекторов процедуры, которые вычисляют периметр и площадь прямоугольника. Теперь реализуйте другое представление для прямоугольников. Можете ли Вы спроектировать свою систему с подходящими барьерами абстракции так, чтобы одни и те же процедуры вычисления периметра и площади работали с любым из Ваших представлений?

2.1.3. Что значит слово «данные»?

Свою реализацию рациональных чисел в разделе 2.1.1 мы начали с определения операций над рациональными числами `add-rat`, `sub-rat` и так далее в терминах трех неопределенных процедур: `make-rat`, `numer` и `denom`. В этот момент мы могли думать об операциях как определяемых через объекты данных — числители, знаменатели и рациональные числа, — поведение которых определялось тремя последними процедурами.

Но что в точности означает слово *данные* (data)? Здесь недостаточно просто сказать «то, что реализуется некоторым набором селекторов и конструкторов». Ясно, что не любой набор из трех процедур может служить основой для реализации рациональных чисел. Нам нужно быть уверенными в том, что если мы конструируем рациональное число `x` из пары целых `n` и `d`, то получение `numer` и `denom` от `x` и деление их друг на друга должно давать тот же результат, что и деление `n` на `d`. Другими словами, `make-rat`, `numer` и `denom` должны удовлетворять следующему условию: для каждого целого числа `n` и не равного нулю целого `d`, если `x` есть `(make-rat n d)`, то

$$\frac{\texttt{(numer x)}}{\texttt{(denom x)}} = \frac{\texttt{n}}{\texttt{d}}$$

Это на самом деле единственное условие, которому должны удовлетворять `make-rat`, `numer` и `denom`, чтобы служить основой для представления рациональных чисел. В общем случае можно считать, что данные — это то, что определяется некоторым набором селекторов и конструкторов, а также некоторыми условиями, которым эти процедуры должны удовлетворять, чтобы быть правильным представлением[5].

Эта точка зрения может послужить для определения не только «высокоуровневых» объектов данных, таких как рациональные числа, но и объектов низкого уровня. Рассмотрим понятие пары, с помощью которого мы определили наши рациональные числа. Мы ведь ни разу не сказали, что такое пара, и указывали только, что для работы с парами язык дает нам процедуры `cons`, `car` и `cdr`. Но единственное, что нам надо знать об этих процедурах — это что если мы склеиваем два объекта при помощи `cons`, то с помощью `car` и `cdr` мы можем получить их обратно. То есть эти операции удовлетворяют условию, что для любых объектов `x` и `y`, если `z` есть `(cons x y)`, то `(car z)` есть `x`, а `(cdr z)` есть `y`. Действительно, мы упомянули, что три эти процедуры включены в наш язык как примитивы. Однако любая тройка процедур, которая удовлетворяет вышеуказанному условию, может использоваться как основа реализации пар.

[5]Как ни странно, эту мысль очень трудно строго сформулировать. Существует два подхода к такой формулировке. Один, начало которому положил Ч. А. Р. Хоар (Hoare 1972), известен как метод *абстрактных моделей* (abstract models). Он формализует спецификацию вида «процедуры плюс условия» вроде описанной выше в примере с рациональными числами. Заметим, что условие на представление рациональных чисел было сформулировано в терминах утверждений о целых числах (равенство и деление). В общем случае абстрактные модели определяют новые типы объектов данных в терминах типов данных, определенных ранее. Следовательно, утверждения об объектах данных могут быть проверены путем сведения их к утверждениям об объектах данных, которые были определены ранее. Другой подход, который был введен Зиллесом из MIT, Гогеном, Тэтчером, Вагнером и Райтом из IBM (см. Thatcher, Wagner, and Wright 1978) и Гаттэгом из университета Торонто (см. Guttag 1977), называется *алгебраическая спецификация* (algebraic specification). Этот подход рассматривает «процедуры» как элементы абстрактной алгебраической системы, чье поведение определяется аксиомами, соответствующими нашим «условиям», и использует методы абстрактной алгебры для проверки утверждений об объектах данных. Оба этих метода описаны в статье Лисков и Зиллеса (Liskov and Zilles 1975).

Эта идея ярко иллюстрируется тем, что мы могли бы реализовать `cons`, `car` и `cdr` без использования каких-либо структур данных, а только при помощи одних процедур. Вот эти определения:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Аргумент не 0 или 1 -- CONS" m))))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

Такое использование процедур совершенно не соответствует нашему интуитивному понятию о том, как должны выглядеть данные. Однако для того, чтобы показать, что это законный способ представления пар, требуется только проверить, что эти процедуры удовлетворяют вышеуказанному условию.

Тонкость здесь состоит в том, чтобы заметить, что значение, возвращаемое `cons`, есть процедура, — а именно процедура `dispatch`, определенная внутри `cons`, которая принимает один аргумент и возвращает либо `x`, либо `y` в зависимости от того, равен ли ее аргумент 0 или 1. Соответственно, `(car z)` определяется как применение `z` к 0. Следовательно, если `z` есть процедура, полученная из `(cons x y)`, то `z`, примененная к 0, вернет `x`. Таким образом, мы показали, что `(car (cons x y))` возвращает `x`, как нам и хотелось. Подобным образом `(cdr (cons x y))` применяет процедуру, возвращаемую `(cons x y)`, к 1, что дает нам `y`. Следовательно, эта процедурная реализация пар законна, и если мы обращаемся к парам только с помощью `cons`, `car` и `cdr`, то мы не сможем отличить эту реализацию от такой, которая использует «настоящие» структуры данных.

Демонстрировать процедурную реализацию имеет смысл не для того, чтобы показать, как работает наш язык (Scheme, и вообще Лисп-системы, реализуют пары напрямую из соображений эффективности), а в том, чтобы показать, что он мог бы работать и так. Процедурная реализация, хотя она и выглядит трюком, — совершенно адекватный способ представления пар, поскольку она удовлетворяет единственному условию, которому должны соответствовать пары. Кроме того, этот пример показывает, что способность работать с процедурами как с объектами автоматически дает нам возможность представлять составные данные. Сейчас это может показаться курьезом, но в нашем программистском репертуаре процедурные представления данных будут играть центральную роль. Такой стиль программирования часто называют *передачей сообщений* (message passing), и в главе 3, при рассмотрении вопросов моделирования, он будет нашим основным инструментом.

Упражнение 2.4.
Вот еще одно процедурное представление для пар. Проверьте для этого представления, что при любых двух объектах `x` и `y` `(car (cons x y))` возвращает `x`.

```
(define (cons x y)
  (lambda (m) (m x y)))
```

```
(define (car z)
  (z (lambda (p q) p)))
```

Каково соответствующее определение `cdr`? (Подсказка: Чтобы проверить, что это работает, используйте подстановочную модель из раздела 1.1.5.)

Упражнение 2.5.
Покажите, что можно представлять пары неотрицательных целых чисел, используя только числа и арифметические операции, если представлять пару a и b как произведение $2^a 3^b$. Дайте соответствующие определения процедур `cons`, `car` и `cdr`.

Упражнение 2.6.
Если представление пар как процедур было для Вас еще недостаточно сумасшедшим, то заметьте, что в языке, который способен манипулировать процедурами, мы можем обойтись и без чисел (по крайней мере, пока речь идет о неотрицательных числах), определив 0 и операцию прибавления 1 так:

```
(define zero (lambda (f) (lambda (x) x)))

(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

Такое представление известно как *числа Чёрча* (Church numerals), по имени его изобретателя, Алонсо Чёрча, того самого логика, который придумал λ-исчисление.

Определите `one` (единицу) и `two` (двойку) напрямую (не через `zero` и `add-1`). (Подсказка: вычислите `(add-1 zero)` с помощью подстановки.) Дайте прямое определение процедуры сложения `+` (не в терминах повторяющегося применения `add-1`).

2.1.4. Расширенный пример: интервальная арифметика

Лиза П. Хакер проектирует систему, которая помогала бы в решении технических задач. Одна из возможностей, которые она хочет реализовать в своей системе, — способность работать с неточными величинами (например, измеренные параметры физических устройств), обладающими известной погрешностью, так что когда с такими приблизительными величинами производятся вычисления, результаты также представляют собой числа с известной погрешностью.

Инженеры-электрики будут с помощью Лизиной системы вычислять электрические величины. Иногда им требуется вычислить сопротивление R_p параллельного соединения двух резисторов R_1 и R_2 по формуле

$$R_p = \frac{1}{1/R_1 + 1/R_2}$$

Обычно сопротивления резисторов известны только с некоторой точностью, которую гарантирует их производитель. Например, покупая резистор с надписью «6.8 Ом с погрешностью 10%», Вы знаете только то, что сопротивление резистора находится между $6.8 - 0.68 = 6.12$ и $6.8 + 0.68 = 7.48$ Ом. Так что если резистор в 6.8 Ом с погрешностью 10% подключен параллельно резистору в 4.7 Ом с погрешностью 5%, то сопротивление этой комбинации может быть примерно от 2.58 Ом (если оба резистора находятся

на нижней границе интервала допустимых значений) до 2.97 Ом (если оба резистора находятся на верхней границе).

Идея Лизы состоит в том, чтобы реализовать «интервальную арифметику» как набор арифметических операций над «интервалами» (объектами, которые представляют диапазоны возможных значений неточной величины). Результатом сложения, вычитания, умножения или деления двух интервалов также будет интервал, который представляет диапазон возможных значений результата.

Лиза постулирует существование абстрактного объекта, называемого «интервал», у которого есть два конца: верхняя и нижняя границы. Кроме того, она предполагает, что имея два конца интервала, мы можем сконструировать его при помощи конструктора `make-interval`. Сначала Лиза пишет процедуру сложения двух интервалов. Она рассуждает так: минимальное возможное значение суммы равно сумме нижних границ интервалов, а максимальное возможное значение сумме верхних границ интервалов.

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))
```

Кроме того, она вычисляет произведение двух интервалов путем нахождения минимума и максимума произведений концов интервалов и использования в качестве границ интервала-результата. (`min` и `max` — примитивы, которые находят минимум и максимум при любом количестве аргументов.)

```
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                   (max p1 p2 p3 p4))))
```

При делении двух интервалов Лиза умножает первый из них на интервал, обратный второму. Заметим, что границами обратного интервала являются числа, обратные верхней и нижней границе исходного интервала, именно в таком порядке.

```
(define (div-interval x y)
  (mul-interval x
                (make-interval (/ 1.0 (upper-bound y))
                               (/ 1.0 (lower-bound y)))))
```

Упражнение 2.7.
Программа Лизы неполна, поскольку она не определила, как реализуется абстракция интервала. Вот определение конструктора интервала:

```
(define (make-interval a b) (cons a b))
```

Завершите реализацию, определив селекторы `upper-bound` и `lower-bound`.

Упражнение 2.8.
Рассуждая в духе Лизы, опишите, как можно вычислить разность двух интервалов. Напишите соответствующую процедуру вычитания, называемую `sub-interval`.

Упражнение 2.9.
Радиус (width) интервала определяется как половина расстояния между его верхней и нижней границами. Радиус является мерой неопределенности числа, которое обозначает интервал. Есть такие математические операции, для которых радиус результата зависит только от радиусов интервалов-аргументов, а есть такие, для которых радиус результата не является функцией радиусов аргументов. Покажите, что радиус суммы (или разности) двух интервалов зависит только от радиусов интервалов, которые складываются (или вычитаются). Приведите примеры, которые показывают, что для умножения или деления это не так.

Упражнение 2.10.
Бен Битобор, системный программист-эксперт, смотрит через плечо Лизы и замечает: неясно, что должно означать деление на интервал, пересекающий ноль. Модифицируйте код Лизы так, чтобы программа проверяла это условие и сообщала об ошибке, если оно возникает.

Упражнение 2.11.
Проходя мимо, Бен делает туманное замечание: «Если проверять знаки концов интервалов, можно разбить `mul-interval` на девять случаев, из которых только в одном требуется более двух умножений». Перепишите эту процедуру в соответствии с предложением Бена.

Отладив программу, Лиза показывает ее потенциальному пользователю, а тот жалуется, что она решает не ту задачу. Ему нужна программа, которая работала бы с числами, представленными в виде срединного значения и аддитивной погрешности; например, ему хочется работать с интервалами вида 3.5 ± 0.15, а не $[3.35, 3.65]$. Лиза возвращается к работе и исправляет этот недочет, добавив дополнительный конструктор и дополнительные селекторы:

```
(define (make-center-width c w)
  (make-interval (- c w) (+ c w)))

(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))

(define (width i)
  (/ (- (upper-bound i) (lower-bound i)) 2))
```

К сожалению, большая часть Лизиных пользователей — инженеры. В реальных технических задачах речь обычно идет об измерениях с небольшой погрешностью, которая измеряется как отношение радиуса интервала к его средней точке. Инженеры обычно указывают в параметрах устройств погрешность в процентах, как в спецификациях резисторов, которые мы привели в пример выше.

Упражнение 2.12.
Определите конструктор `make-center-percent`, который принимает среднее значение и погрешность в процентах и выдает требуемый интервал. Нужно также определить селектор `percent`, который для данного интервала выдает погрешность в процентах. Селектор `center` остается тем же, что приведен выше.

Упражнение 2.13.
Покажите, что, если предположить, что погрешность составляет малую долю величины интервала, то погрешность в процентах произведения двух интервалов можно получить из погрешности в процентах исходных интервалов по простой приближенной формуле. Задачу можно упростить, если предположить, что все числа положительные.

После долгой работы Лиза П. Хакер сдает систему пользователям. Несколько лет спустя, ужезабыв об этом, она получает жалобу от разгневанного пользователя Дайко Поправича. Оказывается, Дайко заметил, что формулу для параллельных резисторов можно записать двумя алгебраически эквивалентными способами:

$$\frac{R_1 R_2}{R_1 + R_2}$$

и

$$\frac{1}{1/R_1 + 1/R_2}$$

Он написал следующие две программы, каждая из которых считает формулу для параллельных резисторов своим способом:

```
(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval one
                  (add-interval (div-interval one r1)
                                (div-interval one r2)))))
```

Дайко утверждает, что для двух способов вычисления Лизина программа дает различные результаты. Это серьезное нарекание.

Упражнение 2.14.
Покажите, что Дайко прав. Исследуйте поведение системы на различных арифметических выражениях. Создайте несколько интервалов A и B и вычислите с их помощью выражения A/A и A/B. Наибольшую пользу Вы получите, если будете использовать интервалы, радиус которых составляет малую часть от среднего значения. Исследуйте результаты вычислений в форме центр/проценты (см. упражнение 2.12).

Упражнение 2.15.
Ева Лу Атор, другой пользователь Лизиной программы, тоже заметила, что алгебраически эквивалентные, но различные выражения могут давать разные результаты. Она говорит, что формула для вычисления интервалов, которая использует Лизину систему, будет давать более узкие границы погрешности, если ее удастся записать так, чтобы ни одна переменная, представляющая неточную величину, не повторялась. Таким образом, говорит она, `par2` «лучше» как программа для параллельных резисторов, чем `par1`. Права ли она? Почему?

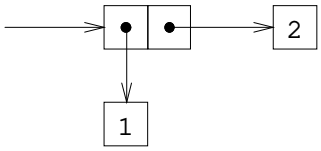


Рис. 2.2. Представление `(cons 1 2)` в виде стрелочной диаграммы.

Упражнение 2.16.
Объясните в общем случае, почему эквивалентные алгебраические выражения могут давать разные результаты. Можете ли Вы представить себе пакет для работы с интервальной арифметикой, который бы не обладал этим недостатком, или такое невозможно? (Предупреждение: эта задача очень сложна.)

2.2. Иерархические данные и свойство замыкания

Как мы уже видели, пары служат элементарным «клеем», с помощью которого можно строить составные объекты данных. На рис. 2.2 показан стандартный способ рисовать пару — в данном случае, пару, которая сформирована выражением `(cons 1 2)`. В этом представлении, которое называется *стрелочная диаграмма* (box-and-pointer notation), каждый объект изображается в виде *стрелки* (pointer), указывающей на какую-нибудь ячейку. Ячейка, изображающая элементарный объект, содержит представление этого объекта. Например, ячейка, соответствующая числу, содержит числовую константу. Изображение пары состоит из двух ячеек, причем левая из них содержит (указатель на) `car` этой пары, а правая — ее `cdr`.

Мы уже видели, что `cons` способен соединять не только числа, но и пары. (Вы использовали это свойство, или, по крайней мере, должны были использовать, когда выполняли упражнения 2.2 и 2.3). Как следствие этого, пары являются универсальным материалом, из которого можно строить любые типы структур данных. На рис. 2.3 показаны два способа соединить числа 1, 2, 3 и 4 при помощи пар.

Возможность создавать пары, элементы которых сами являются парами, определяет значимость списковой структуры как средства представления данных. Мы называем эту возможность *свойством замыкания* (closure property) для `cons`. В общем случае, операция комбинирования объектов данных обладает свойством замыкания в том случае, если результаты соединения объектов с помощью этой операции сами могут соединяться этой же операцией[6]. Замыкание — это ключ к выразительной силе для любого средства комбинирования, поскольку оно позволяет строить *иерархические* (hierarchical) структуры, то есть структуры, которые составлены из частей, которые сами составлены из частей, и так далее.

[6]Такое употребление слова «замыкание» происходит из абстрактной алгебры. Алгебраисты говорят, что множество замкнуто относительно операции, если применение операции к элементам этого множества дает результат, который также является элементом множества. К сожалению, в сообществе программистов, пишущих на Лиспе, словом «замыкание» обозначается еще и совершенно другое понятие: замыканием называют способ представления процедур, имеющих свободные переменные. В этом втором смысле мы слово «замыкание» в книге не используем.

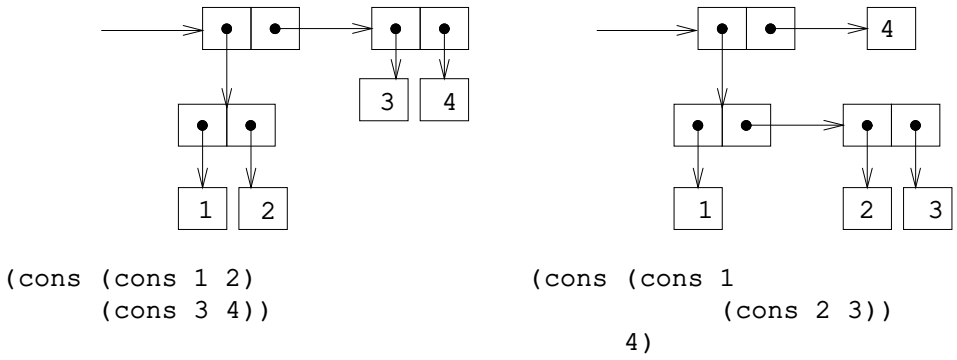


Рис. 2.3. Два способа соединить 1, 2, 3 и 4 с помощью пар.

С самого начала главы 1 мы существенным образом использовали свойство замыкания при работе с процедурами, поскольку все программы, кроме самых простых, опираются на то, что элементы комбинации сами могут быть комбинациями. В этом разделе мы рассмотрим, какое значение замыкание имеет для составных данных. Мы опишем несколько распространенных методов использования пар для представления последовательностей и деревьев, а также построим графический язык, который наглядно иллюстрирует замыкание[7].

2.2.1. Представление последовательностей

Одна из полезных структур, которые можно построить с помощью пар — это *последовательность* (sequence), то есть упорядоченная совокупность объектов данных. Разумеется, существует много способов представления последовательностей при помощи пар. Один, особенно простой, показан на рисунке 2.4, где последовательность 1, 2, 3, 4 представлена как цепочка пар. В каждой паре `car` — это соответствующий член цепочки, а `cdr` — следующая пара цепочки. `Cdr` последней пары указывает на особое значение, не являющееся парой, которое на диаграммах изображается как диагональная линия, а в программах как значение переменной `nil`. Вся последовательность порождается несколькими вложенными операциями `cons`:

[7]Идея, что средство комбинирования должно удовлетворять условию замыкания, очень проста. К сожалению, такие средства во многих популярных языках программирования либо не удовлетворяют этому условию, либо делают использование замыканий неудобным. В Фортране и Бейсике элементы данных обычно группируются путем создания массивов — но массивы, элементы которых сами являются массивами, строить нельзя. Паскаль и Си позволяют иметь структуры, члены которых являются структурами. Однако при этом требуется, чтобы программист напрямую работал с указателями и соблюдал ограничение, по которому каждое поле структуры может содержать только элементы заранее заданной формы. В отличие от Лиспа с его парами, в этих языках нет встроенного универсального клея, который позволял бы легко работать с составными данными единым способом. Это ограничение дало Алану Перлису повод сказать в предисловии к этой книге: «В Паскале обилие объявляемых структур данных ведет к специализации функций, которая сдерживает и наказывает случайное взаимодействие между ними. Лучше иметь 100 функций, которые работают с одной структурой данных, чем 10 функций, работающих с 10 структурами».

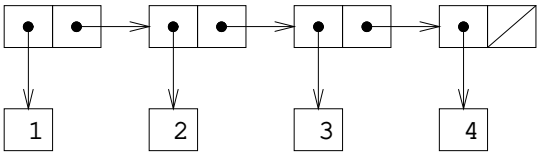


Рис. 2.4. Последовательность 1, 2, 3, 4, представленная в виде цепочки пар.

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))
```

Такая последовательность пар, порождаемая вложенными `cons`-ами, называется *список* (list). В Scheme имеется примитив, который называется `list` и помогает строить списки[8]. Вышеуказанную последовательность можно было бы получить с помощью `(list 1 2 3 4)`. В общем случае

(list ⟨a_1⟩ ⟨a_2⟩ ... ⟨a_n⟩)

эквивалентно

(cons ⟨a_1⟩ (cons⟨a_2⟩ (cons ... (cons ⟨a_n⟩ nil) ...)))

По традиции, Лисп-системы печатают списки в виде последовательности их элементов, заключенной в скобки. Таким образом, объект данных с рисунка 2.4 выводится как `(1 2 3 4)`:

```
(define one-through-four (list 1 2 3 4))

one-through-four
(1 2 3 4)
```

Внимание: не путайте выражение `(list 1 2 3 4)` со списком `(1 2 3 4)`, который является результатом вычисления этого выражения. Попытка вычислить выражение `(1 2 3 4)` приведет к сообщению об ошибке, когда интерпретатор попробует применить процедуру 1 к аргументам 1, 2, 3 и 4.

Мы можем считать, что процедура `car` выбирает первый элемент из списка, а `cdr` возвращает подсписок, состоящий из всех элементов, кроме первого. Вложенные применения `car` и `cdr` могут выбрать второй, третий и последующие элементы списка[9].

[8] В этой книге термин *список* всегда означает цепочку пар, которая завершается маркером конца списка. Напротив, термин *списковая структура* (list structure) относится к любой структуре данных, составленной из пар, а не только к спискам.

[9] Поскольку записывать вложенные применения `car` и `cdr` громоздко, в диалектах Лиспа существуют сокращения — например,

(cadr ⟨*арг*⟩) = (car (cdr ⟨*арг*⟩))

У всех таких процедур имена начинаются с `c`, а кончаются на `r`. Каждое `a` между ними означает операцию `car`, а каждое `d` операцию `cdr`, и они применяются в том же порядке, в каком идут внутри имени. Имена `car` и `cdr` сохраняются, поскольку простые их комбинации вроде `cadr` нетрудно произнести.

Конструктор `cons` порождает список, подобный исходному, но с дополнительным элементом в начале.

```
(car one-through-four)
1

(cdr one-through-four)
(2 3 4)

(car (cdr one-through-four))
2

(cons 10 one-through-four)
(10 1 2 3 4)

(cons 5 one-through-four)
(5 1 2 3 4)
```

Значение `nil`, которым завершается цепочка пар, можно рассматривать как последовательность без элементов, *пустой список* (empty list). Слово *nil* произошло от стяжения латинского *nihil*, что значит «ничто»[10].

Операции со списками

Использованию пар для представления последовательностей элементов в виде списков сопутствуют общепринятые методы программирования, которые, работая со списками, последовательно их «у`cdr`ивают». Например, процедура `list-ref` берет в качестве аргументов список и число n и возвращает n-й элемент списка. Обычно элементы списка нумеруют, начиная с 0. Метод вычисления `list-ref` следующий:

- Если $n = 0$, `list-ref` должна вернуть `car` списка.
- В остальных случаях `list-ref` должна вернуть $(n-1)$-й элемент от `cdr` списка.

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))

(define squares (list 1 4 9 16 25))
(list-ref squares 3)
16
```

[10]Удивительно, сколько энергии при стандартизации диалектов Лиспа было потрачено на споры буквально ни о чем: должно ли слово `nil` быть обычным именем? Должно ли значение `nil` являться символом? Должно ли оно являться списком? Парой? В Scheme `nil` — обычное имя, и в этом разделе мы используем его как переменную, значение которой — маркер конца списка (так же, как `true` — это обычная переменная, значение которой истина). Другие диалекты Лиспа, включая Common Lisp, рассматривают `nil` как специальный символ. Авторы этой книги пережили слишком много скандалов со стандартизацией языков и хотели бы не возвращаться к этим вопросам. Как только в разделе 2.3 мы введем кавычку, мы станем обозначать пустой список в виде `'()`, а от переменной `nil` полностью избавимся.

Часто мы прocdrиваем весь список. Чтобы помочь нам с этим, Scheme включает элементарную процедуру `null?`, которая определяет, является ли ее аргумент пустым списком. Процедура `length`, которая возвращает число элементов в списке, иллюстрирует эту характерную схему использования операций над списками:

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))

(define odds (list 1 3 5 7))

(length odds)
4
```

Процедура `length` реализует простую рекурсивную схему. Шаг редукции таков:

- Длина любого списка равняется 1 плюс длина `cdr` этого списка

Этот шаг последовательно применяется, пока мы не достигнем базового случая:

- Длина пустого списка равна 0.

Мы можем вычислить `length` и в итеративном стиле:

```
(define (length items)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter items 0))
```

Еще один распространенный программистский прием состоит в том, чтобы «cconsить» результат по ходу уcdrивания списка, как это делает процедура `append`, которая берет в качестве аргументов два списка и составляет из их элементов один общий список:

```
(append squares odds)
(1 4 9 16 25 1 3 5 7)

(append odds squares)
(1 3 5 7 1 4 9 16 25)
```

`Append` также реализуется по рекурсивной схеме. Чтобы соединить списки `list1` и `list2`, нужно сделать следующее:

- Если список `list1` пуст, то результатом является просто `list2`.
- В противном случае, нужно соединить `cdr` от `list1` с `list2`, а к результату прибавить `car` от `list1` с помощью `cons`:

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```

Упражнение 2.17.
Определите процедуру `last-pair`, которая возвращает список, содержащий только последний элемент данного (непустого) списка.

```
(last-pair (list 23 72 149 34))
(34)
```

Упражнение 2.18.
Определите процедуру `reverse`, которая принимает список как аргумент и возвращает список, состоящий из тех же элементов в обратном порядке:

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

Упражнение 2.19.
Рассмотрим программу подсчета способов размена из раздела 1.2.2. Было бы приятно иметь возможность легко изменять валюту, которую эта программа использует, так, чтобы можно было, например, вычислить, сколькими способами можно разменять британский фунт. Эта программа написана так, что знание о валюте распределено между процедурами `first-denomination` и `count-change` (которая знает, что существует пять видов американских монет). Приятнее было бы иметь возможность просто задавать список монет, которые можно использовать при размене.

Мы хотим переписать процедуру `cc` так, чтобы ее вторым аргументом был список монет, а не целое число, которое указывает, какие монеты использовать. Тогда у нас могли бы быть списки, определяющие типы валют:

```
(define us-coins (list 50 25 10 5 1))

(define uk-coins (list 100 50 20 10 5 2 1 0.5))
```

Можно было бы вызывать `cc` следующим образом:

```
(cc 100 us-coins)
292
```

Это потребует некоторых изменений в программе `cc`. Ее форма останется прежней, но со вторым аргументом она будет работать иначе, вот так:

```
(define (cc amount coin-values)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (no-more? coin-values)) 0)
        (else
         (+ (cc amount
                (except-first-denomination coin-values))
            (cc (- amount
                   (first-denomination coin-values))
                coin-values)))))
```

Определите процедуры `first-denomination`, `except-first-denomination` и `no-more?` в терминах элементарных операций над списковыми структурами. Влияет ли порядок списка `coin-values` на результат, получаемый `cc`? Почему?

Упражнение 2.20.
Процедуры `+`, `*` и `list` принимают произвольное число аргументов. Один из способов определения таких процедур состоит в использовании *точечной записи* (dotted-tail notation). В определении процедуры список параметров с точкой перед именем последнего члена означает, что, когда процедура вызывается, начальные параметры (если они есть) будут иметь в качестве значений начальные аргументы, как и обычно, но значением последнего параметра будет *список* всех оставшихся аргументов. Например, если дано определение

```
(define (f x y . z) ⟨тело⟩)
```

то процедуру `f` можно вызывать с двумя и более аргументами. Если мы вычисляем

```
(f 1 2 3 4 5 6)
```

то в теле `f` переменная `x` будет равна 1, `y` будет равно 2, а `z` будет списком (3 4 5 6). Если дано определение

```
(define (g . w) ⟨тело⟩)
```

то процедура `g` может вызываться с нулем и более аргументов. Если мы вычислим

```
(g 1 2 3 4 5 6)
```

то в теле `g` значением переменной `w` будет список `(1 2 3 4 5 6)`[11].

Используя эту нотацию, напишите процедуру `same-parity`, которая принимает одно или более целое число и возвращает список всех тех аргументов, у которых четность та же, что у первого аргумента. Например,

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)

(same-parity 2 3 4 5 6 7)
(2 4 6)
```

Отображение списков

Крайне полезной операцией является применение какого-либо преобразования к каждому элементу списка и порождение списка результатов. Например, следующая процедура умножает каждый элемент списка на заданное число.

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))

(scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

[11] Для того, чтобы определить `f` и `g` при помощи `lambda`, надо было бы написать

```
(define f (lambda (x y . z) ⟨тело⟩))
(define g (lambda w ⟨тело⟩))
```

Мы можем выделить здесь общую идею и зафиксировать ее как схему, выраженную в виде процедуры высшего порядка, в точности как в разделе 1.3. Здесь эта процедура высшего порядка называется `map`. `Map` берет в качестве аргументов процедуру от одного аргумента и список, а возвращает список результатов, полученных применением процедуры к каждому элементу списка[12]:

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))

(map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)

(map (lambda (x) (* x x))
     (list 1 2 3 4))
(1 4 9 16)
```

Теперь мы можем дать новое определение `scale-list` через `map`:

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))
```

`Map` является важным конструктом, не только потому, что она фиксирует общую схему, но и потому, что она повышает уровень абстракции при работе со списками. В исходном определении `scale-list` рекурсивная структура программы привлекает внимание к поэлементной обработке списка. Определение `scale-list` через `map` устраняет этот уровень деталей и подчеркивает, что умножение преобразует список элементов в список результатов. Разница между этими двумя определениями состоит не в том, что компьютер выполняет другой процесс (это не так), а в том, что мы думаем об этом процессе по-другому. В сущности, `map` помогает установить барьер абстракции, который отделяет реализацию процедур, преобразующих списки, от деталей того, как выбираются и комбинируются элементы списков. Подобно барьерам на рисунке 2.1, эта абстракция позволяет нам свободно изменять низкоуровневые детали того, как реализованы списки, сохраняя концептуальную схему с операциями, переводящими одни последовательности

[12] Стандартная Scheme содержит более общую процедуру `map`, чем описанная здесь. Этот вариант `map` принимает процедуру от n аргументов и n списков и применяет процедуру ко всем первым элементам списков, всем вторым элементам списков и так далее. Возвращается список результатов. Например:

```
(map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)

(map (lambda (x y) (+ x (* 2 y)))
     (list 1 2 3)
     (list 4 5 6))
(9 12 15)
```

в другие. В разделе 2.2.3 такое использование последовательностей как способ организации программ рассматривается более подробно.

Упражнение 2.21.
Процедура `square-list` принимает в качестве аргумента список чисел и возвращает список квадратов этих чисел.

```
(square-list (list 1 2 3 4))
(1 4 9 16)
```

Перед Вами два различных определения `square-list`. Закончите их, вставив пропущенные выражения:

```
(define (square-list items)
  (if (null? items)
      nil
      (cons ⟨??⟩ ⟨??⟩)))

(define (square-list items)
  (map ⟨??⟩ ⟨??⟩))
```

Упражнение 2.22.
Хьюго Дум пытается переписать первую из процедур `square-list` из упражнения 2.21 так, чтобы она работала как итеративный процесс:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

К сожалению, такое определение `square-list` выдает список результатов в порядке, обратном желаемому. Почему?
Затем Хьюго пытается исправить ошибку, обменяв аргументы `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

И так программа тоже не работает. Объясните это.

Упражнение 2.23.
Процедура `for-each` похожа на `map`. В качестве аргументов она принимает процедуру и список элементов. Однако вместо того, чтобы формировать список результатов, `for-each` просто

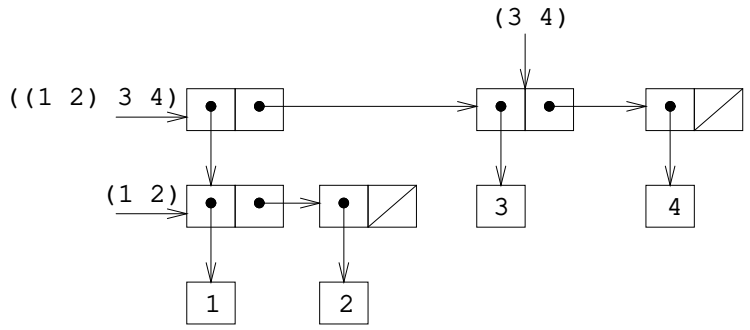


Рис. 2.5. Структура, формируемая `(cons (list 1 2) (list 3 4))`

применяет процедуру по очереди ко всем элементам слева направо. Результаты применения процедуры к аргументам не используются вообще — `for-each` применяют к процедурам, которые осуществляют какое-либо действие вроде печати. Например,

```
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
```

Значение, возвращаемое вызовом `for-each` (оно в листинге не показано) может быть каким угодно, например истина. Напишите реализацию `for-each`.

2.2.2. Иерархические структуры

Представление последовательностей в виде списков естественно распространить на последовательности, элементы которых сами могут быть последовательностями. Например, мы можем рассматривать объект `((1 2) 3 4)`, получаемый с помощью

```
(cons (list 1 2) (list 3 4))
```

как список с тремя членами, первый из которых сам является списком. В сущности, это подсказывается формой, в которой результат печатается интерпретатором. Рисунок 2.5 показывает представление этой структуры в терминах пар.

Еще один способ думать о последовательностях последовательностей — *деревья* (trees). Элементы последовательности являются ветвями дерева, а элементы, которые сами по себе последовательности — поддеревьями. Рисунок 2.6 показывает структуру, изображенную на рис. 2.5, в виде дерева.

Естественным инструментом для работы с деревьями является рекурсия, поскольку часто можно свести операции над деревьями к операциям над их ветвями, которые сами сводятся к операциям над ветвями ветвей, и так далее, пока мы не достигнем листьев дерева. Например, сравним процедуру `length` из раздела 2.2.1 с процедурой `count-leaves`, которая подсчитывает число листьев дерева:

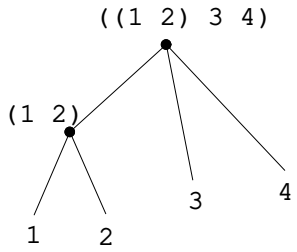


Рис. 2.6. Списковая структура с рис. 2.5, рассматриваемая как дерево.

```
(define x (cons (list 1 2) (list 3 4)))

(length x)
3

(count-leaves x)
4

(list x x)
(((1 2) 3 4) ((1 2) 3 4))

(length (list x x))
2

(count-leaves (list x x))
8
```

Чтобы реализовать `count-leaves`, вспомним рекурсивную схему вычисления `length`:

- Длина списка `x` есть 1 плюс длина `cdr` от `x`.
- Длина пустого списка есть 0.

`Count-leaves` очень похожа на эту схему. Значение для пустого списка остается тем же:

- `Count-leaves` от пустого списка равна 0.

Однако в шаге редукции, когда мы выделяем `car` списка, нам нужно учесть, что `car` сам по себе может быть деревом, листья которого нам требуется сосчитать. Таким образом, шаг редукции таков:

- `Count-leaves` от дерева `x` есть `count-leaves` от `(car x)` плюс `count-leaves` от `(cdr x)`.

Наконец, вычисляя `car`-ы, мы достигаем листьев, так что нам требуется еще один базовый случай:

- `Count-leaves` от листа равна 1.

Писать рекурсивные процедуры над деревьями в Scheme помогает элементарный предикат `pair?`, который проверяет, является ли его аргумент парой. Вот процедура целиком[13]:

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

Упражнение 2.24.
Предположим, мы вычисляем выражение `(list 1 (list 2 (list 3 4)))`. Укажите, какой результат напечатает интерпретатор, изобразите его в виде стрелочной диаграммы, а также его интерпретацию в виде дерева (как на рисунке 2.6).

Упражнение 2.25.
Укажите комбинации `car` и `cdr`, которые извлекают 7 из следующих списков:

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

Упражнение 2.26.
Допустим, мы определили `x` и `y` как два списка:

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

Какой результат напечатает интерпретатор в ответ на следующие выражения:

```
(append x y)
(cons x y)
(list x y)
```

Упражнение 2.27.
Измените свою процедуру `reverse` из упражнения 2.18 так, чтобы получилась процедура `deep-reverse`, которая принимает список в качестве аргумента и возвращает в качестве значения список, где порядок элементов обратный и подсписки также обращены. Например:

```
(define x (list (list 1 2) (list 3 4)))

x
((1 2) (3 4))

(reverse x)
((3 4) (1 2))

(deep-reverse x)
((4 3) (2 1))
```

[13] Порядок первых двух ветвей существен, поскольку пустой список удовлетворяет предикату `null?` и при этом не является парой.

Упражнение 2.28.
Напишите процедуру `fringe`, которая берет в качестве аргумента дерево (представленное в виде списка) и возвращает список, элементы которого — все листья дерева, упорядоченные слева направо. Например,

```
(define x (list (list 1 2) (list 3 4)))

(fringe x)
(1 2 3 4)

(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

Упражнение 2.29.
Бинарный мобиль состоит из двух ветвей, левой и правой. Каждая ветвь представляет собой стержень определенной длины, с которого свисает либо гирька, либо еще один бинарный мобиль. Мы можем представить бинарный мобиль в виде составных данных, соединив две ветви (например, с помощью `list`):

```
(define (make-mobile left right)
  (list left right))
```

Ветвь составляется из длины `length` (которая должна быть числом) и структуры `structure`, которая может быть либо числом (представляющим простую гирьку), либо еще одним мобилем:

```
(define (make-branch length structure)
  (list length structure))
```

а. Напишите соответствующие селекторы `left-branch` и `right-branch`, которые возвращают левую и правую ветви мобиля, а также `branch-length` и `branch-structure`, которые возвращают компоненты ветви.

б. С помощью этих селекторов напишите процедуру `total-weight`, которая возвращает общий вес мобиля.

в. Говорят, что мобиль *сбалансирован*, если момент вращения, действующий на его левую ветвь, равен моменту вращения, действующему на правую ветвь (то есть длина левого стержня, умноженная на вес груза, свисающего с него, равна соответствующему произведению для правой стороны), и если все подмобили, свисающие с его ветвей, также сбалансированы. Напишите предикат, который проверяет мобили на сбалансированность.

г. Допустим, мы изменили представление мобилей, так что конструкторы теперь приняли такой вид:

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

Как много Вам нужно изменить в программах, чтобы перейти на новое представление?

Отображение деревьев

Подобно тому, как `map` может служить мощной абстракцией для работы с последовательностями, `map`, совмещенная с рекурсией, служит мощной абстракцией для работы с деревьями. Например, процедура `scale-tree`, аналогичная процедуре `scale-list` из раздела 2.2.1, принимает в качестве аргумента числовой множитель и дерево, листьями которого являются числа. Она возвращает дерево той же формы, где каждое число умножено на множитель. Рекурсивная схема `scale-tree` похожа на схему `count-leaves`:

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))

(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))
            10)
(10 (20 (30 40) 50) (60 70))
```

Другой способ реализации `scale-tree` состоит в том, чтобы рассматривать дерево как последовательность поддеревьев и использовать `map`. Мы отображаем последовательность, масштабируя по очереди каждое поддерево, и возвращаем список результатов. В базовом случае, когда дерево является листом, мы просто умножаем:

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (scale-tree sub-tree factor)
             (* sub-tree factor)))
       tree))
```

Многие операции над деревьями могут быть реализованы с помощью такого сочетания операций над последовательностями и рекурсии.

Упражнение 2.30.
Определите процедуру `square-tree`, подобную процедуре `square-list` из упражнения 2.21. А именно, `square-tree` должна вести себя следующим образом:

```
(square-tree
 (list 1
       (list 2 (list 3 4) 5)
       (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Определите `square-tree` как прямо (то есть без использования процедур высших порядков), так и с помощью `map` и рекурсии.

Упражнение 2.31.
Абстрагируйте свой ответ на упражнение 2.30, получая процедуру `tree-map`, так, чтобы `square-tree` можно было определить следующим образом:

```
(define (square-tree tree) (tree-map square tree))
```

Упражнение 2.32.
Множество можно представить как список его различных элементов, а множество его подмножеств как список списков. Например, если множество равно `(1 2 3)`, то множество его подмножеств равно `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Закончите следующее определение процедуры, которая порождает множество подмножеств и дайте ясное объяснение, почему она работает:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map ⟨??⟩ rest)))))
```

2.2.3. Последовательности как стандартные интерфейсы

При работе с составными данными мы подчеркивали, что абстракция позволяет проектировать программы, не увязая в деталях представления данных, и оставляет возможность экспериментировать с различными способами представления. В этом разделе мы представляем еще один мощный принцип проектирования для работы со структурами данных — использование *стандартных интерфейсов* (conventional interfaces).

В разделе 1.3 мы видели, как абстракции, реализованные в виде процедур высших порядков, способны выразить общие схемы программ, которые работают с числовыми данными. Наша способность формулировать подобные операции с составными данными существенным образом зависит от того, в каком стиле мы манипулируем своими структурами данных. Например, рассмотрим следующую процедуру, аналогичную `count-leaves` из раздела 2.2.2. Она принимает в качестве аргумента дерево и вычисляет сумму квадратов тех из его листьев, которые являются нечетными числами:

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

При поверхностном взгляде кажется, что эта процедура очень сильно отличается от следующей, которая строит список всех четных чисел Фибоначчи $\mathrm{Fib}(k)$, где k меньше или равно данного целого числа n:

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

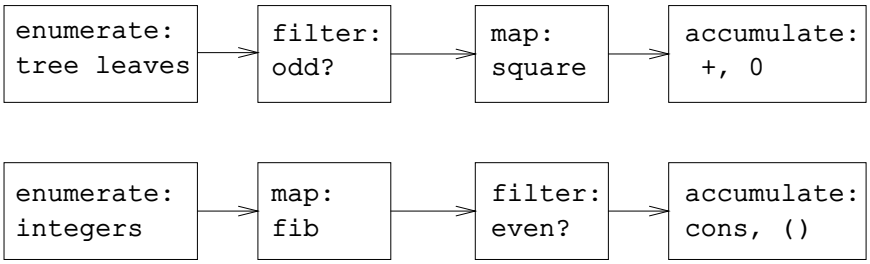


Рис. 2.7. Диаграммы потока сигналов для процедур `sum-odd-squares` (сверху) и `even-fibs` (снизу) раскрывают схожесть этих двух программ.

Несмотря на то, что структурно эти процедуры весьма различны, более абстрактное описание двух процессов вычисления раскрывает немалую долю сходства. Первая программа

- перечисляет листья дерева;
- просеивает их, отбирая нечетные;
- возводит в квадрат каждое из отобранных чисел; и
- накапливает результаты при помощи +, начиная с 0.

Вторая программа

- перечисляет числа от 1 до n;
- вычисляет для каждого из них число Фибоначчи;
- просеивает их, выбирая нечетные; и
- собирает их с помощью `cons`, начиная с пустого списка.

Специалисту по обработке сигналов покажется естественным выразить эти процессы в терминах сигналов, проходящих через ряд стадий, каждая из которых реализует часть плана программы, как это показано на рисунке 2.7. В процедуре `sum-odd-squares` мы начинаем с *перечислителя* (enumerator), который порождает «сигнал», состоящий из листьев данного дерева. Этот сигнал пропускается через *фильтр* (filter), который удаляет все элементы, кроме нечетных. Получившийся после этого сигнал, в свою очередь, проходит *отображение* (map), которое представляет собой «преобразователь», применяющий к каждому элементу процедуру `square`. Наконец, выход отображения идет в *накопитель* (accumulator), который собирает элементы при помощи +, начиная с 0. Для `even-fibs` план аналогичен.

К сожалению, два определения процедур, приведенные выше, не отражают эту структуру потока сигналов. Например, если мы рассмотрим `sum-oddsquares`, мы обнаружим, что перечисление отчасти реализуется проверками `null?` и `pair?`, а отчасти древовидно-рекурсивной структурой процедуры. Подобным образом, накопление отчасти происходит в проверках, а отчасти в сложении, которое выполняется при рекурсивном

вызове. Вообще, никакая отдельная часть этих процедур не соответствует элементу потоковой диаграммы. Наши две процедуры дробят вычисление другим образом, раскидывая перечисление по программе и смешивая его с отображением, просеиванием и накоплением. Если бы мы смогли организовать свои программы так, чтобы структура обработки потока сигналов была ясно видна в написанных нами процедурах, то это сделало бы смысл получаемого кода более прозрачным.

Операции над последовательностями

Итак, наши программы должны яснее отражать структуру потока сигналов. Ключевым моментом здесь будет перенос внимания на «сигналы», которые передаются от одной стадии процесса к другой. Если мы представим эти сигналы в виде списков, то сможем использовать операции над списками, чтобы реализовать обработку на каждом этапе. Например, мы можем реализовать стадии отображения из диаграмм потоков сигналов с помощью процедуры `map` из раздела 2.2.1:

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

Просеивание последовательности, выбирающее только те элементы, которые удовлетворяют данному предикату, осуществляется при помощи

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

Например,

```
(filter odd? (list 1 2 3 4 5))
(1 3 5)
```

Накопление осуществляется посредством

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(accumulate + 0 (list 1 2 3 4 5))
15
(accumulate * 1 (list 1 2 3 4 5))
120
(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)
```

Чтобы реализовать диаграммы потока сигналов, нам остается только перечислить последовательности элементов, с которыми мы будем работать. Для `even-fibs` нужно породить последовательность целых чисел в заданном диапазоне. Это можно сделать так:

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))

(enumerate-interval 2 7)
(2 3 4 5 6 7)
```

Чтобы перечислить листья дерева, можно использовать такую процедуру[14]:

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))

(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

Теперь мы можем переформулировать `sum-odd-squares` и `even-fibs` соответственно тому, как они изображены на диаграммах потока сигналов. В случае `sum-odd-squares` мы вычисляем последовательность листьев дерева, фильтруем ее, оставляя только нечетные числа, возводим каждый элемент в квадрат и суммируем результаты:

```
(define (sum-odd-squares tree)
  (accumulate +
              0
              (map square
                   (filter odd?
                           (enumerate-tree tree)))))
```

В случае с `even-fibs` мы перечисляем числа от 0 до n, порождаем для каждого из них число Фибоначчи, фильтруем получаемую последовательность, оставляя только четные элементы, и собираем результаты в список:

```
(define (even-fibs n)
  (accumulate cons
              nil
              (filter even?
                      (map fib
                           (enumerate-interval 0 n)))))
```

[14]Это в точности процедура `fringe` из упражнения 2.28. Здесь мы ее переименовали, чтобы подчеркнуть, что она входит в семейство общих процедур обработки последовательностей.

Польза от выражения программ в виде операций над последовательностями состоит в том, что эта стратегия помогает нам строить модульные проекты программ, то есть проекты, которые получаются путем сборки из относительно независимых частей. Можно поощрять модульное проектирование, давая разработчику набор стандартных компонент и унифицированный интерфейс, предназначенный для гибкого соединения этих компонентов.

Модульное построение является мощной стратегией управления сложностью в инженерном проектировании. Например, в реальных приложениях по обработке сигналов проектировщики обычно строят системы путем каскадирования элементов, которые выбираются из стандартизованных семейств фильтров и преобразователей. Подобным образом операции над последовательностями составляют библиотеку стандартных элементов, которые мы можем связывать и смешивать. К примеру, можно составить куски из процедур `sum-odd-squares` и `even-fibs` и получить программу, которая строит список квадратов первых `n+1` чисел Фибоначчи:

```
(define (list-fib-squares n)
  (accumulate cons
              nil
              (map square
                   (map fib
                        (enumerate-interval 0 n)))))

(list-fib-squares 10)
(0 1 1 4 9 25 64 169 441 1156 3025)
```

Можно переставить куски и использовать их, чтобы вычислить произведение квадратов нечетных чисел в последовательности:

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate *
              1
              (map square
                   (filter odd? sequence))))

(product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```

Часто встречающиеся приложения по обработке данных можно также формулировать в терминах операций над последовательностями. Допустим, у нас есть последовательность записей о служащих, и нам требуется найти зарплату самого высокооплачиваемого программиста. Пусть у нас будет селектор `salary`, который возвращает зарплату служащего, и предикат `programmer?`, который проверяет, относится ли запись к программисту. Тогда мы можем написать:

```
(define (salary-of-highest-paid-programmer records)
  (accumulate max
              0
              (map salary
                   (filter programmer? records))))
```

Все эти примеры дают лишь слабое представление об огромной области задач, выразимых в виде операций над последовательностями[15].

Последовательности, здесь реализованные в виде списков, служат стандартным интерфейсом, который позволяет комбинировать обрабатывающие модули. Кроме того, если мы представляем все структуры единым образом как последовательности, то нам удается локализовать зависимость структур данных в своих программах в небольшом наборе операций с последовательностями. Изменяя эти последние, мы можем экспериментировать с различными способами представления последовательностей, оставляя неприкосновенной общую структуру своих программ. Этой возможностью мы воспользуемся в разделе 3.5, когда обобщим парадигму обработки последовательностей и введем бесконечные последовательности.

Упражнение 2.33.
Заполните пропущенные выражения, так, чтобы получились определения некоторых базовых операций по работе со списками в виде накопления:

```
(define (map p sequence)
  (accumulate (lambda (x y) ⟨??⟩) nil sequence))

(define (append seq1 seq2)
  (accumulate cons ⟨??⟩ ⟨??⟩))

(define (length sequence)
  (accumulate ⟨??⟩ 0 sequence))
```

Упражнение 2.34.
Вычисление многочлена с переменной x при данном значении x можно сформулировать в виде накопления. Мы вычисляем многочлен

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

по известному алгоритму, называемому *схема Горнера* (Horner's rule), которое переписывает формулу в виде

$$(\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0)$$

Другими словами, мы начинаем с a_n, умножаем его на x, и так далее, пока не достигнем a_0[16]. Заполните пропуски в следующей заготовке так, чтобы получить процедуру, которая вычисляет

[15]Ричард Уотерс (Waters 1979) разработал программу, которая анализирует традиционные программы на Фортране, представляя их в терминах отображений, фильтров и накоплений. Он обнаружил, что 90 процентов кода в Пакете Научных Подпрограмм на Фортране хорошо укладывается в эту парадигму. Одна из причин успеха Лиспа как языка программирования заключается в том, что списки дают стандартное средство представления упорядоченных множеств, с которыми можно работать при помощи процедур высших порядков. Язык программирования APL своей мощности и красоте во многом обязан подобному же выбору. В APL все данные выражаются как массивы, и существует универсальный и удобный набор общих операторов для всевозможных действий над массивами.

[16]Согласно Кнуту (Knuth 1981), это правило было сформулировано У. Г. Горнером в начале девятнадцатого века, но на самом деле его использовал Ньютон более чем на сто лет раньше. По схеме Горнера многочлен вычисляется с помощью меньшего количества сложений и умножений, чем при прямолинейном способе: вычислить сначала $a^n x^n$, затем добавить $a_{n-1}x^{n-1}$ и так далее. На самом деле можно доказать, что любой алгоритм для вычисления произвольных многочленов будет использовать по крайней мере столько сложений и умножений, сколько схема Горнера, и, таким образом, схема Горнера является оптимальным алгоритмом для вычисления многочленов. Это было доказано (для числа сложений) А. М. Островским в статье 1954 года,

многочлены по схеме Горнера. Предполагается, что коэффициенты многочлена представлены в виде последовательности, от a_0 до a_n.

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) ⟨??⟩)
              0
              coefficient-sequence))
```

Например, чтобы вычислить $1 + 3x + 5x^3 + x^5$ в точке $x = 2$, нужно ввести

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

Упражнение 2.35.
Переопределите `count-leaves` из раздела 2.2.2 в виде накопления:

```
(define (count-leaves t)
  (accumulate ⟨??⟩ ⟨??⟩  (map ⟨??⟩  ⟨??⟩)))
```

Упражнение 2.36.
Процедура `accumulate-n` подобна `accumulate`, только свой третий аргумент она воспринимает как последовательность последовательностей, причем предполагается, что все они содержат одинаковое количество элементов. Она применяет указанную процедуру накопления ко всем первым элементам последовательностей, вторым элементам последовательностей и так далее, и возвращает последовательность результатов. Например, если `s` есть последовательность, состоящая из четырех последовательностей, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, то значением `(accumulate-n + 0 s)` будет последовательность `(22 26 30)`. Заполните пробелы в следующем определении `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init ⟨??⟩)
            (accumulate-n op init ⟨??⟩))))
```

Упражнение 2.37.
Предположим, что мы представляем векторы $v = (v_i)$ как последовательности чисел, а матрицы $m = (m_{ij})$ как последовательности векторов (рядов матрицы). Например, матрица

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

представляется в виде последовательности `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. Имея такое представление, мы можем использовать операции над последовательностями, чтобы кратко выразить основные действия над матрицами и векторами. Эти операции (описанные в любой книге по матричной алгебре) следующие:

которая по существу заложила основы современной науки об оптимальных алгоритмах. Аналогичное утверждение для числа умножений доказал В. Я. Пан в 1966 году. Книга Бородина и Мунро (Borodin and Munro 1975) дает обзор этих результатов, а также других достижений в области оптимальных алгоритмов.

Скалярное произведение `(dot-product` v w`)` возвращает сумму $\sum_i v_i w_i$;

Произведение матрицы и вектора `(matrix-*-vector` m v`)` возвращает вектор t, где $t_i = \sum_j m_{ij} v_i$;

Произведение матриц `(matrix-*-matrix` m n`)` возвращает матрицу p, где $p_{ij} = \sum_k m_{ik} n_{kj}$

Транспозиция `(transpose` m`)` возвращает матрицу n, где $n_{ij} = m_{ji}$

Скалярное произведение мы можем определить так[17]:

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Заполните пропуски в следующих процедурах для вычисления остальных матричных операций. (Процедура `accumulate-n` описана в упражнении 2.36.)

```
(define (matrix-*-vector m v)
  (map ⟨??⟩ m))

(define (transpose mat)
  (accumulate-n ⟨??⟩ ⟨??⟩ mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map ⟨??⟩ m)))
```

Упражнение 2.38.
Процедура `accumulate` известна также как `fold-right` (правая свертка), поскольку она комбинирует первый элемент последовательности с результатом комбинирования всех элементов справа от него. Существует также процедура `fold-left` (левая свертка), которая подобна `fold-right`, но комбинирует элементы в противоположном направлении:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

Каковы значения следующих выражений?

```
(fold-right / 1 (list 1 2 3))

(fold-left / 1 (list 1 2 3))

(fold-right list nil (list 1 2 3))

(fold-left list nil (list 1 2 3))
```

[17] Это определение использует расширенную версию `map`, описанную в сноске 12.

Укажите свойство, которому должна удовлетворять `op`, чтобы для любой последовательности `fold-right` и `fold-left` давали одинаковые результаты.

Упражнение 2.39.
Закончите следующие определения `reverse` (упражнение 2.18) в терминах процедур `fold-right` и `fold-left` из упражнения 2.38.

```
(define (reverse sequence)
  (fold-right (lambda (x y) ⟨??⟩) nil sequence))
(define (reverse sequence)
  (fold-left (lambda (x y) ⟨??⟩) nil sequence))
```

Вложенные отображения

Расширив парадигму последовательностей, мы можем включить в нее многие вычисления, которые обычно выражаются с помощью вложенных циклов[18]. Рассмотрим следующую задачу: пусть дано положительное целое число n; найти все такие упорядоченные пары различных целых чисел i и j, где $1 \le j < i \le n$, что $i+j$ является простым. Например, если n равно 6, то искомые пары следующие:

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

Естественный способ организации этого вычисления состоит в том, чтобы породить последовательность всех упорядоченных пар положительных чисел, меньших n, отфильтровать ее, выбирая те пары, где сумма чисел простая, и затем для каждой пары (i, j), которая прошла через фильтр, сгенерировать тройку $(i, j, i+j)$.

Вот способ породить последовательность пар: для каждого целого $i \le n$ перечислить целые числа $j < i$, и для каждых таких i и j породить пару (i, j). В терминах операций над последовательностями, мы производим отображение последовательности `(enumerate-interval 1 n)`. Для каждого i из этой последовательности мы производим отображение последовательности `(enumerate-interval 1 (- i 1))`. Для каждого j в этой последовательности мы порождаем пару `(list i j)`. Это дает нам последовательность пар для каждого i. Скомбинировав последовательности для всех i (путем накопления через `append`), получаем необходимую нам последовательность пар[19].

```
(accumulate append
            nil
            (map (lambda (i)
                   (map (lambda (j) (list i j))
                        (enumerate-interval 1 (- i 1))))
                 (enumerate-interval 1 n)))
```

[18] Этот подход к вложенным отображениям нам показал Дэвид Тёрнер, чьи языки KRC и Миранда обладают изящным формализмом для работы с такими конструкциями. Примеры из этого раздела (см. также упражнение 2.42) адаптированы из Turner 1981. В разделе 3.5.3 мы увидим, как этот подход можно обобщить на бесконечные последовательности.

[19] Здесь мы представляем пару в виде списка из двух элементов, а не в виде лисповской пары. Иначе говоря, «пара» (i, j) представляется как `(list i j)`, а не как `(cons i j)`.

Комбинация из отображения и накопления через `append` в такого рода программах настолько обычна, что мы ее выразим как отдельную процедуру:

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

Теперь нужно отфильтровать эту последовательность пар, чтобы найти те из них, где сумма является простым числом. Предикат фильтра вызывается для каждой пары в последовательности; его аргументом является пара и он должен обращаться к элементам пары. Таким образом, предикат, который мы применяем к каждому элементу пары, таков:

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

Наконец, нужно породить последовательность результатов, отобразив отфильтрованную последовательность пар при помощи следующей процедуры, которая создает тройку, состоящую из двух элементов пары и их суммы:

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

Сочетание всех этих шагов дает нам процедуру целиком:

```
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum?
               (flatmap
                (lambda (i)
                  (map (lambda (j) (list i j))
                       (enumerate-interval 1 (- i 1))))
                (enumerate-interval 1 n)))))
```

Вложенные отображения полезны не только для таких последовательностей, которые перечисляют интервалы. Допустим, нам нужно перечислить все перестановки множества S, то есть все способы упорядочить это множество. Например, перестановки множества $\{1,2,3\}$ — это $\{1,2,3\}$, $\{1,3,2\}$, $\{2,1,3\}$, $\{2,3,1\}$, $\{3,1,2\}$ и $\{3,2,1\}$. Вот план того, как можно породить все перестановки S: Для каждого элемента x из S, нужно рекурсивно породить все множество перестановок $S-x$[20], затем добавить x к началу каждой из них. Для каждого x из S это дает множество всех перестановок S, которые начинаются с x. Комбинация всех последовательностей для всех x дает нам все перестановки S[21]:

```
(define (permutations s)
  (if (null? s)                ; пустое множество?
      (list nil)               ; последовательность,
                               ; содержащая пустое множество
```

[20] Множество $S-x$ есть множество, состоящее из всех элементов S, кроме x.

[21] Точки с запятой в коде на Scheme начинают *комментарии* (comments). Весь текст, начиная от точки с запятой и заканчивая концом строки, интерпретатор игнорирует. В этой книге мы мало используем комментарии; мы стараемся, чтобы программы документировали себя сами при помощи описательных имен переменных.

```
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

Заметим, что такая стратегия сводит задачу порождения перестановок S к задаче порождения перестановок для множества, которое меньше, чем S. В граничном случае мы добираемся до пустого списка, который представляет множество, не содержащее элементов. Для этого множества мы порождаем `(list nil)`, которое является последовательностью из одного члена, а именно множества без элементов. Процедура `remove`, которую мы используем внутри `permutations`, возвращает все элементы исходной последовательности, за исключением данного. Ее можно выразить как простой фильтр:

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
          sequence))
```

Упражнение 2.40.
Определите процедуру `unique-pairs`, которая, получая целое число n, порождает последовательность пар (i, j), таких, что $1 \leq j < i \leq n$. С помощью `unique-pairs` упростите данное выше определение `prime-sum-pairs`.

Упражнение 2.41.
Напишите процедуру, которая находит все такие упорядоченные тройки различных положительных целых чисел i, j и k, меньших или равных данному целому числу n, сумма которых равна данному числу s.

Упражнение 2.42.
В «задаче о восьми ферзях» спрашивается, как расставить на шахматной доске восемь ферзей так, чтобы ни один из них не бил другого (то есть никакие два ферзя не должны находиться на одной вертикали, горизонтали или диагонали). Одно из возможных решений показано на рисунке 2.8. Один из способов решать эту задачу состоит в том, чтобы идти поперек доски, устанавливая по ферзю в каждой вертикали. После того, как $k - 1$ ферзя мы уже разместили, нужно разместить k-го в таком месте, где он не бьет ни одного из тех, которые уже находятся на доске. Этот подход можно сформулировать рекурсивно: предположим, что мы уже породили последовательность из всех возможных способов разместить $k - 1$ ферзей на первых $k - 1$ вертикалях доски. Для каждого из этих способов мы порождаем расширенный набор позиций, добавляя ферзя на каждую горизонталь k-й вертикали. Затем эти позиции нужно отфильтровать, оставляя только те, где ферзь на k-й вертикали не бьется ни одним из остальных. Продолжая этот процесс, мы породим не просто одно решение, а все решения этой задачи.

Это решение мы реализуем в процедуре `queens`, которая возвращает последовательность решений задачи размещения n ферзей на доске $n \times n$. В процедуре `queens` есть внутренняя процедура `queen-cols`, которая возвращает последовательность всех способов разместить ферзей на первых k вертикалях доски.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
```

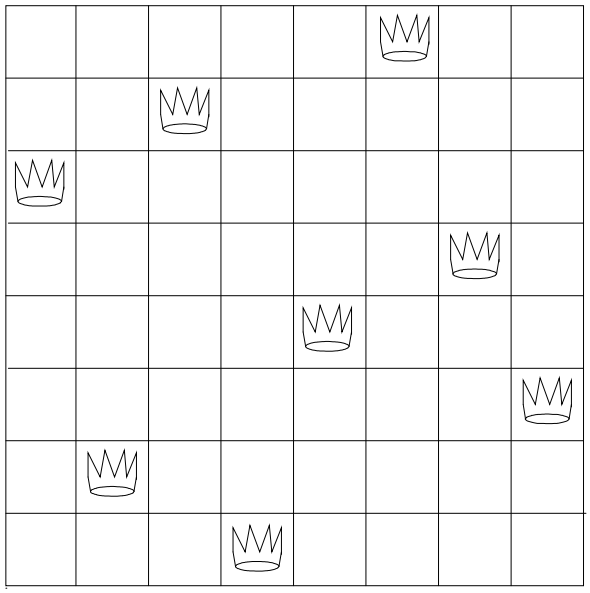


Рис. 2.8. Решение задачи о восьми ферзях.

```
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                   (adjoin-position new-row k rest-of-queens))
                 (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (queen-cols board-size))
```

В этой процедуре `rest-of-queens` есть способ размещения $k-1$ ферзя на первых $k-1$ вертикалях, а `new-row` — это горизонталь, на которую предлагается поместить ферзя с k-й вертикали. Завершите эту программу, реализовав представление множеств позиций ферзей на доске, включая процедуру `adjoin-position`, которая добавляет нового ферзя на определенных горизонтали и вертикали к заданному множеству позиций, и `empty-board`, которая представляет пустое множество позиций. Еще нужно написать процедуру `safe?`, которая для множества позиций определяет, находится ли ферзь с k-й вертикали в безопасности от остальных. (Заметим, что нам требуется проверять только то, находится ли в безопасности новый ферзь — для остальных ферзей безопасность друг от друга уже гарантирована.)

Упражнение 2.43.
У Хьюго Дума ужасные трудности при решении упражнения 2.42. Его процедура `queens` вроде бы работает, но невероятно медленно. (Хьюго ни разу не удается дождаться, пока она решит хотя

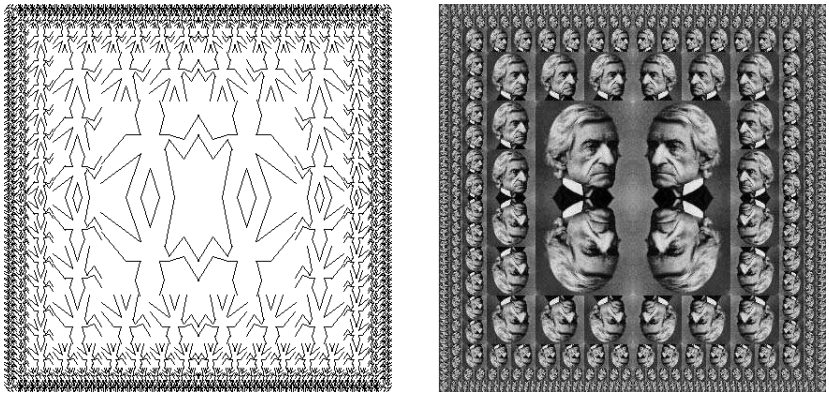


Рис. 2.9. Узоры, порождаемые языком описания изображений.

бы задачу 6×6.) Когда Хьюго просит о помощи Еву Лу Атор, она указывает, что он поменял местами порядок вложенных отображений в вызове процедуры `flatmap`, записав его в виде

```
(flatmap
  (lambda (new-row)
    (map (lambda (rest-of-queens)
           (adjoin-position new-row k rest-of-queens))
         (queen-cols (- k 1))))
  (enumerate-interval 1 board-size))
```

Объясните, почему из-за такой перестановки программа работает медленно. Оцените, насколько долго программа Хьюго будет решать задачу с восемью ферзями, если предположить, что программа, приведенная в упражнении 2.42, решает ее за время T.

2.2.4. Пример: язык описания изображений

В этой главе мы представляем простой язык для рисования картинок, иллюстрирующий силу абстракции данных и свойства замыкания; кроме того, он существенным образом опирается на процедуры высших порядков. Язык этот спроектирован так, чтобы легко было работать с узорами вроде тех, которые показаны на рисунке 2.9, составленными из элементов, которые повторяются в разных положениях и меняют размер[22]. В этом языке комбинируемые объекты данных представляются не как списковая структура, а как процедуры. Точно так же, как `cons`, которая обладает свойством замыкания, позволила нам строить списковые структуры произвольной сложности, операции этого языка, также обладающие свойством замыкания, позволяют нам строить сколь угодно сложные узоры.

[22]Этот язык описания картинок основан на языке, который создал Питер Хендерсон для построения изображений, подобных гравюре М. К. Эшера «Предел квадрата» (см. Henderson 1982). На гравюре изображен повторяющийся с уменьшением элемент, подобно картинкам, получающимся при помощи процедуры `square-limit` из этого раздела.

Язык описания изображений

Когда в разделе 1.1 мы начинали изучать программирование, мы подчеркивали важность описания языка через рассмотрение его примитивов, методов комбинирования и методов абстракции. Мы будем следовать этой схеме и здесь.

Одно из элегантных свойств языка описания изображений состоит в том, что в нем есть только один тип элементов, называемый *рисовалкой* (painter). Рисовалка рисует изображение с необходимым смещением и масштабом, чтобы попасть в указанную рамку в форме параллелограмма. Например, существует элементарная рисовалка `wave`, которая порождает грубую картинку из линий, как показано на рисунке 2.10. Форма изображения зависит от рамки — все четыре изображения на рисунке 2.10 порождены одной и той же рисовалкой `wave`, но по отношению к четырем различным рамкам. Рисовалки могут быть и более изощренными: элементарная рисовалка по имени `rogers` рисует портрет основателя MIT Уильяма Бартона Роджерса, как показано на рисунке 2.11[23]. Четыре изображения на рисунке 2.11 нарисованы относительно тех же рамок, что и картинки `wave` на рисунке 2.10.

При комбинировании изображений мы используем различные операции, которые строят новые рисовалки из рисовалок, полученных в качестве аргументов. Например, операция `beside` получает две рисовалки и порождает новую составную рисовалку, ко-

---

[23]Уильям Бартон Роджерс (1804-1882) был основателем и первым президентом MIT. Будучи геологом и способным педагогом, он преподавал в Колледже Вильгельма и Марии, а также в университете штата Виргиния. В 1859 году он переехал в Бостон, где у него было больше времени для исследований, разработал план создания «политехнического института» и служил первым Инспектором штата Массачусетс по газовым счетчикам.

Когда в 1861 году был основан MIT, Роджерс был избран его первым президентом. Роджерс исповедовал идеал «полезного обучения», отличного от университетского образования его времени с чрезмерным вниманием к классике, которое, как он писал, «стояло на пути более широкого, высокого и практического обучения и преподавания в естественных и общественных науках». Это образование должно было отличаться и от узкого образования коммерческих школ. По словам Роджерса:

> Повсеместно проводимое разделение между практическим и научным работником совершенно бесполезно, и весь опыт нашего времени показывает его полную несостоятельность.

Роджерс был президентом MIT до 1870 года, когда он ушел в отставку по состоянию здоровья. В 1878 году второй президент MIT Джон Ранкл оставил свой пост из-за финансового кризиса, вызванного биржевой паникой 1873 года, и напряженной борьбы с попытками Гарварда поглотить MIT. Роджерс вернулся и оставался на посту президента до 1881 года.

Роджерс умер от приступа во время своей речи перед студентами MIT на выпускной церемонии 1882 года. В речи, посвященной его памяти и произнесенной в том же году, Ранкл приводит последние его слова:

> «Стоя здесь и видя, чем стал Институт, ... я вспоминаю о начале научных исследований. Я вспоминаю, как сто пятьдесят лет назад Стивен Хейлс опубликовал статью на тему о светящемся газе, где он утверждал, что его исследования показали, что 128 гран битумного угля...»
>
> «Битумный уголь» — были его последние слова в этом мире. Он склонился вперед, как будто справляясь со своими заметками, которые лежали перед ним на столе, затем медленно выпрямился, поднял руки, и был перенесен со сцены своих земных забот и триумфов в «завтра смерти», где решены тайны жизни, и бестелесный дух находит неизмеримое наслаждение в созерцании новых и по-прежнему необъяснимых загадок бесконечного будущего.

По словам Фрэнсиса А. Уокера (третьего президента MIT):

> Всю свою жизнь он провел с огромной верой и героизмом, и умер так, как, наверное, и должен был желать столь превосходный рыцарь, в полном вооружении, на своем посту, и во время самого акта исполнения общественных обязанностей.

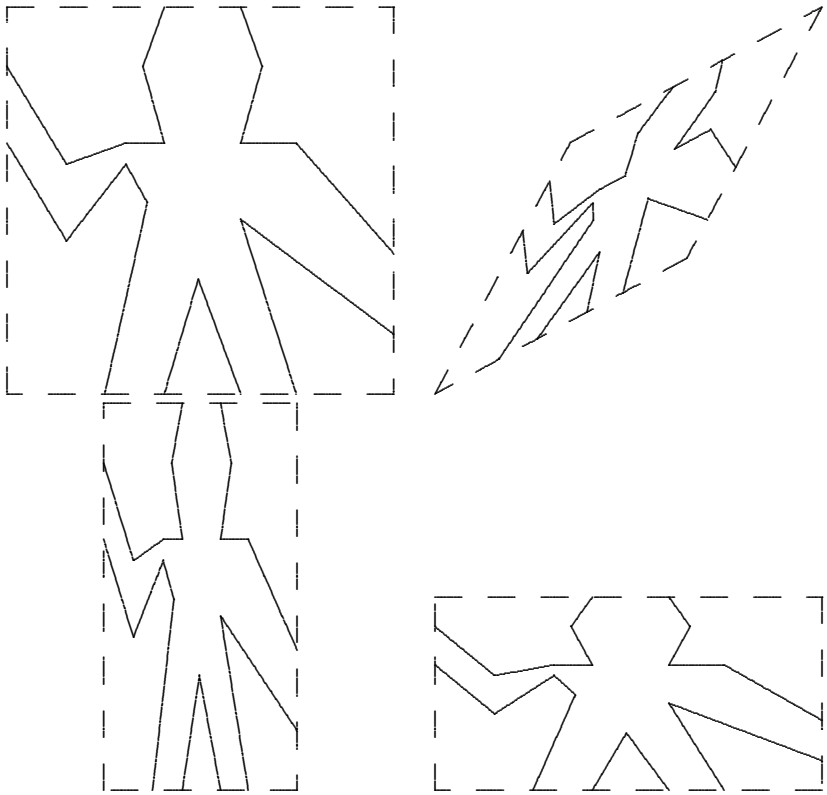


Рис. 2.10. Изображения, порожденные рисовалкой `wave` по отношению к четырем различным рамкам. Рамки, показанные пунктиром, не являются частью изображений.

торая рисует изображение первой рисовалки в левой половине рамки, а изображение второй рисовалки в правой половине рамки. Подобным образом, `below` принимает две рисовалки и порождает составную рисовалку, рисующую изображение первого аргумента под изображением второго аргумента. Некоторые операции преобразуют одну рисовалку и получают другую. Например, `flip-vert` получает рисовалку и порождает новую, рисующую изображение вверх ногами, а `flip-horiz` порождает рисовалку, рисующую изображение исходной в зеркальном отображении.

На картинке 2.12 показан результат работы рисовалки, называемой `wave4`, который строится в два этапа, начиная с `wave`:

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```

Строя таким образом составные рисовалки, мы используем тот факт, что рисовалки замкнуты относительно средств комбинирования нашего языка. `Beside` или `below` от двух рисовалок само является рисовалкой; следовательно, мы можем ее использовать как элемент при построении еще более сложных рисовалок. Так же, как при построении

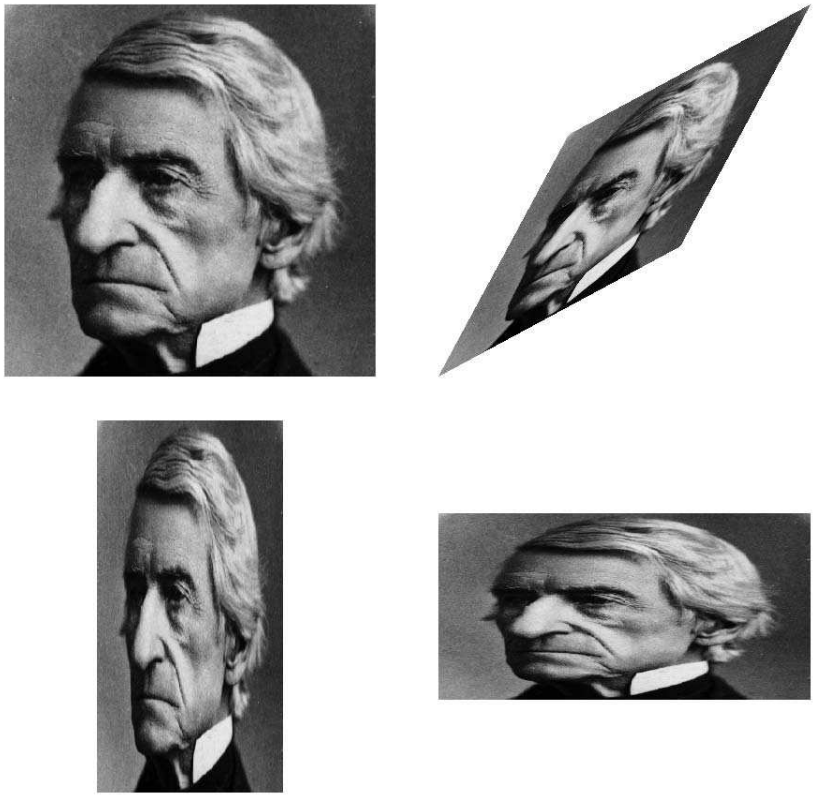
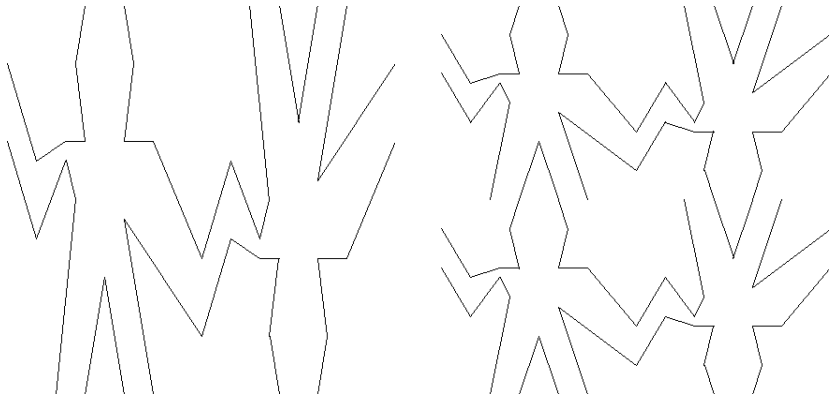


Рис. 2.11. Изображения Уильяма Бартона Роджерса, основателя и первого президента MIT, нарисованные по отношению к тем же четырем рамкам, что и на рисунке 2.10 (первоначальное изображение печатается с разрешения музея MIT).



```
(define wave2
  (beside wave (flip-vert wave)))
```

```
(define wave4
  (below wave2 wave2))
```

Рис. 2.12. Построение составного изображения, начиная с рисовалки `wave` с рисунка 2.10

списковых структур с помощью `cons`, замкнутость наших данных относительно средств комбинирования служит основой способности строить сложные структуры при помощи всего лишь нескольких операций.

Раз мы можем комбинировать рисовалки, нам хотелось бы уметь выделять типичные схемы их комбинирования. Операции над рисовалками мы реализуем как процедуры языка Scheme. Это означает, что нам в языке изображений не требуется специального механизма абстракции: поскольку средства комбинирования являются обычными процедурами Scheme, у нас автоматически есть право делать с операциями над рисовалками все то, что мы можем делать с процедурами. Например, схему построения `wave4` мы можем абстрагировать в виде

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

и определить `wave4` как пример применения этой схемы:

```
(define wave4 (flipped-pairs wave))
```

Мы можем определять и рекурсивные операции. Вот пример, который заставляет рисовалки делиться и ветвиться направо, как показано на рисунках 2.13 и 2.14:

```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```

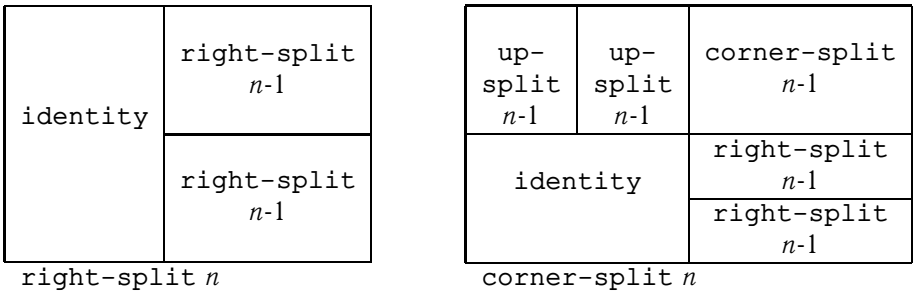


Рис. 2.13. Рекурсивные планы для `right-split` и `corner-split`.

Можно порождать сбалансированные узоры, наращивая их не только направо, но и вверх (см. упражнение 2.44 и рисунки 2.13 и 2.14):

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right corner))))))
```

Соответствующим образом расположив четыре копии `corner-split`, мы получаем схему под названием `square-limit`, применение которой к `wave` и `rogers` показано на рисунке 2.9:

```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

Упражнение 2.44.
Определите процедуру `up-split`, которую использует `corner-split`. Она подобна `right-split`, но только меняет местами роли `below` и `beside`.

Операции высших порядков

В дополнение к абстрагированию схем комбинирования рисовалок, мы можем работать и на более высоком уровне, абстрагируя схемы комбинирования операций над рисовалками. А именно, мы можем рассматривать операции над рисовалками в качестве элементов, подлежащих манипуляции, и писать средства комбинирования этих элементов — операции, которые принимают операции над рисовалками как аргументы и создают новые операции.

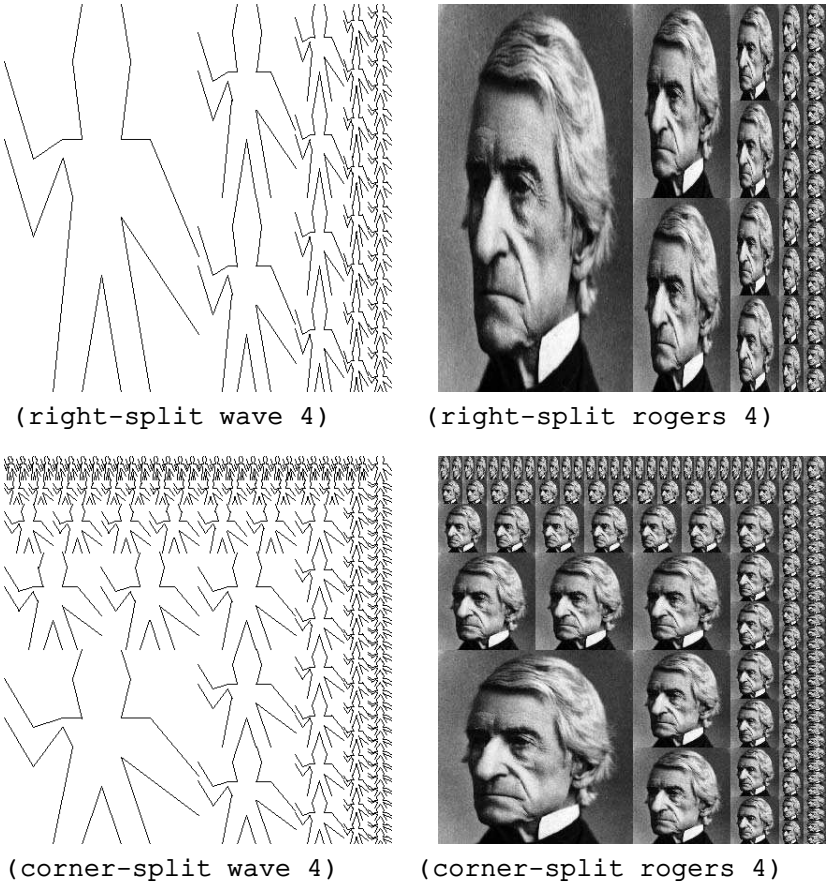


Рис. 2.14. Рекурсивные операции `right-split` и `corner-split` в применении к рисовалкам `wave` и `rogers`. Комбинирование четырех картинок `corner-split` дает симметричные узоры `square-limit`, как показано на рисунке 2.9.

Например, и `flipped-pairs`, и `square-limit` располагают определенным образом в виде квадрата четыре копии порождаемого рисовалкой изображения; они отличаются только тем, как они ориентируют эти копии. Один из способов абстрагировать такую схему комбинирования рисовалок представлен следующей процедурой, которая принимает четыре одноаргументных операции и порождает операцию над рисовалками, которая трансформирует данную ей рисовалку с помощью этих четырех операций и расставляет результаты по квадрату. `Tl`, `tr`, `bl` и `br` — это трансформации, которые следует применить к верхней левой, верхней правой, нижней левой и нижней правой копиям, соответственно.

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

Тогда в терминах `square-of-four` можно определить `flipped-pairs` следующим образом[24]:

```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                  identity flip-vert)))
    (combine4 painter)))
```

а `square-limit` можно выразить как[25]

```
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-horiz identity
                                  rotate180 flip-vert)))
    (combine4 (corner-split painter n))))
```

Упражнение 2.45.
`Right-split` и `up-split` можно выразить как разновидности общей операции разделения. Определите процедуру `split` с таким свойством, что вычисление

```
(define right-split (split beside below))
(define up-split (split below beside))
```

порождает процедуры `right-split` и `up-split` с таким же поведением, как и определенные ранее.

---

[24] Мы также могли бы написать

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

[25] `Rotate180` поворачивает рисовалку на 180 градусов (см. упражнение 2.50). Вместо `rotate180` мы могли бы сказать `(compose flip-vert flip-horiz)`, используя процедуру `compose` из упражнения 1.42.

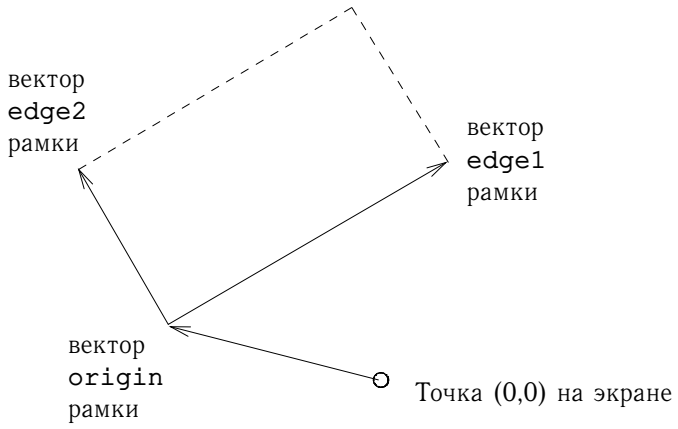


Рис. 2.15. Рамка представляется в виде трех векторов — начальной точки и двух краев.

Рамки

Прежде, чем мы сможем показать, как реализуются рисовалки и средства их комбинирования, нам нужно рассмотреть рамки. Рамку можно описать как три вектора — вектор исходной точки и два вектора краев рамки. Вектор исходной точки Origin указывает смещение исходной точки рамки от некой абсолютной начальной точки, а векторы краев Edge_1 и Edge_2 указывают смещение углов рамки от ее исходной точки. Если края перпендикулярны, рамка будет прямоугольной. В противном случае рамка будет представлять более общий случай параллелограмма. На рис. 2.15 показаны рамка и соответствующие ей вектора. В соответствии с принципами абстракции данных, нам пока незачем указывать, каким образом представляются рамки; нужно только сказать, что есть конструктор `make-frame`, который принимает три вектора и выдает рамку, и что есть еще три селектора, `origin-frame`, `edge1-frame` и `edge2-frame` (см. упражнение 2.47).

Для определения изображений мы будем использовать координаты в единичном квадрате $(0 \leq x, y \leq 1)$. Каждой рамке мы сопоставляем *отображение координат рамки* (frame coordinate map), которое будет использоваться, чтобы сдвигать и масштабировать изображения так, чтобы они умещались в рамку. Это отображение трансформирует единичный квадрат в рамку, переводя вектор $\mathbf{v} = (x, y)$ в сумму векторов

$$\mathrm{Origin}(\mathrm{Frame}) + x \cdot \mathrm{Edge}_1(\mathrm{Frame}) + y \cdot \mathrm{Edge}_2(\mathrm{Frame})$$

Например, (0, 0) отображается в исходную точку рамки, (1, 1) в вершину, противоположную исходной точке по диагонали, а (0.5, 0.5) в центр рамки. Мы можем создать отображение координат рамки при помощи следующей процедуры[26]:

```
(define (frame-coord-map frame)
  (lambda (v)
```

[26] `Frame-coord-map` использует векторные операции, определенные ниже в упражнении 2.46, и мы предполагаем, что они реализованы для какого-нибудь представления векторов. Благодаря абстракции данных, неважно, каково это представление; нужно только, чтобы операции над векторами вели себя правильно.

```
(add-vect
 (origin-frame frame)
 (add-vect (scale-vect (xcor-vect v)
                       (edge1-frame frame))
           (scale-vect (ycor-vect v)
                       (edge2-frame frame))))))
```

Заметим, что применение `frame-coord-map` к рамке дает нам процедуру, которая, получая вектор, возвращает тоже вектор. Если вектор-аргумент находится в единичном квадрате, вектор-результат окажется в рамке. Например,

```
((frame-coord-map a-frame) (make-vect 0 0))
```

возвращает тот же вектор, что и

```
(origin-frame a-frame)
```

Упражнение 2.46.
Двумерный вектор `v`, идущий от начала координат к точке, можно представить в виде пары, состоящей из x-координаты и y-координаты. Реализуйте абстракцию данных для векторов, написав конструктор `make-vect` и соответствующие селекторы `xcor-vect` и `ycor-vect`. В терминах своих селекторов и конструктора реализуйте процедуры `add-vect`, `sub-vect` и `scale-vect`, которые выполняют операции сложения, вычитания векторов и умножения вектора на скаляр:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$
$$(x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2)$$
$$s \cdot (x, y) = (sx, sy)$$

Упражнение 2.47.
Вот два варианта конструкторов для рамок:

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))

(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

К каждому из этих конструкторов добавьте соответствующие селекторы, так, чтобы получить реализацию рамок.

Рисовалки

Рисовалка представляется в виде процедуры, которая, получая в качестве аргумента рамку, рисует определенное изображение, отмасштабированное и сдвинутое так, чтобы уместиться в эту рамку. Это означает, что если есть рисовалка `p` и рамка `f`, то мы можем получить изображение, порождаемое `p`, в `f`, позвав `p` с `f` в качестве аргумента.

Детали того, как реализуются элементарные рисовалки, зависят от конкретных характеристик графической системы и типа изображения, которое надо получить. Например, пусть у нас будет процедура `draw-line`, которая рисует на экране отрезок между

двумя указанными точками. Тогда мы можем создавать из списков отрезков рисовалки для изображений, состоящих из этих отрезков, вроде рисовалки `wave` с рисунка 2.10, таким образом[27]:

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
     (lambda (segment)
       (draw-line
        ((frame-coord-map frame) (start-segment segment))
        ((frame-coord-map frame) (end-segment segment))))
     segment-list)))
```

Отрезки даются в координатах по отношению к единичному квадрату. Для каждого сегмента в списке рисовалка преобразует концы отрезка с помощью отображения координат рамки и рисует отрезок между точками с преобразованными координатами.

Представление рисовалок в виде процедур воздвигает в языке построения изображений мощный барьер абстракции. Мы можем создавать и смешивать множество типов элементарных рисовалок, в зависимости от имеющихся возможностей графики. Детали их реализации несущественны. Любая процедура, если она принимает в качестве аргумента рамку и рисует в ней что-нибудь должным образом отмасштабированное, может служить рисовалкой[28].

Упражнение 2.48.
Направленный отрезок на плоскости можно представить в виде пары векторов: вектор от начала координат до начала отрезка и вектор от начала координат до конца отрезка. Используйте свое представление векторов из упражнения 2.46 и определите представление отрезков с конструктором `make-segment` и селекторами `start-segment` и `end-segment`.

Упражнение 2.49.
С помощью `segments->painter` определите следующие элементарные рисовалки:

а. Рисовалку, которая обводит указанную рамку.

б. Рисовалку, которая рисует «X», соединяя противоположные концы рамки.

в. Рисовалку, которая рисует ромб, соединяя между собой середины сторон рамки.

г. Рисовалку `wave`.

---

[27] Процедура `segments->painter` использует представление отрезков прямых, описанное ниже в упражнении 2.48. Кроме того, она использует процедуру `for-each`, описанную в упражнении 2.23.

[28] Например, рисовалка `rogers` с рисунка 2.11 была получена из полутонового черно-белого изображения. Для каждой точки в указанной рамке рисовалка `rogers` определяет точку исходного изображения, которая в нее отображается, и соответствующим образом ее окрашивает. Разрешая себе иметь различные типы рисовалок, мы пользуемся идеей абстрактных данных, описанной в разделе 2.1.3, где мы говорили, что представление рациональных чисел может быть каким угодно, пока соблюдается соответствующее условие. Здесь мы используем то, что рисовалку можно реализовать как угодно, лишь бы она что-то изображала в указанной рамке. В разделе 2.1.3 показывается и то, как реализовать пары в виде процедур. Рисовалки — это наш второй пример процедурного представления данных.

Преобразование и комбинирование рисовалок

Операции над рисовалками (`flip-vert` или `beside`, например) создают новые рисовалки, которые вызывает исходные рисовалки по отношению к рамкам, производным от рамок-аргументов. Таким образом, скажем, `flip-vert` не требуется знать, как работает рисовалка, чтобы перевернуть ее — ей нужно только уметь перевернуть рамку вверх ногами: перевернутая рисовалка просто использует исходную, но в обращенной рамке.

Операции над рисовалками основываются на процедуре `transform-painter`, которая в качестве аргументов берет рисовалку и информацию о том, как преобразовать рамку, а возвращает новую рисовалку. Когда преобразованная рисовалка вызывается по отношению к какой-либо рамке, она преобразует рамку и вызывает исходную рисовалку по отношению к ней. Аргументами `transform-painter` служат точки (представленные в виде векторов), указывающие углы новой рамки: будучи отображенной на рамку, первая точка указывает исходную точку новой рамки, а две других — концы краевых векторов. Таким образом, аргументы, лежащие в пределах единичного квадрата, определяют рамку, которая содержится внутри исходной рамки.

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
         (make-frame new-origin
                     (sub-vect (m corner1) new-origin)
                     (sub-vect (m corner2) new-origin)))))))
```

Вот как перевернуть изображение в рамке вертикально:

```
(define (flip-vert painter)
  (transform-painter painter
                     (make-vect 0.0 1.0)   ; новая исходная точка
                     (make-vect 1.0 1.0)   ; новый конец  edge1
                     (make-vect 0.0 0.0))) ; новый конец  edge2
```

При помощи `transform-painter` нам нетрудно будет определять новые трансформации. Например, можно определить рисовалку, которая рисует уменьшенную копию исходного изображения в верхней правой четверти рамки:

```
(define (shrink-to-upper-right painter)
  (transform-painter painter
                     (make-vect 0.5 0.5)
                     (make-vect 1.0 0.5)
                     (make-vect 0.5 1.0)))
```

Вот трансформация, которая поворачивает изображение на 90 градусов против часовой стрелки[29]:

[29] `Rotate90` представляет собой чистый поворот только для квадратных рамок, поскольку она еще растягивает и сплющивает изображение так, чтобы оно уместилось в повернутой рамке.

```
(define (rotate90 painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))
```

А эта сжимает изображение по направлению к центру рамки[30]:

```
(define (squash-inwards painter)
  (transform-painter painter
                     (make-vect 0.0 0.0)
                     (make-vect 0.65 0.35)
                     (make-vect 0.35 0.65)))
```

Преобразования рамок являются также основой для определения средств комбинирования двух или более рисовалок. Например, процедура `beside` берет две рисовалки, трансформирует их так, чтобы они работали соответственно в левой и правой половинах рамки-аргумента, и создает новую составную рисовалку. Когда составной рисовалке передается рамка, она вызывает первую из преобразованных рисовалок над левой половиной рамки, а вторую над правой половиной:

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter painter1
                              (make-vect 0.0 0.0)
                              split-point
                              (make-vect 0.0 1.0)))
          (paint-right
           (transform-painter painter2
                              split-point
                              (make-vect 1.0 0.0)
                              (make-vect 0.5 1.0))))
      (lambda (frame)
        (paint-left frame)
        (paint-right frame)))))
```

Обратите внимание, как абстракция данных, и особенно представление рисовалок в виде процедур, облегчает реализацию `beside`. Процедуре `beside` не требуется ничего знать о деталях рисовалок-компонент, кроме того, что каждая из них что-то изобразит в указанной ей рамке.

Упражнение 2.50.
Определите преобразование `flip-horiz`, которое обращает изображение вокруг горизонтальной оси, а также преобразования, которые вращают рисовалки против часовой стрелки на 180 и 270 градусов.

[30] Ромбовидные изображения на рисунках 2.10 и 2.11 были получены с помощью `squash-inwards`, примененной к `wave` и `rogers`.

Упражнение 2.51.
Определите для рисовалок операцию `below`. `Below` принимает в качестве аргументов две рисовалки. Когда получившейся рисовалке передается рамка, она рисует в нижней ее половине при помощи первой рисовалки, а в верхней при помощи второй. Определите `below` двумя способами — один раз аналогично процедуре `beside`, как она приведена выше, а второй раз через `beside` и операции вращения (см. упражнение 2.50).

Уровни языка помогают устойчивому проектированию

Язык построения изображений использует некоторые из важнейших введенных нами идей, относящихся к абстракции процедур и данных. Базовая абстракция данных, рисовалки, реализуется при помощи процедурного представления, и благодаря этому наш язык может работать с различными графическими системами единым образом. Средства комбинирования обладают свойством замыкания, и это позволяет нам легко возводить сложные построения. Наконец, все средства абстракции процедур доступны нам для того, чтобы абстрагировать средства комбинирования рисовалок.

Нам удалось бросить взгляд и еще на одну существеннейшую идею касательно проектирования языков и программ. Это подход *уровневого проектирования* (stratified design), представление, что сложной системе нужно придавать структуру при помощи последовательности уровней, которая описывается последовательностью языков. Каждый из уровней строится путем комбинации частей, которые на этом уровне рассматриваются как элементарные, и части, которые строятся на каждом уровне, работают как элементарные на следующем уровне. Язык, который используется на каждом уровне такого проекта, включает примитивы, средства комбинирования и абстракции, соответствующие этому уровню подробности.

Уровневое проектирование пронизывает всю технику построения сложных систем. Например, при проектировании компьютеров резисторы и транзисторы сочетаются (и описываются при помощи языка аналоговых схем), и из них строятся и-, или- элементы и им подобные, служащие основой языка цифровых схем[31]. Из этих элементов строятся процессоры, шины и системы памяти, которые в свою очередь служат элементами в построении компьютеров при помощи языков, подходящих для описания компьютерной архитектуры. Компьютеры, сочетаясь, дают распределенные системы, которые описываются при помощи языков описания сетевых взаимодействий, и так далее.

Как миниатюрный пример уровневого подхода, наш язык описания изображений использует элементарные объекты (элементарные рисовалки), создаваемые при помощи языка, в котором описываются точки и линии и создаются списки отрезков для рисовалки `segments->painter` либо градации серого цвета в рисовалке вроде `rogers`. Большей частью наше описание языка картинок было сосредоточено на комбинировании этих примитивов с помощью геометрических комбинаторов вроде `beside` и `below`. Работали мы и на более высоком уровне, где `beside` и `below` рассматривались как примитивы, манипулируемые языком, операции которого, такие как `square-of-four`, фиксируют стандартные схемы сочетания геометрических комбинаторов.

Уровневое проектирование помогает придать программам *устойчивость* (robustness), то есть повышает вероятность, что небольшое изменение в спецификации потребует относительно малых изменений в программе. Например, предположим, что нам нужно из-

[31] Один из таких языков описывается в разделе 3.3.4.

менить картинку, основанную на рисовалке `wave`, которая показана на рисунке 2.9. Мы можем работать на самом низком уровне, изменяя конкретный вид элемента `wave`; можем работать на промежуточном уровне и менять то, как `corner-split` воспроизводит `wave`; можем на самом высоком уровне изменять то, как `square-limit` расставляет четыре копии по углам. В общем, каждый уровень такого проекта дает свой словарь для описания характеристик системы и свой тип возможных изменений.

Упражнение 2.52.
Измените предел квадрата рисовалки `wave`, показанный на рисунке 2.9, работая на каждом из вышеописанных уровней. А именно:

а. Добавьте новые отрезки к элементарной рисовалке `wave` из упражнения 2.49 (например, изобразив улыбку).

б. Измените шаблон, который порождает `corner-split` (например, используя только одну копию образов `up-split` и `right-split` вместо двух).

в. Измените версию `square-limit`, использующую `square-of-four`, так, чтобы углы компоновались как-нибудь по-другому. (Например, можно сделать так, чтобы большой мистер Роджерс выглядывал из каждого угла квадрата.)

2.3. Символьные данные

Все составные объекты данных, которые мы до сих пор использовали, состояли, в конечном счете, из чисел. В этом разделе мы расширяем возможности представления нашего языка, разрешая использовать в качестве данных произвольные символы.

2.3.1. Кавычки

Раз теперь нам можно формировать составные данные, используя символы, мы можем пользоваться списками вроде

```
(a b c d)
(23 45 17)
((Norah 12) (Molly 9) (Anna 7) (Lauren 6) (Charlotte 3))
```

Списки, содержащие символы, могут выглядеть в точности как выражения нашего языка:

```
(* (+ 23 45) (+ x 9))

(define (fact n) (if (= n 1) 1 (* n (fact (- n 1)))))
```

Чтобы работать с символами, нам в языке нужен новый элемент: способность *закавычить* (quote) объект данных. Допустим, нам хочется построить список `(a b)`. Этого нельзя добиться через `(list a b)`, поскольку это выражение строит список из *значений* символов `a` и `b`, а не из них самих. Этот вопрос хорошо изучен по отношению к естественным языкам, где слова и предложения могут рассматриваться либо как семантические единицы, либо как строки символов (синтаксические единицы). В естественных языках обычно используют кавычки, чтобы обозначить, что слово или предложение

нужно рассматривать буквально как строку символов. Например, первая буква «Джона» — разумеется, «Д». Если мы говорим кому-то «скажите, как Вас зовут», мы ожидаем услышать имя этого человека. Если же мы говорим кому-то «скажите “как Вас зовут”», то мы ожидаем услышать слова «как Вас зовут». Заметьте, как, для того, чтобы описать, что должен сказать кто-то другой, нам пришлось использовать кавычки[32].

Чтобы обозначать списки и символы, с которыми нужно обращаться как с объектами данных, а не как с выражениями, которые нужно вычислить, мы можем следовать тому же обычаю. Однако наш формат кавычек отличается от принятого в естественных языках тем, что мы ставим знак кавычки (по традиции, это символ одинарной кавычки `'`) только в начале того объекта, который надо закавычить. В Scheme это сходит нам с рук, поскольку для разделения объектов мы полагаемся на пробелы и скобки. Таким образом, значением одинарной кавычки является требование закавычить следующий объект[33].

Теперь мы можем отличать символы от их значений:

```
(define a 1)

(define b 2)

(list a b)
(1 2)

(list 'a 'b)
(a b)

(list 'a b)
(a 2)
```

Кроме того, кавычки позволяют нам вводить составные объекты, используя обычное представление для печати списков:[34].

[32]Когда мы разрешаем в языке кавычки, это разрушает нашу способность говорить о языке в простых терминах, поскольку становится неверным, что равнозначные выражения можно подставлять друг вместо друга. Например, три есть два плюс один, но слово «три» не есть слова «два плюс один». Кавычки являются мощным инструментом, поскольку они дают нам способ строить выражения, которые работают с другими выражениями (как мы убедимся в главе 4, когда станем писать интерпретатор). Однако как только мы разрешаем в языке выражения, которые говорят о других выражениях того же языка, становится очень сложно соблюдать в каком-либо виде принцип «равное можно заменить равным». Например, если мы знаем, что утренняя и вечерняя звезда — одно и то же, то из утверждения «вечерняя звезда — это Венера» мы можем заключить, что «утренняя звезда — это Венера». Однако если нам дано, что «Джон знает, что вечерняя звезда — это Венера», мы не можем заключить, что «Джон знает, что утренняя звезда — это Венера».

[33]Одинарная кавычка отличается от двойной, которую мы использовали для обозначения строк, выводимых на печать. В то время как одинарную кавычку можно использовать для обозначения списков символов, двойная кавычка используется только со строками, состоящими из печатных знаков. Единственное, для чего такие строки используются в нашей книге — это печать.

[34]Строго говоря, то, как мы используем кавычку, нарушает общее правило, что все сложные выражения нашего языка должны отмечаться скобками и выглядеть как списки. Мы можем восстановить эту закономерность, введя особую форму `quote`, которая служит тем же целям, что и кавычка. Таким образом, мы можем печатать `(quote a)` вместо `'a` и `(quote (a b c))` вместо `'(a b c)`. Именно так и работает интерпретатор. Знак кавычки — это просто сокращение, означающее, что следующее выражение нужно завернуть в форму `quote` и получить `(quote` ⟨*выражение*⟩`)`. Это важно потому, что таким образом соблюдается принцип, что с любым выражением, которое видит интерпретатор, можно обращаться как с объектом данных. Например, можно получить выражение `(car '(a b c))`, и это будет то же самое, что и `(car (quote (a b c)))`, вычислив `(list 'car (list 'quote '(a b c)))`.

```
(car '(a b c))
a

(cdr '(a b c))
(b c)
```

Действуя в том же духе, пустой список мы можем получить, вычисляя `'()`, и таким образом избавиться от переменной `nil`.

Еще один примитив, который используется при работе с символами — это `eq?`, который берет в качестве аргументов два символа и проверяет, совпадают ли они[35].

С помощью `eq?` можно реализовать полезную процедуру, называемую `memq`. Она принимает два аргумента, символ и список. Если символ не содержится в списке (то есть, не равен в смысле `eq?` ни одному из элементов списка), то `memq` возвращает ложь. В противном случае она возвращает подсписок списка, начиная с первого вхождения символа:

```
(define (memq item x)
  (cond ((null? x) false)
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

Например, значение

```
(memq 'apple '(pear banana prune))
```

есть ложь, в то время как значение

```
(memq 'apple '(x (apple sauce) y apple pear))
```

есть `(apple pear)`.

Упражнение 2.53.
Что напечатает интерпретатор в ответ на каждое из следующих выражений?

```
(list 'a 'b 'c)

(list (list 'george))

(cdr '((x1 x2) (y1 y2)))

(cadr '((x1 x2) (y1 y2)))

(pair? (car '(a short list)))

(memq 'red '((red shoes) (blue socks)))

(memq 'red '(red shoes blue socks))
```

[35] Можно считать, что два символа «совпадают», если они состоят из одних и тех же печатных знаков в одинаковом порядке. Такое определение обходит важный вопрос, который мы пока не готовы обсуждать: значение «одинаковости» в языке программирования. К нему мы вернемся в главе 3 (раздел 3.1.3).

Упражнение 2.54.
Предикат `equal?` для двух списков возвращает истину, если они содержат одни и те же элементы в одинаковом порядке. Например,

```
(equal? '(this is a list) '(this is a list))
```

истинно, но

```
(equal? '(this is a list) '(this (is a) list))
```

ложно. Более точно, можно определить `equal?` рекурсивно в терминах базового равенства символов `eq?`, сказав, что `a` равно `b`, если оба они символы и для них выполняется `eq?` либо оба они списки и при этом верно, что `(car a)` равняется в смысле `equal? (car b)`, а `(cdr a)` равняется в смысле `equal? (cdr b)`. Пользуясь этой идеей, напишите `equal?` в виде процедуры[36].

Упражнение 2.55.
Ева Лу Атор вводит при работе с интерпретатором выражение

```
(car ''abracadabra)
```

К ее удивлению, интерпретатор печатает `quote`. Объясните.

2.3.2. Пример: символьное дифференцирование

Как иллюстрацию к понятию символьной обработки, а также как дополнительный пример абстракции данных, рассмотрим построение процедуры, которая производит символьное дифференцирование алгебраических выражений. Нам хотелось бы, чтобы эта процедура принимала в качестве аргументов алгебраическое выражение и переменную, и чтобы она возвращала производную выражения по отношению к этой переменной. Например, если аргументами к процедуре служат $ax^2 + bx + c$ и x, процедура должна возвращать $2ax + b$. Символьное дифференцирование имеет для Лиспа особое историческое значение. Оно было одним из побудительных примеров при разработке компьютерного языка для обработки символов. Более того, оно послужило началом линии исследований, приведшей к разработке мощных систем для символической математической работы, которые сейчас все больше используют прикладные математики и физики.

При разработке программы для символьного дифференцирования мы будем следовать той же самой стратегии абстракции данных, согласно которой мы действовали при разработке системы рациональных чисел в разделе 2.1.1. А именно, сначала мы разработаем алгоритм дифференцирования, который работает с абстрактными объектами, такими как «суммы», «произведения» и «переменные», не обращая внимания на то, как они должны быть представлены. Только после этого мы обратимся к задаче представления.

[36] На практике программисты используют `equal?` для сравнения не только символов, но и чисел. Числа не считаются символами. Вопрос о том, выполняется ли `eq?` для двух чисел, которые равны между собой (в смысле `=`), очень сильно зависит от конкретной реализации. Более правильное определение `equal?` (например, то, которое входит в Scheme как элементарная процедура) должно содержать условие, что если и `a`, и `b` являются числами, то `equal?` для них выполняется тогда, когда они численно равны.

Программа дифференцирования с абстрактными данными

Чтобы упростить задачу, мы рассмотрим простую программу символьного дифференцирования, которая работает с выражениями, построенными только при помощи операций сложения и умножения с двумя аргументами. Дифференцировать любое такое выражение можно, применяя следующие правила редукции:

$$\frac{dc}{dx} = 0 \text{ для константы либо переменной, отличной от } x$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u(\frac{dv}{dx}) + v(\frac{du}{dx})$$

Заметим, что два последних правила по сути своей рекурсивны. То есть, чтобы получить производную суммы, нам сначала нужно получить производные слагаемых и их сложить. Каждое из них в свою очередь может быть выражением, которое требуется разложить на составляющие. Разбивая их на все более мелкие части, мы в конце концов дойдем до стадии, когда все части являются либо константами, либо переменными, и их производные будут равны либо 0, либо 1.

Чтобы воплотить эти правила в виде процедуры, мы позволим себе немного помечтать, подобно тому, как мы делали при реализации рациональных чисел. Если бы у нас был способ представления алгебраических выражений, мы могли бы проверить, является ли выражение суммой, произведением, константой или переменной. Можно было бы извлекать части выражений. Например, для суммы мы хотели бы уметь получать первое и второе слагаемое. Еще нам нужно уметь составлять выражения из частей. Давайте предположим, что у нас уже есть процедуры, которые реализуют следующие селекторы, конструкторы и предикаты:

- `(variable? e)` Является ли `e` переменной?
- `(same-variable? v1 v2)` Являются ли `v1` и `v2` одной и той же переменной?
- `(sum? e)` Является ли `e` суммой?
- `(addend e)` Первое слагаемое суммы `e`.
- `(augend e)` Второе слагаемое суммы `e`.
- `(make-sum a1 a2)` Строит сумму `a1` и `a2`.
- `(product? e)` Является ли `e` произведением?
- `(multiplier e)` Первый множитель произведения `e`.
- `(multiplicand e)` Второй множитель произведения `e`.
- `(make-product m1 m2)` Строит произведение `m1` и `m2`.

При помощи этих процедур и элементарного предиката `number?`, который распознает числа, мы можем выразить правила дифференцирования в виде следующей процедуры:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "неизвестный тип выражения -- DERIV" exp))))
```

Процедура `deriv` заключает в себе весь алгоритм дифференцирования. Поскольку она выражена в терминах абстрактных данных, она будет работать, как бы мы ни представили алгебраические выражения, если только у нас будут соответствующие селекторы и конструкторы. Именно этим вопросом нам и нужно теперь заняться.

Представление алгебраических выражений

Можно представить себе множество способов представления алгебраических выражений с помощью списковых структур. Например, можно использовать списки символов, которые отражали бы обычную алгебраическую нотацию, так что $ax + b$ представлялось бы как список `(a * x + b)`. Однако естественней всего использовать ту же скобочную префиксную запись, с помощью которой в Лиспе представляются комбинации; то есть представлять $ax + b$ в виде `(+ (* a x) b)`. Тогда наше представление данных для задачи дифференцирования будет следующим:

- Переменные — это символы. Они распознаются элементарным предикатом `symbol?`:

```
(define (variable? x) (symbol? x))
```

- Две переменные одинаковы, если для представляющих их символов выполняется `eq?`:

```
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
```

- Суммы и произведения конструируются как списки:

```
(define (make-sum a1 a2) (list '+ a1 a2))

(define (make-product m1 m2) (list '* m1 m2))
```

- Сумма — это список, первый элемент которого символ `+`:

```
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
```

- Первое слагаемое — это второй элемент списка, представляющего сумму:

```
(define (addend s) (cadr s))
```

- Второе слагаемое — это третий элемент списка, представляющего сумму:

```
(define (augend s) (caddr s))
```

- Произведение — это список, первый элемент которого символ `*`:

```
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
```

- Первый множитель — это второй элемент списка, представляющего произведение:

```
(define (multiplier p) (cadr p))
```

- Второй множитель — это третий элемент списка, представляющего произведение:

```
(define (multiplicand p) (caddr p))
```

Таким образом, нам осталось только соединить это представление с алгоритмом, заключенным в процедуре `deriv`, и мы получаем работающую программу символьного дифференцирования. Посмотрим на некоторые примеры ее поведения:

```
(deriv '(+ x 3) 'x)
(+ 1 0)

(deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))

(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
   (* (+ (* x 0) (* 1 y))
      (+ x 3)))
```

Ответы, которые выдает программа, правильны; однако их нужно упрощать. Верно, что

$$\frac{d(xy)}{dx} = x \cdot 0 + 1 \cdot y$$

но нам хотелось бы, чтобы программа знала, что $x \cdot 0 = 0$, $1 \cdot y = y$, а $0 + y = y$. Ответом на второй пример должно быть просто y. Как видно из третьего примера, при усложнении выражений упрощение превращается в серьезную проблему.

Наши теперешние затруднения очень похожи на те, с которыми мы столкнулись при реализации рациональных чисел: мы не привели ответы к простейшей форме. Чтобы произвести приведение рациональных чисел, нам потребовалось изменить только конструкторы и селекторы в нашей реализации. Здесь мы можем применить подобную же

стратегию. Процедуру `deriv` мы не будем изменять вовсе. Вместо этого мы изменим `make-sum` так, что если оба слагаемых являются числами, она их сложит и вернет их сумму. Кроме того, если одно из слагаемых равно 0, то `make-sum` вернет другое.

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
```

Здесь используется процедура `=number?`, которая проверяет, не равно ли выражение определенному числу:

```
(define (=number? exp num)
  (and (number? exp) (= exp num)))
```

Подобным же образом мы изменим и `make-product`, так. чтобы встроить в него правила, что нечто, умноженное на 0, есть 0, а умноженное на 1 равно самому себе:

```
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

Вот как эта версия работает на наших трех примерах:

```
(deriv '(+ x 3) 'x)
1

(deriv '(* x y) 'x)
y

(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

Хотя это заметное улучшение, третий пример показывает, что нужно многое еще сделать, прежде чем мы получим программу, приводящую выражения к форме, которую мы согласимся считать «простейшей». Задача алгебраического упрощения сложна, среди прочего, еще и потому, что форма, которая является простейшей для одних целей, может таковой не являться для других.

Упражнение 2.56.
Покажите, как расширить простейшую программу дифференцирования так, чтобы она воспринимала больше разных типов выражений. Например, реализуйте правило взятия производной

$$\frac{d(u^n)}{dx} = nu^{n-1}(\frac{du}{dx})$$

добавив еще одну проверку к программе `deriv` и определив соответствующие процедуры `exponentiation?`, `base`, `exponent` и `make-exponentiation` (обозначать возведение в степень можно символом

`**`). Встройте правила, что любое выражение, возведенное в степень 0, дает 1, а возведенное в степень 1 равно самому себе.

Упражнение 2.57.
Расширьте программу дифференцирования так, чтобы она работала с суммами и произведениями любого (больше двух) количества термов. Тогда последний из приведенных выше примеров мог бы быть записан как

```
(deriv '(* x y (+ x 3)) 'x)
```

Попытайтесь сделать это, изменяя только представление сумм и произведений, не трогая процедуру `deriv`. Тогда, например, процедура `addend` будет возвращать первое слагаемое суммы, а `augend` сумму остальных.

Упражнение 2.58.
Предположим, что нам захотелось изменить программу дифференцирования так, чтобы она работала с обычной математической нотацией, где `+` и `*` не префиксные, а инфиксные операции. Поскольку программа взятия производных определена в терминах абстрактных данных, мы можем изменять представление выражений, с которыми она работает, меняя только предикаты, селекторы и конструкторы, определяющие представление алгебраических выражений, с которыми должен работать дифференциатор.

а. Покажите, как это сделать так, чтобы брать производные от выражений, представленных в инфиксной форме, например `(x + (3 * (x + (y + 2))))`. Для упрощения задачи предположите, что `+` и `*` всегда принимают по два аргумента, и что в выражении расставлены все скобки.

б. Задача становится существенно сложней, если мы разрешаем стандартную алгебраическую нотацию, например `(x + 3 * (x + y + 2))`, которая опускает ненужные скобки и предполагает, что умножение выполняется раньше, чем сложение. Можете ли Вы разработать соответствующие предикаты, селекторы и конструкторы для этой нотации так, чтобы наша программа взятия производных продолжала работать?

2.3.3. Пример: представление множеств

В предыдущих примерах мы построили представления для двух типов составных объектов: для рациональных чисел и для алгебраических выражений. В одном из этих примеров перед нами стоял выбор, упрощать ли выражение при его конструировании или при обращении; в остальном же выбор представления наших структур через списки был простым делом. Когда мы обращаемся к представлению множеств, выбор представления не так очевиден. Здесь существует несколько возможных представлений, и они значительно отличаются друг от друга в нескольких аспектах.

Говоря неформально, множество есть просто набор различных объектов. Чтобы дать ему более точное определение, можно использовать метод абстракции данных. А именно, мы определяем «множество», указывая операции, которые можно производить над множествами. Это операции `union-set` (объединение), `intersection-set` (пересечение), `element-of-set?` (проверка на принадлежность) и `adjoin-set` (добавление элемента). `Element-of-set?` — это предикат, который определяет, является ли данный объект элементом множества. `Adjoin-set` принимает как аргументы объект и множество, и возвращает множество, которое содержит все элементы исходного множества плюс добавленный элемент. `Union-set` вычисляет объединение двух множеств,

то есть множество, содержащее те элементы, которые присутствуют хотя бы в одном из аргументов. `Intersection-set` вычисляет пересечение двух множеств, то есть множество, которое содержит только те элементы, которые присутствуют в обоих аргументах. С точки зрения абстракции данных, мы имеем право взять любое представление, позволяющее нам использовать эти операции способом, который согласуется с вышеуказанной интерпретацией[37].

Множества как неупорядоченные списки

Можно представить множество как список, в котором ни один элемент не содержится более одного раза. Пустое множество представляется пустым списком. При таком представлении `element-of-set?` подобен процедуре `memq` из раздела 2.3.1. Она использует не `eq?`, а `equal?`, так что элементы множества не обязаны быть символами:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
```

Используя эту процедуру, мы можем написать `adjoin-set`. Если объект, который требуется добавить, уже принадлежит множеству, мы просто возвращаем исходное множество. В противном случае мы используем `cons`, чтобы добавить объект к списку. представляющему множество:

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

Для `intersection-set` можно использовать рекурсивную стратегию. Если мы знаем, как получить пересечение `set2` и `cdr` от `set1`, нам нужно только понять, надо ли добавить к нему `car` от `set1`. Это зависит от того, принадлежит ли `(car set1)` еще и `set2`. Получается такая процедура:

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

[37] Если нам хочется быть более формальными, мы можем определить «соответствие вышеуказанной интерпретации» как условие, что операции удовлетворяют некоторому набору правил вроде следующих:

- Для любого множества `S` и любого объекта `x`, `(element-of-set? x (adjoin-set x S))` истинно (неформально: «добавление объекта к множеству дает множество, содержащее этот объект»).
- Для любых двух множеств `S` и `T` и любого объекта `x`, `(element-of-set? x (union-set S T))` равно `(or (element-of-set? x S) (element-of-set? x T))` (неформально: «элементы `(union-set S T)` — это те элементы, которые принадлежат либо `S`, либо `T`»).
- Для любого объекта `x`, `(element-of-set? x '())` ложно (неформально: «ни один объект не принадлежит пустому множеству»).

Один из вопросов, которые должны нас заботить при разработке реализации — эффективность. Рассмотрим число шагов, которые требуют наши операции над множествами. Поскольку все они используют `element-of-set?`, скорость этой операции оказывает большое влияние на скорость реализации в целом. Теперь заметим, что для того, чтобы проверить, является ли объект элементом множества, процедуре `element-of-set?` может потребоваться просмотреть весь список. (В худшем случае оказывается, что объекта в списке нет.) Следовательно, если в множестве n элементов, `element-of-set?` может затратить до n шагов. Таким образом, число требуемых шагов растет как $\Theta(n)$. Число шагов, требуемых `adjoin-set`, которая эту операцию использует, также растет как $\Theta(n)$. Для `intersection-set`, которая проделывает `element-of-set?` для каждого элемента `set1`, число требуемых шагов растет как произведение размеров исходных множеств, или $\Theta(n^2)$ для двух множеств размера n. То же будет верно и для `union-set`.

Упражнение 2.59.
Реализуйте операцию `union-set` для представления множеств в виде неупорядоченных списков.

Упражнение 2.60.
Мы указали, что множество представляется как список без повторяющихся элементов. Допустим теперь, что мы разрешаем повторяющиеся элементы. Например, множество $\{1, 2, 3\}$ могло бы быть представлено как список `(2 3 2 1 3 2 2)`. Разработайте процедуры `element-of-set?`, `adjoin-set`, `union-set` и `intersection-set`, которые бы работали с таким представлением. Как соотносится эффективность этих операций с эффективностью соответствующих процедур для представления без повторений? Существуют ли приложения, в которых Вы бы использовали скорее это представление, чем представление без повторений?

Множества как упорядоченные списки

Один из способов ускорить операции над множествами состоит в том, чтобы изменить представление таким образом, чтобы элементы множества перечислялись в порядке возрастания. Для этого нам потребуется способ сравнения объектов, так, чтобы можно было сказать, какой из них больше. Например, символы мы могли бы сравнивать лексикографически, или же мы могли бы найти какой-нибудь способ ставить каждому объекту в соответствие некоторое уникальное число и затем сравнивать объекты путем сравнения соответствующих чисел. Чтобы упростить обсуждение, мы рассмотрим только случай, когда элементами множества являются числа, так что мы сможем сравнивать элементы при помощи > и <. Мы будем представлять множество чисел как список его элементов в возрастающем порядке. В то время как первая наша реализация позволяла нам представлять множество $\{1, 3, 6, 10\}$ путем перечисления его элементов в произвольном порядке, в новом представлении разрешен только список `(1 3 6 10)`.

Одно из преимуществ упорядочения проявляется в `element-of-set?`: проверяя наличие элемента, нам больше незачем просматривать все множество. Если мы достигли элемента, который больше того объекта, который мы ищем, мы можем уже сказать, что искомого в списке нет:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
```

```
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

Сколько шагов мы на этом выигрываем? В худшем случае, объект, который мы ищем, может быть наибольшим в множестве, так что число шагов то же, что и для неупорядоченного представления. С другой стороны, если мы ищем элементы разных размеров, можно ожидать, что иногда мы сможем останавливаться близко к началу списка, а иногда нам все же потребуется просмотреть большую его часть. В среднем мы можем ожидать, что потребуется просмотреть около половины элементов множества. Таким образом, среднее число требуемых шагов будет примерно $n/2$. Это все еще рост порядка $\Theta(n)$, но это экономит нам в среднем половину числа шагов по сравнению с предыдущей реализацией.

Более впечатляющее ускорение мы получаем в `intersection-set`. При неупорядоченном представлении эта операция требовала $\Theta(n^2)$ шагов, поскольку мы производили полный поиск в `set2` для каждого элемента `set1`. Однако при упорядоченном представлении мы можем воспользоваться более разумным методом. Начнем со сравнения первых элементов двух множеств, `x1` и `x2`. Если `x1` равно `x2`, мы получаем один элемент пересечения, а остальные элементы пересечения мы можем получить, пересекая оставшиеся элементы списков-множеств. Допустим, однако, что `x1` меньше, чем `x2`. Поскольку `x2` — наименьший элемент `set2`, мы можем немедленно заключить, что `x1` больше нигде в `set2` не может встретиться и, следовательно, не принадлежит пересечению. Следовательно пересечение двух множеств равно пересечению `set2` с `cdr` от `set1`. Подобным образом, если `x2` меньше, чем `x1`, то пересечение множеств получается путем пересечения `set1` с `cdr` от `set2`. Вот процедура:

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1
                     (intersection-set (cdr set1)
                                       (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((< x2 x1)
               (intersection-set set1 (cdr set2)))))))
```

Чтобы оценить число шагов, необходимое для этого процесса, заметим, что на каждом шагу мы сводим задачу нахождения пересечения к вычислению пересечения меньших множеств — убирая первый элемент либо из `set1`, либо из `set2`, либо из обоих. Таким образом, число требуемых шагов не больше суммы размеров `set1` и `set2`, а не их произведения, как при неупорядоченном представлении. Это рост $\Theta(n)$, а не $\Theta(n^2)$ — заметное ускорение, даже для множеств небольшого размера.

Упражнение 2.61.
Напишите реализацию `adjoin-set` для упорядоченного представления. По аналогии с `element-of-set?` покажите, как использовать упорядочение, чтобы получить процедуру, которая в среднем требует только половину числа шагов, которое требуется при неупорядоченном представлении.

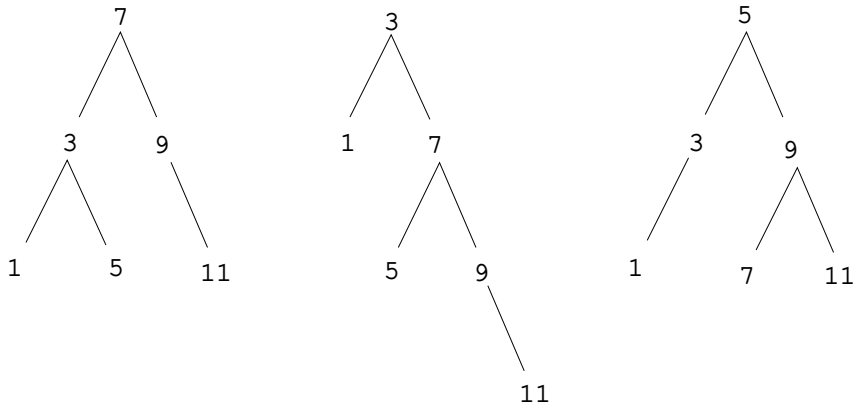


Рис. 2.16. Различные бинарные деревья, представляющие множество $\{1, 3, 5, 7, 9, 11\}$.

Упражнение 2.62.
Дайте представление порядка $\Theta(n)$ для операции `union-set` с представлением в виде упорядоченных списков.

Множества как бинарные деревья

Можно добиться еще лучших результатов, чем при представлении в виде упорядоченных списков, если расположить элементы множества в виде дерева. Каждая вершина дерева содержит один элемент множества, называемый «входом» этой вершины, и указатели (возможно, пустые) на две другие вершины. «Левый» указатель указывает на элементы, меньшие, чем тот, который содержится в вершине, а «правый» на элементы, большие, чем тот, который содержится в вершине. На рисунке 2.16 показано несколько вариантов представления множества $\{1, 3, 5, 7, 9, 11\}$ в виде дерева. Одно и то же множество может быть представлено в виде дерева несколькими различными способами. Единственное, чего мы требуем от правильного представления — это чтобы все элементы левого поддерева были меньше, чем вход вершины, а элементы правого поддерева больше.

Преимущество древовидного представления следующее. Предположим, мы хотим проверить, содержится ли в множестве число x. Начнем с того, что сравним x со входом начальной вершины. Если x меньше его, то мы уже знаем, что достаточно просмотреть только левое поддерево; если x больше, достаточно просмотреть правое поддерево. Если дерево «сбалансировано», то каждое из поддеревьев будет по размеру примерно вполовину меньше. Таким образом, за один шаг мы свели задачу поиска в дереве размера n к задаче поиска в дереве размера $n/2$. Поскольку размер дерева уменьшается вдвое на каждом шаге, следует ожидать, что число шагов, требуемых для поиска в дереве размера n, растет как $\Theta(\log n)$[38]. Для больших множеств это будет заметным ускорением по

[38]Уменьшение размера задачи вдвое на каждом шагу является определяющей характеристикой логарифмического роста, как мы видели на примере алгоритма быстрого возведения в степень в разделе 1.2.4 и метода половинного деления в разделе 1.3.3.

сравнению с предыдущими реализациями.

Деревья мы можем представлять при помощи списков. Каждая вершина будет списком из трех элементов: вход вершины, левое поддерево и правое поддерево. Пустой список на месте левого или правого поддерева будет означать, что в этом месте никакое поддерево не присоединяется. Мы можем описать это представление при помощи следующих процедур[39]:

```
(define (entry tree) (car tree))

(define (left-branch tree) (cadr tree))

(define (right-branch tree) (caddr tree))

(define (make-tree entry left right)
  (list entry left right))
```

Теперь можно написать процедуру `element-of-set?` с использованием вышеописанной стратегии:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Добавление элемента к множеству реализуется похожим образом и также требует $\Theta(\log n)$ шагов. Чтобы добавить объект `x`, мы сравниваем его с входом вершины и определяем, должны ли мы добавить `x` к левой или правой ветви, а добавив x к соответствующей ветви, мы соединяем результат с изначальным входом и второй ветвью. Если x равен входу, мы просто возвращаем вершину. Если нам требуется добавить x к пустому дереву, мы порождаем дерево, которое содержит x на входе и пустые левое и правое поддеревья. Вот процедура:

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set))))))
```

[39] Мы представляем множества при помощи деревьев, а деревья при помощи списков — получается абстракция данных на основе другой абстракции данных. Процедуры `entry`, `left-branch`, `right-branch` и `make-tree` мы можем рассматривать как способ изолировать абстракцию «бинарное дерево» от конкретного способа, которым мы желаем представить такое дерево в виде списковой структуры.

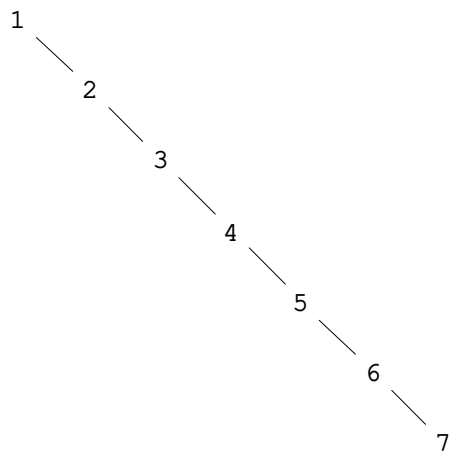


Рис. 2.17. Несбалансированное дерево, порожденное последовательным присоединением элементов от 1 до 7.

Утверждение, что поиск в дереве можно осуществить за логарифмическое число шагов, основывается на предположении, что дерево «сбалансировано», то есть что левое и правое его поддеревья содержат приблизительно одинаковое число элементов, так что каждое поддерево содержит приблизительно половину элементов своего родителя. Но как нам добиться того, чтобы те деревья, которые мы строим, были сбалансированы? Даже если мы начинаем со сбалансированного дерева, добавление элементов при помощи `adjoin-set` может дать несбалансированный результат. Поскольку позиция нового добавляемого элемента зависит от того, как этот элемент соотносится с объектами, уже содержащимися в множестве, мы имеем право ожидать, что если мы будем добавлять элементы «случайным образом», в среднем дерево будет получаться сбалансированным. Однако такой гарантии у нас нет. Например, если мы начнем с пустого множества и будем добавлять по очереди числа от 1 до 7, то получится весьма несбалансированное дерево, показанное на рисунке 2.17. В этом дереве все левые поддеревья пусты, так что нет никакого преимущества по сравнению с простым упорядоченным списком. Одним из способов решения этой проблемы было бы определение операции, которая переводит произвольное дерево в сбалансированное с теми же элементами. Тогда мы сможем проводить преобразование через каждые несколько операций `adjoin-set`, чтобы поддерживать множество в сбалансированном виде. Есть и другие способы решения этой задачи. Большая часть из них связана с разработкой новых структур данных, для которых и поиск, и вставка могут производиться за $\Theta(\log n)$ шагов[40].

[40]Примерами таких структур могут служить *B-деревья* (B-trees) и *красно-черные деревья* (red-black trees). Существует обширная литература по структурам данных, посвященная этой задаче. См. Cormen, Leiserson, and Rivest 1990.

Упражнение 2.63.
Каждая из следующих двух процедур преобразует дерево в список.

```
(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1 (right-branch tree))))))

(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list (right-branch tree)
                                          result-list)))))
  (copy-to-list tree '()))
```

а. Для всякого ли дерева эти процедуры дают одинаковый результат? Если нет, то как их результаты различаются? Какой результат дают эти две процедуры для деревьев с рисунка 2.16?

б. Одинаков ли порядок роста этих процедур по отношению к числу шагов, требуемых для преобразования сбалансированного дерева с n элементами в список? Если нет, которая из них растет медленнее?

Упражнение 2.64.
Следующая процедура `list->tree` преобразует упорядоченный список в сбалансированное бинарное дерево. Вспомогательная процедура `partial-tree` принимает в качестве аргументов целое число n и список по крайней мере из n элементов, и строит сбалансированное дерево из первых n элементов дерева. Результат, который возвращает `partial-tree`, — это пара (построенная через `cons`), `car` которой есть построенное дерево, а `cdr` — список элементов, не включенных в дерево.

```
(define (list->tree elements)
  (car (partial-tree elements (length elements))))

(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result (partial-tree (cdr non-left-elts)
                                              right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts (cdr right-result)))
                (cons (make-tree this-entry left-tree right-tree)
                      remaining-elts))))))))
```

а. Дайте краткое описание, как можно более ясно объясняющее работу `partialtree`. Нарисуйте дерево, которое `partial-tree` строит из списка `(1 3 5 7 9 11)`

б. Каков порядок роста по отношению к числу шагов, которые требуются процедуре `list->tree` для преобразования дерева из n элементов?

Упражнение 2.65.
Используя результаты упражнений 2.63 и 2.64, постройте реализации `union-set` и `intersection-set` порядка $\Theta(n)$ для множеств, реализованных как (сбалансированные) бинарные деревья[41].

Множества и поиск информации

Мы рассмотрели способы представления множеств при помощи списков и увидели, как выбор представления для объектов данных может сильно влиять на производительность программ, использующих эти данные. Еще одной причиной нашего внимания к множествам было то, что описанные здесь методы снова и снова возникают в приложениях, связанных с поиском данных.

Рассмотрим базу данных, содержащую большое количество записей, например, сведения о кадрах какой-нибудь компании или о транзакциях в торговой системе. Как правило, системы управления данными много времени проводят, занимаясь поиском и модификацией данных в записях; следовательно, им нужны эффективные методы доступа к записям. Для этого часть каждой записи выделяется как идентифицирующий *ключ* (key). Ключом может служить что угодно, что однозначно определяет запись. В случае записей о кадрах это может быть номер карточки сотрудника. Для торговой системы это может быть номер транзакции. Каков бы ни был ключ, когда мы определяем запись в виде структуры данных, нам нужно указать процедуру выборки ключа, которая возвращает ключ, связанный с данной записью.

Пусть мы представляем базу данных как множество записей. Чтобы получить запись с данным ключом, мы используем процедуру `lookup`, которая принимает как аргументы ключ и базу данных и возвращает запись, содержащую указанный ключ, либо ложь, если такой записи нет. `Lookup` реализуется почти так же, как `element-of-set?`. Например, если множество записей реализуется как неупорядоченный список, мы могли бы написать

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) false)
        ((equal? given-key (key (car set-of-records)))
         (car set-of-records))
        (else (lookup given-key (cdr set-of-records)))))
```

Конечно, существуют лучшие способы представить большие множества, чем в виде неупорядоченных списков. Системы доступа к информации, в которых необходим «произвольный доступ» к записям, как правило, реализуются с помощью методов, основанных на деревьях, вроде вышеописанной системы с бинарными деревьями. При разработке таких систем методология абстракции данных оказывается весьма полезной. Проектировщик может создать исходную реализацию с помощью простого, прямолинейного представления вроде неупорядоченных списков. Для окончательной версии это не подходит,

[41] Упражнениями 2.63–2.65 мы обязаны Полу Хилфингеру.

но такой вариант можно использовать как «поспешную и небрежную» реализацию базы данных, на которой тестируется остальная часть системы. Позже представление данных можно изменить и сделать более изощренным. Если доступ к базе данных происходит в терминах абстрактных селекторов и конструкторов, такое изменение представления данных не потребует никаких модификаций в остальной системе.

Упражнение 2.66.
Реализуйте процедуру `lookup` для случая, когда множество записей организовано в виде бинарного дерева, отсортированного по числовым значениям ключей.

2.3.4. Пример: деревья кодирования по Хаффману

Этот раздел дает возможность попрактиковаться в использовании списковых структур и абстракции данных для работы с множествами и деревьями. Они применяются к методам представления данных как последовательностей из единиц и нулей (битов). Например, стандартный код ASCII, который используется для представления текста в компьютерах, кодирует каждый символ как последовательность из семи бит. Семь бит позволяют нам обозначить 2^7, то есть 128 различных символов. В общем случае, если нам требуется различать n символов, нам потребуется $\log_2 n$ бит для каждого символа. Если все наши сообщения составлены из восьми символов A, B, C, D, E, F, G, и H, мы можем использовать код с тремя битами для каждого символа, например

A	000	C	010	E	100	G	110
B	001	D	011	F	101	H	111

С таким кодом, сообщение
BACADAEAFABBAAAGAH
кодируется как строка из 54 бит
001000010000011000100000101000001001000000000110000111

Такие коды, как ASCII и наш код от A до H, известны под названием кодов *с фиксированной длиной*, поскольку каждый символ сообщения они представляют с помощью одного и того же числа битов. Иногда полезно использовать и коды *с переменной длиной* (variable-length), в которых различные символы могут представляться различным числом битов. Например, азбука Морзе не для всех букв алфавита использует одинаковое число точек и тире. В частности, E, наиболее частая (в английском) буква, представляется с помощью одной точки. В общем случае, если наши сообщения таковы, что некоторые символы встречаются очень часто, а некоторые очень редко, то мы можем кодировать свои данные более эффективно (т. е. с помощью меньшего числа битов на сообщение), если более частым символам мы назначим более короткие коды. Рассмотрим следующий код для букв с A по H:

A	0	C	1010	E	1100	G	1110
B	100	D	1011	F	1101	H	1111

С таким кодом то же самое сообщение преобразуется в строку
100010100101101100011010100100000111001111

В этой строке 42 бита, так что она экономит более 20% места по сравнению с приведенным выше кодом с фиксированной длиной.

Одна из сложностей при работе с кодом с переменной длиной состоит в том, чтобы узнать, когда при чтении последовательности единиц и нулей достигнут конец символа. В азбуке Морзе эта проблема решается при помощи специального *кода-разделителя* (separator code) (в данном случае паузы) после последовательности точек и тире для каждой буквы. Другое решение состоит в том, чтобы построить систему кодирования так, чтобы никакой полный код символа не совпадал с началом (или *префиксом*) кода никакого другого символа. Такой код называется *префиксным* (prefix). В вышеприведенном примере A кодируется 0, а B 100, так что никакой другой символ не может иметь код, который начинается на 0 или 100.

В общем случае можно добиться существенной экономии, если использовать коды с переменной длиной, использующие относительные частоты символов в подлежащих кодированию сообщениях. Одна из схем такого кодирования называется кодированием по Хаффману, в честь своего изобретателя, Дэвида Хаффмана. Код Хаффмана может быть представлен как бинарное дерево, на листьях которого лежат кодируемые символы. В каждом нетерминальном узле находится множество символов с тех листьев, которые лежат под данным узлом. Кроме того, каждому символу на листе дерева присваивается вес (представляющий собой относительную частоту), а каждый нетерминальный узел имеет вес, который равняется сумме весов листьев, лежащих под данным узлом. Веса не используются в процессе кодирования и декодирования. Ниже мы увидим, как они оказываются полезными при построении дерева.

Рисунок 2.18 изображает дерево Хаффмана для кода от A до H, показанного выше. Веса в вершинах дерева указывают, что дерево строилось для сообщений, где A встречается с относительной частотой 8, B с относительной частотой 3, а все остальные буквы с относительной частотой 1.

Имея дерево Хаффмана, можно найти код любого символа, если начать с корня и двигаться вниз до тех пор, пока не будет достигнута концевая вершина, содержащая этот символ. Каждый раз, как мы спускаемся по левой ветви, мы добавляем 0 к коду, а спускаясь по правой ветви, добавляем 1. (Мы решаем, по какой ветке двигаться, проверяя, не является ли одна из веток концевой вершиной, а также содержит ли множество при вершине символ, который мы ищем.) Например, начиная с корня на картине 2.18, мы попадаем в концевую вершину D, сворачивая на правую дорогу, затем на левую, затем на правую, затем, наконец, снова на правую ветвь; следовательно, код для D — 1011.

Чтобы раскодировать последовательность битов при помощи дерева Хаффмана, мы начинаем с корня и просматриваем один за другим биты в последовательности, чтобы решить, нужно ли нам спускаться по левой или по правой ветви. Каждый раз, как мы добираемся до листовой вершины, мы порождаем новый символ сообщения и возвращаемся к вершине дерева, чтобы найти следующий символ. Например, пусть нам дано дерево, изображенное на рисунке, и последовательность 10001010. Начиная от корня, мы идем по правой ветви (поскольку первый бит в строке 1), затем по левой (поскольку второй бит 0), затем опять по левой (поскольку и третий бит 0). Здесь мы попадаем в лист, соответствующий B, так что первый символ декодируемого сообщения — B. Мы снова начинаем от корня и идем налево, поскольку следующий бит строки 0. Тут мы попадаем в лист, изображающий символ A. Мы опять начинаем от корня с остатком строки 1010, двигаемся направо, налево, направо, налево и приходим в C. Таким образом, все сообщение было BAC.

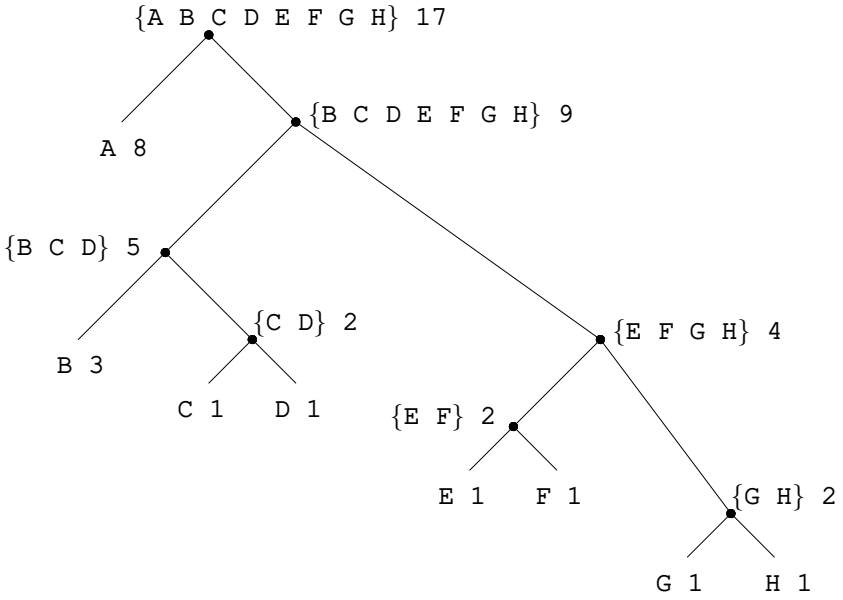


Рис. 2.18. Дерево кодирования по Хаффману.

Порождение деревьев Хаффмана

Если нам дан «алфавит» символов и их относительные частоты, как мы можем породить «наилучший» код? (Другими словами, какое дерево будет кодировать сообщения при помощи наименьшего количества битов?) Хаффман дал алгоритм для решения этой задачи и показал, что получаемый этим алгоритмом код — действительно наилучший код с переменной длиной для сообщений, где относительная частота символов соответствует частотам, для которых код строился. Здесь мы не будем доказывать оптимальность кодов Хаффмана, но покажем, как эти коды строятся[42].

Алгоритм порождения дерева Хаффмана весьма прост. Идея состоит в том, чтобы упорядочить дерево так, чтобы символы с наименьшей частотой оказались дальше всего от корня. Начнем с множества терминальных вершин, содержащих символы и их частоты, как указано в исходных данных, из которых нам надо построить дерево. Теперь найдем два листа с наименьшими весами и сольем их, получая вершину, у которой предыдущие две являются левым и правым потомками. Вес новой вершины равен сумме весов ее ветвей. Исключим два листа из исходного множества и заменим их новой вершиной. Продолжим этот процесс. На каждом шаге будем сливать две вершины с самыми низкими весами, исключая их из множества и заменяя вершиной, для которой они являются левой и правой ветвями. Этот процесс заканчивается, когда остается только одна вершина, которая и является корнем всего дерева. Вот как было порождено дерево Хаффмана на рисунке 2.18:

[42]Обсуждение математических свойств кодов Хаффмана можно найти в Hamming 1980.

Исходный набор листьев	{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}
Слияние	{(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}
Слияние	{(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}
Слияние	{(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}
Слияние	{(A 8) (B 3) ({C D} 2) ({E F G H} 4)}
Слияние	{(A 8) ({B C D} 5) ({E F G H} 4)}
Слияние	{(A 8) ({B C D E F G H} 9)}
Окончательное слияние	{({A B C D E F G H} 17)}

Алгоритм не всегда приводит к построению единственно возможного дерева, поскольку на каждом шаге выбор вершин с наименьшим весом может быть не единственным. Выбор порядка, в котором будут сливаться две вершины (то есть, какая из них будет левым, а какая правым поддеревом) также произволен.

Представление деревьев Хаффмана

В следующих упражнениях мы будем работать с системой, которая использует деревья Хаффмана для кодирования и декодирования сообщений и порождает деревья Хаффмана в соответствии с вышеописанным алгоритмом. Начнем мы с обсуждения того, как представляются деревья.

Листья дерева представляются в виде списка, состоящего из символа `leaf` (лист), символа, содержащегося в листе, и веса:

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (leaf? object)
  (eq? (car object) 'leaf))

(define (symbol-leaf x) (cadr x))

(define (weight-leaf x) (caddr x))
```

Дерево в общем случае будет списком из левой ветви, правой ветви, множества символов и веса. Множество символов будет просто их списком, а не каким-то более сложным представлением. Когда мы порождаем дерево слиянием двух вершин, мы получаем вес дерева как сумму весов этих вершин, а множество символов как объединение множеств их символов. Поскольку наши множества представлены в виде списка, мы можем породить объединение при помощи процедуры `append`, определенной нами в разделе 2.2.1:

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))
```

Если мы порождаем дерево таким образом, то у нас будут следующие селекторы:

```
(define (left-branch tree) (car tree))

(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))

(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

Процедуры `symbols` и `weight` должны вести себя несколько по-разному в зависимости от того, вызваны они для листа или для дерева общего вида. Это простые примеры *обобщенных процедур* (generic procedures) (процедур, которые способны работать более, чем с одним типом данных), о которых мы будем говорить намного более подробно в разделах 2.4 и 2.5.

Процедура декодирования

Следующая процедура реализует алгоритм декодирования. В качестве аргументов она принимает список из единиц и нулей, а также дерево Хаффмана.

```
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
               (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "плохой бит -- CHOOSE-BRANCH" bit))))
```

Процедура `decode-1` принимает два аргумента: список остающихся битов и текущую позицию в дереве. Она двигается «вниз» по дереву, выбирая левую или правую ветвь в зависимости от того, ноль или единица следующий бит в списке (этот выбор делается в процедуре `choose-branch`). Когда она достигает листа, она возвращает символ из него как очередной символ сообщения, присоединяя его посредством `cons` к результату декодирования остатка сообщения, начиная от корня дерева. Обратите внимание на проверку ошибок в конце `choose-branch`, которая заставляет программу протестовать, если во входных данных обнаруживается что-либо помимо единиц и нулей.

Множества взвешенных элементов

В нашем представлении деревьев каждая нетерминальная вершина содержит множество символов, которое мы представили как простой список. Однако алгоритм порождения дерева, который мы обсуждали выше, требует, чтобы мы работали еще и с множествами листьев и деревьев, последовательно сливая два наименьших элемента. Поскольку нам нужно будет раз за разом находить наименьший элемент множества, удобно для такого множества использовать упорядоченное представление.

Мы представим множество листьев и деревьев как список элементов, упорядоченный по весу в возрастающем порядке. Следующая процедура `adjoinset` для построения множеств подобна той, которая описана в упражнении 2.61; однако элементы сравниваются по своим весам, и никогда не бывает так, что добавляемый элемент уже содержится в множестве.

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
```

Следующая процедура принимает список пар вида символ–частота, например `((A 4) (B 2) (C 1) (D 1))`, и порождает исходное упорядоченное множество листьев, готовое к слиянию по алгоритму Хаффмана:

```
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair)
                               (cadr pair))
                    (make-leaf-set (cdr pairs))))))
```

Упражнение 2.67.
Пусть нам даны дерево кодирования и пример сообщения:

```
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                  (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree (make-leaf 'D 1)
                                   (make-leaf 'C 1)))))

(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
```

Раскодируйте сообщение при помощи процедуры `decode`.

Упражнение 2.68.
Процедура `encode` получает в качестве аргументов сообщение и дерево, и порождает список битов, который представляет закодированное сообщение.

```
(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree))))
```

`Encode-symbol` — процедура, которую Вы должны написать, возвращает список битов, который кодирует данный символ в соответствии с заданным деревом. Вы должны спроектировать `encode-symbol` так, чтобы она сообщала об ошибке, если символ вообще не содержится в дереве. Проверьте свою процедуру, закодировав тот результат, который Вы получили в упражнении 2.67, с деревом-примером и проверив, совпадает ли то, что получаете Вы, с исходным сообщением.

Упражнение 2.69.
Следующая процедура берет в качестве аргумента список пар вида символ-частота (где ни один символ не встречается более, чем в одной паре) и порождает дерево кодирования по Хаффману в соответствии с алгоритмом Хаффмана.

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))
```

Приведенная выше процедура `make-leaf-set` преобразует список пар в упорядоченное множество пар. Вам нужно написать процедуру `successive-merge`, которая при помощи `make-code-tree` сливает наиболее легкие элементы множества, пока не останется только один элемент, который и представляет собой требуемое дерево Хаффмана. (Эта процедура устроена немного хитро, но она не такая уж сложная. Если Вы видите, что строите сложную процедуру, значит, почти наверняка Вы делаете что-то не то. Можно извлечь немалое преимущество из того, что мы используем упорядоченное представление для множеств.)

Упражнение 2.70.
Нижеприведенный алфавит из восьми символов с соответствующими им относительными частотами был разработан, чтобы эффективно кодировать слова рок-песен 1950-х годов. (Обратите внимание, что «символы» «алфавита» не обязаны быть отдельными буквами.)

A	2	NA	16
BOOM	1	SHA	3
GET	2	YIP	9
JOB	2	WAH	1

При помощи `generate-huffman-tree` (упр. 2.69) породите соответствующее дерево Хаффмана, и с помощью `encode` закодируйте следующее сообщение:

Get a job
Sha na na na na na na na na
Get a job
Sha na na na na na na na na
Wah yip yip yip yip yip yip yip yip yip
Sha boom

Сколько битов потребовалось для кодирования? Каково наименьшее число битов, которое потребовалось бы для кодирования этой песни, если использовать код с фиксированной длиной для алфавита из восьми символов?

Упражнение 2.71.
Допустим, у нас есть дерево Хаффмана для алфавита из n символов, и относительные частоты символов равны $1, 2, 4, \ldots, 2^{n-1}$. Изобразите дерево для $n = 5$; для $n = 10$. Сколько битов в таком дереве (для произвольного n) требуется, чтобы закодировать самый частый символ? Самый редкий символ?

Упражнение 2.72.
Рассмотрим процедуру кодирования, которую Вы разработали в упражнении 2.68. Каков порядок роста в терминах количества шагов, необходимых для кодирования символа? Не забудьте включить число шагов, требуемых для поиска символа в каждой следующей вершине. Ответить на этот вопрос в общем случае сложно. Рассмотрите особый случай, когда относительные частоты символов таковы, как описано в упражнении 2.71, и найдите порядок роста (как функцию от n) числа шагов, необходимых, чтобы закодировать самый частый и самый редкий символ алфавита.

2.4. Множественные представления для абстрактных данных

В предыдущих разделах мы описали абстракцию данных, методологию, позволяющую структурировать системы таким образом, что бóльшую часть программы можно специфицировать независимо от решений, которые принимаются при реализации объектов, обрабатываемых программой. Например, в разделе 2.1.1 мы узнали, как отделить задачу проектирования программы, которая пользуется рациональными числами, от задачи реализации рациональных чисел через элементарные механизмы построения составных данных в компьютерном языке. Главная идея состояла в возведении барьера абстракции, — в данном случае, селекторов и конструкторов для рациональных чисел (`make-rat`, `numer`, `denom`), — который отделяет то, как рациональные числа используются, от их внутреннего представления через списковые структуры. Подобный же барьер абстракции отделяет детали процедур, реализующих рациональную арифметику (`add-rat`, `sub-rat`, `mul-rat` и `div-rat`), от «высокоуровневых» процедур, которые используют рациональные числа. Получившаяся программа имеет структуру, показанную на рис. 2.1.

Такие барьеры абстракции — мощное средство управления сложностью проекта. Изолируя внутренние представления объектов данных, нам удается разделить задачу построения большой программы на меньшие задачи, которые можно решать независимо друг от друга. Однако такой тип абстракции данных еще недостаточно мощен, поскольку не всегда имеет смысл говорить о «внутреннем представлении» объекта данных.

Например, может оказаться более одного удобного представления для объекта данных, и мы можем захотеть проектировать системы, которые способны работать с множественными представлениями. В качестве простого примера, комплексные числа можно представить двумя почти эквивалентными способами: в декартовой форме (действительная и мнимая часть) и в полярной форме (модуль и аргумент). Иногда лучше подходит декартова форма, а иногда полярная. В сущности, вполне возможно представить себе систему, в которой комплексные числа представляются обоими способами, а процедуры-операции над комплексным числами способны работать с любым представлением.

Еще важнее то, что часто программные системы разрабатываются большим количеством людей в течение долгого времени, в соответствии с требованиями, которые также

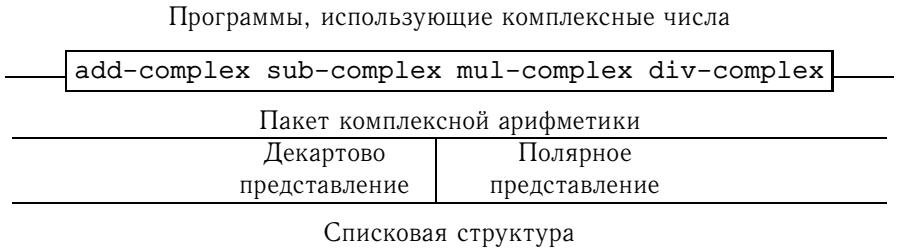


Рис. 2.19. Барьеры абстракции данных в системе работы с комплексными числами.

со временем меняются. В такой ситуации просто невозможно заранее всем договориться о выборе представления данных. Так что в дополнение к барьерам абстракции данных, которые отделяют представление данных от их использования, нам нужны барьеры абстракции, которые отделяют друг от друга различные проектные решения и позволяют различным решениям сосуществовать в рамках одной программы. Более того, поскольку часто большие программы создаются путем комбинирования существующих модулей, созданных независимо друг от друга, нам требуются соглашения, которые позволяли бы программистам добавлять модули к большим системам *аддитивно* (additively), то есть без перепроектирования и переписывания этих модулей.

В этом разделе мы научимся работать с данными, которые могут быть представлены в разных частях программы различными способами. Это требует построения *обобщенных процедур* (generic procedures) — процедур, работающих с данными, которые могут быть представлены более чем одним способом. Наш основной метод построения обобщенных процедур будет состоять в том, чтобы работать в терминах объектов, обладающих *метками типа* (type tags), то есть объектов, явно включающих информацию о том, как их надо обрабатывать. Кроме того, мы обсудим *программирование, управляемое данными* (data-directed programming) — мощную и удобную стратегию реализации, предназначенную для аддитивной сборки систем с обобщенными операциями.

Мы начнем с простого примера комплексных чисел. Мы увидим, как метки типа и стиль, управляемый данными, позволяют нам создать отдельные декартово и полярное представления комплексных чисел, и при этом поддерживать понятие абстрактного объекта «комплексное число» . Мы добьемся этого, определив арифметические процедуры для комплексных чисел (`add-complex`, `sub-complex`, `mul-complex` и `div-complex`) в терминах обобщенных селекторов, которые получают части комплексного числа независимо от того, как оно представлено. Получающаяся система работы с комплексными числами, как показано на рис. 2.19, содержит два типа барьеров абстракции. «Горизонтальные» барьеры играют ту же роль, что и на рис. 2.1. Они отделяют «высокоуровневые» операции от «низкоуровневых» представлений. В дополнение к этому, существует еще «вертикальный» барьер, который дает нам возможность отдельно разрабатывать и добавлять альтернативные представления.

В разделе 2.5 мы покажем, как с помощью меток типа и стиля программирования, управляемого данными, создать арифметический пакет общего назначения. Такой пакет дает пользователю процедуры (`add`, `mul` и т.д.), с помощью которых можно манипулировать всеми типами «чисел», и если нужно, его можно легко расширить, когда потре-

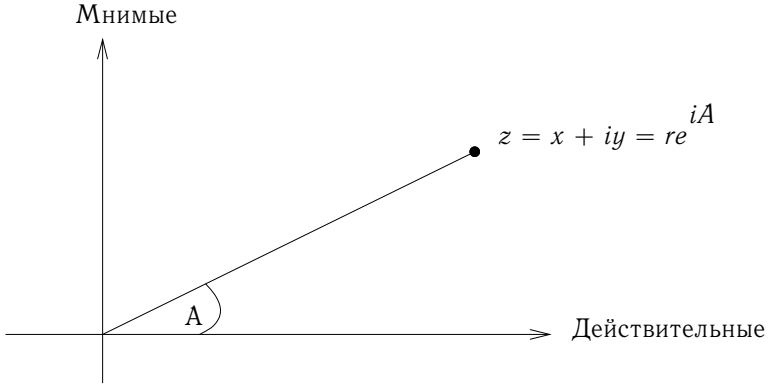


Рис. 2.20. Комплексные числа как точки на плоскости

буется новый тип чисел. В разделе 2.5.3 мы покажем, как использовать обобщенную арифметику в системе, работающей с символьной алгеброй.

2.4.1. Представления комплексных чисел

В качестве простого, хотя и нереалистичного, примера программы, использующей обобщенные операции, мы разработаем систему, которая производит арифметические операции над комплексными числами. Начнем мы с обсуждения двух возможных представлений комплексного числа в виде упорядоченной пары: декартова форма (действительная и мнимая части) и полярная форма (модуль и аргумент)[43]. В разделе 2.4.2 будет показано, как оба представления можно заставить сосуществовать в рамках одной программы при помощи меток типа и обобщенных операций.

Подобно рациональным числам, комплексные числа естественно представлять в виде упорядоченных пар. Множество комплексных чисел можно представлять себе как двумерное пространство с двумя перпендикулярными осями: «действительной» и «мнимой» (см. рис. 2.20). С этой точки зрения комплексное число $z = x + iy$ (где $i^2 = -1$) можно представить как точку на плоскости, действительная координата которой равна x, а мнимая y. В этом представлении сложение комплексных чисел сводится к сложению координат:

$$\text{Действительная-часть}(z_1 + z_2) =$$
$$= \text{Действительная-часть}(z_1) + \text{Действительная-часть}(z_2)$$

$$\text{Мнимая-часть}(z_1 + z_2) = \text{Мнимая-часть}(z_1) + \text{Мнимая-часть}(z_2)$$

[43] В реальных вычислительных системах, как правило, декартова форма предпочтительнее полярной из-за ошибок округления при преобразованиях между этими двумя формами. Именно поэтому пример с комплексными числами нереалистичен. Тем не менее, он служит ясной иллюстрацией строения системы, использующей обобщенные операции, и хорошим введением в более содержательные системы, которые мы строим далее по ходу этой главы.

При умножении комплексных чисел естественней думать об их представлении в полярной форме, в виде модуля и аргумента (r и A на рис. 2.20). Произведение двух комплексных чисел есть вектор, получаемый путем растягивания одного комплексного числа на модуль другого и поворота на его же аргумент:

$$\text{Модуль}(z_1 \cdot z_2) = \text{Модуль}(z_1) \cdot \text{Модуль}(z_2)$$
$$\text{Аргумент}(z_1 \cdot z_2) = \text{Аргумент}(z_1) + \text{Аргумент}(z_2)$$

Таким образом, есть два различных представления для комплексных чисел, и каждое из них удобнее для какого-то набора операций. Однако с точки зрения человека, который пишет программу с использованием комплексных чисел, принцип абстракции данных утверждает, что все операции, работающие с комплексными числами, должны работать независимо от того, какую интерпретацию использует компьютер. Например, часто бывает нужно получить модуль комплексного числа, представленного в декартовых координатах. Подобным образом, часто полезно уметь определять действительную часть комплексного числа, представленного в полярных координатах.

При разработке такой системы мы можем следовать той самой стратегии абстракции данных, которую мы использовали в пакете работы с рациональными числами в разделе 2.1.1. Предположим, что операции над комплексными числами реализованы в терминах четырех селекторов: `real-part`, `imag-part`, `magnitude` и `angle`. Предположим еще, что у нас есть две процедуры для построения комплексных чисел: `make-from-real-imag` возвращает комплексное число с указанными действительной и мнимой частями, а `make-from-mag-ang` возвращает комплексное число с указанными модулем и аргументом. Эти процедуры обладают такими свойствами, что для любого комплексного числа `z`

```
(make-from-real-imag (real-part z) (imag-part z))
```

и

```
(make-from-mag-ang (magnitude z) (angle z))
```

порождают комплексные числа, равные `z`.

Используя такие конструкторы и селекторы, мы можем реализовать арифметику комплексных чисел через «абстрактные данные», определяемые этими конструкторами и селекторами, в точности как мы это делали для рациональных чисел в разделе 2.1.1. Как показывают вышеуказанные формулы, можно складывать и вычитать комплексные числа в терминах действительной и мнимой части, а умножать и делить в терминах модуля и аргумента:

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))

(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))
```

```
(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))

(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
```

Для того, чтобы придать пакету работы с комплексными числами окончательный вид, нам осталось выбрать представление и реализовать конструкторы и селекторы в терминах элементарных чисел и элементарной списковой структуры. Есть два очевидных способа это сделать: можно представлять комплексное число как пару в «декартовой форме» (действительная часть, мнимая часть) либо в «полярной форме» (модуль, аргумент). Какой вариант мы выберем?

Чтобы говорить о конкретных вариантах, предположим, что двое программистов, Бен Битобор и Лиза П. Хакер, независимо друг от друга разрабатывают представления для системы, работающей с комплексными числами. Бен решает представлять комплексные числа в декартовой форме. При таком решении доступ к действительной и мнимой частям комплексного числа, а также построение его из действительной и мнимой частей реализуются прямолинейно. Чтобы найти модуль и аргумент, а также чтобы построить комплексное число с заданными модулем и аргументом, он использует тригонометрические соотношения

$$x = r\cos A \qquad y = r\sin A;$$

$$r = \sqrt{x^2 + y^2} \quad A = \operatorname{arctg}(y, x)$$

которые связывают действительную и мнимую части (x, y) с модулем и аргументом (r, A)[44]. Таким образом, реализация Бена определяется следующими селекторами и конструкторами:

```
(define (real-part z) (car z))

(define (imag-part z) (cdr z))

(define (magnitude z)
  (sqrt (+ (square (real-part z)) (square (imag-part z)))))

(define (angle z)
  (atan (imag-part z) (real-part z)))

(define (make-from-real-imag x y) (cons x y))

(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
```

Напротив, Лиза решает представить комплексные числа в полярной форме. Для нее доступ к модулю и аргументу тривиален, но для получения действительной и мнимой

[44]Функция взятия арктангенса, которая здесь используется, вычисляется процедурой Scheme `atan`. Она берет два аргумента y и x и возвращает угол, тангенс которого равен y/x. Знаки аргументов определяют, в каком квадранте находится угол.

части ей приходится использовать тригонометрические тождества. Вот представление Лизы:

```
(define (real-part z)
  (* (magnitude z) (cos (angle z))))

(define (imag-part z)
  (* (magnitude z) (sin (angle z))))

(define (magnitude z) (car z))

(define (angle z) (cdr z))

(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))

(define (make-from-mag-ang r a) (cons r a))
```

Дисциплина абстракции данных обеспечивает то, что одни и те же реализации процедур `add-complex`, `sub-complex`, `mul-complex` и `div-complex` будут работать как с Беновым представлением, так и с Лизиным.

2.4.2. Помеченные данные

Можно рассматривать абстракцию данных как применение принципа «наименьших обязательств». Реализуя систему обработки комплексных чисел в разделе 2.4.1, мы можем использовать либо декартово представление от Бена, либо полярное от Лизы. Барьер абстракции, который образуют селекторы и конструкторы, позволяет нам до последнего момента отложить выбор конкретного представления для наших объектов данных, и таким образом сохранить максимальную гибкость в проекте нашей системы.

Принцип наименьших обязательств можно довести до еще бóльших крайностей. Если нам понадобится, мы можем сохранить неопределенность представления даже *после* того, как мы спроектировали селекторы и конструкторы, и использовать *и* представление Бена, *и* представление Лизы. Однако если оба представления участвуют в одной и той же системе, нам потребуется какой-нибудь способ отличить данные в полярной форме от данных в декартовой форме. Иначе, если нас попросят, например, вычислить `magnitude` от пары $(3, 4)$, мы не будем знать, надо ли ответить 5 (интерпретируя число в декартовой форме) или 3 (интерпретируя его в полярной форме). Естественный способ добиться необходимого различия состоит в том, чтобы использовать *метку типа* (type tag) — символ `rectangular` или `polar` — как часть каждого комплексного числа. Тогда, когда нам понадобится что-то делать с комплексным числом, мы можем при помощи этой метки решить, который селектор требуется применить.

Чтобы работать с помеченными данными, мы предположим, что у нас есть процедуры `type-tag` и `contents`, которые извлекают из элемента данных метку и собственно содержимое (полярные либо декартовы координаты, если речь идет о комплексном числе). Кроме того, мы постулируем процедуру `attach-tag`, которая берет метку и

содержимое, и выдает помеченный объект данных. Простейший способ реализовать эти процедуры — использовать обыкновенную списковую структуру:

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Некорректные помеченные данные -- TYPE-TAG" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Некорректные помеченные данные -- CONTENTS" datum)))
```

При помощи этих процедур мы можем определить предикаты `rectangular?` и `polar?`, которые распознают, соответственно, декартово и полярное представление:

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))

(define (polar? z)
  (eq? (type-tag z) 'polar))
```

Теперь, когда у нас имеются метки типов, Бен и Лиза могут переделать свой код так, чтобы позволить своим разнородным представлениям сосуществовать в одной и той же системе. Каждый раз, когда Бен создает комплексное число, он помечает его как декартово. Каждый раз, когда Лиза создает комплексное число, она помечает его как полярное. В дополнение к этому, Бен и Лиза должны сделать так, чтобы не было конфликта имен между названиями их процедур. Один из способов добиться этого — Бену добавить слово `rectangular` к названиям всех своих процедур представления данных, а Лизе добавить `polar` к своим. Вот переработанное декартово представление Бена из раздела 2.4.1:

```
(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
           (square (imag-part-rectangular z)))))

(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))

(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
```

```
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))
```

а вот переработанное полярное представление Лизы:

```
(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))

(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))

(define (magnitude-polar z) (car z))

(define (angle-polar z) (cdr z))

(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
               (cons (sqrt (+ (square x) (square y)))
                     (atan y x))))

(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
```

Каждый обобщенный селектор реализуется как процедура, которая проверяет метку своего аргумента и вызывает подходящую процедуру для обработки данных нужного типа. Например, для того, чтобы получить действительную часть комплексного числа, `real-part` смотрит на метку и решает, звать ли Бенову `real-part-rectangular` или Лизину `real-part-polar`. В каждом из этих случаев мы пользуемся процедурой `contents`, чтобы извлечь голый, непомеченный элемент данных и передать его либо в декартову, либо в полярную процедуру:

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Неизвестный тип -- REAL-PART" z))))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Неизвестный тип -- IMAG-PART" z))))

(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
```

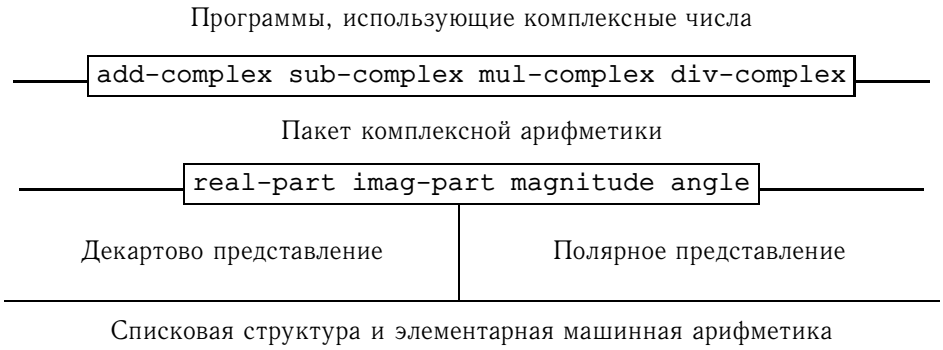


Рис. 2.21. Структура обобщенной системы комплексной арифметики.

```
         (magnitude-polar (contents z)))
        (else (error "Неизвестный тип -- MAGNITUDE" z))))

(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Неизвестный тип -- ANGLE" z))))
```

Для реализации арифметических операций с комплексными числами мы по-прежнему можем использовать старые процедуры `add-complex`, `sub-complex`, `mul-complex` и `div-complex` из раздела 2.4.1, поскольку вызываемые ими селекторы обобщенные и, таким образом, могут работать с любым из двух представлений. Например, процедура `add-complex` по-прежнему выглядит как

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
```

Наконец, нам надо решить, порождать ли комплексные числа в Беновом или Лизином представлении. Одно из разумных решений состоит в том, чтобы порождать декартовы числа, когда нам дают действительную и мнимую часть, и порождать полярные числа, когда нам дают модуль и аргумент:

```
(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))
(define (make-from-mag-ang r a)
  (make-from-mag-ang-polar r a))
```

Структура получившейся системы комплексной арифметики показана на рисунке 2.21. Система разбита на три относительно независимых части: операции арифметики комплексных чисел, полярная реализация Лизы и декартова реализация Бена. Полярная и

декартова реализации могли быть написаны Беном и Лизой по отдельности, и любую из них может использовать в качестве внутреннего представления третий программист, чтобы реализовать процедуры арифметики комплексных чисел в терминах абстрактного интерфейса конструкторов и селекторов.

Поскольку каждый объект данных помечен своим типом, селекторы работают с данными обобщенным образом. Это означает, что каждый селектор по определению обладает поведением, которое зависит от того, к какому типу данных он применяется. Следует обратить внимание на общий механизм доступа к отдельным представлениям: внутри любой данной реализации представления (скажем, внутри полярного пакета Лизы) комплексное число представляется нетипизированной парой (модуль, аргумент). Когда обобщенный селектор обращается к данным полярного типа, он отрывает метку и передает содержимое Лизиному коду. И наоборот, когда Лиза строит число для общего пользования, она помечает его тип, чтобы процедуры более высокого уровня могли его должным образом распознать. Такая дисциплина снятия и добавления меток при передаче объектов данных с уровня на уровень может быть ценной стратегией организации данных и программ, как мы увидим в разделе 2.5.

2.4.3. Программирование, управляемое данными, и аддитивность

Общая стратегия проверки типа данных и вызова соответствующей процедуры называется *диспетчеризацией по типу* (dispatching on type). Это хороший способ добиться модульности при проектировании системы. С другой стороны, такая реализация диспетчеризации, как в разделе 2.4.2, имеет два существенных недостатка. Один заключается в том, что обобщенные процедуры интерфейса (`real-part`, `imag-part`, `magnitude` и `angle`) обязаны знать про все имеющиеся способы представления. Предположим, к примеру, что нам хочется ввести в нашу систему комплексных чисел еще одно представление. Нам нужно будет сопоставить этому представлению тип, а затем добавить в каждую из обобщенных процедур интерфейса по варианту для проверки на этот новый тип и вызова селектора, соответствующего его представлению.

Второй недостаток этого метода диспетчеризации состоит в том, что, хотя отдельные представления могут проектироваться раздельно, нам нужно гарантировать, что никакие две процедуры во всей системе не называются одинаково. Вот почему Бену и Лизе пришлось изменить имена своих первоначальных процедур из раздела 2.4.1.

Оба эти недостатка являются следствием того, что наш метод реализации обобщенных интерфейсов *неаддитивен*. Программист, реализующий обобщенные процедуры-селекторы, должен их переделывать каждый раз, как добавляется новое представление, а авторы, создающие отдельные представления, должны изменять свой код, чтобы избежать конфликтов имен. В каждом из этих случаев изменения, которые требуется внести в код, тривиальны, но их все равно нужно делать, и отсюда проистекают неудобства и ошибки. Для системы работы с комплексными числами в ее нынешнем виде это проблема небольшая, но попробуйте представить, что есть не два, а сотни различных представлений комплексных чисел. И что есть много обобщенных селекторов, которые надо поддерживать в интерфейсе абстрактных данных. Представьте даже, что ни один программист не знает всех интерфейсных процедур всех реализаций. Проблема эта реальна, и с ней приходится разбираться в программах вроде систем управления базами данных большого калибра.

Операции	Типы	
	Polar	Rectangular
`real-part`	`real-part-polar`	`real-part-rectangular`
`imag-part`	`imag-part-polar`	`imag-part-rectangular`
`magnitude`	`magnitude-polar`	`magnitude-rectangular`
`angle`	`angle-polar`	`angle-rectangular`

Рис. 2.22. Таблица операций в системе комплексных чисел.

Нам нужен способ еще более модуляризовать устройство системы. Это позволяет метод программирования, который называется *программирование, управляемое данными* (data-directed programming). Чтобы понять, как работает этот метод, начнем с наблюдения: каждый раз, когда нам приходится работать с набором обобщенных операций, общих для множества различных типов, мы, в сущности, работаем с двумерной таблицей, где по одной оси расположены возможные операции, а по другой всевозможные типы. Клеткам таблицы соответствуют процедуры, которые реализуют каждую операцию для каждого типа ее аргумента. В системе комплексной арифметики из предыдущего раздела соответствие между именем операции, типом данных и собственно процедурой было размазано по условным предложениям в обобщенных процедурах интерфейса. Но ту же самую информацию можно было бы организовать в виде таблицы, как показано на рис. 2.22.

Программирование, управляемое данными, — метод проектирования программ, позволяющий им напрямую работать с такого рода таблицей. Механизм, который связывает код комплексных арифметических операций с двумя пакетами представлений, мы ранее реализовали в виде набора процедур, которые явно осуществляют диспетчеризацию по типу. Здесь мы реализуем этот интерфейс через одну процедуру, которая ищет сочетание имени операции и типа аргумента в таблице, чтобы определить, какую процедуру требуется применить, а затем применяет ее к содержимому аргумента. Если мы так сделаем, то, чтобы добавить к системе пакет с новым представлением, нам не потребуется изменять существующие процедуры; понадобится только добавить новые клетки в таблицу.

Чтобы реализовать этот план, предположим, что у нас есть две процедуры `put` и `get`, для манипуляции с таблицей операций и типов:

- `(put` ⟨*оп*⟩ ⟨*тип*⟩ ⟨*элемент*⟩`)` вносит ⟨*элемент*⟩ в таблицу, в клетку, индексом которой служат операция ⟨*оп*⟩ и тип ⟨*тип*⟩.
- `(get` ⟨*оп*⟩ ⟨*тип*⟩`)` ищет в таблице ячейку с индексом ⟨*оп*⟩,⟨*тип*⟩ и возвращает ее содержимое. Если ячейки нет, `get` возвращает ложь.

Пока что мы предположим, что `get` и `put` входят в наш язык. В главе 3 (раздел 3.3.3) мы увидим, как реализовать эти и другие операции для работы с таблицами.

Программирование, управляемое данными, в системе с комплексными числами можно использовать так: Бен, который разрабатывает декартово представление, пишет код в точности как он это делал сначала. Он определяет набор процедур, или *пакет* (package), и привязывает эти процедуры к остальной системе, добавляя в таблицу ячейки, которые сообщают системе, как работать с декартовыми числами. Это происходит при вызове

следующей процедуры:

```
(define (install-rectangular-package)
   ;; внутренние процедуры
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))

   ;; интерфейс к остальной системе
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Обратите внимание, что внутренние процедуры — те самые, которые Бен писал, когда он в разделе 2.4.1 работал сам по себе. Никаких изменений, чтобы связать их с остальной системой, не требуется. Более того, поскольку определения процедур содержатся внутри процедуры установки, Бену незачем беспокоиться о конфликтах имен с другими процедурами вне декартова пакета. Чтобы связать их с остальной системой, Бен устанавливает свою процедуру `real-part` под именем операции `real-part` и типом `(rectangular)`, и то же самое он проделывает с другими селекторами[45]. Его интерфейс также определяет конструкторы, которые может использовать внешняя система[46]. Они совпадают с конструкторами, которые Бен определяет для себя, но вдобавок прикрепляют метку.

Лизин полярный пакет устроен аналогично:

```
(define (install-polar-package)
   ;; внутренние процедуры
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z)
```

[45]Мы используем список `(rectangular)`, а не символ `rectangular`, чтобы предусмотреть возможность операций с несколькими аргументами, не все из которых одинакового типа.

[46]Тип, под которым устанавливаются конструкторы, необязательно делать списком, поскольку конструктор всегда вызывается для того, чтобы породить один объект определенного типа.

```
    (* (magnitude z) (cos (angle z))))
  (define (imag-part z)
    (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))

   ;; интерфейс к остальной системе
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Несмотря на то, что Бен и Лиза используют свои исходные процедуры с совпадающими именами (например, `real-part`), эти определения теперь внутренние для различных процедур (см. раздел 1.1.8), так что никакого конфликта имен не происходит.

Селекторы комплексной арифметики обращаются к таблице посредством общей процедуры-«операции» `apply-generic`, которая применяет обобщенную операцию к набору аргументов. `Apply-generic` ищет в таблице ячейку по имени операции и типам аргументов и применяет найденную процедуру, если она существует[47]:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "Нет метода для этих типов -- APPLY-GENERIC"
            (list op type-tags))))))
```

При помощи `apply-generic` можно определить обобщенные селекторы так:

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

[47] `Apply-generic` пользуется точечной записью, описанной в упражнении 2.20, поскольку различные обобщенные операции могут принимать различное число аргументов. В `apply-generic` значением `op` является первый аргумент вызова `apply-generic`, а значением `args` список остальных аргументов.

Кроме того, `apply-generic` пользуется элементарной процедурой `apply`, которая принимает два аргумента: процедуру и список. `Apply` вызывает процедуру, используя элементы списка как аргументы. Например,

```
(apply + (list 1 2 3 4))
```

возвращает 10.

Заметим, что они не изменяются, если в систему добавляется новое представление.

Кроме того, мы можем из той же таблицы получать конструкторы, которые будут использоваться программами, внешними по отношению к пакетам, для изготовления комплексных чисел из действительной и мнимой части либо из модуля и аргумента. Как и в разделе 2.4.2, мы порождаем декартово представление, если нам дают действительную и мнимую часть, и полярное, если дают модуль и аргумент:

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))

(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

Упражнение 2.73.
В разделе 2.3.2 описывается программа, которая осуществляет символьное дифференцирование:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        ⟨здесь можно добавить еще правила⟩
        (else (error "неизвестный тип выражения -- DERIV" exp))))
```

Можно считать, что эта программа осуществляет диспетчеризацию по типу выражения, которое требуется продифференцировать. В этом случае «меткой типа» элемента данных является символ алгебраической операции (например, `+`), а операция, которую нужно применить – `deriv`. Эту программу можно преобразовать в управляемый данными стиль, если переписать основную процедуру взятия производной в виде

```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp) (if (same-variable? exp var) 1 0))
         (else ((get 'deriv (operator exp)) (operands exp)
                                            var))))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))
```

а. Объясните, что происходит в приведенном фрагменте кода. Почему нельзя включить в операцию выбора, управляемого данными, предикаты `number?` и `variable?`?

б. Напишите процедуры для вычисления производных от суммы и произведения, а также дополнительный код, чтобы добавить их к таблице, которой пользуется приведенный фрагмент.

в. Выберите еще какое-нибудь правило дифференцирования, например для возведения в степень (упражнение 2.56), и установите его в систему.

г. В этой простой алгебраической системе тип выражения — это алгебраическая операция верхнего уровня. Допустим, однако, что мы индексируем процедуры противоположным образом, так что строка диспетчеризации в `deriv` выглядит как

```
((get (operator exp) 'deriv) (operands exp) var)
```

Какие изменения потребуются в системе дифференцирования?

Упражнение 2.74.
Insatiable Enterprises, Inc. — децентрализованная компания-конгломерат, которая состоит из большого количества независимых подразделений, раскиданных по всему миру. Недавно вычислительные мощности компании были связаны умной вычислительной сетью, создающей для пользователя иллюзию, что он работает с единым компьютером. Президент компании, когда она в первый раз пытается воспользоваться способностью системы осуществлять доступ к файлам подразделений, с изумлением и ужасом обнаруживает, что, несмотря на то, что все эти файлы реализованы в виде структур данных на Scheme, конкретная структура данных отличается от подразделения к подразделению. Спешно созывается совещание менеджеров подразделений, чтобы найти стратегию, которая позволила бы собрать файлы в единую систему для удовлетворения нужд главного офиса, и одновременно сохранить существующую автономию подразделений.

Покажите, как такую стратегию можно реализовать при помощи программирования, управляемого данными. К примеру, предположим, что сведения о персонале каждого подразделения устроены в виде единого файла, который содержит набор записей, проиндексированных по имени служащего. Структура набора данных от подразделения к подразделению различается. Более того, каждая запись сама по себе — набор сведений (в разных подразделениях устроенный по-разному), в котором информация индексируется метками вроде `address` (адрес) или `salary` (зарплата). В частности:

а. Для главного офиса реализуйте процедуру `get-record`, которая получает запись, относящуюся к указанному служащему, из указанного файла персонала. Процедура должна быть применима к файлу любого подразделения. Объясните, как должны быть структурированы файлы отдельных подразделений. В частности, какую информацию о типах нужно хранить?

б. Для главного офиса реализуйте процедуру `get-salary`, которая возвращает зарплату указанного служащего из файла любого подразделения. Как должна быть устроена запись, чтобы могла работать эта процедура?

в. Для главного офиса напишите процедуру `find-employee-record`. Она должна искать в файлах всех подразделений запись указанного служащего и возвращать эту запись. Предположим, что в качестве аргументов эта процедура принимает имя служащего и список файлов всех подразделений.

г. Какие изменения требуется внести в систему, чтобы внести в центральную систему информацию о новых служащих, когда Insatiable поглощает новую компанию?

Передача сообщений

Основная идея программирования, управляемого данными, состоит в том, чтобы работать с обобщенными операциями в программах при помощи явных манипуляций с таблицами операций и типов, вроде таблицы на рисунке 2.22. В стиле программирования, который мы применяли в разделе 2.4.2, диспетчеризация по типу организуется

внутри каждой операции, и каждая операция должна сама заботиться о своей диспетчеризации. Это, в сущности, разбивает таблицу операций и типов на строки, и каждая обобщенная операция представляет собой строку таблицы.

Альтернативой такой стратегии реализации будет разбить таблицу по столбцам и вместо «умных операций», которые диспетчируют по типам данных, работать с «умными объектами данных», которые диспетчируют по именам операций. Мы можем этого добиться, если устроим все так, что объект данных, например комплексное число в декартовом представлении, будет представляться в виде процедуры, которая в качестве входа воспринимает имя операции и осуществляет соответствующее ей действие. При такой организации можно написать `make-from-real-imag` в виде

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else
           (error "Неизвестная оп. -- MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

Соответствующая процедура `apply-generic`, которая применяет обобщенную операцию к аргументу, просто скармливает имя операции объекту данных и заставляет его делать всю работу[48]:

```
(define (apply-generic op arg) (arg op))
```

Обратите внимание, что значение, возвращаемое из `make-from-real-imag`, является процедурой — это внутренняя процедура `dispatch`. Она вызывается, когда `apply-generic` требует выполнить обобщенную операцию.

Такой стиль программирования называется *передача сообщений* (message passing). Имя происходит из представления, что объект данных — это сущность, которая получает имя затребованной операции как «сообщение». Мы уже встречались с примером передачи сообщений в разделе 2.1.3, где мы видели, как `cons`, `car` и `cdr` можно определить безо всяких объектов данных, с одними только процедурами. Теперь мы видим, что передача сообщений не математический трюк, а полезный метод организации систем с обобщенными операциями. В оставшейся части этой главы мы будем продолжать пользоваться программированием, управляемым данными, а не передачей сообщений, и рассмотрим обобщенные арифметические операции. Мы вернемся к передаче сообщений в главе 3, и увидим, что она может служить мощным инструментом для структурирования моделирующих программ.

Упражнение 2.75.

Реализуйте в стиле передачи сообщений конструктор `make-from-mag-ang`. Он должен быть аналогичен приведенной выше процедуре `make-from-real-imag`.

[48] У такой организации есть ограничение: она допускает обобщенные процедуры только от одного аргумента.

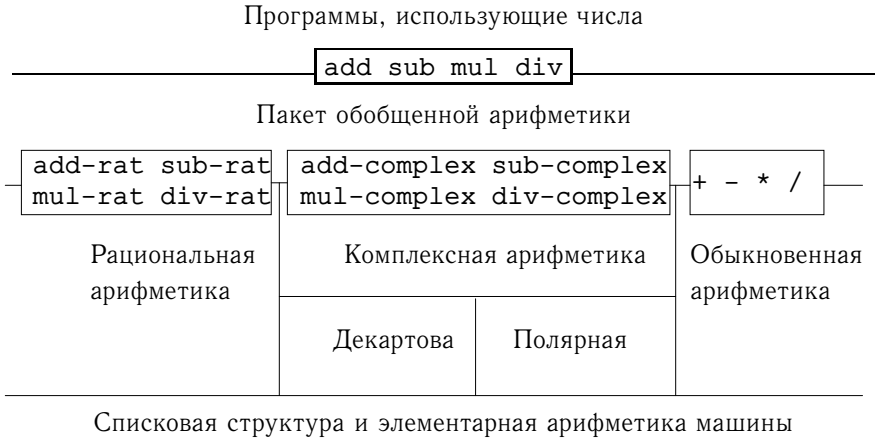


Рис. 2.23. Обобщенная арифметическая истема.

Упражнение 2.76.
Когда большая система с обобщенными операциями развивается, могут потребоваться новые типы объектов данных или новые операции. Для каждой из трех стратегий — обобщенные операции с явной диспетчеризацией, стиль, управляемый данными, и передача сообщений, – опишите, какие изменения нужно произвести в системе, чтобы добавить новый тип или новую операцию. Какая организация лучше подходит для системы, в которую часто добавляются новые типы? Какая для системы, где часто появляются новые операции?

2.5. Системы с обобщенными операциями

В предыдущем разделе мы увидели, как проектировать системы, где объекты данных могут быть представлены более чем одним способом. Основная идея состоит в том, чтобы связать код, который определяет операции над данными, и многочисленные реализации данных, при помощи обобщенных процедур интерфейса. Теперь мы увидим, что ту же самую идею можно использовать не только для того, чтобы определять обобщенные операции для нескольких реализаций одного типа, но и для того, чтобы определять операции, обобщенные относительно нескольких различных типов аргументов. Мы уже встречались с несколькими различными пакетами арифметических операций: элементарная арифметика (`+`, `-`, `*`, `/`), встроенная в наш язык, арифметика рациональных чисел (`add-rat`, `sub-rat`, `mul-rat`, `div-rat`) из раздела 2.1.1 и арифметика комплексных чисел, которую мы реализовали в разделе 2.4.3. Теперь мы, используя методы программирования, управляемого данными, создадим пакет арифметических операций, который включает все уже построенные нами арифметические пакеты.

На рисунке 2.23 показана структура системы, которую мы собираемся построить. Обратите внимание на барьеры абстракции. С точки зрения человека, работающего с «числами», есть только одна процедура `add`, которая работает, какие бы числа ей ни дали. `Add` является частью обобщенного интерфейса, который позволяет программам,

пользующимся числами, одинаковым образом обращаться к раздельным пакетам обыкновенной, рациональной и комплексной арифметики. Всякий конкретный арифметический пакет (например, комплексная арифметика) сам по себе доступен через обобщенные процедуры (например, `add-complex`), которые связывают пакеты, предназначенные для различных реализаций (таких, как декартовы и полярные числа). Более того, структура системы аддитивна, так что можно проектировать отдельные арифметические пакеты независимо и сочетать их, получая обобщенную арифметическую систему.

2.5.1. Обобщенные арифметические операции

Задача проектирования обобщенных арифметических операций аналогична задаче проектирования обобщенных операций с комплексными числами. К примеру, нам бы хотелось иметь обобщенную процедуру сложения `add`, которая действовала бы как обычное элементарное сложение `+` по отношению к обычным числам, как `add-rat` по отношению к рациональным числам и как `add-complex` по отношению к комплексным. Реализовать `add` и прочие обобщенные арифметические операции мы можем, следуя той же стратегии, которую мы использовали в разделе 2.4.3 для обобщенных селекторов комплексных чисел. К каждому числу мы прикрепим метку типа и заставим обобщенную процедуру передавать управление в нужный пакет в соответствии с типами своих аргументов.

Обобщенные арифметические процедуры определяются следующим образом:

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

Начнем с установки пакета для работы с *обычными* числами, то есть элементарными числами нашего языка. Мы пометим их символом `scheme-number`. Арифметические операции этого пакета — это элементарные арифметические процедуры (так что нет никакой нужды определять дополнительные процедуры для обработки непомеченных чисел). Поскольку каждая из них принимает по два аргумента, в таблицу они заносятся с ключом-списком `(scheme-number scheme-number)`:

```
(define (install-scheme-number-package)
  (define (tag x)
    (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number
       (lambda (x) (tag x)))
  'done)
```

Пользователи пакета Схемных чисел будут создавать (помеченные) элементарные числа с помощью процедуры

```
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```

Теперь, когда каркас обобщенной арифметической системы построен, мы можем без труда добавлять новые типы чисел. Вот пакет, который реализует арифметику рациональных чисел. Обратите внимание, что благодаря аддитивности мы можем без изменений использовать код рациональной арифметики из раздела 2.1.1 в виде внутренних процедур пакета:

```
(define (install-rational-package)
   ;; внутренние процедуры
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  (define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  (define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y))))
  (define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (denom x) (numer y))))
   ;; интерфейс к остальной системе
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
       (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
       (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational)
       (lambda (x y) (tag (div-rat x y))))

  (put 'make 'rational
       (lambda (n d) (tag (make-rat n d))))
  'done)

(define (make-rational n d)
  ((get 'make 'rational) n d))
```

Мы можем установить подобный пакет и для комплексных чисел, используя метку complex. При создании пакета мы извлекаем из таблицы операции make-from-real-imag и make-from-mag-ang, определенные в декартовом и полярном пакетах. Аддитивность позволяет нам использовать без изменений в качестве внутренних операций процедуры add-complex, sub-complex, mul-complex и div-complex из раздела 2.4.1.

```
(define (install-complex-package)
   ;; процедуры, импортируемые из декартова
   ;; и полярного пакетов
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))
   ;; внутренние процедуры
  (define (add-complex z1 z2)
    (make-from-real-imag (+ (real-part z1) (real-part z2))
                         (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (- (real-part z1) (real-part z2))
                         (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
    (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                       (+ (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                       (- (angle z1) (angle z2))))

   ;; интерфейс к остальной системе
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2) (tag (add-complex z1 z2))))
  (put 'sub '(complex complex)
       (lambda (z1 z2) (tag (sub-complex z1 z2))))
  (put 'mul '(complex complex)
       (lambda (z1 z2) (tag (mul-complex z1 z2))))
  (put 'div '(complex complex)
       (lambda (z1 z2) (tag (div-complex z1 z2))))
  (put 'make-from-real-imag 'complex
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'complex
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Вне комплексного пакета программы могут создавать комплексные числа либо из действительной и мнимой части, либо из модуля и аргумента. Обратите внимание, как нижележащие процедуры, которые были изначально определены в декартовом и полярном пакете, экспортируются в комплексный пакет, а оттуда во внешний мир.

```
(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
```

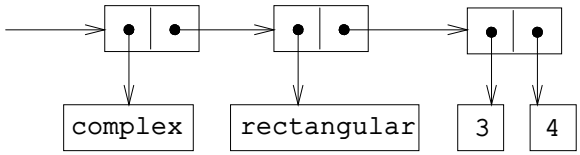


Рис. 2.24. Представление $3 + 4i$ в декартовой форме.

```
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))
```

Здесь мы имеем двухуровневую систему меток. Типичное комплексное число, например $3 + 4i$ в декартовой форме, будет представлено так, как показано на рисунке 2.24. Внешняя метка (`complex`) используется, чтобы отнести число к пакету комплексных чисел. Внутри комплексного пакета вторая метка (`rectangular`) относит число к декартову пакету. В большой и сложной системе может быть несколько уровней, каждый из которых связан со следующим при помощи обобщенных операций. Когда объект данных передается «вниз», внешняя метка, которая используется для отнесения к нужному пакету, отрывается (при помощи вызова `contents`), и следующий уровень меток (если таковой имеется) становится доступным для дальнейшей диспетчеризации.

В приведенных пакетах мы использовали `add-rat`, `add-complex` и другие арифметические процедуры ровно в таком виде, как они были написаны с самого начала. Но когда эти определения оказываются внутри различных процедур установки, отпадает необходимость давать им различные имена: мы могли бы просто назвать их в обоих пакетах `add`, `sub`, `mul` и `div`.

Упражнение 2.77.
Хьюго Дум пытается вычислить выражение `(magnitude z)`, где `z` — объект, показанный на рис. 2.24. К своему удивлению, вместо ответа 5 он получает сообщение об ошибке от `apply-generic`, гласящее, что у операции `magnitude` нет методов для типа `(complex)`. Он показывает результат Лизе П. Хакер. Та заявляет: "Дело в том, что селекторы комплексных чисел для чисел с меткой `complex` определены не были, а были только для чисел с меткой `polar` и `rectangular`. Все, что требуется, чтобы заставить это работать — это добавить к пакету `complex` следующее:"

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

Подробно опишите, почему это работает. В качестве примера, проследите все процедуры, которые вызываются при вычислении `(magnitude z)`, где `z` — объект, показанный на рис. 2.24. В частности, сколько раз вызывается `apply-generic`? На какую процедуру она диспетчирует в каждом случае?

Упражнение 2.78.
В пакете `scheme-number` внутренние процедуры, в сущности, ничего не делают, только вызывают элементарные процедуры `+`, `-`, и т.д. Прямо использовать примитивы языка не было возможности,

поскольку наша система меток типов требует, чтобы каждый объект данных был снабжен меткой. Однако на самом деле все реализации Лиспа имеют систему типов, которую они используют внутри себя. Элементарные процедуры вроде `symbol?` или `number?` определяют, относится ли объект к определенному типу. Измените определения `type-tag`, `contents` и `attach-tag` из раздела 2.4.2 так, чтобы наша обобщенная система использовала внутреннюю систему типов Scheme. То есть, система должна работать так же, как раньше, но только обычные числа должны быть представлены просто в виде чисел языка Scheme, а не в виде пары, у которой первый элемент символ `scheme-number`.

Упражнение 2.79.
Определите обобщенный предикат равенства `equ?`, который проверяет два числа на равенство, и вставьте его в пакет обобщенной арифметики. Операция должна работать для обычных чисел, рациональных и комплексных.

Упражнение 2.80.
Определите обобщенный предикат `=zero?`, который проверяет, равен ли его аргумент нулю, и вставьте его в пакет обобщенной арифметики. Предикат должен работать для обычных, рациональных и комплексных чисел.

2.5.2. Сочетание данных различных типов

Мы видели, как можно построить объединенную арифметическую систему, которая охватывает обыкновенные числа, комплексные числа, рациональные числа и любые другие типы чисел, которые нам может потребоваться изобрести, но мы упустили важный момент. Операции, которые мы до сих пор определили, рассматривают различные типы данных как совершенно независимые. Таким образом, есть отдельные пакеты для сложения, например, двух обыкновенных чисел и двух комплексных чисел. Мы до сих пор не учитывали того, что имеет смысл определять операции, которые пересекают границы типов, например, сложение комплексного числа с обычным. Мы затратили немалые усилия, чтобы воздвигнуть барьеры между частями наших программ, так, чтобы их можно было разрабатывать и понимать по отдельности. Нам бы хотелось добавить операции со смешанными типами по возможности аккуратно, так, чтобы мы их могли поддерживать, не нарушая всерьез границ модулей.

Один из способов управления операциями со смешанными типами состоит в том, чтобы определить отдельную процедуру для каждого сочетания типов, для которых операция имеет смысл. Например, мы могли бы расширить пакет работы с комплексными числами и включить туда процедуру сложения комплексных чисел с обычными, занося ее в таблицу с меткой `(complex scheme-number)`[49]:

```
 ;; включается в пакет комплексных чисел
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x)
                       (imag-part z)))

(put 'add '(complex scheme-number)
     (lambda (z x) (tag (add-complex-to-schemenum z x))))
```

[49]Придется к тому же написать почти такую же процедуру для типа `(scheme-number complex)`.

Этот метод работает, но он очень громоздок. При такой системе стоимость введения нового типа не сводится к тому, чтобы построить пакет процедур для этого типа, но включает еще построение и установку процедур, осуществляющих операции со смешанными типами. Это запросто может потребовать больше кода, чем нужно, чтобы определить операции над самим типом. Кроме того, этот метод подрывает нашу способность сочетать отдельные пакеты аддитивно, или, по крайней мере, ограничивать степень, в которой реализация отдельного пакета должна принимать другие пакеты в расчет. Скажем, в вышеприведенном примере, кажется естественным, чтобы ответственность за обработку смешанных операций с обычными и комплексными числами лежала на комплексном пакете. Однако сочетание рациональных и комплексных чисел может осуществляться комплексным пакетом, рациональным пакетом, или каким-нибудь третьим, который пользуется операциями, извлеченными из этих двух. Формулировка ясных правил разделения ответственности между пакетами может стать непосильной задачей при разработке систем с многими пакетами и многими смешанными операциями.

Приведение типов

В ситуации общего вида, когда совершенно несвязанные друг с другом операции применяются к совершенно друг с другом не связанным типам, явное написание операций со смешанными типами, как бы это ни было громоздко, — все, на что мы можем рассчитывать. К счастью, обычно мы можем воспользоваться дополнительной структурой, которая часто в скрытом виде присутствует в нашей системе типов. Часто различные типы данных не совсем независимы, и каким-то образом объекты одного типа можно рассматривать как объекты другого. Такой процесс называется *приведением типов* (coercion). Например, если нас просят найти некоторую арифметическую комбинацию обычного числа и комплексного, то мы можем рассматривать обычное число как такое комплексное, у которого мнимая часть равна нулю. Это сводит нашу задачу к сочетанию двух комплексных чисел, а с этим может стандартным способом справиться пакет комплексной арифметики.

В общем случае мы можем реализовать эту идею, проектируя процедуры приведения типа, которые переводят объект одного типа в эквивалентный ему объект другого типа. Вот типичная процедура приведения типов, которая преобразует данное обыкновенное число в комплексное, у которого есть действительная часть, а мнимая равна нулю:

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

Мы записываем процедуры приведения типа в специальную таблицу приведения типов, проиндексированную именами двух типов:

```
(put-coercion 'scheme-number 'complex scheme-number->complex)
```

(Предполагается, что для работы с этой таблицей существуют процедуры `put-coercion` и `get-coercion`.) Как правило, часть ячеек этой таблицы будет пуста, потому что в общем случае невозможно привести произвольный объект произвольного типа ко всем остальным типам. К примеру, нет способа привести произвольное комплексное число к обыкновенному, так что в таблице не появится общая процедура `complex->scheme-number`.

Когда таблица приведения типов построена, мы можем работать с приведением стандартным образом, приспособив для этого процедуру `apply-generic` из раздела 2.4.3. Когда нас просят применить операцию, мы первым делом, как и раньше, проверяем, не определена ли уже операция для типов аргументов. Если да, мы вызываем процедуру, найденную в таблице операций и типов. Если нет, мы пробуем применить приведение типов. Для простоты мы рассматриваем только тот случай, когда аргументов два[50]. Мы проверяем таблицу преобразования типов и смотрим, можно ли объект первого типа привести ко второму типу. Если да, осуществляем приведение и снова пробуем операцию. Если объекты первого типа в общем случае ко второму не приводятся, мы пробуем приведение в обратном направлении и смотрим, нет ли способа привести второй аргумент к типу первого. Наконец, если нет никакого известного способа привести один тип к другому, мы сдаемся. Вот эта процедура:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                         (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                         (apply-generic op a1 (t2->t1 a2)))
                        (else
                         (error "Нет метода для этих типов"
                                (list op type-tags))))))
              (error "Нет метода для этих типов"
                     (list op type-tags)))))))
```

Такая схема приведения типов имеет много преимуществ перед методом явного определения смешанных операций, как это описано выше. Хотя нам по-прежнему требуется писать процедуры приведения для связи типов (возможно, n^2 процедур для системы с n типами), для каждой пары типов нам нужно написать только одну процедуру, а не по процедуре на каждый набор типов и каждую обобщенную операцию[51]. Здесь мы рассчитываем на то, что требуемая трансформация типов зависит только от самих типов, и не зависит от операции, которую требуется применить.

[50] Обобщение см. в упражнении 2.82.

[51] Если мы умные, мы обычно можем обойтись меньше, чем n^2 процедурами приведения типа. Например, если мы знаем, как из типа 1 получить тип 2, а из типа 2 тип 3, то можно использовать это знание для преобразования из 1 в 3. Это может сильно уменьшить количество процедур, которые надо явно задавать при введении нового типа в систему. Если нам не страшно ввести в свою систему требуемый уровень изощренности, мы можем заставить ее искать по «графу» отношений между типами и автоматически порождать все процедуры приведения типов, которые можно вывести из тех, которые явно заданы.

Рис. 2.25. Башня типов

С другой стороны, могут существовать приложения, для которых наша схема приведения недостаточно обща. Даже когда ни один из объектов, которые требуется сочетать, не может быть приведен к типу другого, операция может оказаться применимой, если преобразовать оба объекта к третьему типу. Чтобы справиться с такой степенью сложности и по-прежнему сохранить модульность в наших программах, обычно необходимо строить такие системы, которые еще в большей степени используют структуру в отношениях между типами, как мы сейчас расскажем.

Иерархии типов

Описанная выше схема приведения типов опиралась на существование естественных отношений между парами типов. Часто в отношениях типов между собой существует более «глобальная» структура. Предположим, например, что мы строим обобщенную арифметическую систему, которая должна работать с целыми, рациональными, действительными и комплексными числами. В такой системе вполне естественно будет рассматривать целое число как частный случай рационального, которое в свою очередь является частным случаем действительного числа, которое опять-таки частный случай комплексного числа. Здесь мы имеем так называемую *иерархию типов* (hierarchy of types) в которой, например, целые числа являются *подтипом* (subtype) рациональных чисел (то есть всякая операция, которую можно применить к рациональному числу, применима и к целым). Соответственно, мы говорим, что рациональные числа являются *надтипом* (supertype) целых. Та конкретная иерархия, с которой мы имеем дело здесь, имеет очень простой вид, а именно, у каждого типа не более одного надтипа и не более одного подтипа. Такая структура, называемая *башня типов* (tower), показана на рис. 2.25.

Если у нас имеется башня типов, то задача добавления нового типа в систему сильно упрощается, поскольку требуется указать только то, каким образом новый тип включается в ближайший надтип сверху и то, каким образом он является надтипом типа, который находится прямо под ним. Например, если мы хотим к комплексному числу добавить целое, нам не нужно специально определять процедуру приведения типа `integer->complex`. Вместо этого мы определяем, как можно перевести целое число в рациональное, рациональное в действительное, и как действительное число переводится в комплексное. Потом мы позволяем системе преобразовать целое число в комплексное через все эти промежуточные шаги и складываем два комплексных числа.

Можно переопределить процедуру `apply-generic` следующим образом: для каж-

дого типа требуется указать процедуру `raise`, которая «поднимает» объекты этого типа на один уровень в башне. В таком случае, когда системе требуется обработать объекты различных типов, она может последовательно поднимать объекты более низких типов, пока все объекты не окажутся на одном и том же уровне башни. (Упражнения 2.83 и 2.84 касаются деталей реализации такой стратегии.)

Еще одно преимущество башни состоит в том, что легко реализуется представление о том, что всякий тип «наследует» операции своего надтипа. Например, если мы не даем особой процедуры для нахождения действительной части целого числа, мы все равно можем ожидать, что `real-part` будет для них определена в силу того, что целые числа являются подтипом комплексных. В случае башни мы можем устроить так, чтобы это происходило само собой, модифицировав `apply-generic`. Если требуемая операция не определена непосредственно для типа данного объекта, мы поднимаем его до надтипа и пробуем еще раз. Так мы ползем вверх по башне, преобразуя по пути свой аргумент, пока мы либо не найдем уровень, на котором требуемую операцию можно произвести, либо не доберемся до вершины (и в таком случае мы сдаемся).

Еще одно преимущество башни над иерархией более общего типа состоит в том, что она дает нам простой способ «опустить» объект данных до его простейшего представления. Например, если мы складываем $2 + 3i$ с $4 - 3i$, было бы приятно в качестве ответа получить целое 6, а не комплексное $6 + 0i$. В упражнении 2.85 обсуждается способ, которым такую понижающую операцию можно реализовать. (Сложность в том, что нам нужен общий способ отличить объекты, которые можно понизить, вроде $6 + 0i$, от тех, которые понизить нельзя, например $6 + 2i$.)

Неадекватность иерархий

Если типы данных в нашей системе естественным образом выстраиваются в башню, это сильно упрощает задачу работы с обобщенными операциями над различными типами, как мы только что видели. К сожалению, обычно это не так. На рисунке 2.26 показано более сложное устройство набора типов, а именно отношения между различными типами геометрических фигур. Мы видим, что в общем случае у типа может быть более одного подтипа. Например, и треугольники, и четырехугольники являются разновидностями многоугольников. В дополнение к этому, у типа может быть более одного надтипа. Например, равнобедренный прямоугольный треугольник можно рассматривать и как равнобедренный, и как прямоугольный. Вопрос с множественными надтипами особенно болезнен, поскольку из-за него теряется единый способ «поднять» тип по иерархии. Нахождение «правильного» надтипа, в котором требуется применить операцию к объекту, может потребовать долгого поиска по всей сети типов внутри процедуры вроде `apply-generic`. Поскольку в общем случае у типа несколько подтипов, существует подобная проблема и в сдвиге значения «вниз» по иерархии. Работа с большим количеством связанных типов без потери модульности при разработке больших систем – задача очень трудная, и в этой области сейчас ведется много исследований[52].

[52] Данное утверждение, которое присутствует и в первом издании этой книги, сейчас столь же верно, как и двенадцать лет назад. Разработка удобного, достаточно общего способа выражать отношения между различными типами сущностей (то, что философы называют «онтологией»), оказывается невероятно сложным делом. Основная разница между той путаницей, которая была десять лет назад, и той, которая есть сейчас, состоит в том, что теперь множество неадекватных онтологических теорий оказалось воплощено в массе соответственно неадекватных языков программирования. Например, львиная доля сложности объектно-

Рис. 2.26. Отношения между типами геометрических фигур.

Упражнение 2.81.
Хьюго Дум заметил, что `apply-generic` может пытаться привести аргументы к типу друг друга даже тогда, когда их типы и так совпадают. Следовательно, решает он, нам нужно вставить в таблицу приведения процедуры, которые «приводят» аргументы каждого типа к нему самому. Например, в дополнение к приведению `scheme-number->complex`, описанному выше, он бы написал еще:

```
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
(put-coercion 'scheme-number 'scheme-number
              scheme-number->scheme-number)
(put-coercion 'complex 'complex complex->complex)
```

а. Если установлены процедуры приведения типов, написанные Хьюго, что произойдет, когда `apply-generic` будет вызвана с двумя аргументами типа `scheme-number` или двумя аргументами типа `complex` для операции, которая не находится в таблице для этих типов? Допустим, например, что мы определили обобщенную процедуру возведения в степень:

```
(define (exp x y) (apply-generic 'exp x y))
```

и добавили процедуру возведения в степень в пакет чисел Scheme и ни в какой другой:

```
 ;; Следующие строки добавляются в пакет scheme-number
(put 'exp '(scheme-number scheme-number)
     (lambda (x y) (tag (expt x y))))  ;используется
                                       ;элементарная  expt
```

Что произойдет, если мы позовем `exp` с двумя комплексными числами в качестве аргументов?

б. Прав ли Хьюго, что нужно что-то сделать с приведением однотипных аргументов, или все и так работает правильно?

в. Измените `apply-generic` так, чтобы она не пыталась применить приведение, если у обоих аргументов один и тот же тип.

Упражнение 2.82.
Покажите, как обобщить `apply-generic` так, чтобы она обрабатывала приведение в общем случае с несколькими аргументами. Один из способов состоит в том, чтобы попытаться сначала привести все аргументы к типу первого, потом к типу второго, и так далее. Приведите пример, когда эта стратегия (а также двухаргументная версия, описанная выше) недостаточно обща. (Подсказка: рассмотрите случай, когда в таблице есть какие-то подходящие операции со смешанными типами, но обращения к ним не произойдет.)

Упражнение 2.83.
Предположим, что Вы разрабатываете обобщенную арифметическую систему для работы с башней типов, показанной на рис. 2.25: целые, рациональные, действительные, комплексные. Для каждого

ориентированных языков программирования — и мелких невразумительных различий между современными объектно-ориентированными языками, — сосредоточена в том, как рассматриваются обобщенные операции над взаимосвязанными типами. Наше собственное описание вычислительных объектов в главе 3 полностью избегает этих вопросов. Читатели, знакомые с объектно-ориентированным программированием, заметят, что нам есть, что сказать в главе 3 о локальном состоянии, но мы ни разу не упоминаем «классы» или «наследование». Мы подозреваем, что на самом деле эти проблемы нельзя рассматривать только в терминах проектирования языков программирования, без обращения к работам по представлению знаний и автоматическому логическому выводу.

из типов (кроме комплексного), разработайте процедуру, поднимающую объект на один уровень в башне. Покажите, как ввести обобщенную операцию `raise`, которая будет работать для всех типов (кроме комплексных чисел).

Упражнение 2.84.
Используя операцию `raise` из упражнения 2.83, измените процедуру `apply-generic` так, чтобы она приводила аргументы к одному типу путем последовательного подъема, как описано в этом разделе. Потребуется придумать способ проверки, какой из двух типов выше по башне. Сделайте это способом, «совместимым» с остальной системой, так, чтобы не возникало проблем при добавлении к башне новых типов.

Упражнение 2.85.
В этом разделе упоминался метод «упрощения» объекта данных путем спуска его по башне насколько возможно вниз. Разработайте процедуру `drop`, которая делает это для башни, описанной в упражнении 2.83. Ключ к задаче состоит в том, что надо решить некоторым общим способом, можно ли понизить объект в типе. Например, комплексное число $1.5+0i$ можно опустить до `real`, комплексное число $1+0i$ до `integer`, а комплексное число $2+3i$ никуда понизить нельзя. Вот план того, как определить, можно ли понизить объект: для начала определите обобщенную операцию `project`, которая «сталкивает» объект вниз по башне. Например, проекция комплексного числа будет состоять в отбрасывании его мнимой части. Тогда число можно сдвинуть вниз в том случае, если, спроецировав его, а затем подняв обратно до исходного типа, мы получаем нечто, равное исходному числу. Покажите как реализовать эту идею в деталях, написав процедуру `drop`, которая опускает объект как можно ниже. Потребуется разработать различные операции проекции[53] и установить `project` в системе в качестве обобщенной операции. Вам также потребуется обобщенный предикат равенства, подобный описанному в упражнении 2.79. Наконец, используя `drop`, перепишите `apply-generic` из упражнения 2.84, чтобы она «упрощала» свои результаты.

Упражнение 2.86.
Допустим, нам хочется работать с комплексными числами, чьи действительные и мнимые части, модули и аргументы могут быть обыкновенными числами, рациональными числами либо любыми другими, какие нам захочется добавить к системе. Опишите и реализуйте изменения в системе, которые потребуются, чтобы добавить такую возможность. Вам придется определить операции вроде `sine` (синус) и `cosine` (косинус), обобщенные на обыкновенные и рациональные числа.

2.5.3. Пример: символьная алгебра

Обработка символьных алгебраических выражений представляет собой сложный процесс, который иллюстрирует многие тяжелейшие проблемы, возникающие при проектировании больших систем. В общем случае, алгебраическое выражение можно рассматривать как иерархическую структуру, дерево операций, применяемых к операндам. Мы можем строить алгебраические выражения, начиная с элементарных объектов, таких, как константы и переменные, и комбинируя их с помощью алгебраических операций, таких, как сложение и умножение. Как и в других языках, мы формируем абстракции, которые позволяют нам именовать составные объекты при помощи простых терминов.

[53] Действительное число можно спроецировать на целое при помощи примитива `round`, который возвращает целое число, ближайшее к своему аргументу.

В символьной алгебре типичными абстракциями являются такие понятия, как линейная комбинация, многочлен, рациональная или тригонометрическая функция. Мы можем рассматривать их как составные «типы», которые часто бывают полезны при управлении обработкой выражений. Например, выражение

$$x^2 \sin(y^2 + 1) + \cos 2y + \cos(y^3 - 2y^2)$$

можно рассматривать как многочлен по x с коэффициентами, которые являются тригонометрическими функциями многочленов по y, чьи коэффициенты, в свою очередь, целые числа.

Здесь мы не будем пытаться разработать полную систему для работы с алгебраическими выражениями. Такие системы — очень сложные программы, использующие глубокие математические знания и элегантные алгоритмы. Мы собираемся описать только одну простую, но важную часть алгебраических операций — арифметику многочленов. Мы проиллюстрируем типы решений, которые приходится принимать разработчику подобной системы, и то, как применить идеи абстракции данных и обобщенных операций, чтобы с их помощью организовать работу.

Арифметика многочленов

Первая задача при разработке системы для проведения арифметических операций над многочленами — решить, что именно представляет собой многочлен. Обычно многочлены определяют по отношению к тем или иным переменным. Ради простоты, мы ограничимся многочленами только с одной переменной[54]. Мы определяем многочлен как сумму термов, каждый из которых представляет собой либо коэффициент, либо переменную, возведенную в степень, либо произведение того и другого. Коэффициент определяется как алгебраическое выражение, не зависящее от переменной многочлена. Например,

$$5x^2 + 3x + 7$$

есть простой многочлен с переменной x, а

$$(y^2 + 1)x^3 + (2y)x + 1$$

есть многочлен по x, коэффициенты которого — многочлены по y.

Уже здесь мы сталкиваемся с несколькими неудобными деталями. Является ли первый из приведенных многочленов тем же объектом, что $5y^2 + 3y + 7$? Разумный ответ на этот вопрос таков: «если мы рассматриваем многочлен как чисто математическую функцию, то да, но если как синтаксическую форму, то нет». Второй пример алгебраически эквивалентен многочлену по y, коэффициенты которого — многочлены по x. Должна ли наша система распознавать это? Наконец, существуют другие способы представления многочленов — например, как произведение линейных множителей, как множество корней (для многочлена с одной переменной), или как список значений многочлена в за-

[54] С другой стороны, мы разрешаем многочлены, коэффициенты которых сами по себе являются многочленами от других переменных. По существу, это дает нам такую же выразительную силу, что и у полной системы со многими переменными, хотя и ведет к проблемам приведения, как это обсуждается ниже.

данном множестве точек[55]. Мы можем обойти эти вопросы, решив, что в нашей системе алгебраических вычислений «многочлен» будет определенной синтаксической формой, а не ее математическим значением.

Теперь пора подумать, как мы будем осуществлять арифметические операции над многочленами. В нашей упрощенной системе мы рассмотрим только сложение и умножение. Более того, мы будем настаивать, чтобы два многочлена, над которыми проводится операция, имели одну и ту же переменную.

К проектированию системы мы приступим, следуя уже знакомой нам дисциплине абстракции данных. Мы будем представлять многочлены в виде структуры данных под названием *poly*, которая состоит из переменной и набора термов. Мы предполагаем, что имеются селекторы `variable` и `term-list`, которые получают из poly эти данные, и конструктор `make-poly`, который собирает poly из переменной и списка термов. Переменная будет просто символом, так что для сравнения переменных мы сможем использовать процедуру `same-variable?` из раздела 2.3.2. Следующие процедуры определяют сложение и умножение многочленов:

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1)
                            (term-list p2)))
      (error "Многочлены от разных переменных -- ADD-POLY"
             (list p1 p2))))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (mul-terms (term-list p1)
                            (term-list p2)))
      (error "Многочлены от разных переменных -- MUL-POLY"
             (list p1 p2))))
```

Чтобы включить многочлены в нашу обобщенную арифметическую систему, нам потребуется снабдить их метками типа. Мы будем пользоваться меткой `polynomial` и вносить соответствующие операции над помеченными многочленами в таблицу операций. Весь свой код мы включим в процедуру установки пакета многочленов, подобно пакетам из раздела 2.5.1:

```
(define (install-polynomial-package)
   ;; внутренние процедуры
   ;; представление poly
  (define (make-poly variable term-list)
    (cons variable term-list))
```

[55] В случае многочленов с одной переменной задание значений многочлена в определенном множестве точек может быть особенно удачным представлением. Арифметика многочленов получается чрезвычайно простой. Чтобы получить, скажем, сумму двух представленных таким образом многочленов, достаточно сложить значения в соответствующих точках. Чтобы перейти обратно к более привычному представлению, можно использовать формулу интерполяции Лагранжа, которая показывает, как восстановить коэффициенты многочлена степени n, имея его значения в $n + 1$ точке.

```
(define (variable p) (car p))
(define (term-list p) (cdr p))
⟨процедуры  same-variable? и  variable? из раздела 2.3.2⟩

 ;; представление термов и списков термов
⟨процедуры  adjoin-term ...  coeff из текста ниже⟩

(define (add-poly p1 p2) ... )
⟨процедуры, которыми пользуется  add-poly⟩

(define (mul-poly p1 p2) ... )
⟨процедуры, которыми пользуется  mul-poly⟩

 ;; интерфейс к остальной системе
(define (tag p) (attach-tag 'polynomial p))
(put 'add '(polynomial polynomial)
     (lambda (p1 p2) (tag (add-poly p1 p2))))
(put 'mul '(polynomial polynomial)
     (lambda (p1 p2) (tag (mul-poly p1 p2))))

(put 'make 'polynomial
     (lambda (var terms) (tag (make-poly var terms))))
'done)
```

Сложение многочленов происходит по термам. Термы одинакового порядка (то есть имеющие одинаковую степень переменной многочлена) нужно скомбинировать. Это делается при помощи порождения нового терма того же порядка, в котором коэффициент является суммой коэффициентов слагаемых. Термы одного слагаемого, для которых нет соответствия в другом, просто добавляются к порождаемому многочлену-сумме.

Для того, чтобы работать со списками термов, мы предположим, что имеется конструктор `the-empty-termlist`, который возвращает пустой список термов, и конструктор `adjoin-term`, который добавляет к списку термов еще один. Кроме того, мы предположим, что имеется предикат `empty-termlist?`, который говорит, пуст ли данный список, селектор `first-term`, который получает из списка термов тот, у которого наибольший порядок, и селектор `rest-terms`, который возвращает все термы, кроме того, у которого наибольший порядок. Мы предполагаем, что для работы с термами у нас есть конструктор `make-term`, строящий терм с указанными порядком и коэффициентом, и селекторы `order` и `coeff`, которые, соответственно, возвращают порядок и коэффициент терма. Эти операции позволяют нам рассматривать и термы, и их списки как абстракции данных, о конкретной реализации которых мы можем позаботиться отдельно.

Вот процедура, которая строит список термов для суммы двух многочленов[56]:

```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
```

[56]Эта операция очень похожа на процедуру объединения множеств `union-set`, которую мы разработали в упражнении 2.62. На самом деле, если мы будем рассматривать многочлены как множества, упорядоченные по степени переменной, то программа, которая порождает список термов для суммы, окажется почти идентична `union-set`.

```
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1)) (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term
                   t1 (add-terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term
                   t2 (add-terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term
                   (make-term (order t1)
                              (add (coeff t1) (coeff t2)))
                   (add-terms (rest-terms L1)
                              (rest-terms L2)))))))))
```

Самая важная деталь, которую здесь надо заметить, — это что для сложения коэффициентов комбинируемых термов мы использовали обобщенную процедуру `add`. Это влечет глубокие последствия, как мы увидим ниже.

Чтобы перемножить два списка термов, мы умножаем каждый терм из первого списка на все термы второго, используя в цикле `mul-term-by-allterms`, которая умножает указанный терм на все термы указанного списка. Получившиеся списки термов (по одному на каждый терм в первом списке) накапливаются и образуют сумму. Перемножение двух термов дает терм, порядок которого равен сумме порядков множителей, а коэффициент равен произведению коэффициентов множителей:

```
(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2))))

(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (adjoin-term
         (make-term (+ (order t1) (order t2))
                    (mul (coeff t1) (coeff t2)))
         (mul-term-by-all-terms t1 (rest-terms L))))))
```

Вот и все, что нам требуется для сложения и умножения многочленов. Обратите внимание, что, поскольку мы работаем с термами при помощи обобщенных процедур `add` и `mul`, наш пакет работы с многочленами автоматически оказывается в состоянии обрабатывать любой тип коэффициента, о котором знает обобщенный арифметический пакет. Если мы подключим механизм приведения типов, подобный тому, который обсуждался в разделе 2.5.2, то мы автоматически окажемся способны производить операции над многочленами с коэффициентами различных типов, например

$$[3x^2 + (2 + 3i)x + 7] \cdot [x^4 + \frac{2}{3}x^2 + (5 + 3i)]$$

Поскольку мы установили процедуры сложения и умножения многочленов `add-poly` и `mul-poly` в обобщенной арифметической системе в качестве операций `add` и `mul` для типа `polynomial`, наша система оказывается автоматически способна производить операции над многочленами вроде

$$[(y+1)x^2 + (y^2+1)x + (y-1)] \cdot [(y-2)x + (y^3+7)]$$

Причина этого в том, что, когда система пытается скомбинировать коэффициенты, она диспетчирует через `add` и `mul`. Поскольку коэффициенты сами по себе являются многочленами (по y), они будут скомбинированы при помощи `add-poly` и `mul-poly`. В результате получается своего рода «рекурсия, управляемая данными», где, например, вызов `mul-poly` приводит к рекурсивным вызовам `mul-poly` для того, чтобы скомбинировать коэффициенты. Если бы коэффициенты коэффициентов сами по себе были бы многочленами (это может потребоваться, если надо представить многочлены от трех переменных), программирование, управляемое данными, позаботится о том, чтобы система прошла еще через один уровень рекурсивных вызовов, и так далее, на столько уровней структуры, сколько требуют данные[57].

Представление списков термов

Наконец, мы сталкиваемся с задачей реализовать хорошее представление для списков термов. Список термов, в сущности, есть множество коэффициентов, проиндексированное порядком терма. Следовательно, любой из методов представления множеств, описанных в 2.3.3, годится для этой задачи. С другой стороны, наши процедуры `add-terms` и `mul-terms` всегда обрабатывают списки термов последовательно от наибольшего порядка к наименьшему, так что мы будем использовать некоторую разновидность упорядоченного представления.

Как нам устроить структуру данных, которая представляет список термов? Одно из соображений — «плотность» многочленов, с которыми мы будем работать. Многочлен называется *плотным* (dense), если в термах с большинством порядков у него ненулевые коэффициенты. Если же в нем много нулевых коэффициентов, он называется *разреженным* (sparse). Например,

$$A : x^5 + 2x^4 + 3x^2 - 2x - 5$$

плотный многочлен, а

$$B : x^{100} + 2x^2 + 1$$

разреженный.

Списки термов плотных многочленов эффективнее всего представлять в виде списков коэффициентов. Например, A в приведенном примере удобно представляется в виде `(1`

[57] Чтобы все это работало совершенно гладко, потребуется добавить в нашу систему обобщенной арифметики возможность привести «число» к типу многочлена, рассматривая его как многочлен степени ноль, коэффициентом которого является данное число. Это нужно, если мы хотим осуществлять операции вроде

$$[x^2 + (y+1)x + 5] + [x^2 + 2x + 1]$$

где требуется сложить коэффициент $y+1$ с коэффициентом 2.

`2 0 3 -2 -5)`. Порядок терма в таком представлении есть длина списка, начинающегося с этого коэффициента, уменьшенная на 1[58]. Для разреженного многочлена вроде B такое представление будет ужасным: получится громадный список нулей, в котором изредка попадаются одинокие ненулевые термы. Более разумно представление разреженного многочлена в виде списка ненулевых термов, где каждый терм есть список, содержащий порядок терма и коэффициент при этом порядке. При такой схеме многочлен B эффективно представляется в виде `((100 1) (2 2) (0 1))`. Поскольку большинство операций над многочленами применяется к разреженным многочленам, мы используем это представление. Мы предполагаем, что список термов представляется в виде списка, элементами которого являются термы, упорядоченные от бóльшего порядка к меньшему. После того, как решение принято, реализация селекторов и конструкторов для термов и списков термов не представляет трудностей[59]:

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))

(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

где `=zero?` работает так, как определяется в упражнении 2.80 (см. также ниже упражнение 2.87).

Пользователи многочленного пакета будут создавать (помеченные) многочлены при помощи процедуры:

```
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

Упражнение 2.87.
Установите `=zero?` для многочленов в обобщенный арифметический пакет. Это позволит `adjoin-term` работать с многочленами, чьи коэффициенты сами по себе многочлены.

---

[58] В этих примерах многочленов мы предполагаем, что реализовали обобщенную арифметическую систему при помощи механизма типов, предложенного в упражнении 2.78. Таким образом, коэффициенты, которые являются обыкновенными числами, будут представлены самими числами, а не парами с первым элементом — символом `scheme-number`.

[59] Хотя мы предполагаем, что списки термов упорядочены, мы реализовали `adjoin-term` путем простого `cons` к существующему списку термов. Нам это может сойти с рук, пока мы гарантируем, что процедуры (вроде `add-terms`), которые используют `adjoin-term`, всегда вызывают ее с термом бóльшего порядка, чем уже есть в списке. Если бы нам не хотелось давать такую гарантию, мы могли бы реализовать `adjoin-term` подобно конструктору `adjoin-set` для представления множеств в виде упорядоченных списков (упражнение 2.61).

Упражнение 2.88.
Расширьте систему многочленов так, чтобы она включала вычитание многочленов. (Подсказка: может оказаться полезным определить обобщенную операцию смены знака.)

Упражнение 2.89.
Определите процедуры, которые реализуют представление в виде списка термов, описанное выше как подходящее для плотных многочленов.

Упражнение 2.90.
Допустим, что мы хотим реализовать систему многочленов, которая эффективна как для плотных, так и для разреженных многочленов. Один из способов это сделать заключается в том, чтобы разрешить в системе оба типа представления. Ситуация аналогична примеру с комплексными числами из раздела 2.4, где мы позволили сосуществовать декартову и полярному представлению. Чтобы добиться этого, нам придется различать виды списков термов и сделать операции над списками термов обобщенными. Перепроектируйте систему с многочленами так, чтобы это обобщение было реализовано. Это потребует большого труда, а не только локальных изменений.

Упражнение 2.91.
Многочлены с одной переменной можно делить друг на друга, получая частное и остаток. Например, $\dfrac{x^5-1}{x^2-1} = x^3 + x$, остаток $x - 1$.

Деление можно производить в столбик. А именно, разделим старший член делимого на старший член делителя. В результате получится первый терм частного. Затем умножим результат на делитель, вычтем получившийся многочлен из делимого и, рекурсивно деля разность на делитель, получим оставшуюся часть частного. Останавливаемся, когда порядок делителя превысит порядок делимого, и объявляем остатком то, что тогда будет называться делимым. Кроме того, если когда-нибудь делимое окажется нулем, возвращаем ноль в качестве и частного, и остатка.

Процедуру `div-poly` можно разработать, следуя образцу `add-poly` и `mul-poly`. Процедура проверяет, одна ли и та же у многочленов переменная. Если это так, `div-poly` откусывает переменную и передает задачу в `div-terms`, которая производит операцию деления над списками термов. Наконец, `div-poly` прикрепляет переменную к результату, который выдает `div-terms`. Удобно сделать так, чтобы `div-terms` выдавала и частное, и остаток при делении. Она может брать в качестве аргументов два терма и выдавать список, состоящий из списка термов частного и списка термов остатка.

Закончите следующее определение `div-terms`, вставив недостающие выражения. Используйте ее, чтобы реализовать `div-poly`, которая получает в виде аргументов два экземпляра poly, а выдает список из poly–частного и poly–остатка.

```
(define (div-terms L1 L2)
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     ⟨рекурсивно вычислить оставшуюся
```

```
        часть результата⟩
        ))
⟨сформировать окончательный результат⟩
))))))
```

Иерархии типов в символьной алгебре

Наша система обработки многочленов показывает, как объекты одного типа (многочлены) могут на самом деле быть составными сущностями, содержащими в качестве частей объекты многих различных типов. При определении обобщенных операций это не составляет никакой реальной сложности. Нужно только установить соответствующие обобщенные операции для выполнения необходимых действий над частями составных типов. В сущности, мы видели, что многочлены образуют своего рода «рекурсивную абстракцию данных», в том смысле, что части многочленов сами по себе могут быть многочленами. Наши обобщенные операции и наш стиль программирования, управляемого данными, могут справиться с такими трудностями без особого труда.

С другой стороны, алгебра многочленов представляет собой систему, в которой типы данных нельзя естественным образом выстроить в виде башни. Например, могут существовать многочлены по x, коэффициенты которых являются многочленами по y. Но могут существовать и многочлены по y, коэффициенты которых являются многочленами по x. Никакой из этих типов не находится «выше» другого ни в каком естественным смысле, и тем не менее элементы этих двух множеств часто требуется складывать. Для этого существует несколько способов. Одна из возможностей состоит в том, чтобы преобразовывать один из многочленов к типу другого путем раскрытия и переупорядочения термов, так, чтобы у обоих многочленов оказалась одна и та же главная переменная. Можно навязать данным башнеподобную структуру путем упорядочения переменных, и, таким образом, всегда преобразовывать любой многочлен к «канонической форме», где переменная с наибольшим приоритетом всегда доминирует, а переменные с меньшим оказываются зарыты в коэффициенты. Такая стратегия работает довольно хорошо, только преобразование может без особой необходимости «раздуть» многочлен, так что его станет неудобно читать и, возможно, менее эффективно обрабатывать. Для этой области структура башни определенно не является естественной, как и для любой другой области, где пользователь может изобретать новые типы динамически, используя старые в различных комбинирующих формах, таких как тригонометрические функции, последовательности степеней или интегралы.

Не должно вызывать удивления то, что управление приведением типов представляет серьезную проблему при разработке крупных систем алгебраических манипуляций. Существенная часть сложности таких систем связана с отношениями между различными типами. В сущности, можно честно признать, что мы до сих пор не до конца понимаем приведение типов. Мы даже не до конца осознаем понятие типа данных. Однако то, что мы знаем, дает нам солидные принципы структурирования и модуляризации, которые помогают в разработке больших систем.

Упражнение 2.92.
Использовав упорядочение переменных, расширьте пакет работы с многочленами так, чтобы сложение и умножение многочленов работало для многочленов с несколькими переменными. (Это не простая задача!)

Расширенное упражнение: рациональные функции

Можно расширить обобщенную арифметическую систему и включить в нее *рациональные функции* (rational functions). Это «дроби», в которых числитель и знаменатель являются многочленами, например

$$\frac{x+1}{x^3+1}$$

Система должна уметь складывать, вычитать. умножать и делить рациональные функции, а также осуществлять вычисления вроде

$$\frac{x+1}{x^3+1}+\frac{x}{x^2-1}=\frac{x^3+2x^2+3x+1}{x^4+x^3-x-1}$$

(здесь сумма упрощена при помощи сокращения общих множителей. Обычное «перекрестное умножение» дало бы многочлен четвертой степени в числителе и пятой в знаменателе.)

Если мы изменим пакет арифметики рациональных чисел так, чтобы он использовал обобщенные операции, то он будет делать то, что нам требуется, за исключением задачи приведения к наименьшему знаменателю.

Упражнение 2.93.
Модифицируйте пакет арифметики рациональных чисел, заставив его пользоваться обобщенными операциями, но при этом измените `make-rat`, чтобы она не пыталась сокращать дроби. Проверьте систему, применив `make-rational` к двум многочленам, и получив рациональную функцию

```
(define p1 (make-polynomial 'x '((2 1)(0 1))))
(define p2 (make-polynomial 'x '((3 1)(0 1))))
(define rf (make-rational p2 p1))
```

Сложите теперь `rf` саму с собой, используя `add`. Вы увидите, что процедура сложения не приводит дроби к наименьшему знаменателю.

Приводить дроби многочленов к наименьшему знаменателю мы можем, используя ту же самую идею, которой мы воспользовались для целых чисел: изменить `make-rat`, чтобы она делила и числитель, и знаменатель на их наибольший общий делитель. Понятие «наибольшего общего делителя» имеет смысл для многочленов. Более того, вычислять НОД для многочленов можно с помощью, в сущности, того же алгоритма Евклида, который работает на целых числах[60]. Вот целочисленная версия:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

[60]То, что алгоритм Евклида работает для многочленов, в алгебре формализуется утверждением, что многочлены образуют структуру, называемую *Евклидовым кольцом* (Euclidean ring). Евклидово кольцо — это структура, на которой определены сложение, вычитание и коммутативное умножение, а также некоторый способ сопоставить каждому элементу кольца x «меру» — неотрицательное целое число $m(x)$, обладающую следующими свойствами: $m(xy) \geq m(x)$ для любых ненулевых x и y, а также для любых x и y существует q, такое, что $y = qx + r$ и либо $r = 0$, либо $m(r) < m(x)$. С абстрактной точки зрения, это все, что нужно, чтобы доказать, что алгоритм Евклида работает. В случае целых чисел, мера m каждого числа есть его модуль. Для структуры многочленов мерой служит степень многочлена.

Взяв ее за основу, мы можем проделать очевидные изменения и определить операцию извлечения НОД, которая работает на списках термов:

```
(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (remainder-terms a b))))
```

где `remainder-terms` извлекает компоненту списка, соответствующую остатку, из списка, который возвращает операция деления списков термов `divterms`, реализованная в упражнении 2.91.

Упражнение 2.94.
Используя `div-terms`, напишите процедуру `remainder-terms`, и с ее помощью определите `gcd-terms`, как показано выше. Напишите теперь процедуру `gcd-polys`, которая вычисляет НОД двух многочленов. (Процедура должна сообщать об ошибке, если входные объекты являются многочленами от разных переменных.) Установите в систему обобщенную операцию `greatest-common-divisor`, которая для многочленов сводится к `gcd-poly`, а для обыкновенных чисел к обыкновенному `gcd`. В качестве проверки, попробуйте ввести

```
(define p1 (make-polynomial 'x '((4 1) (3 -1) (2 -2) (1 2))))
(define p2 (make-polynomial 'x '((3 1) (1 -1))))
(greatest-common-divisor p1 p2)
```

и проверьте результат вручную.

Упражнение 2.95.
Пусть P_1, P_2 и P_3 – многочлены

$$P_1 : x^2 - 2x + 1$$
$$P_2 : 11x^2 + 1$$
$$P_3 : 13x + 5$$

Теперь пусть Q_1 будет произведение P_1 и P_2, а Q_2 произведение P_1 и P_3. При помощи `greatest-common-divisor` (упражнение 2.94) вычислите НОД Q_1 и Q_2. Обратите внимание, что ответ не совпадает с P_1. Этот пример вводит в вычисление операции с нецелыми числами, и это создает сложности для алгоритма вычисления НОД[61]. Чтобы понять, что здесь происходит, попробуйте включить трассировку в `gcd-terms` при вычислении НОД либо проведите деление вручную.

Проблему, которую демонстрирует упражнение 2.95, можно решить, если мы используем следующий вариант алгоритма вычисления НОД (который работает только для многочленов с целыми коэффициентами). Прежде, чем проводить деление многочленов при вычислении НОД, мы умножаем делимое на целую константу, которая выбирается так, чтобы в процессе деления не возникло никаких дробей. Результат вычисления будет отличаться от настоящего НОД на целую константу, но при приведении рациональных функций к наименьшему знаменателю это несущественно; будет проведено деление и числителя, и знаменателя на НОД, так что константный множитель сократится.

[61] В системах вроде MIT Scheme получится многочлен, который на самом деле является делителем Q_1 и Q_2, но с рациональными коэффициентами. Во многих других реализациях Scheme, где при делении целых чисел могут получаться десятичные числа ограниченной точности, может оказаться, что мы не получим правильного делителя.

Выражаясь более точно, если P и Q — многочлены, определим O_1 как порядок P (то есть порядок его старшего терма), а O_2 как порядок Q. Пусть c будет коэффициент старшего терма Q. В таком случае, можно показать, что если мы домножим P на *множитель целости* (integerizing factor) $c^{1+O_1-O_2}$, то получившийся многочлен можно будет поделить на Q алгоритмом `div-terms`, получив результат, в котором не будет никаких дробей. Операция домножения делимого на такую константу, а затем деления, иногда называется *псевдоделением* (pseudodivision) P на Q. Остаток такого деления называется *псевдоостатком* (pseudoremainder).

Упражнение 2.96.

а. Напишите процедуру `pseudoremainder-terms`, которая работает в точности как `remainder-terms`, но только прежде, чем позвать `div-terms`, домножает делимое на множитель целости, описанный выше. Модифицируйте `gcd-terms` так, чтобы она использовала `pseudoremainder-terms`, и убедитесь, что теперь в примере из упражнения 2.95 `greatest-common-divisor` выдает ответ с целыми коэффициентами.

б. Теперь у НОД целые коэффициенты, но они больше, чем коэффициенты P_1. Измените `gcd-terms`, чтобы она убирала общий множитель из коэффициентов ответа путем деления всех коэффициентов на их (целочисленный) НОД.

Итак, вот как привести рациональную функцию к наименьшему знаменателю:

- Вычислите НОД числителя и знаменателя, используя версию `gcd-terms` из упражнения 2.96.

- Когда Вы получаете НОД, домножьте числитель и знаменатель на множитель целости, прежде чем делить на НОД, чтобы при делении не получить дробных коэффициентов. В качестве множителя можно использовать старший коэффициент НОД, возведенный в степень $1 + O_1 - O_2$, где O_2 – порядок НОД, а O_1 — максимум из порядков числителя и знаменателя. Так Вы добьетесь того, чтобы деление числителя и знаменателя на НОД не привносило дробей.

- В результате этой операции Вы получите числитель и знаменатель с целыми коэффициентами. Обычно из-за всех множителей целости коэффициенты окажутся очень большими, стало быть, на последнем шаге следует избавиться от лишних множителей, вычислив (целый) наибольший общий делитель числителя и знаменателя и поделив на него все термы.

Упражнение 2.97.

а. Реализуйте этот алгоритм как процедуру `reduce-terms`, которая принимает в качестве аргументов два списка термов `n` и `d` и возвращает список из `nn` и `dd`, которые представляют собой `n` и `d`, приведенные к наименьшему знаменателю по вышеописанному алгоритму. Напишите, кроме того, процедуру `reduce-poly`, подобную `add-poly`, которая проверяет, чтобы два poly имели одну и ту же переменную. Если это так, `reduce-poly` откусывает эту переменную и передает оставшуюся часть задачи в `reduce-terms`, а затем прикрепляет переменную обратно к двум спискам термов, которые получены из `reduce-terms`.

б. Определите процедуру, аналогичную `reduce-terms`, которая делает то, что делала для целых чисел исходная `make-rat`:

```
(define (reduce-integers n d)
  (let ((g (gcd n d)))
    (list (/ n g) (/ d g))))
```

и определите `reduce` как обобщенную операцию, которая вызывает `apply-generic` и диспетчирует либо к `reduce-poly` (если аргументы — многочлены), либо к `reduce-integers` (для аргументов типа `scheme-number`). Теперь Вы легко можете заставить пакет рациональной арифметики приводить дроби к наименьшему знаменателю, потребовав от `make-rat` звать `reduce` прежде, чем сочетать данные числитель и знаменатель в процессе порождения рационального числа. Теперь система обрабатывает рациональные выражения и для целых чисел, и для многочленов. Чтобы проверить программу, попробуйте пример, который приведен в начале этого расширенного упражнения:

```
(define p1 (make-polynomial 'x '((1 1)(0 1))))
(define p2 (make-polynomial 'x '((3 1)(0 -1))))
(define p3 (make-polynomial 'x '((1 1))))
(define p4 (make-polynomial 'x '((2 1)(0 -1))))

(define rf1 (make-rational p1 p2))
(define rf2 (make-rational p3 p4))

(add rf1 rf2)
```

Посмотрите, удалось ли Вам получить правильный ответ, правильно приведенный к наименьшему знаменателю.

Вычисление НОД находится в центре всякой системы, работающей с рациональными числами. Алгоритм, который мы использовали в тексте, хотя математически он естествен, работает очень медленно. Медлительность эта проистекает отчасти из большого количества операций деления, а отчасти из огромного размера промежуточных коэффициентов, которые порождаются в ходе псевдоделения. Одна из активно разрабатываемых областей в теории систем алгебраических манипуляций – построение более быстрых алгоритмов для вычисления НОД многочленов[62].

[62]Изящный и чрезвычайно эффективный метод вычисления НОД многочленов был открыт Ричардом Зиппелем (Zippel 1979). Этот метод — вероятностный алгоритм, подобно быстрому тесту на простоту числа, описанному в главе 1. Книга Зиппеля Zippel 1993 описывает этот метод, а также другие способы нахождения НОД многочленов.

Глава 3

Модульность, объекты и состояние

Μεταβάλλον ἀναπαύεται
(Изменяясь, оно остается неподвижным)
Гераклит

Plus ça change, plus c'est la même chose
Альфонс Карр

В предыдущих главах мы ввели основные элементы, из которых строятся программы. Мы видели, как элементарные процедуры и элементарные данные, сочетаясь, образуют составные сущности; мы стали понимать, что без абстракции нельзя справиться со сложностью больших систем. Однако этих инструментов недостаточно для разработки программ. Для эффективного синтеза программ требуются также организационные принципы, которые помогали бы нам сформулировать общий проект программы. В частности, нам нужны стратегии для построения больших программ по принципу *модульности* (modularity): чтобы программы «естественным» образом делились на логически цельные куски, которые можно разрабатывать и поддерживать независимо друг от друга.

Существует мощная стратегия разработки, которая особенно хорошо подходит для построения программ, моделирующих физические системы: воспроизводить в структуре программы структуру моделируемой системы. Для каждого объекта в системе мы строим соответствующий ему вычислительный объект. Для каждого действия в системе определяем в рамках нашей вычислительной модели символьную операцию. Используя эту стратегию, мы надеемся, что расширение нашей модели на новые объекты или действия не потребует стратегических изменений в программе, а позволит обойтись только добавлением новых символьных аналогов этих объектов или действий. Если наша организация системы окажется удачной, то для добавления новых возможностей или отладки старых нам придется работать только с ограниченной частью системы.

Таким образом, способ, которым мы организуем большую программу, в значительной степени диктуется нашим восприятием моделируемой системы. В этой главе мы исследуем две важных организационных стратегии, которые соответствуют двум достаточно различным взглядам на мир и структуру систем. Первая из них сосредотачивается на *объектах* (objects), и большая система рассматривается как собрание индивидуальных объектов, поведение которых может меняться со временем. Альтернативная стратегия строится вокруг *потоков* (streams) информации в системе, во многом подобно тому, как в электронике рассматриваются системы обработки сигналов.

Как подход, основанный на объектах, так и подход, основанный на потоках, высвечивают важные вопросы, касающиеся языков программирования. При работе с объектами

нам приходится думать о том, как вычислительный объект может изменяться и при этом сохранять свою индивидуальность. Из-за этого нам придется отказаться от подстановочной модели вычислений (раздел 1.1.5) в пользу более механистичной и в то же время менее привлекательной теоретически *модели с окружениями* (environment model). Сложности, связанные с объектами, их изменением и индивидуальностью являются фундаментальным следствием из потребности ввести понятие времени в вычислительные модели. Эти сложности только увеличиваются, когда мы добавляем возможность параллельного выполнения программ. Получить наибольшую отдачу от потокового подхода удается тогда, когда моделируемое время отделяется от порядка событий, происходящих в компьютере в процессе вычисления. Мы достигнем этого при помощи метода, называемого *задержанными вычислениями* (delayed evaluation).

3.1. Присваивание и внутреннее состояние объектов

Обычно мы считаем, что мир состоит из отдельных объектов, и у каждого из них есть состояние, которое изменяется со временем. Мы говорим, что объект «обладает состоянием», если на поведение объекта влияет его история. Например, банковский счет обладает состоянием потому, что ответ на вопрос «Могу ли я снять 100 долларов?» зависит от истории занесения и снятия с него денег. Состояние объекта можно описать набором из одной или более *переменных состояния* (state variables), которые вместе содержат достаточно информации, чтобы определить текущее поведение объекта. В простой банковской системе состояние счета можно охарактеризовать его текущим балансом, вместо того, чтобы запоминать всю историю транзакций с этим счетом.

Если система состоит из многих объектов, они редко совершенно независимы друг от друга. Каждый из них может влиять на состояние других при помощи актов взаимодействия, связывающих переменные состояния одного объекта с переменными других объектов. На самом деле, взгляд, согласно которому система состоит из отдельных объектов, полезнее всего в том случае, когда ее можно разделить на несколько подсистем, в каждой из которых внутренние связи сильнее, чем связи с другими подсистемами.

Такая точка зрения на систему может служить мощной парадигмой для организации вычислительных моделей системы. Чтобы такая модель была модульной, ее требуется разделить на вычислительные объекты, моделирующие реальные объекты системы. Каждый вычислительный объект должен содержать собственные *внутренние переменные состояния* (local state variables), описывающие состояние реального объекта. Поскольку объекты в моделируемой системе меняются со временем, переменные состояния соответствующих вычислительных объектов также должны изменяться. Если мы решаем, что поток времени в системе будет моделироваться временем, проходящим в компьютере, то нам требуется способ строить вычислительные объекты, поведение которых меняется по мере выполнения программы. В частности, если нам хочется моделировать переменные состояния обыкновенными символическими именами в языке программирования, в языке должен иметься *оператор присваивания* (assignment operator), который позволял бы изменять значение, связанное с именем.

3.1.1. Внутренние переменные состояния

Чтобы показать, что мы имеем в виду, говоря о вычислительном объекте, состояние которого меняется со временем, давайте промоделируем ситуацию снятия денег с банковского счета. Воспользуемся для этого процедурой `withdraw`, которая в качестве аргумента принимает сумму, которую требуется снять. Если на счету имеется достаточно средств, чтобы осуществить операцию, то `withdraw` возвращает баланс, остающийся после снятия. В противном случае `withdraw` возвращает сообщение «Недостаточно денег на счете». Например, если вначале на счету содержится 100 долларов, мы получим следующую последовательность результатов:

```
(withdraw 25)
75

(withdraw 25)
50

(withdraw 60)
"Недостаточно денег на счете"

(withdraw 15)
35
```

Обратите внимание, что выражение `(withdraw 25)`, будучи вычислено дважды, дает различные результаты. Это новый тип поведения для процедуры. До сих пор все наши процедуры можно было рассматривать как описания способов вычисления математических функций. Вызов процедуры вычислял значение функции для данных аргументов, и два вызова одной и той же процедуры с одинаковыми аргументами всегда приводили к одинаковому результату[1].

При реализации `withdraw` мы используем переменную `balance`, которая показывает остаток денег на счете, и определяем `withdraw` в виде процедуры, которая обращается к этой переменной. Процедура `withdraw` проверяет, что значение `balance` не меньше, чем значение аргумента `amount`. Если это так, `withdraw` уменьшает значение `balance` на `amount` и возвращает новое значение `balance`. В противном случае она возвращает сообщение «Недостаточно денег на счете». Вот определения `balance` и `withdraw`:

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Недостаточно денег на счете"))
```

[1] На самом деле это не совсем правда. Одно исключение — генератор случайных чисел из раздела 1.2.6. Второе связано с таблицами операций и типов, которые мы ввели в разделе 2.4.3, где значения двух вызовов `get` с одними и теми же аргументами зависели от того, какие были в промежутке между ними вызовы `put`. С другой стороны, пока мы не ввели присваивание, мы лишены возможности самим создавать такие процедуры.

Значение переменной `balance` уменьшается, когда мы выполняем выражение

```
(set! balance (- balance amount))
```

Здесь используется особая форма `set!`, синтаксис которой выглядит так:

(`set!` ⟨*имя*⟩ ⟨*новое-значение*⟩)

Здесь ⟨*имя*⟩ — символ, а ⟨*новое-значение*⟩ – произвольное выражение. `Set!` заменяет значение ⟨*имени*⟩ на результат, полученный при вычислении ⟨*нового-значения*⟩. В данном случае, мы изменяем `balance` так, что его новое значение будет результатом вычитания `amount` из предыдущего значения `balance`[2].

Кроме того, `withdraw` использует особую форму `begin`, когда проверка `if` выдает истину, и требуется вычислить два выражения: сначала уменьшить `balance`, а затем вернуть его значение. В общем случае вычисление выражения

(`begin` ⟨*выражение*$_1$⟩ ⟨*выражение*$_2$⟩ ... ⟨*выражение*$_k$⟩)

приводит к последовательному вычислению выражений от ⟨*выражения*$_1$⟩ до ⟨*выражения*$_k$⟩, и значение последнего выражения ⟨*выражение*$_k$⟩ возвращается в качестве значения всей формы `begin`[3].

Хотя процедура `withdraw` и работает так, как мы того хотели, переменная `balance` представляет собой проблему. `Balance`, как она описана выше, является переменной, определенной в глобальном окружении, и любая процедура может прочитать или изменить ее значение. Намного лучше было бы, если бы `balance` можно было сделать внутренней переменной для `withdraw`, так, чтобы только `withdraw` имела доступ к ней напрямую, а любая другая процедура — только посредством вызовов `withdraw`. Так можно будет более точно смоделировать представление о `balance` как о внутренней переменной состояния, с помощью которой `withdraw` следит за состоянием счета.

Сделать `balance` внутренней по отношению к `withdraw` мы можем, переписав определение следующим образом:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Недостаточно денег на счете"))))
```

Здесь мы, используя `let`, создаем окружение с внутренней переменной `balance`, которой вначале присваивается значение 100. Внутри этого локального окружения мы при

[2]Значение выражения `set!` зависит от реализации. `Set!` нужно использовать только ради эффекта, который оно оказывает, а не ради значения, которое оно возвращает.

Имя `set!` отражает соглашение, принятое в Scheme: операциям, которые изменяют значения переменных (или структуры данных, как мы увидим в разделе 3.3) даются имена с восклицательным знаком на конце. Это напоминает соглашение называть предикаты именами, которые оканчиваются на вопросительный знак.

[3]Неявно мы уже использовали в своих программах `begin`, поскольку в Scheme тело процедуры может быть последовательностью выражений. Кроме того, в каждом подвыражении `cond` следствие может состоять не из одного выражения, а из нескольких.

помощи `lambda` определяем процедуру, которая берет в качестве аргумента `amount` и действует так же, как наша старая процедура `withdraw`. Эта процедура — возвращаемая как результат выражения `let`, — и есть `new-withdraw`. Она ведет себя в точности так же, как, как `withdraw`, но ее переменная `balance` недоступна для всех остальных процедур[4].

`Set!` в сочетании с локальными переменными — общая стратегия программирования, которую мы будем использовать для построения вычислительных объектов, обладающих внутренним состоянием. К сожалению, при использовании этой стратегии возникает серьезная проблема: когда мы только вводили понятие процедуры, мы ввели также подстановочную модель вычислений (раздел 1.1.5) для того, чтобы объяснить, что означает применение процедуры к аргументам. Мы сказали, что оно должно интерпретироваться как вычисление тела процедуры, в котором формальные параметры заменяются на свои значения. К сожалению, как только мы вводим в язык присваивание, подстановка перестает быть адекватной моделью применения процедуры. (Почему это так, мы поймем в разделе 3.1.3.) В результате, с технической точки зрения мы сейчас не умеем объяснить, почему процедура `new-withdraw` ведет себя именно так, как описано выше. Чтобы действительно понять процедуры, подобные `new-withdraw`, нам придется разработать новую модель применения процедуры. В разделе 3.2 мы введем такую модель, попутно объяснив `set!` и локальные переменные. Однако сначала мы рассмотрим некоторые вариации на тему, заданную `new-withdraw`.

Следующая процедура, `make-withdraw`, создает «обработчики снятия денег со счетов». Формальный параметр `balance`, передаваемый в `make-withdraw`, указывает начальную сумму денег на счету[5].

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете")))
```

При помощи `make-withdraw` можно следующим образом создать два объекта `W1` и `W2`:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))

(W1 50)
50

(W2 70)
30
```

[4]По терминологии, принятой при описании языков программирования, переменная `balance` *инкапсулируется* (is encapsulated) внутри процедуры `new-withdraw`. Инкапсуляция отражает общий принцип проектирования систем, известный как *принцип сокрытия* (the hiding principle): систему можно сделать более модульной и надежной, если защищать ее части друг от друга; то есть, разрешать доступ к информации только тем частям системы, которым «необходимо это знать».

[5]В отличие от предыдущей процедуры `new-withdraw`, здесь нам необязательно использовать `let`, чтобы сделать `balance` локальной переменной, поскольку формальные параметры и так локальны. Это станет яснее после обсуждения модели вычисления с окружениями в разделе 3.2. (См. также упражнение 3.10.)

```
(W2 40)
"Недостаточно денег на счете"

(W1 40)
10
```

Обратите внимание, что W1 и W2 — полностью независимые объекты, каждый со своей локальной переменной balance. Снятие денег с одного счета не влияет на другой.

Мы можем создавать объекты, которые будут разрешать не только снятие денег, но и их занесение на счет, и таким образом можно смоделировать простые банковские счета. Вот процедура, которая возвращает объект-«банковский счет» с указанным начальным балансом:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Неизвестный вызов -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Каждый вызов make-account создает окружение с локальной переменной состояния balance. Внутри этого окружения make-account определяет процедуры deposit и withdraw, которые обращаются к balance, а также дополнительную процедуру dispatch, которая принимает «сообщение» в качестве ввода, и возвращает одну из двух локальных процедур. Сама процедура dispatch возвращается как значение, которое представляет объект-банковский счет. Это не что иное, как стиль программирования с *передачей сообщений* (message passing), который мы видели в разделе 2.4.3, но только здесь мы его используем в сочетании с возможностью изменять локальные переменные.

Make-account можно использовать следующим образом:

```
(define acc (make-account 100))

((acc 'withdraw) 50)
50

((acc 'withdraw) 60)
"Недостаточно денег на счете"

((acc 'deposit) 40)
```

```
90

((acc 'withdraw) 60)
30
```

Каждый вызов `acc` возвращает локально определенную процедуру `deposit` или `withdraw`, которая затем применяется к указанной сумме. Точно так же, как это было с `make-withdraw`, второй вызов `make-account`

```
(define acc2 (make-account 100))
```

создает совершенно отдельный объект-счет, который поддерживает свою собственную переменную `balance`.

Упражнение 3.1.
Накопитель (accumulator) — это процедура, которая вызывается с одним численным аргументом и собирает свои аргументы в сумму. При каждом вызове накопитель возвращает сумму, которую успел накопить. Напишите процедуру `make-accumulator`, порождающую накопители, каждый из которых поддерживает свою отдельную сумму. Входной параметр `make-accumulator` должен указывать начальное значение суммы; например,

```
(define A (make-accumulator 5))

(A 10)
15

(A 10)
25
```

Упражнение 3.2.
При тестировании программ удобно иметь возможность подсчитывать, сколько раз за время вычислений была вызвана та или иная процедура. Напишите процедуру `make-monitored`, принимающую в качестве параметра процедуру `f`, которая сама по себе принимает один входной параметр. Результат, возвращаемый `make-monitored` — третья процедура, назовем ее `mf`, которая подсчитывает, сколько раз она была вызвана, при помощи внутреннего счетчика. Если на входе `mf` получает специальный символ `how-many-calls?`, она возвращает значение счетчика. Если же на вход подается специальный символ `reset-count`, `mf` обнуляет счетчик. Для любого другого параметра `mf` возвращает результат вызова `f` с этим параметром и увеличивает счетчик. Например, можно было бы сделать отслеживаемую версию процедуры `sqrt`:

```
(define s (make-monitored sqrt))

(s 100)
10

(s 'how-many-calls?)
1
```

Упражнение 3.3.
Измените процедуру `make-account` так, чтобы она создавала счета, защищенные паролем. А именно, `make-account` должна в качестве дополнительного аргумента принимать символ, например

```
(define acc (make-account 100 'secret-password))
```

Получившийся объект-счет должен обрабатывать запросы, только если они сопровождаются паролем, с которым счет был создан, а в противном случае он должен жаловаться:

```
((acc 'secret-password 'withdraw) 40)
60

((acc 'some-other-password 'deposit) 50)
"Неверный пароль"
```

Упражнение 3.4.
Модифицируйте процедуру `make-account` из упражнения 3.3, добавив еще одну локальную переменную, так, чтобы, если происходит более семи попыток доступа подряд с неверным паролем, вызывалась процедура `call-the-cops` (вызвать полицию).

3.1.2. Преимущества присваивания

Как нам предстоит увидеть, введение присваивания в наш язык программирования ведет к множеству сложных концептуальных проблем. Тем не менее, представление о системе как о наборе объектов, имеющих внутреннее состояние, — мощное средство для обеспечения модульности проекта. В качестве примера рассмотрим строение процедуры `rand`, которая, будучи вызванной, каждый раз возвращает случайное целое число.

Вовсе не так просто определить, что значит «случайное». Вероятно, имеется в виду, что последовательные обращения к `rand` должны порождать последовательность чисел, которая обладает статистическими свойствами равномерного распределения. Здесь мы не будем обсуждать способы порождения подобных последовательностей. Вместо этого предположим, что у нас есть процедура `rand-update`, которая обладает следующим свойством: если мы начинаем с некоторого данного числа x_1 и строим последовательность

$$x_2 = \texttt{(rand-update } x_1\texttt{)}$$
$$x_3 = \texttt{(rand-update } x_2\texttt{)}$$

то последовательность величин $x_1, x_2, x_3 \ldots$ будет обладать требуемыми математическими свойствами[6].

Мы можем реализовать `rand` как процедуру с внутренней переменной состояния `x`, которая инициализируется некоторым заранее заданным значением `random-init`. Каждый вызов `rand` вычисляет `rand-update` от текущего значения `x`, возвращает это значение как случайное число, и, кроме того, сохраняет его как новое значение `x`.

[6]Один из распространенных способов реализации `rand-update` состоит в том, чтобы положить новое значение x равным $ax + b \bmod m$, где a, b и m — соответствующим образом подобранные целые числа. Глава 3 книги Knuth 1981 содержит подробное обсуждение методов порождения последовательностей случайных чисел и обеспечения их статистических свойств. Обратите внимание, что `rand-update` вычисляет математическую функцию: если ей дважды дать один и тот же вход, она вернет одинаковый результат. Таким образом, последовательность чисел, порождаемая `rand-update`, никоим образом не «случайна», если мы настаиваем на том, что в последовательности «случайных» чисел следующее число не должно иметь никакого отношения к предыдущему. Отношение между «настоящей» случайностью и так называемыми *псевдослучайными* (pseudorandom) последовательностями, которые порождаются путем однозначно определенных вычислений и тем не менее обладают нужными статистическими свойствами, — непростой вопрос, связанный со сложными проблемами математики и философии. Для прояснения этих вопросов много сделали Колмогоров, Соломонофф и Хайтин; обсуждение можно найти в Chaitin 1975.

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

Разумеется, ту же последовательность случайных чисел мы могли бы получить без использования присваивания, просто напрямую вызывая `rand-update`. Однако это означало бы, что всякая часть программы, которая использует случайные числа, должна явно запоминать текущее значение `x`, чтобы передать его как аргумент `rand-update`. Чтобы понять, насколько это было бы неприятно, рассмотрим использование случайных чисел для реализации т. н. *моделирования методом Монте-Карло* (Monte Carlo simulation).

Метод Монте-Карло состоит в том, чтобы случайным образом выбирать тестовые точки из большого множества и затем делать выводы на основании вероятностей, оцениваемых по результатам тестов. Например, можно получить приближенное значение π, используя тот факт, что для двух случайно выбранных целых чисел вероятность отсутствия общих множителей (то есть, вероятность того, что их наибольший общий делитель будет равен 1) составляет $6/\pi^2$[7]. Чтобы получить приближенное значение π, мы производим большое количество тестов. В каждом тесте мы случайным образом выбираем два числа и проверяем, не равен ли их НОД единице. Доля тестов, которые проходят, дает нам приближение к $6/\pi^2$, и отсюда мы получаем приближенное значение π.

В центре нашей программы находится процедура `monte-carlo`, которая в качестве аргументов принимает количество попыток тестирования, а также сам тест — процедуру без аргументов, возвращающую при каждом вызове либо истину, либо ложь. `Monte-carlo` запускает тест указанное количество раз и возвращает число, обозначающее долю попыток, в которых тест вернул истинное значение.

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))

(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))
```

Теперь попробуем осуществить то же вычисление, используя `rand-update` вместо `rand`, как нам пришлось бы поступить, если бы у нас не было присваивания для моделирования локального состояния:

[7] Эта теорема доказана Э. Чезаро. Обсуждение и доказательство можно найти в разделе 4.5.2 книги Knuth 1981.

```
(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init))))

(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0)
               (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1)
                     (+ trials-passed 1)
                     x2))
              (else
               (iter (- trials-remaining 1)
                     trials-passed
                     x2))))))
  (iter trials 0 initial-x))
```

Хотя программа по-прежнему проста, в ней обнаруживается несколько болезненных нарушений принципа модульности. В первой версии программы нам удалось, используя `rand`, выразить метод Монте-Карло напрямую как обобщенную процедуру `monte-carlo`, которая в качестве аргумента принимает произвольную процедуру `experiment`. Во втором варианте программы, где у генератора случайных чисел нет локального состояния, `random-gcd-test` приходится непосредственно возиться со случайными числами `x1` и `x2` и передавать в итеративном цикле `x2` в качестве нового входа `rand-update`. Из-за того, что обработка случайных чисел происходит явно, структура накопления результатов тестов начинает зависеть от того, что наш тест использует именно два случайных числа, тогда как для других тестов Монте-Карло может потребоваться, скажем, одно или три. Даже процедура верхнего уровня `estimate-pi` вынуждена заботиться о том, чтобы предоставить начальное значение случайного числа. Поскольку внутренности генератора случайных чисел просачиваются наружу в другие части программы, задача изолировать идею метода Монте-Карло так, чтобы применять ее затем к другим задачам, осложняется. В первом варианте программы присваивание инкапсулирует состояние генератора случайных чисел внутри `rand`, так что состояние генератора остается независимым от остальной программы.

Общее явление, наблюдаемое на примере с методом Монте-Карло, таково: с точки зрения одной части сложного процесса кажется, что другие части изменяются со временем. Они обладают скрытым локальным состоянием. Если мы хотим, чтобы структура программ, которые мы пишем, отражала такое разделение на части, мы создаем вычислительные объекты (например, банковские счета или генераторы случайных чисел), поведение которых изменяется со временем. Состояние мы моделируем при помощи локальных переменных, а изменение состояния — при помощи присваивания этим переменным.

Здесь возникает соблазн закрыть обсуждение и сказать, что, введя присваивание и метод сокрытия состояния в локальных переменных, мы обретаем способность структурировать системы более модульным образом, чем если бы нам пришлось всем состоянием манипулировать явно, с передачей дополнительных параметров. К сожалению, как мы увидим, все не так просто.

Упражнение 3.5.
Интегрирование методом Монте-Карло (Monte Carlo integration) — способ приближенного вычисления определенных интегралов при помощи моделирования методом Монте-Карло. Рассмотрим задачу вычисления площади фигуры, описываемой предикатом $P(x, y)$, который истинен для точек (x, y), принадлежащих фигуре, и ложен для точек вне фигуры. Например, область, содержащаяся в круге с радиусом 3 и центром в точке $(5, 7)$, описывается предикатом, проверяющим $(x-5)^2 + (y-7)^2 \leq 3^2$. Чтобы оценить площадь фигуры, описываемой таким предикатом, для начала выберем прямоугольник, который содержит нашу фигуру. Например, прямоугольник с углами $(2, 4)$ и $(8, 10)$, расположенными по диагонали, содержит вышеописанный круг. Нужный нам интеграл — площадь той части прямоугольника, которая лежит внутри фигуры. Мы можем оценить интеграл, случайным образом выбирая точки (x, y), лежащие внутри прямоугольника, и проверяя для каждой точки $P(x, y)$, чтобы определить, лежит ли точка внутри фигуры. Если мы проверим много точек, доля тех, которые окажутся внутри области, даст нам приближенное значение отношения площадей фигуры и прямоугольника. Таким образом, домножив это значение на площадь прямоугольника, мы получим приближенное значение интеграла.

Реализуйте интегрирование методом Монте-Карло в виде процедуры `estimateintegral`, которая в качестве аргументов принимает предикат `P`, верхнюю и нижнюю границы прямоугольника `x1`, `x2`, `y1` и `y2`, а также число проверок, которые мы должны осуществить, чтобы оценить отношение площадей. Ваша процедура должна использовать ту же самую процедуру `monte-carlo`, которая выше использовалась для оценки значения π. Оцените π при помощи `estimate-integral`, измерив площадь единичного круга.

Вам может пригодиться процедура, которая выдает число, случайно выбранное внутри данного отрезка. Нижеприведенная процедура `random-in-range` решает эту задачу, используя процедуру `random`, введенную в разделе 1.2.6, которая возвращает неотрицательное число меньше своего аргумента[8].

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

Упражнение 3.6.
Полезно иметь возможность сбросить генератор случайных чисел, чтобы получить последовательность, которая начинается с некоторого числа. Постройте новую процедуру `rand`, которая вызывается с аргументом. Этот аргумент должен быть либо символом `generate`, либо символом `reset`. Процедура работает так: `(rand 'generate)` порождает новое случайное число; `((rand 'reset)` ⟨*новое-значение*⟩`)` сбрасывает внутреннюю переменную состояния в указанное ⟨*новое-значение*⟩. Таким образом, сбрасывая значения, можно получать повторяющиеся последовательности. Эта возможность очень полезна при тестировании и отладке программ, использующих случайные числа.

3.1.3. Издержки, связанные с введением присваивания

Как мы только что видели, операция `set!` позволяет моделировать объекты, обладающие внутренним состоянием. Однако за это преимущество приходится платить. Наш язык программирования нельзя больше описывать при помощи подстановочной модели

[8] В MIT Scheme есть такая процедура. Если `random` на вход дается точное целое число (как в разделе 1.2.6), она возвращает точное целое число, но если ей дать десятичную дробь (как в этом примере), она и возвращает десятичную дробь.

применения процедур, которую мы ввели в разделе 1.1.5. Хуже того, не существует простой модели с «приятными» математическими свойствами, которая бы адекватно описывала работу с объектами и присваивание в языках программирования.

Пока мы не применяем присваивание, два вычисления одной и той же процедуры с одними и теми же аргументами всегда дают одинаковый результат. Стало быть, можно считать, что процедуры вычисляют математические функции. Соответственно, программирование, в котором присваивание не используется (как у нас в первых двух главах этой книги), известно как *функциональное программирование* (functional programming).

Чтобы понять, как присваивание усложняет ситуацию, рассмотрим упрощенную версию `make-withdraw` из раздела 3.1.1, которая не проверяет, достаточно ли на счете денег:

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))

(define W (make-simplified-withdraw 25))

(W 20)
5

(W 10)
-5
```

Сравним эту процедуру со следующей процедурой `make-decrementer`, которая не использует `set!`:

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

`make-decrementer` возвращает процедуру, которая вычитает свой аргумент из определенного числа `balance`, но при последовательных вызовах ее действие не накапливается, как при использовании `make-simplified-withdraw`:

```
(define D (make-decrementer 25))

(D 20)
5

(D 10)
15
```

Мы можем объяснить, как работает `make-decrementer`, при помощи подстановочной модели. Например, рассмотрим, как вычисляется выражение

```
((make-decrementer 25) 20)
```

Сначала мы упрощаем операторную часть комбинации, подставляя в теле `make-decrementer` вместо `balance` 25. Выражение сводится к

```
((lambda (amount) (- 25 amount)) 20)
```

Теперь мы применяем оператор к операнду, подставляя 20 вместо `amount` в теле `lambda`-выражения:

```
(- 25 20)
```

Окончательный результат равен 5.

Посмотрим, однако, что произойдет, если мы попробуем применить подобный подстановочный анализ к `make-simplified-withdraw`:

```
((make-simplified-withdraw 25) 20)
```

Сначала мы упрощаем оператор, подставляя вместо `balance` 25 в теле `make-simplified-withdraw`. Таким образом, наше выражение сводится к[9]

```
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

Теперь мы применяем оператор к операнду, подставляя в теле `lambda`-выражения 20 вместо `amount`:

```
(set! balance (- 25 20)) 25
```

Если бы мы следовали подстановочной модели, нам пришлось бы сказать, что вычисление процедуры состоит в том, чтобы сначала присвоить переменной `balance` значение 5, а затем в качестве значения вернуть 25. Но это дает неверный ответ. Чтобы получить правильный ответ, нам пришлось бы как-то отличить первое вхождение `balance` (до того, как сработает `set!`) от второго (после выполнения `set!`). Подстановочная модель на это не способна.

Проблема здесь состоит в том, что подстановка предполагает, что символы в нашем языке — просто имена для значений. Но как только мы вводим `set!` и представление, что значение переменной может изменяться, переменная уже не может быть всего лишь именем. Теперь переменная некоторым образом соответствует месту, в котором может храниться значение, и значение это может меняться. В разделе 3.2 мы увидим, как в нашей модели вычислений роль этого «места» играют окружения.

Тождественность и изменение

Проблема, который здесь встает, глубже, чем просто поломка определенной модели вычислений. Как только мы вводим в наши вычислительные модели понятие изменения, многие другие понятия, которые до сих пор были ясны, становятся сомнительными. Рассмотрим вопрос, что значит, что две вещи суть «одно и то же».

Допустим, мы два раза зовем `make-decrementer` с одним и тем же аргументом, и получаем две процедуры:

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))
```

[9] Мы не производим подстановку вхождения `balance` в выражение `set!`, поскольку ⟨*имя*⟩ в `set!` не вычисляется. Если бы мы провели подстановку, получилось бы `(set! 25 (- 25 amount))`, а это не имеет никакого смысла.

Являются ли D1 и D2 одним и тем же объектом? Можно сказать, что да, поскольку D1 и D2 обладают одинаковым поведением — каждая из этих процедур вычитает свой аргумент из 25. В сущности, в любом вычислении можно подставить D1 вместо D2, и результат не изменится.

Напротив, рассмотрим два вызова make-simplified-withdraw:

```
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-withdraw 25))
```

Являются ли W1 и W2 одним и тем же? Нет, конечно, потому что вызовы W1 и W2 приводят к различным результатам, как показывает следующая последовательность вычислений:

```
(W1 20)
5

(W1 20)
-15

(W2 20)
5
```

Хотя W1 и W2 «равны друг другу» в том смысле, что оба они созданы вычислением одного и того же выражения (make-simplified-withdraw 25), неверно, что в любом выражении можно заменить W1 на W2, не повлияв при этом на результат его вычисления.

Язык, соблюдающий правило, что в любом выражении «одинаковое можно подставить вместо одинакового», не меняя его значения, называется *референциально прозрачным* (referentially transparent). Если мы включаем в свой компьютерный язык set!, его референциальная прозрачность нарушается. Становится сложно определить, где можно упростить выражение, подставив вместо него равносильное. Следовательно, рассуждать о программах, в которых используется присваивание, оказывается гораздо сложнее.

С потерей референциальной прозрачности становится сложно формально описать понятие о том, что два объекта – один и тот же объект. На самом деле, смысл выражения «то же самое» в реальном мире, который наши программы моделируют, сам по себе недостаточно ясен. В общем случае, мы можем проверить, являются ли два как будто бы одинаковых объекта одним и тем же, только изменяя один из них и наблюдая, изменился ли таким же образом и другой. Но как мы можем узнать, «изменился» ли объект? Только рассмотрев один и тот же объект дважды и проверив, не различается ли некоторое его свойство между двумя наблюдениями. Таким образом, мы не можем определить «изменение», не имея заранее понятия «идентичности», а идентичность мы не можем определить, не рассмотрев результаты изменений.

В качестве примера того, как эти вопросы возникают в программировании, рассмотрим ситуацию, где у Петра и у Павла есть по банковскому счету в 100 долларов. Здесь не все равно, смоделируем мы это через

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

или

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

В первом случае, два счета различны. Действия, которые производит Петр, не меняют счет Павла, и наоборот. Однако во втором случае мы сказали, что `paul-acc` — это *та же самая вещь*, что и `peter-acc`. Теперь у Петра и у Павла есть совместный банковский счет, и если Петр возьмет сколько-то с `peter-acc`, то у Павла на `paul-acc` будет меньше денег. При построении вычислительных моделей сходство между этими двумя несовпадающими ситуациями может привести к путанице. В частности, в случае с совместным счетом может особенно мешать то, что у одного объекта (банковского счета) есть два имени (`peter-acc` и `paul-acc`); если мы ищем в программе все места, где может меняться `paul-acc`, надо смотреть еще и где меняется `peter-acc`[10].

В связи с этими замечаниями обратите внимание на то, что если бы Петр и Павел могли только проверять свой платежный баланс, но не менять его, то вопрос «один ли у них счет?» не имел бы смысла. В общем случае, если мы никогда не меняем объекты данных, то можно считать, что каждый объект представляет собой в точности совокупность своих частей. Например, рациональное число определяется своим числителем и знаменателем. Однако при наличии изменений такой взгляд становится ошибочным, поскольку теперь у каждого объекта есть «индивидуальность», которая отличается от тех частей, из которых он состоит. Банковский счет останется «тем же самым» счетом, даже если мы снимем с него часть денег; и наоборот, можно иметь два разных счета с одинаковым состоянием. Такие сложности — следствие не нашего языка программирования, а нашего восприятия банковского счета как объекта. Скажем, рациональное число мы обычно не рассматриваем как изменяемый объект со своей индивидуальностью, у которого можно было бы изменить числитель и по-прежнему иметь дело с «тем же» числом.

Ловушки императивного программирования

В противоположность функциональному программированию, стиль программирования, при котором активно используется присваивание, называется *императивное программирование* (imperative programming). Кроме того, что возникают сложности с вычислительными моделями, программы, написанные в императивном стиле, подвержены таким ошибкам, которые в функциональных программах не возникают. Вспомним, к примеру, итеративную программу для вычисления факториала из раздела 1.2.1:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
```

[10]Когда у вычислительного объекта имеется несколько имён, эти имена называются *псевдонимами* (aliasing). Ситуация с совместным банковским счетом — простой пример псевдонимов. В разделе 3.3 мы увидим значительно более сложные примеры, скажем, «различные» составные структуры с общими частями. Если мы забудем, что «побочным эффектом» в результате изменения одного объекта может стать изменение «другого» объекта, поскольку «разные» объекты — на самом деле один и тот же под разными псевдонимами, то могут возникнуть ошибки. Эти так называемые *ошибки побочных эффектов* (side-effect bugs) настолько трудно обнаруживать и анализировать, что некоторые исследователи выступали с предложениями не допускать в языках программирования побочные эффекты и псевдонимы (Lampson et al. 1981; Morris, Schmidt, and Wadler 1980).

```
            (+ counter 1))))
  (iter 1 1))
```

Вместо того, чтобы передавать аргументы во внутреннем итеративном цикле, мы могли бы написать процедуру в более императивном стиле с использованием присваивания для обновления значений переменных `product` и `counter`:

```
(define (factorial n)

  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

Результаты, выдаваемые программой, при этом не меняются, но возникает маленькая ловушка. Как определить порядок присваиваний? В имеющемся виде программа корректна. Однако если бы мы записали присваивания в обратном порядке:

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

— получился бы другой, неверный результат. Вообще, программирование с использованием присваивания заставляет нас тщательно следить за порядком присваиваний, так, чтобы в каждом использовалась правильная версия значения переменных, которые меняются. В функциональных программах такие сложности просто не возникают[11].

Сложность императивных программ еще увеличивается, если мы начинаем рассматривать приложения, где одновременно выполняется несколько процессов. К этому мы еще вернемся в разделе 3.4. Однако сначала мы обратимся к задаче построения вычислительной модели для выражений, содержащих присваивание, а также изучим, как использовать объекты с локальным состоянием при проектировании моделирующих программ.

Упражнение 3.7.
Рассмотрим объекты-банковские счета, создаваемые процедурой `make-account`, и снабженные паролями, как это описано в упражнении 3.3. Предположим, что наша банковская система требует от нас умения порождать совместные счета. Напишите процедуру `make-joint`, которая это делает. `Make-joint` должна принимать три аргумента. Первый из них — защищенный паролем

[11] Поэтому странно и смешно, что вводные курсы программирования часто читаются в глубоко императивном стиле. Может быть, сказываются остатки распространенного в 60-е и 70-е годы представления, что программы, которые вызывают процедуры, непременно будут менее эффективны, чем те, которые производят присваивания. (Steele 1977 развенчивает этот аргумент.) С другой стороны, возможно, считается, что новичкам легче представить пошаговое присваивание, чем вызов процедуры. Так или иначе, программистам часто приходится заботиться о вопросе «присвоить сначала эту переменную или ту?», а это усложняет программирование и затемняет важные идеи.

счет. Второй обязан совпадать с паролем, с которым этот счет был создан, иначе `make-joint` откажется работать. Третий аргумент — новый пароль. Например, если банковский счет `peter-account` был создан с паролем `open-sesame`, то

```
(define paul-acc
  (make-joint peter-acc 'open-sesame 'rosebud))
```

позволит нам проводить операции с `peter-account`, используя имя `paul-acc` и пароль `rosebud`. Вам может потребоваться переработать решение упражнения 3.3, чтобы добавить эту новую возможность.

Упражнение 3.8.
Когда в разделе 1.1.3 мы определяли модель вычислений, мы сказали, что первым шагом при вычислении выражения является вычисление его подвыражений. Однако мы нигде не указали порядок, в котором проходит вычисление подвыражений (слева направо или справа налево). Когда мы вводим присваивание, порядок, в котором вычисляются аргументы процедуры, может повлиять на результат. Определите простую процедуру `f`, так, чтобы вычисление `(+ (f 0) (f 1))` возвращало 0, если аргументы `+` вычисляются слева направо, и 1, если они вычисляются справа налево.

3.2. Модель вычислений с окружениями

Когда в главе 1 мы вводили понятие составной процедуры, то для того, чтобы определить, что значит применение процедуры к аргументам, мы пользовались подстановочной моделью вычислений (раздел 1.1.5):

- Чтобы применить составную процедуру к аргументам, нужно вычислить тело процедуры, подставив вместо каждого формального параметра соответствующий ему аргумент.

Как только мы вводим в язык программирования присваивание, это определение перестает быть адекватным. А именно, в разделе 3.1.3 указывалось, что в присутствии присваивания переменную уже нельзя рассматривать просто как имя для значения. Переменная должна каким-то образом обозначать «место», где значение может храниться. В нашей новой модели вычислений такие места будут находиться в структурах, которые мы называем *окружениями* (environments).

Окружение представляет собой последовательность *кадров* (frames). Каждый кадр есть (возможно, пустая) таблица *связываний* (bindings), которые сопоставляют имена переменных соответствующим значениям. (Каждый кадр должен содержать не более одного связывания для каждой данной переменной.) Кроме того, в каждом кадре имеется указатель на *объемлющее окружение* (enclosing environment), кроме тех случаев, когда в рамках текущего обсуждения окружение считается *глобальным* (global). *Значение переменной* (value of a variable) по отношению к данному окружению есть значение, которое находится в связывании для этой переменной в первом кадре окружения, содержащем такое связывание. Если в последовательности кадров ни один не указывает значения для данной переменной, говорят, что переменная *несвязана* (unbound) в окружении.

На рисунке 3.1 изображена простая структура окружений, которая состоит из трех кадров, помеченных числами I, II и III. На этой диаграмме A, B, C и D — указатели

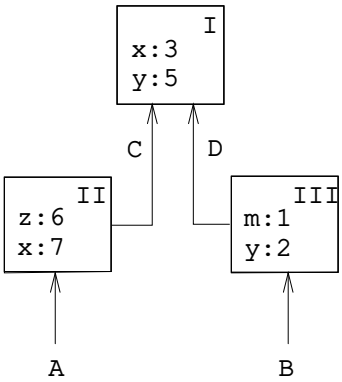


Рис. 3.1. Простой пример структуры окружений

на окружения. C и D указывают на одно и то же окружение. В кадре II связываются переменные `z` и `x`, а в кадре I переменные `y` и `x`. В окружении D переменная `x` имеет значение 3. В окружении B значение переменной `x` также равно 3. Это определяется следующим образом: мы рассматриваем первый кадр в последовательности (кадр III) и не находим там связывания для переменной `x`, так что мы переходим к объемлющему окружению D и находим связывание в кадре I. С другой стороны, в окружении A значение переменной `x` равно 7, поскольку первый кадр окружения (кадр II) содержит связывание `x` со значением 7. По отношению к окружению A говорится, что связывание `x` со значением 7 в кадре II *скрывает* (shadows) связывание `x` со значением 3 в кадре I.

Окружение играет важную роль в процессе вычисления, поскольку оно определяет контекст, в котором выражение должно вычисляться. В самом деле, можно сказать, что выражения языка программирования сами по себе не имеют значения. Выражение приобретает значение только по отношению к окружению, в контексте которого оно вычисляется. Даже интерпретация столь простого выражения, как `(+ 1 1)`, зависит от нашего понимания, что мы работаем в контексте, где `+` является символом сложения. Таким образом, в нашей модели мы всегда будем говорить о вычислении выражения относительно некоторого окружения. При описании взаимодействия с интерпретатором мы будем предполагать, что существует глобальное окружение, состоящее из одного кадра (без объемлющего окружения), и что глобальное окружение содержит значения для символов, обозначающих элементарные процедуры. Например, информация о том, что `+` служит символом сложения, выражается как утверждение, что в глобальном окружении символ `+` связан с элементарной процедурой сложения.

3.2.1. Правила вычисления

Общее описание того, как интерпретатор вычисляет комбинацию, остается таким же, как оно было введено в разделе 1.1.3:

- Для того, чтобы вычислить комбинацию, нужно:
  - Вычислить подвыражения комбинации[12].
  - Применить значение выражения-оператора к значениям выражений-операндов.

Модель вычисления с окружениями заменяет подстановочную модель, по-своему определяя, что значит применить составную процедуру к аргументам.

В модели вычисления с окружениями процедура всегда представляется в виде пары, состоящей из кода и указателя на некое окружение. Процедура создается единственным способом: вычислением `lambda`-выражения. Такое вычисление дает в качестве результата процедуру, код которой берется из тела `lambda`-выражения, а окружение совпадает с окружением, в котором было вычислено выражение, чьим значением является процедура. Например, рассмотрим определение процедуры

```
(define (square x)
  (* x x))
```

которое вычисляется в глобальном окружении. Синтаксис определения процедуры — всего лишь синтаксический сахар для подразумеваемой `lambda`. С тем же успехом можно было написать выражение

```
(define square
  (lambda (x) (* x x)))
```

которое вычисляет `(lambda (x) (* x x))` и связывает символ `square` с полученным значением, все это в глобальном окружении.

Рис. 3.2 показывает результат вычисления `lambda`-выражения. Объект-процедура представляет собой пару, код которой указывает, что процедура принимает один формальный параметр, а именно `x`, а тело ее `(* x x)`. Окружение процедуры — это указатель на глобальное окружение, поскольку именно в нем вычислялось `lambda`-выражение, при помощи которого процедура была порождена. К глобальному кадру добавилось новое связывание, которое сопоставляет процедурный объект символу `square`. В общем случае `define` создает определения, добавляя новые связывания в кадры.

Теперь, когда мы рассмотрели, как процедуры создаются, мы можем описать, как они применяются. Модель с окружениями говорит: чтобы применить процедуру к аргументам, создайте новое окружение, которое содержит кадр, связывающий параметры со значениями аргументов. Объемлющим окружением для нового кадра служит окружение, на которое указывает процедура. Теперь требуется выполнить тело процедуры в этом новом окружении.

Чтобы проиллюстрировать, как работает это новое правило, на рис. 3.3 показана структура окружений, создаваемая при вычислении выражения `(square 5)` в глобальном окружении, если `square` — процедура, порожденная на рисунке 3.2. Применение

[12]Присваивание вносит одну тонкость в шаг 1 правила вычисления. Как показано в упражнении 3.8, присваивание позволяет нам писать выражения, которые имеют различные значения в зависимости от того, в каком порядке вычисляются подвыражения комбинации. Таким образом, чтобы быть точными, мы должны были бы указать порядок вычислений на шаге 1 (например, слева направо или справа налево). Однако этот порядок всегда должен рассматриваться как деталь реализации, и писать программы, которые зависят от порядка вычисления аргументов, не следует. К примеру, продвинутый компилятор может оптимизировать программу, изменяя порядок, в котором вычисляются подвыражения.

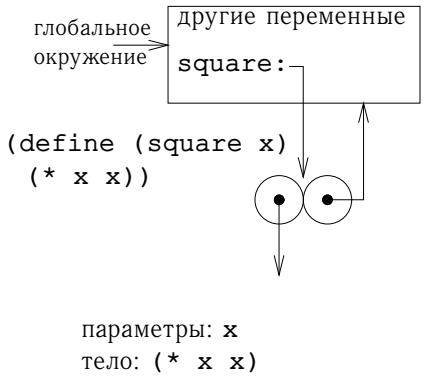


Рис. 3.2. Структура окружений, порождаемая вычислением `(define (square x) (* x x))` в глобальном окружении.

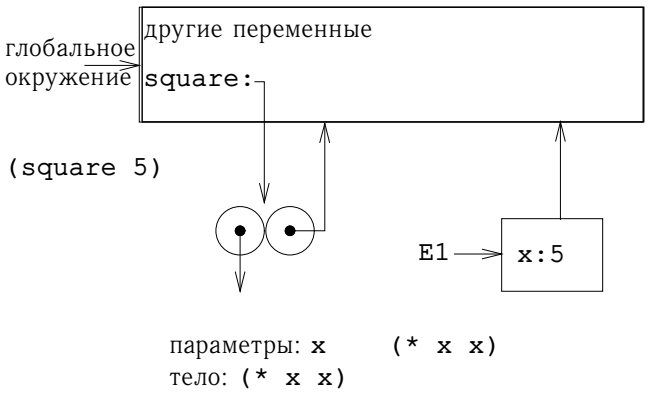


Рис. 3.3. Окружение, создаваемое при вычислении `(square 5)` в глобальном окружении.

процедуры приводит к созданию нового окружения, которое на рисунке обозначено как E1, и это окружение начинается с кадра, в котором `x`, формальный параметр процедуры, связан с аргументом 5. Указатель, который ведет из этого кадра вверх, показывает, что объемлющим для этого окружения является глобальное. Глобальное окружение выбирается потому, что именно на него ссылается процедурный объект `square`. Внутри E1 мы вычисляем тело процедуры, `(* x x)`. Поскольку значение `x` в E1 равно 5, результатом будет `(* 5 5)`, или 25.

Модель вычисления с окружениями можно вкратце описать двумя правилами:

- Процедурный объект применяется к набору аргументов при помощи создания кадра, связывания формальных параметров процедуры с аргументами вызова, и, наконец, вычисления тела процедуры в контексте этого свежесозданного окружения. В качестве объемлющего окружения новый кадр имеет окружение, содержащееся в применяемом процедурном объекте.

• Процедура создается при вычислении `lambda`-выражения по отношению к некоторому окружению. Получающийся процедурный объект есть пара, состоящая из текста `lambda`-выражения и указателя на окружение, в котором процедура была создана.

Кроме того, мы указываем, что когда символ определяется при помощи `define`, в текущем кадре окружения создается связывание, и символу присваивается указанное значение[13]. Наконец, мы описываем поведение `set!`, операции, из-за которой нам, собственно, и пришлось ввести модель с окружениями. Вычисление выражения `(set!` ⟨*переменная*⟩ ⟨*значение*⟩`)` в некотором окружении заставляет интерпретатор найти связывание переменной в окружении и изменить это связывание так, чтобы оно указывало на новое значение. А именно, нужно найти первый кадр окружения, в котором содержится связывание для переменной, и изменить этот кадр. Если переменная в окружении не связана, `set!` сигнализирует об ошибке.

Все эти правила вычисления, хотя они значительно сложнее, чем в подстановочной модели, достаточно просты. Более того, модель вычислений, несмотря на свою абстрактность, дает правильное описание того, как интерпретатор вычисляет выражения. В главе 4 мы увидим, как эта модель может служить основой для реализации работающего интерпретатора. В последующих разделах анализ нескольких примеров программ раскрывает некоторые детали этой модели.

3.2.2. Применение простых процедур

Когда в разделе 1.1.5 мы описывали подстановочную модель, мы показали, как вычисление комбинации `(f 5)` дает результат 136, если даны следующие определения:

```
(define (square x)
  (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

Теперь мы можем проанализировать тот же самый пример, используя модель с окружениями. На рисунке 3.4 изображены три процедурных объекта, созданные вычислением в глобальном окружении определений `f`, `square`, и `sum-of-squares`. Каждый процедурный объект состоит из куска кода и указателя на глобальное окружение.

На рисунке 3.5 мы видим структуру окружений, созданную вычислением выражения `(f 5)`. Вызов `f` создает новое окружение E1, начинающееся с кадра, в котором `a`, формальный параметр `f`, связывается с аргументом 5. В окружении E1 мы вычисляем тело `f`:

```
(sum-of-squares (+ a 1) (* a 2))
```

[13]Если в текущем кадре уже имелось связывание для указанной переменной, то это связывание изменяется. Это правило удобно, поскольку позволяет переопределять символы; однако оно означает, что при помощи `define` можно изменять значение символов, а это влечет за собой все проблемы, связанные с присваиванием, без явного использования `set!`. По этой причине некоторые предпочитают, чтобы переопределение существующего символа вызывало предупреждение или сообщение об ошибке.

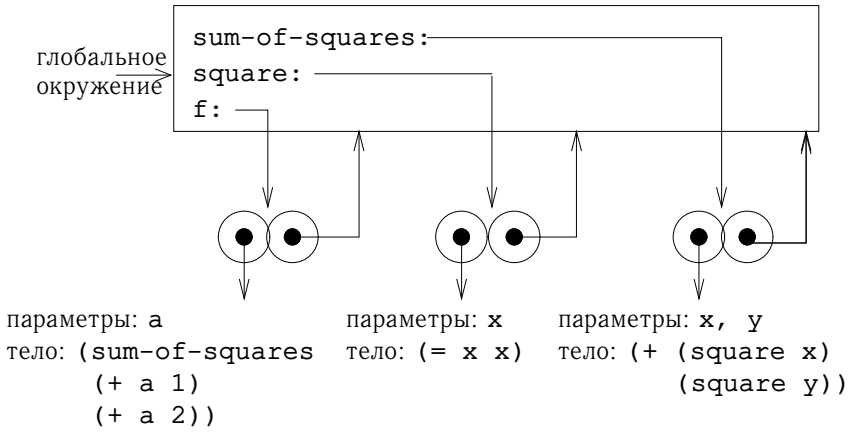


Рис. 3.4. Процедурные объекты в глобальном кадре окружения.

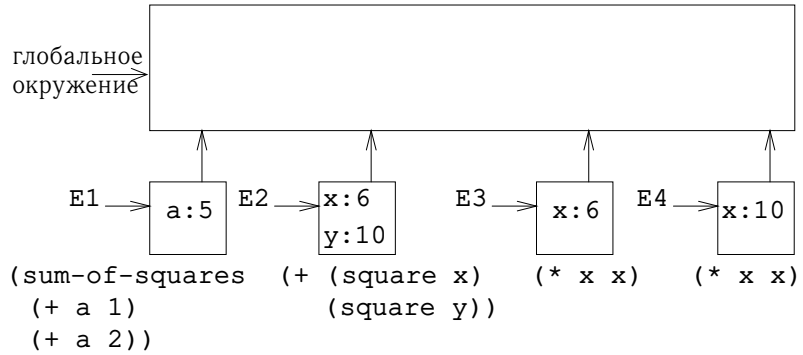


Рис. 3.5. Окружения, созданные при вычислении `(f 5)` с использованием процедур, изображенных на рис. 3.4

Для вычисления этой комбинации сначала мы вычисляем подвыражения. Значение первого подвыражения, `sum-of-squares` — процедурный объект. (Обратите внимание, как мы находим этот объект: сначала мы просматриваем первый кадр E1, который не содержит связывания для переменной `sum-of-squares`. Затем мы переходим в объемлющее окружение, а именно глобальное, и там находим связывание, которое показано на рис. 3.4.) В оставшихся двух подвыражениях элементарные операции `+` и `*` применяются при вычислении комбинаций `(+ a 1)` и `(* a 2)`, и дают, соответственно, результаты 6 и 10.

Теперь мы применяем процедурный объект `sum-of-squares` к аргументам 6 и 10. При этом создается новое окружение E2, в котором формальные параметры `x` и `y` связываются со значениями аргументов. Внутри E2 мы вычисляем комбинацию `(+ (square x) (square y))`. Для этого нам требуется вычислить `(square x)`, причем значение `square` мы находим в глобальном окружении, а `x` равен 6. Мы опять создаем новое окружение, E3, где `x` связан со значением 6, и где мы вычисляем тело `square`, то есть `(* x x)`. Кроме того, как часть вычисления `sum-of-squares`, нам нужно вычислить подвыражение `(square y)`, где `y` равен 10. Этот второй вызов `square` создает еще одно окружение E4, в котором `x`, формальный параметр `square`, связан со значением 10. Внутри E4 нам нужно вычислить `(* x x)`.

Важно заметить, что каждый вызов `square` создает новое окружение с новым связыванием для `x`. Теперь мы видим, как разделение кадров служит для того, чтобы разные локальные переменные по имени `x` не смешивались. Заметим, кроме того, что все кадры, созданные процедурой `square`, указывают на глобальное окружение, поскольку указатель именно на это окружение содержится в процедурном объекте `square`.

После того, как подвыражения вычисляются, они возвращают значения. Значения, порожденные двумя вызовами `square`, складываются в `sum-ofsquares`, и этот результат возвращается процедурой `f`. Поскольку сейчас наше внимание сосредоточено на структурах окружений, мы не будем здесь разбираться, как значения передаются от вызова к вызову; однако на самом деле это важная часть процесса вычисления, и мы детально рассмотрим ее в главе 5.

Упражнение 3.9.
В разделе 1.2.1 мы с помощью подстановочной модели анализировали две процедуры вычисления факториала, рекурсивную

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

и итеративную

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Продемонстрируйте, какие структуры окружений возникнут при вычислении `(factorial 6)` с каждой из версий процедуры `factorial`[14].

3.2.3. Кадры как хранилище внутреннего состояния

Теперь мы можем обратиться к модели с окружениями и рассмотреть, как можно с помощью процедур и присваивания представлять объекты, обладающие внутренним состоянием. В качестве примера возьмем «обработчик снятия денег со счета» из раздела 3.1.1, который создается вызовом процедуры

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете")))
```

Опишем вычисление

```
(define W1 (make-withdraw 100))
```

за которым следует

```
(W1 50)
```

На рисунке 3.6 показан результат определения `make-withdraw` в глобальном окружении. Получается процедурный объект, который содержит ссылку на глобальное окружение. До сих пор мы не видим особых отличий от тех примеров, которые мы уже рассмотрели, кроме того, что тело процедуры само по себе является лямбда-выражением.

Интересная часть вычисления начинается тогда, когда мы применяем процедуру `make-withdraw` к аргументу:

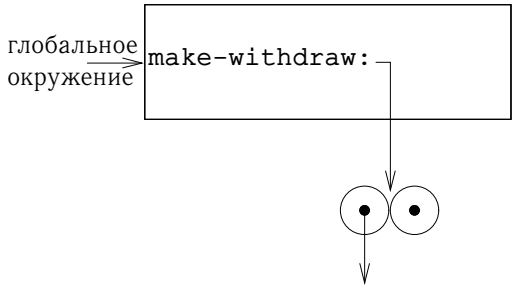
```
(define W1 (make-withdraw 100))
```

Сначала, как обычно, мы создаем окружение E1, где формальный параметр `balance` связан с аргументом 100. Внутри этого окружения мы вычисляем тело `make-withdraw`, а именно `lambda`-выражение. При этом создается новый процедурный объект, код которого определяется `lambda`-выражением, а окружение равно E1, окружению, в котором вычисляется `lambda` при создании процедуры. Полученный процедурный объект возвращается в качестве значения процедуры `make-withdraw`. Это значение присваивается переменной `W1` в глобальном окружении, поскольку выражение `define` вычисляется именно в нем. Получившаяся структура окружений изображена на рисунке 3.7.

Теперь можно проанализировать, что происходит, когда `W1` применяется к аргументу:

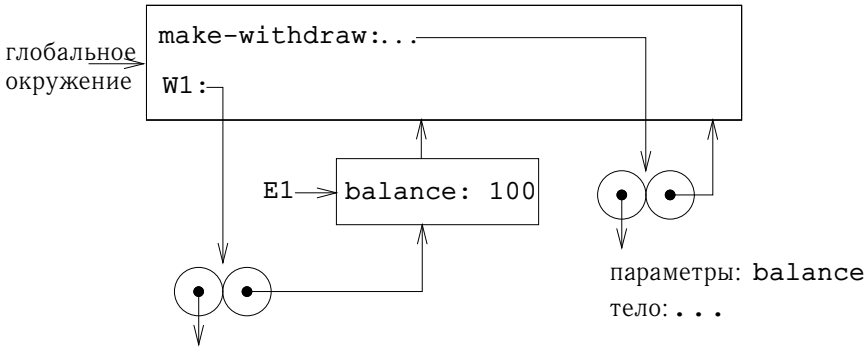
```
(W1 50)
50
```

[14] Модель с окружениями неспособна проиллюстрировать утверждение из раздела 1.2.1, что интерпретатор может, используя хвостовую рекурсию, вычислять процедуры, подобные `fact-iter`, в фиксированном объеме памяти. Мы рассмотрим хвостовую рекурсию, когда будем изучать управляющую структуру интерпретатора в разделе 5.4.



параметры: `balance`
тело:
```
(lambda (amount)
        (if (>= balance amount)
            (begin (set! balance (- balance amount))
            balance)
        "Недостаточно денег на счете"))
```

Рис. 3.6. Результат определения `make-withdraw` в глобальном окружении.



параметры: `amount`
тело:
```
(if (>= balance amount)
        (begin (set! balance (-balance amount))
        balance)
    "Недостаточно денег на счете"))
```

Рис. 3.7. Результат вычисления `(define W1 (make-withdraw 100))`.

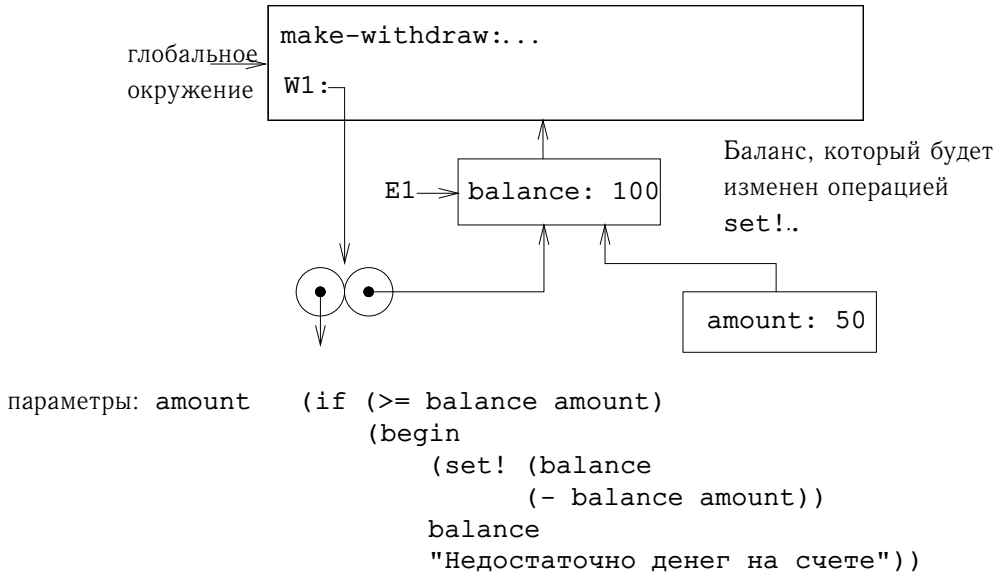


Рис. 3.8. Окружения, создаваемые при применении процедурного объекта `W1`.

Для начала мы конструируем кадр, в котором `amount`, формальный параметр `W1`, связывается со значением 50. Здесь крайне важно заметить, что у этого кадра в качестве объемлющего окружения выступает не глобальное окружение, а E1, поскольку именно на него указывает процедурный объект `W1`. В этом новом окружении мы вычисляем тело процедуры:

```
(if (>= balance amount)
    (begin (set! balance (- balance amount))
           balance)
    "Недостаточно денег на счете")
```

Получается структура окружений, изображенная на рисунке 3.8. Вычисляемое выражение обращается к переменным `amount` и `balance`. `Amount` находится в первом кадре окружения, а `balance` мы найдем, проследовав по указателю на объемлющее окружение E1.

Когда выполняется `set!`, связывание переменной `balance` в E1 изменяется. После завершения вызова `W1` значение `balance` равно 50, а `W1` по-прежнему указывает на кадр, который содержит переменную `balance`. Кадр, содержащий `amount` (тот, в котором мы выполняли код, изменяющий `balance`), больше не нужен, поскольку создавший его вызов процедуры закончен, и никаких указателей на этот кадр из других частей окружения нет. В следующий раз, когда мы позовем `W1`, создастся новый кадр, в котором будет связана переменная `amount`, и для которого объемлющим окружением снова будет E1. Мы видим, что E1 служит «местом», в котором хранится локальная переменная окружения для процедурного объекта `W1`. На рисунке 3.9 изображена ситуация

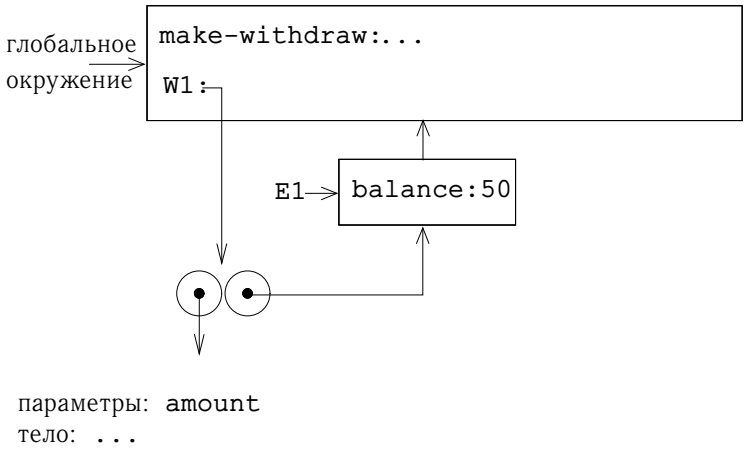


Рис. 3.9. Окружения после вызова W1.

после вызова W1.

Рассмотрим, что произойдет, когда мы создадим другой объект для «снятия денег», вызвав make-withdraw второй раз:

```
(define W2 (make-withdraw 100))
```

При этом получается структура окружений, изображенная на рисунке 3.10. Мы видим, что W2 — процедурный объект, то есть пара, содержащая код и окружение. Окружение E2 для W2 было создано во время вызова make-withdraw. Оно содержит кадр со своим собственным связыванием переменной balance. С другой стороны, код у W1 и W2 один и тот же: это код, определяемый lambda-выражением в теле make-withdraw[15]. Отсюда мы видим, почему W1 и W2 ведут себя как независимые объекты. Вызовы W1 работают с переменной состояния balance, которая хранится в E1, а вызовы W2 с переменной balance, хранящейся в E2. Таким образом, изменения внутреннего состояния одного объекта не действуют на другой.

Упражнение 3.10.
В процедуре make-withdraw локальная переменная balance создается в виде параметра make-withdraw. Можно было бы создать локальную переменную и явно, используя let, а именно:

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Недостаточно денег на счете"))))
```

[15]Разделяют ли W1 и W2 общий физический код, хранимый в компьютере, или каждый из них хранит собственную копию кода — это деталь реализации. В интерпретаторе, который мы создадим в главе 4, код будет общим.

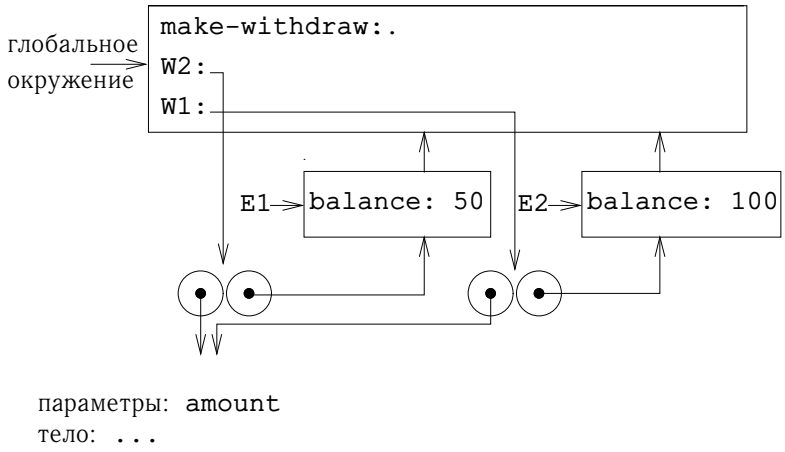


Рис. 3.10. Создание второго объекта при помощи `(define W2 (make-withdraw 100))`

Напомним, что в разделе 1.3.2 говорится, что `let` всего лишь синтаксический сахар для вызова процедуры:

```
(let ((⟨пер⟩ ⟨выр⟩))
   ⟨тело⟩)
```

интерпретируется как альтернативный синтаксис для

```
((lambda (⟨пер⟩) ⟨тело⟩) ⟨выр⟩)
```

С помощью модели с окружениями проанализируйте альтернативную версию `makewithraw`. Нарисуйте картинки, подобные приведенным в этом разделе, для выражений

```
(define W1 (make-withdraw 100))

(W1 50)

(define W2 (make-withdraw 100))
```

Покажите, что две версии `make-withdraw` создают объекты с одинаковым поведением. Как различаются структуры окружений в двух версиях?

3.2.4. Внутренние определения

В разделе 1.1.8 мы познакомились с идеей, что процедуры могут содержать внутренние определения, в результате чего возникает блочная структура, как, например, в следующей процедуре вычисления квадратного корня:

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
```

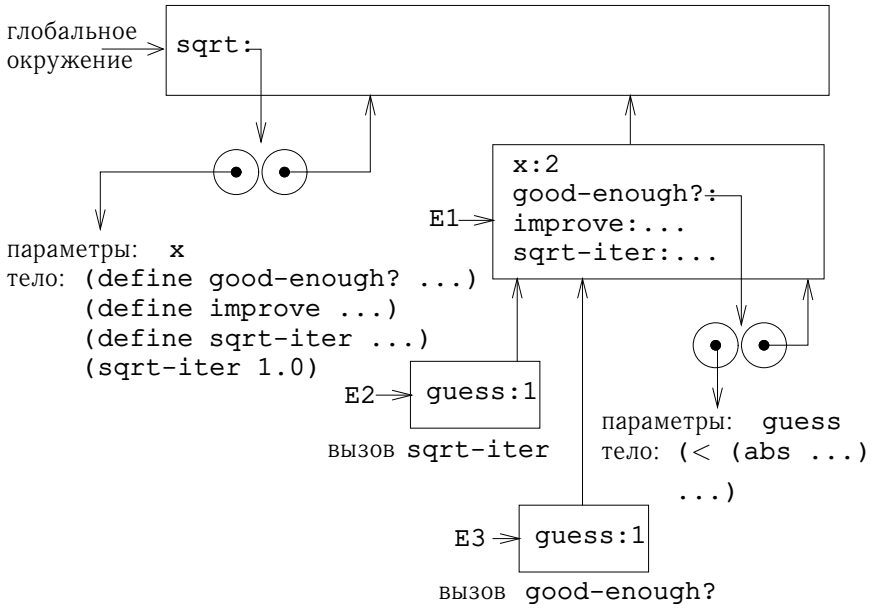


Рис. 3.11. Процедура `sqrt` с внутренними определениями.

```
(define (improve guess)
  (average guess (/ x guess)))
(define (sqrt-iter guess)
  (if (good-enough? guess)
      guess
      (sqrt-iter (improve guess))))
(sqrt-iter 1.0))
```

Теперь с помощью модели с окружениями мы можем увидеть, почему эти внутренние определения работают так, как должны. На рисунке 3.11 изображен момент во время вычисления выражения `(sqrt 2)`, когда внутренняя процедура `good-enough?` вызвана в первый раз со значением `guess`, равным 1.

Рассмотрим структуру окружения. Символ `sqrt` в глобальном окружении связан с процедурным объектом, ассоциированное окружение которого — глобальное окружение. Когда мы вызвали процедуру `sqrt`, появилось окружение E1, зависимое от глобального, в котором параметр `x` связан со значением 2. Затем мы вычислили тело `sqrt` внутри E1. Поскольку первое выражение в теле `sqrt` есть

```
(define (good-enough? guess)
  (< (abs (- (square guess) x)) 0.001))
```

вычисление этого выражения привело к определению процедуры `good-enough?` в окружении E1. Выражаясь более точно, к первому кадру E1 был добавлен символ `good-enough?`, связанный с процедурным объектом, ассоциированным окружением которого является E1. Подобным образом в качестве процедур внутри E1 были определены

improve и sqrt-iter. Краткости ради на рис. 3.11 показан только процедурный объект, соответствующий good-enough?.

После того, как были определены внутренние процедуры, мы вычислили выражение (sqrt-iter 1.0), по-прежнему в окружении E1. То есть, процедурный объект, связанный в E1 с именем sqrt-iter, был вызван с аргументом 1. При этом появилось окружение E2, в котором guess, параметр sqrt-iter, связан со значением 1. В свою очередь, sqrt-iter вызвала good-enough? со значением guess (из E2) в качестве аргумента. Получилось еще одно окружение, E3, в котором guess (параметр good-enough?) связан со значением 1. Несмотря на то, что и sqrt-iter, и good-enough? имеют по параметру с одинаковым именем guess, это две различные переменные, расположенные в разных кадрах. Кроме того, и E2, и E3 в качестве объемлющего окружения имеют E1, поскольку как sqrt-iter, так и good-enough? в качестве окружения содержат указатель на E1. Одним из следствий этого является то, что символ x в теле good-enough? обозначает связывание x, в окружении E1, а точнее, то значение x, с которым была вызвана исходная процедура sqrt.

Таким образом, модель вычислений с окружениями объясняет две ключевых особенности, которые делают внутренние определения процедур полезным способом модуляризации программ:

- Имена внутренних процедур не путаются с именами, внешними по отношению к охватывающей процедуре, поскольку локальные имена процедур будут связываться в кадре, который процедура создает при своем запуске, а не в глобальном окружении.

- Внутренние процедуры могут обращаться к аргументам охватывающих процедур, просто используя имена параметров как свободные переменные. Это происходит потому, что тело внутренней процедуры выполняется в окружении, подчиненном окружению, где вычисляется объемлющая процедура.

Упражнение 3.11.
В разделе 3.2.3 мы видели, как модель с окружениями описывает поведение процедур, обладающих внутренним состоянием. Теперь мы рассмотрели, как работают локальные определения. Типичная процедура с передачей сообщений пользуется и тем, и другим. Рассмотрим процедуру моделирования банковского счета из раздела 3.1.1:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Неизвестный вызов -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Покажите, какая структура окружений создается последовательностью действий

```
(define acc (make-account 50))

((acc 'deposit) 40)
90

((acc 'withdraw) 60)
30
```

Где хранится внутреннее состояние `acc`? Предположим, что мы определяем еще один счет

```
(define acc2 (make-account 100))
```

Каким образом удается не смешивать внутренние состояния двух счетов? Какие части структуры окружений общие у `acc` и `acc2`?

3.3. Моделирование при помощи изменяемых данных

В главе 2 составные данные использовались как средство построения вычислительных объектов, состоящих из нескольких частей, с целью моделирования объектов реального мира, обладающих несколькими свойствами. В этой главе мы ввели дисциплину абстракции данных, согласно которой структуры данных описываются в терминах конструкторов, которые создают объекты данных, и селекторов, которые обеспечивают доступ к частям составных объектов. Однако теперь мы знаем, что есть еще один аспект работы с данными, который остался незатронутым в главе 2. Желание моделировать системы, которые состоят из объектов, обладающих изменяющимся состоянием, вызывает потребность не только создавать составные объекты данных и иметь доступ к их частям, но и изменять их. Чтобы моделировать объекты с изменяющимся состоянием, мы будем проектировать абстракции данных, которые, помимо конструкторов и селекторов, включают *мутаторы* (mutators), модифицирующие объекты данных. Например, моделирование банковской системы требует от нас способности изменять балансы счетов. Таким образом, структура данных, изображающая банковский счет, может обладать операцией

```
(set-balance! ⟨счет⟩ ⟨новое-значение⟩)
```

которая присваивает балансу указанного счета указанное значение. Объекты данных, для которых определены мутаторы, называются *изменяемыми объектами данных* (mutable data objects).

В главе 2 в качестве универсального «клея» для построения составных данных мы ввели пары. Этот раздел мы начинаем с определения мутаторов для пар, так, чтобы пары могли служить строительным материалом для построения изменяемых объектов данных. Мутаторы значительно увеличивают выразительную силу пар и позволяют нам строить структуры данных помимо последовательностей и деревьев, с которыми мы имели дело в разделе 2.2. Кроме того, мы строим несколько примеров моделей, где сложные системы представляются в виде множества объектов, обладающих внутренним состоянием.

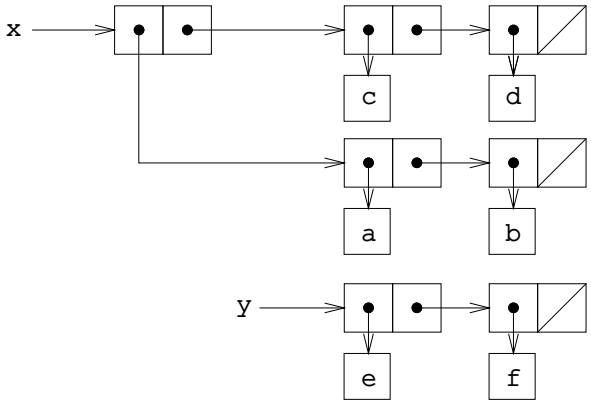


Рис. 3.12. Списки `x`: `((a b) c d)` и `y`: `(e f)`.

3.3.1. Изменяемая списковая структура

Базовые операции над парами — `cons`, `car` и `cdr` — можно использовать для построения списковой структуры и для извлечения частей списковой структуры, однако изменять списковую структуру они не позволяют. То же верно и для операций со списками, которые мы до сих пор использовали, таких, как `append` и `list`, поскольку эти последние можно определить в терминах `cons`, `car` и `cdr`. Для модификации списковых структур нам нужны новые операции.

Элементарные мутаторы для пар называются `setcar!` и `set-cdr!`. `Setcar!` принимает два аргумента, первый из которых обязан быть парой. Он модифицирует эту пару, подставляя вместо указателя `car` указатель на свой второй аргумент[16].

В качестве примера предположим, что переменная `x` имеет значением список `((a b) c d)`, а переменная `y` список `(e f)`, как показано на рисунке 3.12. Вычисление выражения `(set-car! x y)` изменяет пару, с которой связана переменная `x`, заменяя ее `car` на значение `y`. Результат этой операции показан на рисунке 3.13. Структура `x` изменилась, и теперь ее можно записать как `((e f) c d)`. Пары представляющие список `(a b)`, на которые указывал замененный указатель, теперь отделены от исходной структуры[17].

Сравните рисунок 3.13 с рис. 3.14, на котором представлен результат выполнения `(define z (cons y (cdr x)))`, где `x` и `y` имеют исходные значения с рис. 3.12. Здесь переменная `z` оказывается связана с новой парой, созданной операцией `cons`; список, который является значением `x`, не меняется.

Операция `set-cdr!` подобна `set-car!`. Единственная разница состоит в том, что заменяется не указатель `car`, а указатель `cdr`. Результат применения `(set-cdr! x y)`

[16]Значения, которые возвращают `set-car!` и `set-cdr!`, зависят от реализации. Подобно `set!`, эти операции должны использоваться исключительно ради своего побочного эффекта.

[17]Здесь мы видим, как операции изменения данных могут создавать «мусор», который не является частью никакой доступной структуры. В разделе 5.3.2 мы увидим, что системы управления памятью Лиспа включают *сборщик мусора* (garbage collector), который находит и освобождает память, используемую ненужными парами.

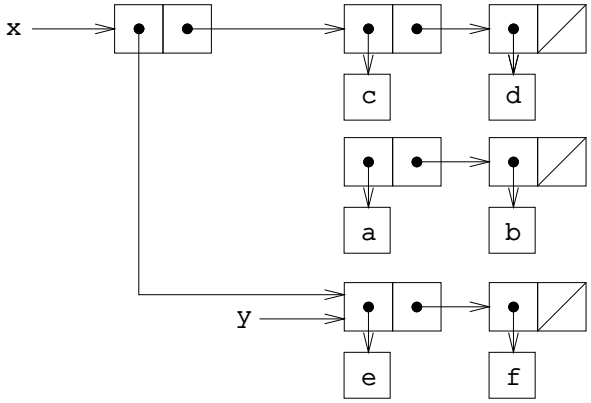


Рис. 3.13. Результат применения `(set-car! x y)` к спискам, изображенным на рис. 3.12.

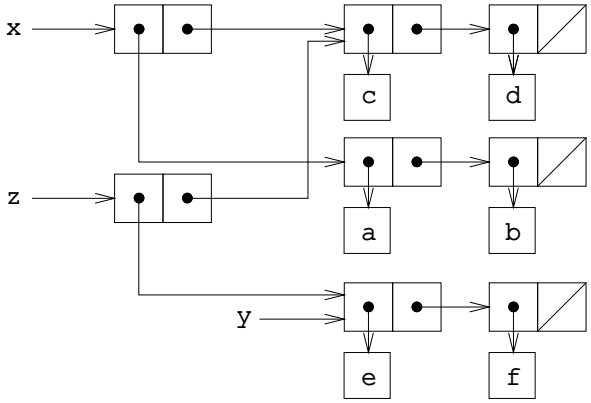


Рис. 3.14. Результат применения `(define z (cons y (cdr x))` к спискам, показанным на рис. 3.12.

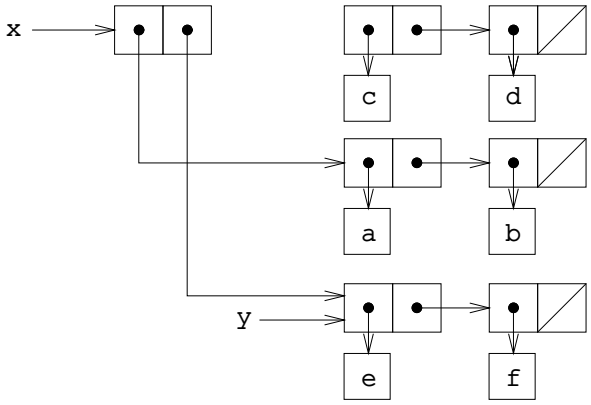


Рис. 3.15. Результат применения `(set-cdr! x y)` к спискам с рис. 3.12.

к спискам, изображенным на рис. 3.12, показан на рис. 3.15. Здесь указатель `cdr` в составе `x` заменился указателем на `(e f)`. Кроме того, список `(c d)`, который был `cdr`-ом `x`, оказывается отделенным от структуры.

`Cons` создает новую списковую структуру, порождая новые пары, а `setcar!` и `set-cdr!` изменяют существующие. В сущности, мы могли бы реализовать `cons` при помощи этих двух мутаторов и процедуры `get-new-pair`, которая возвращает новую пару, не являющуюся частью никакой существующей списковой структуры. Мы порождаем новую пару, присваиваем ее указателям `car` и `cdr` нужные значения, и возвращаем новую пару в качестве результата `cons`[18]:

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

Упражнение 3.12.
В разделе 2.2.1 была введена следующая процедура для добавления одного списка к другому:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

`Append` порождает новый список, по очереди наращивая элементы `x` в начало `y`. Процедура `append!` подобна `append`, но только она является не конструктором, а мутатором. Она склеивает списки вместе, изменяя последнюю пару `x` так, что ее `cdr` становится равным `y`. (Вызов `append!` с пустым `x` является ошибкой.)

[18] `Get-new-pair` — одна из операций, которые требуется предоставить как часть системы управления памятью в рамках реализации Лиспа. Мы рассмотрим эти вопросы в разделе 5.3.1.

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

Здесь `last-pair` — процедура, которая возвращает последнюю пару своего аргумента:

```
(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))
```

Рассмотрим последовательность действий

```
(define x (list 'a 'b))

(define y (list 'c 'd))

(define z (append  x y))

z
(a b c d)

(cdr x)
⟨ответ⟩

(define w (append! x y))

w
(a b c d)

(cdr x)
⟨ответ⟩
```

Каковы будут пропущенные ⟨*ответы*⟩? Объясните, нарисовав стрелочные диаграммы.

Упражнение 3.13.
Рассмотрим следующую процедуру `make-cycle`, которая пользуется `last-pair` из упражнения 3.12:

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)
```

Нарисуйте стрелочную диаграмму, которая изображает структуру `z`, созданную таким кодом:

```
(define z (make-cycle (list 'a 'b 'c)))
```

Что случится, если мы попробуем вычислить `(last-pair z)`?

Упражнение 3.14.
Следующая процедура, хотя и сложна для понимания, вполне может оказаться полезной:

```
(define (mystery x)
  (define (loop x y)
```

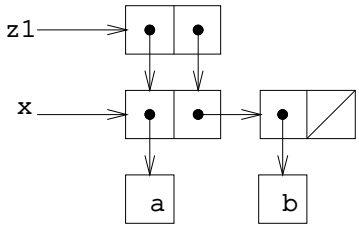


Рис. 3.16. Список `z1`, порождаемый выражением `(cons x x)`.

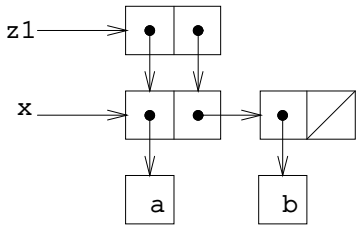


Рис. 3.17. Список z2, порождаемый выражением `(cons (list 'a 'b) (list 'a 'b))`.

```
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '()))
```

`Loop` пользуется «временной» переменной `temp`, чтобы сохранить старое значение `cdr` пары `x`, поскольку `set-cdr!` на следующей строке его разрушает. Объясните, что за задачу выполняет `mystery`. Предположим, что переменная `v` определена выражением `(define v (list 'a 'b 'c 'd))`. Нарисуйте диаграмму, которая изображает список, являющийся значением `v`. Допустим, что теперь мы выполняем `(define w (mystery v))`. Нарисуйте стрелочные диаграммы, которые показывают структуры `v` и `w` после вычисления этого выражения. Что будет напечатано в качестве значений `v` и `w`?

Разделение данных и их идентичность

В разделе 3.1.3 мы упоминали теоретические вопросы «идентичности» и «изменения», которые возникают с появлением присваивания. Эти вопросы начинают иметь практическое значение тогда, когда отдельные пары *разделяются* (are shared) между различными объектами данных. Рассмотрим, например, структуру, которая создается таким кодом:

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

Как показано на рис. 3.16, `z1` представляет собой пару, в которой `car` и `cdr` указывают на одну и ту же пару `x`. Разделение `x` между `car` и `cdr` пары `z1` возникает оттого, что `cons` реализован простейшим способом. В общем случае построение списков с помощью `cons` приводит к возникновению сложносвязанной сети пар, в которой многие пары разделяются между многими различными структурами.

В противоположность рис. 3.16, рис. 3.17 показывает структуру, которая порождается кодом

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

В этой структуре пары двух списков `(a b)` различны, притом, что сами символы разделяются[19].

Если мы рассматриваем `z1` и `z2` как списки, они представляют «один и тот же» список `((a b) a b)`. Вообще говоря, разделение данных невозможно заметить, если мы работаем со списками только при помощи операций `cons`, `car` и `cdr`. Однако если мы вводим мутаторы, работающие со списковой структурой, разделение данных начинает иметь значение. Как пример случая, когда разделение влияет на результат, рассмотрим следующую процедуру, которая изменяет `car` структуры, к которой она применяется:

```
(define (set-to-wow! x)
  (set-car! (car x) 'wow)
  x)
```

Несмотря на то, что `z1` и `z2` имеют «одинаковую» структуру, применение к ним процедуры `set-to-wow!` дает различные результаты. В случае с `z1` изменение `car` влияет и на `cdr`, поскольку здесь `car` и `cdr` — это одна и та же пара. В случае с `z2`, `car` и `cdr` различны, так что `set-to-wow!` изменяет только `car`:

```
z1
((a b) a b)

(set-to-wow! z1)
((wow b) wow b)

z2
((a b) a b)

(set-to-wow! z2)
((wow b) a b)
```

Один из способов распознать разделение данных в списковых структурах — это воспользоваться предикатом `eq?`, который мы ввели в разделе 2.3.1 как метод проверки двух символов на равенство. В более общем случае `(eq? x y)` проверяет, являются ли `x` и `y` одним объектом (то есть, равны ли `x` и `y` друг другу как указатели). Так что,

[19]Пары различаются потому, что каждый вызов `cons` порождает новую пару. Символы разделяются; в Scheme существует только один символ для каждого данного имени. Поскольку Scheme не дает возможности изменять символ, это разделение невозможно заметить. Заметим, кроме того, что именно разделение позволяет нам сравнивать символы при помощи `eq?`, который просто проверяет равенство указателей.

если z1 и z2 определены как на рисунках 3.16 и 3.17, (eq? (car z1) (cdr z1)) будет истинно, а (eq? (car z2) (cdr z2)) ложно.

Как будет видно в последующих разделах, с помощью разделения данных мы значительно расширим репертуар структур данных, которые могут быть представлены через пары. С другой стороны, разделение сопряжено с риском, поскольку изменения в одних структурах могут затрагивать и другие структуры, разделяющие те части, которые подвергаются изменению. Операции изменения set-car! и set-cdr! нужно использовать осторожно; если у нас нет точного понимания, какие из наших объектов разделяют данные, изменение может привести к неожиданным результатам[20].

Упражнение 3.15.
Нарисуйте стрелочные диаграммы, объясняющие, как set-to-wow! действует на структуры z1 и z2 из этого раздела.

Упражнение 3.16.
Бен Битобор решил написать процедуру для подсчета числа пар в любой списковой структуре. «Это легко, — думает он. — Число пар в любой структуре есть число пар в car плюс число пар в cdr плюс один на текущую пару». И он пишет следующую процедуру:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

Покажите, что эта процедура ошибочна. В частности, нарисуйте диаграммы, представляющие списковые структуры ровно из трех пар, для которых Бенова процедура вернет 3; вернет 4; вернет 7; вообще никогда не завершится.

Упражнение 3.17.
Напишите правильную версию процедуры count-pairs из упражнения 3.16, которая возвращает число различных пар в любой структуре. (Подсказка: просматривайте структуру, поддерживая при этом вспомогательную структуру, следящую за тем, какие пары уже были посчитаны.)

Упражнение 3.18.
Напишите процедуру, которая рассматривает список и определяет, содержится ли в нем цикл, то есть, не войдет ли программа, которая попытается добраться до конца списка, продвигаясь по полям cdr, в бесконечный цикл. Такие списки порождались в упражнении 3.13.

[20]Тонкости работы с разделением изменяемых данных отражают сложности с понятием «идентичности» и «изменения», о которых мы говорили в разделе 3.1.3. Там мы отметили, что введение в наш язык понятия изменения требует, чтобы у составного объекта была «индивидуальность», которая представляет собой нечто отличное от частей, из которых он состоит. В Лиспе мы считаем, что именно эта «индивидуальность» проверяется предикатом eq?, то есть сравнением указателей. Поскольку в большинстве реализаций Лиспа указатель — это, в сущности, адрес в памяти, мы «решаем проблему» определения индивидуальности объектов, постановив, что «сам» объект данных есть информация, хранимая в некотором наборе ячеек памяти компьютера. Для простых лисповских программ этого достаточно, но такой метод не способен разрешить общий вопрос «идентичности» в вычислительных моделях.

Упражнение 3.19.
Переделайте упражнение 3.18, используя фиксированное количество памяти. (Тут нужна достаточно хитрая идея.)

Изменение как присваивание

Когда мы вводили понятие составных данных, в разделе 2.1.3 мы заметили, что пары можно представить при помощи одних только процедур:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Неопределенная операция -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))
```

То же наблюдение верно и для изменяемых данных. Изменяемые объекты данных можно реализовать при помощи процедур и внутреннего состояния. Например, можно расширить приведенную реализацию пар, так, чтобы `set-car!` и `set-cdr!` обрабатывались по аналогии с реализацией банковских счетов через `make-account` из раздела 3.1.1:

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Неопределенная операция -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))

(define (set-car! z new-value)
  ((z 'set-car!) new-value)
  z)

(define (set-cdr! z new-value)
  ((z 'set-cdr!) new-value)
  z)
```

Операция	Результат
`(define q (make-queue)`	
`(insert-queue! q 'a)`	a
`(insert-queue! q 'b)`	a b
`(delete-queue! q)`	b
`(insert-queue! q 'c)`	b c
`(insert-queue! q 'd)`	b c d
`(delete-queue! q)`	c d

Рис. 3.18. Операции над очередью.

Теоретически, чтобы описать поведение изменяемых данных, не требуется ничего, кроме присваивания. Как только мы вводим в наш язык `set!`, мы сталкиваемся со всеми проблемами, не только собственно присваивания, но и вообще изменяемых данных[21].

Упражнение 3.20.
Нарисуйте диаграммы окружений, изображающие выполнение последовательности выражений

```
(define x (cons 1 2))
(define z (cons x x))
(set-car! (cdr z) 17)

(car x)
17
```

с помощью вышеприведенной процедурной реализации пар. (Ср. с упражнением 3.11.)

3.3.2. Представление очередей

Мутаторы `set-car!` и `set-cdr!` позволяют нам строить из пар такие структуры, какие мы не смогли бы создать только при помощи `cons`, `car` и `cdr`. В этом разделе будет показано, как представить структуру данных, которая называется очередь. В разделе 3.3.3 мы увидим, как реализовать структуру, называемую таблицей.

Очередь (queue) представляет собой последовательность, в которую можно добавлять элементы с одного конца (он называется *хвостом* (rear)) и убирать с другого (он называется *головой* (front)). На рисунке 3.18 изображено, как в изначально пустую очередь добавляются элементы `a` и `b`. Затем `a` убирается из очереди, в нее добавляются `c` и `d`, потом удаляется `b`. Поскольку элементы удаляются всегда в том же порядке, в котором они были добавлены, иногда очередь называют буфером *FIFO* (англ. first in, first out — первым вошел, первым вышел).

С точки зрения абстракции данных, можно считать, что очередь определяется следующим набором операций:

[21] С другой стороны, с точки зрения реализации, присваивание требует модификации окружения, которое само по себе является изменяемой структурой данных. Таким образом, присваивание и изменяемость данных обладают равной мощностью: каждое из них можно реализовать при помощи другого.

• конструктор `(make-queue)` возвращает пустую очередь (очередь, в которой нет ни одного элемента).

• два селектора: `(empty-queue?` ⟨*очередь*⟩`)` проверяет, пуста ли очередь, `(front-queue` ⟨*очередь*⟩`)` возвращает объект, находящийся в голове очереди. Если очередь пуста, он сообщает об ошибке. Очередь не модифицируется.

• Два мутатора: `(insert-queue!` ⟨*очередь*⟩ ⟨*элемент*⟩`)` вставляет элемент в хвост очереди и возвращает в качестве значения измененную очередь; `(delete-queue!` ⟨*очередь*⟩`)` удаляет элемент в голове очереди и возвращает в качестве значения измененную очередь. Если перед уничтожением элемента очередь оказывается пустой, выводится сообщение об ошибке.

Поскольку очередь есть последовательность элементов, ее, разумеется, можно было бы представить как обыкновенный список; головой очереди был бы `car` этого списка, вставка элемента в очередь сводилась бы к добавлению нового элемента в конец списка, а уничтожение элемента из очереди состояло бы просто во взятии `cdr` списка. Однако такая реализация неэффективна, поскольку для вставки элемента нам пришлось бы просматривать весь список до конца. Поскольку единственный доступный нам метод просмотра списка — это последовательное применение `cdr`, такой просмотр требует $\Theta(n)$ шагов для очереди с `n` членами. Простое видоизменение спискового представления преодолевает этот недостаток, позволяя нам реализовать операции с очередью так, чтобы все они требовали $\Theta(1)$ шагов; то есть, чтобы число шагов алгоритма не зависело от длины очереди.

Сложность со списковым представлением возникает из-за необходимости искать конец списка. Искать приходится потому, что, хотя стандартный способ представления списка в виде цепочки пар дает нам указатель на начало списка, легкодоступного указателя на конец он не дает. Модификация, обходящая этот недостаток, состоит в том, чтобы представлять очередь в виде списка, и держать еще дополнительный указатель на его последнюю пару. В таком случае, когда требуется вставить элемент, мы можем просто посмотреть на этот указатель и избежать за счет этого просмотра всего списка.

Очередь, таким образом, представляется в виде пары указателей, `frontptr` и `rear-ptr`, которые обозначают, соответственно, первую и последнюю пару обыкновенного списка. Поскольку нам хочется, чтобы очередь была объектом с собственной индивидуальностью, соединить эти два указателя можно с помощью `cons`, так что собственно очередь будет результатом `cons` двух указателей. Такое представление показано на рис. 3.19.

Во время определения операций над очередью мы пользуемся следующими процедурами, которые позволяют нам читать и записывать указатели на начало и конец очереди:

```
(define (front-ptr queue) (car queue))

(define (rear-ptr queue) (cdr queue))

(define (set-front-ptr! queue item) (set-car! queue item))

(define (set-rear-ptr! queue item) (set-cdr! queue item))
```

Теперь можно реализовать сами операции над очередью. Очередь будет считаться

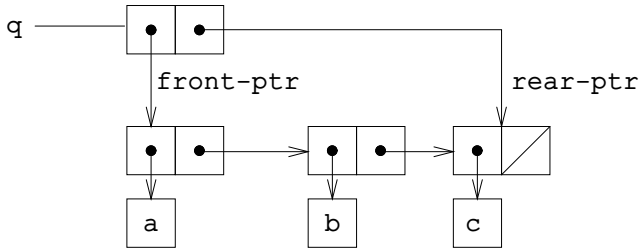


Рис. 3.19. Реализация очереди в виде списка с указателями на начало и конец.

пустой, если ее головной указатель указывает на пустой список:

```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

Конструктор `make-queue` возвращает в качестве исходно пустой очереди пару, в которой и `car`, и `cdr` являются пустыми списками:

```
(define (make-queue) (cons '() '()))
```

При обращении к элементу в голове очереди мы возвращаем `car` пары, на которую указывает головной указатель:

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT вызвана с пустой очередью" queue)
      (car (front-ptr queue))))
```

Чтобы вставить элемент в конец очереди, мы используем метод, результат которого показан на рисунке 3.20. Первым делом мы создаем новую пару, `car` которой содержит вставляемый элемент, а `cdr` — пустой список. Если очередь была пуста, мы перенаправляем на эту пару и головной, и хвостовой указатели. В противном случае, мы изменяем последнюю пару очереди так, чтобы следующей была новая пара, и хвостовой указатель тоже перенаправляем на нее же.

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```

Чтобы уничтожить элемент в голове очереди, мы просто переставляем головной указатель на второй элемент очереди, а его можно найти в `cdr` первого элемента

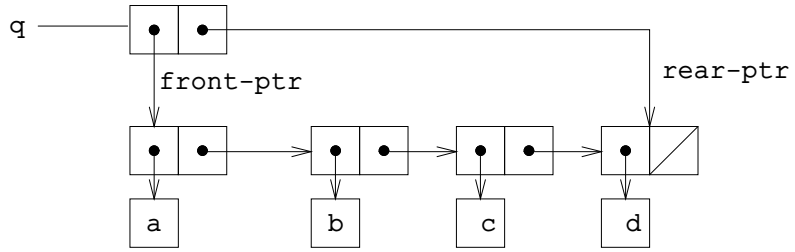


Рис. 3.20. Результат применения `(insert-queue! q 'd)` к очереди с рисунка 3.19

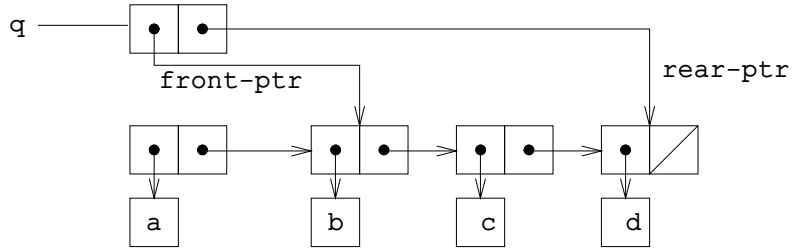


Рис. 3.21. Результат применения `(delete-queue! q)` к очереди с рис. 3.20.

(см. рис. 3.21)[22]:

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! вызвана с пустой очередью" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

Упражнение 3.21.
Бен Битобор решает протестировать вышеописанную реализацию. Он вводит процедуры в интерпретаторе Лиспа и тестирует их:

```
(define q1 (make-queue))

(insert-queue! q1 'a)
((a) a)

(insert-queue! q1 'b)
```

[22]В случае, если первый элемент — одновременно и последний, после его уничтожения головной указатель окажется пустым списком, и это будет означать, что очередь пуста; нам незачем заботиться о хвостовом указателе, который по-прежнему будет указывать на уничтоженный элемент, поскольку `empty-queue?` смотрит только на голову.

```
((a b) b)

(delete-queue! q1)
((b) b)

(delete-queue! q1)
(() b)
```

«Ничего не работает! — жалуется он. — Ответ интерпретатора показывает, что последний элемент попадает в очередь два раза. А когда я оба элемента уничтожаю, второе b по-прежнему там сидит, так что очередь не становится пустой, хотя должна бы». Ева Лу Атор говорит, что Бен просто не понимает, что происходит. «Дело не в том, что элементы два раза оказываются в очереди, — объясняет она. — Дело в том, что стандартная лисповская печаталка не знает, как устроено представление очереди. Если ты хочешь, чтобы очередь правильно печаталась, придется написать специальную процедуру распечатки очередей». Объясните, что имеет в виду Ева Лу. В частности, объясните, почему в примерах Бена на печать выдается именно такой результат. Определите процедуру `print-queue`, которая берет на входе очередь и выводит на печать последовательность ее элементов.

Упражнение 3.22.
Вместо того, чтобы представлять очередь как пару указателей, можно построить ее в виде процедуры с внутренним состоянием. Это состояние будет включать указатели на начало и конец обыкновенного списка. Таким образом, `make-queue` будет иметь вид

```
(define (make-queue)
  (let ((front-ptr ...)
        (rear-ptr ...))
    ⟨определения внутренних процедур⟩
    (define (dispatch m) ...)
    dispatch))
```

Закончите определение `make-queue` и реализуйте операции над очередями с помощью этого представления.

Упражнение 3.23.
Дек (deque, double-ended queue, «двусторонняя очередь») представляет собой последовательность, элементы в которой могут добавляться и уничтожаться как с головы, так и с хвоста. На деках определены такие операции: конструктор `make-deque`, предикат `empty-deque?`, селекторы `front-deque` и `rear-deque`, и мутаторы `frontinsertdeque!`, `rear-insert-deque!`, `front-delete-deque!` и `rear-delete-deque!`. Покажите, как представить дек при помощи пар, и напишите реализацию операций[23].Все операции должны выполняться за $\Theta(1)$ шагов.

3.3.3. Представление таблиц

Когда в главе 2 мы изучали различные способы представления множеств, то в разделе 2.3.3 была упомянута задача поддержания таблицы с идентифицирующими ключами. При реализации программирования, управляемого данными, в разделе 2.4.3, активно

[23]Осторожно, не заставьте ненароком интерпретатор печатать циклическую структуру (см. упр. 3.13).

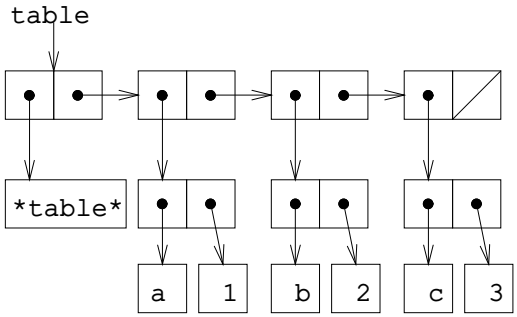


Рис. 3.22. Таблица, представленная в виде списка с заголовком.

использовались двумерные таблицы, в которых информация заносится и ищется с использованием двух ключей. Теперь мы увидим, как такие таблицы можно строить при помощи изменяемых списковых структур.

Сначала рассмотрим одномерную таблицу, где каждый элемент хранится под отдельным ключом. Ее мы реализуем как список записей, каждая из которых представляет собой пару, состоящую из ключа и связанного с ним значения. Пары связаны вместе в список при помощи цепочки пар, в каждой из которых `car` указывают на одну из записей. Эти связующие пары называются *хребтом* (backbone) таблицы. Для того, чтобы у нас было место, которое мы будем изменять при добавлении новой записи, таблицу мы строим как *список с заголовком* (headed list). У такого списка есть в начале специальная хребтовая пара, в которой хранится фиктивная «запись» — в данном случае произвольно выбранный символ `*table*`. На рисунке 3.22 изображена стрелочная диаграмма для таблицы

```
a: 1
b: 2
c: 3
```

Информацию из таблицы можно извлекать при помощи процедуры `lookup`, которая получает ключ в качестве аргумента, а возвращает связанное с ним значение (либо ложь, если в таблице с этим ключом никакого значения не связано). `Lookup` определена при помощи операции `assoc`, которая требует в виде аргументов ключ и список записей. Обратите внимание, что `assoc` не видит фиктивной записи. `Assoc` возвращает запись, которая содержит в `car` искомый ключ[24]. Затем `lookup` проверяет, что запись, возвращенная `assoc`, не есть ложь, и возвращает значение (то есть `cdr`) записи.

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))
```

[24]Поскольку `assoc` пользуется `equal?`, в качестве ключей она может распознавать символы, числа и списковые структуры.

```
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

Чтобы вставить в таблицу значение под данным ключом, сначала мы с помощью `assoc` проверяем, нет ли уже в таблице записи с этим ключом. Если нет, мы формируем новую запись, «`cons`ивая» ключ со значением, и вставляем ее в начало списка записей таблицы, после фиктивной записи. Если же в таблице уже была запись с этим ключом, мы переставляем `cdr` записи на указанное новое значение. Заголовок таблицы используется как неподвижное место, которое мы можем изменять при порождении новой записи[25].

```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))))
  'ok)
```

Для того, чтобы создать таблицу, мы просто порождаем список, содержащий символ `*table*`:

```
(define (make-table)
  (list '*table*))
```

Двумерные таблицы

В двумерной таблице каждое значение индексируется двумя ключами. Такую таблицу мы можем построить как одномерную таблицу, в которой каждый ключ определяет подтаблицу. На рисунке 3.23 изображена стрелочная диаграмма для таблицы

```
math:
    +:  43
    -:  45
    *:  42
letters:
    a:  97
    b:  98
```

содержащей две подтаблицы (подтаблицам не требуется специального заголовочного символа, поскольку для этой цели служит ключ, идентифицирующий подтаблицу).

Когда мы ищем в таблице элемент, сначала при помощи первого ключа мы находим нужную подтаблицу. Затем при помощи второго ключа мы определяем запись внутри подтаблицы.

[25] Таким образом, первая хребтовая пара является объектом, который представляет «саму» таблицу; то есть, указатель на таблицу — это указатель на эту пару. Таблица всегда начинается с одной и той же хребтовой пары. Будь это устроено иначе, пришлось бы возвращать из `insert!` новое начало таблицы в том случае, когда создается новая запись.

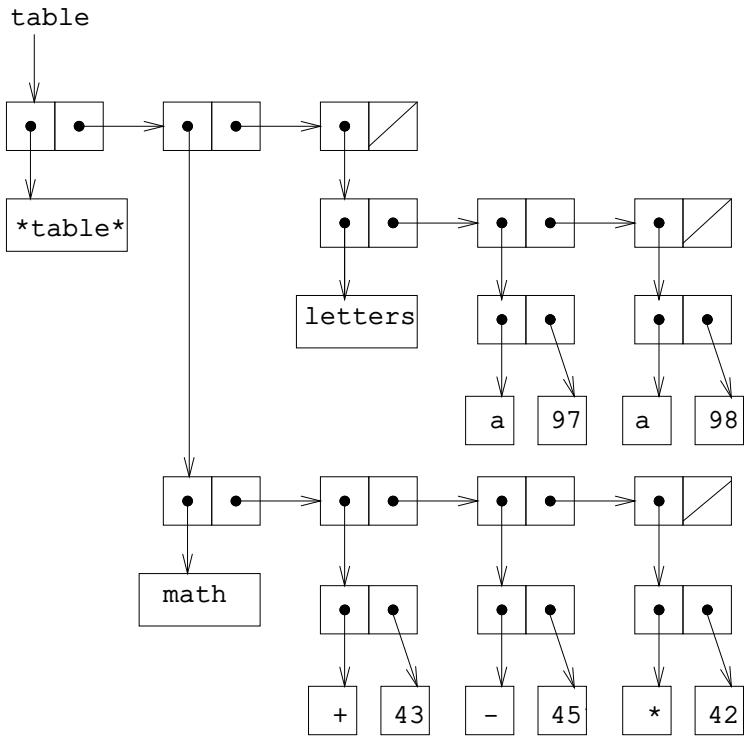


Рис. 3.23. Двумерная таблица.

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              false))
        false)))
```

Чтобы вставить в таблицу новый элемент под двумя ключами, мы при помощи `assoc` проверяем, соответствует ли какая-нибудь подтаблица первому ключу. Если нет, строим новую подтаблицу, содержащую единственную запись (`key-2`, `value`), и заносим ее в таблицу под первым ключом. Если для первого ключа уже существует подтаблица, мы вставляем новую запись в эту подтаблицу, используя вышеописанный метод вставки для одномерных таблиц:

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr table)))))
  'ok)
```

Создание локальных таблиц

Операции `lookup` и `insert!`, которые мы определили, принимают таблицу в качестве аргумента. Это позволяет писать программы, которые обращаются более, чем к одной таблице. Другой способ работы с множественными таблицами заключается в том, чтобы иметь для каждой из них свои отдельные процедуры `lookup` и `insert!`. Мы можем этого добиться, представив таблицу в процедурном виде, как объект, который поддерживает внутреннюю таблицу как часть своего локального состояния. Когда ему посылают соответствующее сообщение, этот «табличный объект» выдает процедуру, с помощью которой можно работать с его внутренним состоянием. Вот генератор двумерных таблиц, представленных таким способом:

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
```

```
            (if record
                (cdr record)
                false))
          false)))
  (define (insert! key-1 key-2 value)
    (let ((subtable (assoc key-1 (cdr local-table))))
      (if subtable
          (let ((record (assoc key-2 (cdr subtable))))
            (if record
                (set-cdr! record value)
                (set-cdr! subtable
                          (cons (cons key-2 value)
                                (cdr subtable)))))
          (set-cdr! local-table
                    (cons (list key-1
                                (cons key-2 value))
                          (cdr local-table)))))
    'ok)
  (define (dispatch m)
    (cond ((eq? m 'lookup-proc) lookup)
          ((eq? m 'insert-proc!) insert!)
          (else (error "Неизвестная операция -- TABLE" m))))
  dispatch))
```

`Make-table` позволяет нам реализовать операции `get` и `put` из раздела 2.4.3, так:

```
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```

`Get` в качестве аргументов берет два ключа, а `put` два ключа и значение. Обе операции обращаются к одной и той же локальной таблице, которая инкапсулируется в объекте, созданном посредством вызова `make-table`.

Упражнение 3.24.
В реализациях таблиц в этом разделе ключи всегда проверяются на равенство с помощью `equal?` (который, в свою очередь, зовется из `assoc`). Это не всегда то, что нужно. Например, можно представить себе таблицу с числовыми ключами, где не требуется точного совпадения с числом, которое мы ищем, а нужно только совпадение с определенной допустимой ошибкой. Постройте конструктор таблиц `make-table`, который в качестве аргумента принимает процедуру `same-key?` для проверки равенства ключей. `Make-table` должна возвращать процедуру `dispatch`. через которую можно добраться до процедур `lookup` и `insert!` локальной таблицы.

Упражнение 3.25.
Обобщая случаи одно- и двумерных таблиц, покажите, как можно реализовать таблицу, в которой элементы хранятся с произвольным количеством ключей и различные значения могут храниться с различным количеством ключей. Процедуры `lookup` и `insert!` должны принимать на входе список ключей, с которыми требуется обратиться к таблице.

Упражнение 3.26.
При поиске в таблице, как она реализована выше, приходится просматривать список записей. В сущности, это представление с неупорядоченным списком из раздела 2.3.3. Для больших таблиц может оказаться эффективнее организовать таблицу иначе. Опишите реализацию таблицы, в которой записи (ключ, значение) организованы в виде бинарного дерева, в предположении, что ключи можно каким-то образом упорядочить (например, численно или по алфавиту). (Ср. с упражнением 2.66 из главы 2.)

Упражнение 3.27.
Мемоизация (memoization) (называемая также *табуляризация* (tabulation)) — прием, который позволяет процедуре записывать в локальной таблице единожды вычисленные значения. Такой прием может сильно повысить производительность программы. Мемоизированная процедура поддерживает таблицу, где сохраняются результаты предыдущих вызовов, а в качестве ключей используются аргументы, относительно которых эти результаты были получены. Когда от мемоизированной процедуры требуют вычислить значение, сначала она проверят в таблице, нет ли там уже нужного значения, и если да, то она просто возвращает это значение. Если нет, то она вычисляет значение обычным способом и заносит его в таблицу. В качестве примера мемоизации, вспомним экспоненциальный процесс вычисления чисел Фибоначчи из раздела 1.2.2:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Мемоизированная версия той же самой процедуры выглядит так:

```
(define memo-fib
  (memoize (lambda (n)
             (cond ((= n 0) 0)
                   ((= n 1) 1)
                   (else (+ (memo-fib (- n 1))
                            (memo-fib (- n 2))))))))
```

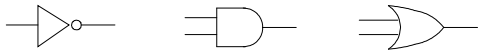
а процедура `memoize` определяется так:

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

Нарисуйте диаграмму окружений, анализирующую вычисление `(memo-fib 3)`. Объясните, почему `memo-fib` вычисляет n-е число Фибоначчи за число шагов, пропорциональное n. Стала бы схема работать, если бы мы определили `memo-fib` просто как `(memoize fib)`?

3.3.4. Имитация цифровых схем

Проектирование сложных цифровых систем, таких, как компьютеры, является важной отраслью инженерной деятельности. Цифровые системы строятся путем соединения



Inverter (инвертор) And-gate (И-элемент) Or-gate (ИЛИ-элемент)

Рис. 3.24. Элементарные функциональные элементы в имитаторе цифровых схем.

простых элементов. Хотя поведение этих составляющих элементов примитивно, сети, из них собранные, могут обладать весьма сложным поведением. Компьютерная имитация проектируемых электронных схем служит важным инструментом для инженеров-специалистов по цифровым системам. В этом разделе мы спроектируем систему для имитационного моделирования цифровых схем. Система эта будет служить примером программ особого вида, называемых *имитация, управляемая событиями* (event-driven simulation), в которых действия («события») вызывают другие события, которые происходят спустя некоторое время и при этом в свою очередь вызывают события, и так далее.

Наша вычислительная модель цифровой схемы будет состоять из объектов, соответствующих элементарным компонентам, из которых строится схема. Имеются *провода* (wires), несущие *цифровые сигналы* (digital signals). В каждый данный момент цифровой сигнал может иметь только одно из двух возможных значений, 0 или 1. Кроме того, имеются различные виды *функциональных элементов* (function boxes), которые соединяют провода, несущие входные сигналы, с выходными проводами. Такие элементы порождают выходные сигналы, вычисляя их на основе входных сигналов. Выходной сигнал задерживается на время, зависящее от типа функционального элемента. Например, *инвертор* (inverter) — элементарный функциональный элемент, который обращает свой входной сигнал. Если входной сигнал инвертора становится 0, то на одну инверторную задержку позже сигнал на выходе станет равен 1. Если входной сигнал станет 1, то на инверторную задержку позже на выходе появится 0. Инвертор символически изображен на рис. 3.24. *И-элемент* (and-gate), также показанный на рис. 3.24, имеет два входа и один выход. Он обеспечивает на выходе сигнал, равный *логическому И* (logical and) от входов. Это означает, что если оба входных сигнала становятся равными 1, то одну И-задержку спустя И-элемент заставит свой выходной сигнал стать 1; в противном случае на выходе будет 0.

ИЛИ-элемент (or-gate) представляет собой подобный же элементарный функциональный элемент, который обеспечивает на выходе сигнал, равный *логическому ИЛИ* (logical or) своих входов. А именно, выходной сигнал станет равен 1, если хотя бы один из входных сигналов окажется 1; в противном случае на выходе будет 0.

Соединяя элементарные функции, можно получать более сложные. Для этого надо подсоединять выходы одних функциональных элементов ко входам других. Например, схема *полусумматора* (half-adder) на рис. 3.25 состоит из ИЛИ-элемента, двух И-элементов и инвертора. Полусумматор получает два входа, A и B, и имеет два выхода, S и C. S становится 1, когда ровно один из сигналов A и B равен 1, а C тогда, когда и A, и B равны 1. Из схемы можно видеть, что по причине задержек выходные сигналы могут генерироваться в разное время. Отсюда происходят многие сложности в проектировании цифровых схем.

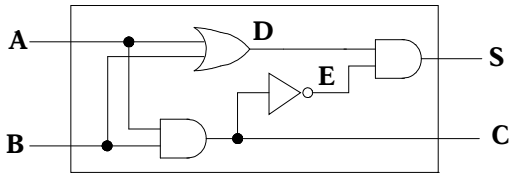


Рис. 3.25. Полусумматор.

Теперь мы построим программу для имитации цифровых логических схем, которые мы хотим изучать. Программа будет строить вычислительные объекты, моделирующие провода, которые «содержат» сигналы. Функциональные элементы будут моделироваться процедурами, которые обеспечивают нужное отношение между сигналами.

Одним из базовых элементов нашей имитации будет процедура `make-wire`, которая порождает провода. Например, мы можем создать шесть проводов так:

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

Мы подсоединяем функциональный элемент к проводу во время вызова процедуры, которая создает данный вид элемента. Аргументами порождающей процедуры служат провода, подсоединяемые к элементу. Например, если мы умеем создавать И-элементы, ИЛИ-элементы и инверторы, мы можем собрать полусумматор, изображенный на рисунке 3.25:

```
(or-gate a b d)
ok

(and-gate a b c)
ok

(inverter c e)
ok

(and-gate d e s)
ok
```

Даже лучше того, можно присвоить этой операции имя, определив процедуру `half-adder`, конструирующую схему, используя четыре внешних провода, которые нужно подсоединить к полусумматору:

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
```

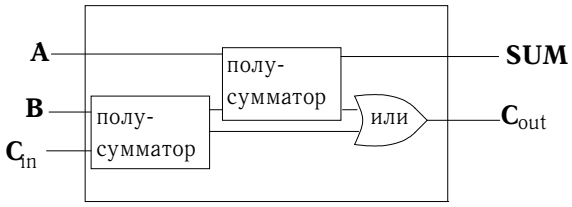


Рис. 3.26. Сумматор.

```
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

Преимущество этого определения в том, что теперь мы можем использовать `half-adder` как строительный блок при создании более сложных схем. Например, на рисунке 3.26 изображен *сумматор* (full-adder), состоящий из двух полусумматоров и ИЛИ-элемента[26]. Сумматор можно сконструировать так:

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

Определив `full-adder` как процедуру, мы можем ее использовать как строительный блок для еще более сложных схем. (См., например, упражнение 3.30.)

В сущности, наша имитация дает инструмент, с помощью которого строится язык описания схем. Принимая общую точку зрения на языки, с которой мы приступили к изучению Лиспа в разделе 1.1, можно сказать, что элементарные функциональные элементы являются примитивами языка, связывание их проводами представляет собой средство комбинирования, а определение шаблонных схем в виде процедур служит средством абстракции.

Элементарные функциональные элементы.

Элементарные функциональные элементы изображают «силы», через посредство которых изменение сигнала в одном проводе влечет изменение сигнала в других проводах. Для построения функциональных элементов мы будем пользоваться следующими операциями над проводами:

[26]Сумматор — основной элемент схем, используемых для сложения двоичных чисел. Здесь A и B — биты на соответствующих позициях двух складываемых чисел, а C_{in} — бит переноса из позиции на одну правее. Схема генерирует SUM, бит суммы для соответствующей позиции, и C_{out}, бит переноса для распространения налево.

- `(get-signal` ⟨*провод*⟩`)` возвращает текущее значение сигнала в проводе.
- `(set-signal!` ⟨*провод*⟩ ⟨*новое-значение*⟩`)` заменяет значение сигнала в проводе на указанное.
- `(add-action!` ⟨*провод*⟩ ⟨*процедура без аргументов*⟩`)` указывает, чтобы процедура-аргумент вызывалась каждый раз, когда сигнальный провод изменяет значение. Такие процедуры служат передаточным механизмом, с помощью которого изменение значения сигнала в одном проводе передается другим проводам. В дополнение, мы будем пользоваться процедурой `after-delay`, которая принимает значение задержки и процедуру. Она выполняет процедуру после истечения задержки.

При помощи этих процедур можно определить элементарные функции цифровой логики. Чтобы соединить вход с выходом через инвертор, мы используем `add-action!` и ассоциируем со входным проводом процедуру, которая будет вызываться всякий раз, когда сигнал на входе элемента изменит значение. Процедура вычисляет `logical-not` (логическое отрицание) входного сигнала, а затем, переждав `inverter-delay`, устанавливает выходной сигнал в новое значение:

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input)
  'ok)

(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Неправильный сигнал" s))))
```

И-элемент устроен немного сложнее. Процедура-действие должна вызываться, когда меняется любое из значений на входе. Она при этом через процедуру, подобную `logical-not`, вычисляет `logical-and` (логическое И) значений сигналов на входных проводах, и затем требует, чтобы изменение значения выходного провода произошло спустя задержку длиной в `and-gate-delay`.

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok)
```

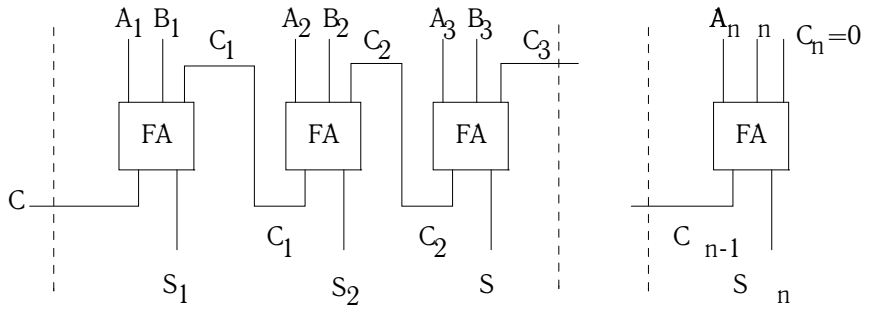


Рис. 3.27. Каскадный сумматор для n-битных чисел.

Упражнение 3.28.
Определите ИЛИ-элемент как элементарный функциональный блок. Ваш конструктор `or-gate` должен быть подобен `and-gate`.

Упражнение 3.29.
Еще один способ создать ИЛИ-элемент — это собрать его как составной блок из И-элементов и инверторов. Определите процедуру `or-gate`, которая это осуществляет. Как время задержки ИЛИ-элемента выражается через `and-gate-delay` и `inverter-delay`?

Упражнение 3.30.
На рисунке 3.27 изображен *каскадный сумматор* (ripple-carry adder), полученный выстраиванием в ряд n сумматоров. Это простейшая форма параллельного сумматора для сложения двух n-битных двоичных чисел. На входе мы имеем A_1, A_2, A_3,... A_n и B_1, B_2, B_3, ... B_n — два двоичных числа, подлежащих сложению (каждый из A_k и B_k имеет значение либо 0, либо 1). Схема порождает S_1, S_2, S_3, ... S_n — первые n бит суммы, и C – бит переноса после суммы. Напишите процедуру `riple-carry-adder`, которая бы моделировала эту схему. Процедура должна в качестве аргументов принимать три списка по n проводов в каждом (A_k, B_k и S_k), а также дополнительный провод C. Главный недостаток каскадных сумматоров в том, что приходится ждать, пока сигнал распространится. Какова задержка, требуемая для получения полного вывода n-битного каскадного сумматора, выраженная в зависимости от задержек И-, ИЛИ-элементов и инверторов?

Представление проводов

Провод в нашей имитации будет вычислительным объектом с двумя внутренними переменными состояния: значение сигнала `signal-value` (вначале равное 0) и набор процедур-действий `action-procedures`, подлежащих исполнению, когда сигнал изменяется. Мы реализуем провод в стиле с передачей сообщений, как набор локальных процедур плюс процедура диспетчеризации, которая выбирает требуемую внутреннюю операцию. Точно так же мы строили объект-банковский счет в разделе 3.1.1.

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '()))
    (define (set-my-signal! new-value)
```

```
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures))
          'done))
    (define (accept-action-procedure! proc)
      (set! action-procedures (cons proc action-procedures))
      (proc))
    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure!)
            (else (error "Неизвестная операция -- WIRE" m))))
    dispatch))
```

Внутренняя процедура `set-my-signal!` проверяет, отличается ли новое значение сигнала в проводе от старого. Если да, то она запускает все процедуры-действия при помощи процедуры `call-each`, которая по очереди вызывает элементы списка безаргументных процедур:

```
(define (call-each procedures)
  (if (null? procedures)
      'done
      (begin
        ((car procedures))
        (call-each (cdr procedures)))))
```

Внутренняя процедура `accept-action-procedure!` добавляет процедуру-аргумент к списку действий, а затем один раз запускает новую процедуру. (См. упражнение 3.31.)

Располагая вышеописанной процедурой `dispatch`, мы можем написать следующие процедуры для доступа к внутренним операциям над проводами[27]:

```
(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))

(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))
```

Провода, которые содержат меняющиеся со временем сигналы и могут подсоединяться к одному объекту за другим, — типичный образец изменяющихся объектов. Мы смоде-

[27] Эти процедуры — всего лишь синтаксический сахар, который позволяет нам работать с внутренними процедурами объектов, используя обычный синтаксис процедурного вызова. Поразительно, что мы так просто можем менять местами роли процедур и данных. Например, когда мы пишем `(wire 'get-signal)`, мы представляем себе провод `wire` как процедуру, вызываемую с сообщением `get-signal` на входе. С другой стороны, запись `(get-signal wire)` поощряет нас думать о `wire` как об объекте данных, который поступает на вход процедуре `get-signal`. Истина состоит в том, что в языке, где с процедурами можно работать как с объектами, никакого фундаментального различия между «процедурами» и «данными» не существует, и мы имеем право выбирать такой синтаксический сахар, который позволит программировать в удобном для нас стиле.

лировали их в виде процедур с внутренними переменными состояния, которые изменяются присваиванием. При создании нового провода создается новый набор переменных состояния (в выражении `let` внутри `make-wire`), а также порождается и возвращается новая процедура `dispatch`, которая захватывает окружение с новыми переменными состояния.

Провода разделяются между различными устройствами, к ним подсоединенными. Таким образом, изменение, произведенное при взаимодействии с одним устройством, скажется на всех других устройствах, связанных с этим проводом. Провод передает изменение своим соседям, вызывая процедуры-действия, зарегистрированные в нем в момент установления соединения.

План действий

Теперь для завершения модели нам остается только написать `after-delay`. Здесь идея состоит в том, чтобы организовать структуру данных под названием *план действий* (agenda), где будет храниться расписание того, что нам надо сделать. Для планов действий определены следующие операции:

- `(make-agenda)` возвращает новый пустой план действий.
- `(empty-agenda?` ⟨*план-действий*⟩`)` истинно, если план пуст.
- `(first-agenda-item` ⟨*план-действий*⟩`)` возвращает первый элемент плана.
- `(remove-first-agenda-item!` ⟨*план-действий*⟩`)` модифицирует план, убирая из него первый элемент.
- `(add-to-agenda!` ⟨*время*⟩ ⟨*действие*⟩ ⟨*план-действий*⟩`)` модифицирует план, добавляя указанную процедуру-действие, которую нужно запустить в указанное время.
- `(current-time` ⟨*план-действий*⟩`)` возвращает текущее время модели.

Экземпляр плана, которым мы будем пользоваться, будет обозначаться `the-agenda`. Процедура `after-delay` добавляет новый элемент в план `the-agenda`:

```
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda))
```

Имитация управляется процедурой `propagate`, которая работает с `theagenda`, по очереди выполняяпроцедуры, содержащиеся в плане. В общем случае, при работе модели в план добавляются новые элементы, а `propagate` продолжает работу, пока план не становится пустым:

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

Пример работы модели

Следующая процедура, которая навешивает на провод «тестер», показывает имитационную модель в действии. Тестер говорит проводу, что, каждый раз, когда сигнал изменяет значение, нужно напечатать новое значение сигнала, а также текущее время и имя провода:

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (newline)
                 (display name)
                 (display " ")
                 (display (current-time the-agenda))
                 (display "  New-value = ")
                 (display (get-signal wire)))))
```

Сначала мы инициализируем план действий и указываем задержки для элементарных функциональных элементов:

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

Затем мы создаем четыре провода и к двум из них подсоединяем тестеры:

```
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))

(probe 'sum sum)
sum 0 New-value = 0

(probe 'carry carry)
carry 0 New-value = 0
```

Затем мы связываем провода, образуя схему полусумматора (как на рис. 3.25), устанавливаем сигнал на входе `input-1` в 1, и запускаем модель:

```
(half-adder input-1 input-2 sum carry)
ok

(set-signal! input-1 1)
done

(propagate)
sum 8 New-value = 1
done
```

Сигнал `sum` становится 1 в момент времени 8. Мы находимся в 8 единицах от начала работы модели. В этот момент мы можем установить сигнал на входе `input-2` в 1 и дать изменению распространиться:

```
(set-signal! input-2 1)
done

(propagate)
carry 11 New-value = 1
sum 16 New-value = 0
done
```

Сигнал `carry` становится равным 1 в момент 11, а `sum` становится 0 в момент 16.

Упражнение 3.31.
Внутренняя процедура `accept-action-procedure!`, определенная в `make-wire`, требует, чтобы в момент, когда процедура-действие добавляется к проводу, она немедленно исполнялась. Объясните, зачем требуется такая инициализация. В частности, проследите работу процедуры `half-adder` из этого текста и скажите, как отличалась бы реакция системы, если бы `accept-action-procedure!` была определена как

```
(define (accept-action-procedure! proc)
  (set! action-procedures (cons proc action-procedures)))
```

Реализация плана действий

Наконец, мы описываем детали структуры данных плана действий, которая хранит процедуры, предназначенные для исполнения в будущем.

План состоит из *временных отрезков* (time segments). Каждый временной отрезок является парой, состоящей из числа (значения времени) и очереди (см. упражнение 3.32), которая содержит процедуры, предназначенные к исполнению в этот временной отрезок.

```
(define (make-time-segment time queue)
  (cons time queue))

(define (segment-time s) (car s))

(define (segment-queue s) (cdr s))
```

Мы будем работать с очередями временных отрезков при помощи операций, описанных в разделе 3.3.2.

Сам по себе план действий является одномерной таблицей временных отрезков. От таблиц, описанных в разделе 3.3.3, он отличается тем, что сегменты отсортированы в порядке возрастания времени. В дополнение к этому мы храним *текущее время* (current time) (т. е. время последнего исполненного действия) в голове плана. Свежесозданный план не содержит временных отрезков, а его текущее время равно 0[28]:

[28] Подобно таблицам из раздела 3.3.3, план действий — это список с заголовком, но, поскольку в заголовке хранится время, не нужно дополнительного заголовка-пустышки (вроде символа `*table*`, которым мы пользовались в таблицах).

```
(define (make-agenda) (list 0))

(define (current-time agenda) (car agenda))

(define (set-current-time! agenda time)
  (set-car! agenda time))

(define (segments agenda) (cdr agenda))

(define (set-segments! agenda segments)
  (set-cdr! agenda segments))

(define (first-segment agenda) (car (segments agenda)))

(define (rest-segments agenda) (cdr (segments agenda)))
```

План пуст, если в нем нет ни одного временного отрезка:

```
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

Для того, чтобы добавить в план новое действие, прежде всего мы проверяем, не пуст ли он. Если пуст, мы создаем для действия новый отрезок и вставляем его в план. Иначе мы просматриваем план, глядя на времена отрезков. Если мы находим отрезок с назначенным временем, мы добавляем действие к соответствующей очереди. Если же мы обнаруживаем время, большее, чем назначенное, мы вставляем новый отрезок перед текущим. Если мы доходим до конца плана, мы вставляем новый отрезок в конец.

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments))
                       action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr!
               segments
               (cons (make-new-time-segment time action)
                     (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments!
```

```
        agenda
        (cons (make-new-time-segment time action)
              segments))
       (add-to-segments! segments))))
```

Процедура, которая убирает из плана первый элемент, уничтожает элемент в начале очереди первого отрезка времени. Если в результате отрезок становится пустым, мы изымаем его из списка отрезков[29]:

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

Первый элемент плана находится в начале очереди в первом временном отрезке. Каждый раз, когда мы обращаемся к такому элементу, мы обновляем текущее время[30].

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "План пуст -- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

Упражнение 3.32.
Процедуры, предназначенные к выполнению в каждом временном отрезке, хранятся в виде очереди. Таким образом, процедуры для каждого отрезка вызываются в том же порядке, в котором они были добавлены к плану (первый пришел, первый ушел). Объясните, почему требуется использовать именно такой порядок. В частности, проследите поведение И-элемента, входы которого меняются с 0 на 1 и с 1 на 0 одновременно и скажите, как отличалось бы поведение, если бы мы хранили процедуры отрезка в обыкновенном списке, добавляя и убирая их только с головы (последний пришел, первый ушел).

3.3.5. Распространение ограничений

Традиционно компьютерные программы организованы как однонаправленные вычисления, выполняющие вычисления над указанными аргументами и получающие указанные значения. С другой стороны, часто системы приходится моделировать в виде отношений между величинами. Например, математическая модель механической структуры

[29]Обратите внимание, что в этой процедуре выражение `if` не имеет ⟨*альтернативы*⟩. Такие «односторонние предложения `if`» используются, когда требуется решить, нужно ли какое-то действие, а не выбрать одно из двух выражений. Если предикат ложен, а ⟨*альтернатива*⟩ отсутствует, значение предложения `if` не определено.

[30]Таким образом, текущее время всегда будет совпадать с временем последнего обработанного действия. Благодаря тому, что это время хранится в голове плана, оно всегда доступно, даже если соответствующий отрезок времени был уничтожен.

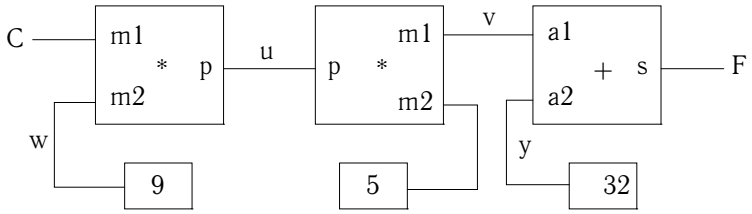


Рис. 3.28. Уравнение $9C = 5(F - 32)$, выраженное в виде сети ограничений.

может включать информацию, что деформация d металлического стержня связана уравнением $dAE = FL$ с приложенной к нему силой F, его длиной L, поперечным сечением A и модулем упругости E. Такое уравнение не является однонаправленным. Имея любые четыре величины, мы можем вычислить пятую. Однако при переводе уравнения на традиционный компьютерный язык нам придется выбрать величину, которая вычисляется на основе остальных четырех, так что процедура для вычисления площади A не может быть использована для вычисления деформации d, хотя вычисление A и d основаны на одном и том же уравнении[31].

В этом разделе мы набросаем эскиз языка, который позволит нам работать в терминах самих отношений. Минимальными составляющими этого языка будут служить *элементарные ограничения* (primitive constraints), которые говорят, что между величинами существуют определенные связи. Например, `(adder a b c)` означает, что величины a, b и c должны быть связаны уравнением $a + b = c$, `(multiplier x y z)` выражает ограничение $xy = z$, а `(constant 3.14 x)` говорит, что значение x обязано равняться 3.14.

Наш язык предоставляет средства комбинирования элементарных ограничений, чтобы с их помощью выражать более сложные отношения. Сочетания образуют *сети ограничений* (constraint networks), в которых ограничения связаны *соединителями* (connectors). Соединитель — это объект, который «содержит» значение, способное участвовать в одном или нескольких ограничениях. К примеру, мы знаем, что связь между температурами по Цельсию и по Фаренгейту выглядит как $9C = 5(F - 32)$ Такое ограничение можно изобразить в виде сети, состоящей из элементарных ограничений — сумматора, умножителей и констант (рисунок 3.28). На этом рисунке слева мы видим блок умножителя с тремя выводами, обозначенными $m1$, $m2$ и p. Вывод $m1$ присоединен к соединителю C, который будет хранить температуру по Цельсию. Вывод $m2$ присоединен к соединителю w, который, кроме того, связан с блоком-константой, содержащим 9. Вывод p, про который блок-умножитель говорит, что он должен быть произведением $m1$ и $m2$, связан с выводом p другого блока-умножителя, чей вывод $m2$ связан с константой 5, а $m1$ присоединен к одному из слагаемых суммы.

[31] Распространение ограничений появилось в системе SKETCHPAD Айвена Сазерленда (Sutherland 1963), невероятно опередившей свое время. Изящная система распространения ограничений, основанная на языке Smalltalk, была разработана Аланом Борнингом (Borning 1977) в исследовательском центре компании Xerox в Пало Альто. Сассман, Столлман и Стил применили распространение ограничений к анализу электрических цепей (Sussman and Stallman 1975; Sussman and Steele 1980). TK!Solver (Konopasek and Jayaraman 1984) представляет собой богатую среду моделирования, основанную на ограничениях.

Вычисления в такой сети происходят следующим образом: когда соединителю дается значение (пользователем либо блоком-ограничением, с которым он связан), соединитель пробуждает все связанные с ним ограничения (кроме того, которое само его пробудило), и сообщает им, что у него появилось значение. Каждый пробужденный блок-ограничение опрашивает свои выводы, чтобы определить, достаточно ли у него информации, чтобы найти значение для какого-нибудь еще соединителя. Если да, блок присваивает соединителю значение, и тогда уже он пробуждает связанные с ним ограничения, и так далее. Например, при преобразовании между градусами Цельсия и Фаренгейта, значения w, x и y сразу устанавливаются блоками-константами соответственно в 9, 5 и 32. Соединители пробуждают умножители и сумматор, которые убеждаются, что у них не хватает информации, чтобы продолжить. Если пользователь (или какая-то другая часть сети) установит значение C в 25, пробудится левый умножитель, и сделает u равным $25 \cdot 9 = 225$. Затем u разбудит второй умножитель, который присвоит v значение 45, а v разбудит сумматор, и тот сделает значение F равным 77.

Использование системы ограничений

Чтобы при помощи системы ограничений провести вышеописанное вычисление, сначала мы порождаем два соединителя, `C` и `F`, вызовами конструктора `make-connector`, и связываем `C` и `F` в требуемую нам сеть:

```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F)
ok
```

Процедура, создающая сеть, определяется так:

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

Эта процедура порождает внутренние соединители `u`, `v`, `w`, `x` и `y`, а затем связывает их, как показано на рис. 3.28, при помощи элементарных ограничений `adder`, `multiplier` и `constant`. Как и при моделировании цифровых схем в разделе 3.3.4, способность выражать комбинации базовых элементов в виде процедур автоматически сообщает нашему языку средство абстракции для составных объектов.

Чтобы наблюдать сеть в действии, мы подсоединим тестеры к соединителям `C` и `F` при помощи процедуры `probe`, подобной той, которая следила за сигналами в проводах

в разделе 3.3.4. Установка тестера на соединителе ведет к тому, что каждый раз, когда он получает значение, печатается сообщение:

```
(probe "по Цельсию" C)
(probe "по Фаренгейту" F)
```

Затем мы присваиваем значение 25 соединителю C. (Третий аргумент процедуры set-value! сообщает C, что директива исходит от пользователя.)

```
(set-value! C 25 'user)
Тестер: по Цельсию = 25
Тестер: по Фаренгейту = 77
done
```

Тестер на C просыпается и печатает значение. Кроме того, C распространяет значение по сети, как описано выше. В результате F становится равным 77, и тестер на F об этом сообщает.

Теперь можно попробовать присвоить F новое значение, скажем, 212:

```
(set-value! F 212 'user)
Ошибка! Противоречие (77 212)
```

Соединитель жалуется, что обнаружил противоречие: его значение равно 77, а при этом кто-то пытается установить его в 212. Если мы и вправду хотим снова воспользоваться сетью с новыми значениями, можно попросить C забыть свое старое значение:

```
(forget-value! C 'user)
Тестер: по Цельсию = ?
Тестер: по Фаренгейту = ?
done
```

C видит, что user, который изначально присвоил ему значение, отменяет его, так что C соглашается потерять значение, как показывает тестер, и информирует об этом остальную сеть. Эта информация в конце концов добирается до F, и у F уже не остается причин считать, что его значение равно 77. Так что F тоже теряет значение, и тестер это отображает.

Теперь, когда у F больше нет значения, мы можем установить его в 212:

```
(set-value! F 212 'user)
Тестер: по Фаренгейту = 212
Тестер: по Цельсию = 100
done
```

Это новое значение, распространяясь по сети, заставляет C получить значение 100, и тестер на C это регистрирует. Заметим, что одна и та же сеть используется и для того, чтобы на основе F получить C и для того, чтобы на основе C получить F. Эта ненаправленность вычислений является отличительной чертой систем, основанных на ограничениях.

Реализация системы ограничений

Система ограничений реализована на основе процедурных объектов с внутренним состоянием, очень похоже на модель цифровых схем из раздела 3.3.4. Хотя базовые объекты системы с ограничениями несколько более сложны, система в целом проще за счет того, что незачем заботиться о планах действий и логических задержках.

Базовые операции над соединителями таковы:

- `(has-value?` ⟨*соединитель*⟩)сообщает, есть ли у соединителя значение.
- `(get-value` ⟨*соединитель*⟩)возвращает текущее значение соединителя.
- `(set-value!` ⟨*соединитель*⟩ ⟨*новое-знач*⟩ ⟨*информант*⟩) сообщает соединителю, что информант требует установить в нем новое значение.
- `(forget-value!` ⟨*соединитель*⟩ ⟨*отказник*⟩) сообщает соединителю, что отказник просит его забыть значение.
- `(connect` ⟨*соединитель*⟩ ⟨*новое-огр*⟩) говорит соединителю, что он участвует в новом ограничении.

Соединители общаются с ограничениями при помощи процедур `inform-about-value`, которая говорит ограничению, что у соединителя есть значение, и `inform-about-no-value`, которая сообщает ограничению, что соединитель утратил значение.

`Adder` порождает ограничение-сумматор между соединителями-слагаемыми `a1` и `a2` исоединителем-суммой `sum`. Сумматор реализован в виде процедуры с внутренним состоянием (процедура `me`):

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2))
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
                       (- (get-value sum) (get-value a1))
                       me))
          ((and (has-value? a2) (has-value? sum))
           (set-value! a1
                       (- (get-value sum) (get-value a2))
                       me))))
  (define (process-forget-value)
    (forget-value! sum me)
    (forget-value! a1 me)
    (forget-value! a2 me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
```

```
          (else
           (error "Неизвестный запрос -- ADDER" request))))
  (connect a1 me)
  (connect a2 me)
  (connect sum me)
  me)
```

`Adder` связывает новый сумматор с указанными соединителями и возвращает его в качестве значения. Процедура `me`, которая представляет сумматор, работает как диспетчер для внутренних процедур. Для доступа к диспетчеру используются следующие «синтаксические интерфейсы» (см. примечание 27 в разделе 3.3.4):

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))

(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

Внутренняя процедура сумматора `process-new-value` вызывается, когда сумматору сообщают, что один из его соединителей получил значение. Сумматор проверяет, имеют ли значения одновременно `a1` и `a2`. Если да, то он говорит `sum`, чтобы тот установил значение в сумму двух слагаемых. Аргумент `informant` процедуры `set-value!` равен `me`, то есть самому объекту-сумматору. Если неверно, что и `a1` и `a2` имеют значения, то сумматор проверяет, имеют ли одновременно значения `a1` и `sum`. Если да, то он устанавливает `a2` в их разность. Наконец, если значения есть у `a2` и `sum`, это дает сумматору достаточно информации, чтобы установить `a1`. Если сумматору сообщают, что один из соединителей потерял значение, то он просит все свои соединители избавиться от значений. (На самом деле будут отброшены только значения, установленные самим сумматором.) Затем он зовет `process-new-value`. Смысл этого последнего шага в том, что один или более соединителей по-прежнему могут обладать значением (то есть, у соединителя могло быть значение, не установленное сумматором), и эти значения может быть необходимо распространить через сумматор.

Умножитель очень похож на сумматор. Он устанавливает свой вывод `product` в 0, если хотя бы один множитель равен 0, даже в том случае, когда второй множитель неизвестен.

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                       (* (get-value m1) (get-value m2))
                       me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2
                       (/ (get-value product) (get-value m1))
                       me))
```

```
          ((and (has-value? product) (has-value? m2))
           (set-value! m1
                       (/ (get-value product) (get-value m2))
                       me))))
  (define (process-forget-value)
    (forget-value! product me)
    (forget-value! m1 me)
    (forget-value! m2 me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error "Неизвестный запрос -- MULTIPLIER" request))))
  (connect m1 me)
  (connect m2 me)
  (connect product me)
  me)
```

Конструктор `constant` просто устанавливает значение указанного соединителя. Сообщение `I-have-a-value` либо `I-lost-my-value`, посланные блоку-константе, приводят к ошибке.

```
(define (constant value connector)
  (define (me request)
    (error "Неизвестный запрос -- CONSTANT" request))
  (connect connector me)
  (set-value! connector value me)
  me)
```

Наконец, тестер печатает сообщение о присваивании или потере значения в указанном соединителе:

```
(define (probe name connector)
  (define (print-probe value)
    (newline)
    (display "Тестер: ")
    (display name)
    (display " = ")
    (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value)
    (print-probe "?"))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
```

```
          (process-forget-value))
         (else
          (error "Неизвестный запрос -- PROBE" request))))
  (connect connector me)
  me)
```

Представление соединителей

Соединитель представляется в виде процедурного объекта с внутренними переменными состояния: `value`, значение соединителя; `informant`, объект, который установил значение соединителя; и `constraints`, множество ограничений, в которых участвует соединитель.

```
(define (make-connector)
  (let ((value false) (informant false) (constraints '()))
    (define (set-my-value newval setter)
      (cond ((not (has-value? me))
             (set! value newval)
             (set! informant setter)
             (for-each-except setter
                              inform-about-value
                              constraints))
            ((not (= value newval))
             (error "Противоречие" (list value newval)))
            (else 'ignored)))
    (define (forget-my-value retractor)
      (if (eq? retractor informant)
          (begin (set! informant false)
                 (for-each-except retractor
                                  inform-about-no-value
                                  constraints))
          'ignored))
    (define (connect new-constraint)
      (if (not (memq new-constraint constraints))
          (set! constraints
                (cons new-constraint constraints)))
      (if (has-value? me)
          (inform-about-value new-constraint))
      'done)
    (define (me request)
      (cond ((eq? request 'has-value?)
             (if informant true false))
            ((eq? request 'value) value)
            ((eq? request 'set-value!) set-my-value)
            ((eq? request 'forget) forget-my-value)
            ((eq? request 'connect) connect)
            (else (error "Неизвестная операция -- CONNECTOR"
                         request))))
    me))
```

Внутренняя процедура соединителя `set-my-value` зовется, когда поступает требование установить значение соединителя. Если у соединителя нет текущего значения, он его устанавливает и запоминает ограничение, которое потребовало установки значения, в переменной `informant`[32]. Затем соединитель оповещает все связанные с ним ограничения, кроме того, которое потребовало установить значение. Это проделывается с помощью следующего итератора, который применяет указанную процедуру ко всем элементам списка, кроме одного.

```
(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items)))))
  (loop list))
```

Если от соединителя требуют забыть значение, он запускает внутреннюю процедуру `forget-my-value`, которая первым делом убеждается, что запрос исходит от того же самого объекта, который значение установил. Если это так, соединитель оповещает связанные с ним ограничения о потере значения.

Внутренняя процедура `connect` добавляет указанное ограничение к списку ограничений, если его там еще нет. Затем, если у соединителя есть значение, он сообщает об этом ограничению.

Процедура соединителя `me` служит диспетчером для остальных внутренних процедур, а кроме того, представляет соединитель как объект. Следующие процедуры предоставляют синтаксический интерфейс к диспетчеру:

```
(define (has-value? connector)
  (connector 'has-value?))

(define (get-value connector)
  (connector 'value))

(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))

(define (forget-value! connector retractor)
  ((connector 'forget) retractor))

(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

Упражнение 3.33.
С помощью элементарных ограничений сумматор, умножитель и константа, определите процедуру `averager` (усреднитель), которая принимает три соединителя `a`, `b` и `c`, и обеспечивает условие, что значение `c` равно среднему арифметическому значений `a` и `b`.

[32] `Setter` может и не быть ограничением. В примере с температурой мы использовали символ `user` в качестве значения `setter`.

Упражнение 3.34.
Хьюго Дум хочет построить квадратор, блок-ограничение с двумя выводами, такое, что значение соединителя `b` на втором выводе всегда будет равно квадрату значения соединителя `a` на первом выводе. Он предлагает следующее простое устройство на основе умножителя:

```
(define (squarer a b)
  (multiplier a a b))
```

В такой идее есть существенная ошибка. Объясните ее.

Упражнение 3.35.
Бен Битобор объясняет Хьюго, что один из способов избежать неприятностей в упражнении 3.34 — определить квадратор как новое элементарное ограничение. Заполните недостающие части в Беновой схеме процедуры, реализующей такое ограничение:

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "квадрат меньше 0 -- SQUARER" (get-value b))
            ⟨альтернатива1⟩)
        ⟨альтернатива2⟩))
  (define (process-forget-value) ⟨тело1⟩)
  (define (me request) ⟨тело2⟩)
  ⟨остаток определения⟩
  me)
```

Упражнение 3.36.
Допустим, что мы выполняем следующую последовательность действий в глобальном окружении:

```
(define a (make-connector))
(define b (make-connector))
(set-value! a 10 'user)
```

В какой-то момент при вычислении `set-value!` будет выполнено следующее выражение из внутренней процедуры соединителя:

```
(for-each-except setter inform-about-value constraints)
```

Нарисуйте диаграмму, изображающую окружение, в котором выполняется указанное выражение.

Упражнение 3.37.
Процедура `celsius-fahrenheit-converter` выглядит громоздко по сравнению со стилем определения в формате выражения:

```
(define (celsius-fahrenheit-converter x)
  (c+ (c* (c/ (cv 9) (cv 5))
          x)
      (cv 32)))

(define C (make-connector))
(define F (celsius-fahrenheit-converter C))
```

Здесь c+, c* и т. п. — «ограничительные» версии арифметических операций. Например, c+ берет в виде аргументов два соединителя, и возвращает соединитель, который связан с ними ограничением-сумматором:

```
(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))
```

Определите аналогичные процедуры для c-, c*, c/ и cv (константа), так, чтобы можно было определять составные ограничения, как в вышеприведенном примере[33].

3.4. Параллелизм: время имеет значение

Мы убедились в мощности вычислительных объектов с внутренним состоянием в качестве инструмента моделирования. Однако, как было сказано в разделе 3.1.3, за эту мощность приходится платить потерей референциальной прозрачности, которая ведет в дебри вопросов об идентичности и изменении, и необходимостью замены подстановочной модели вычислений на более сложную модель с окружениями.

Главная проблема, стоящая за сложностями состояния, идентичности и изменения, состоит в том, что, введя присваивание, мы вынуждены внести в свои вычислительные модели понятие *времени* (time). До того, как появилось присваивание, наши программы от времени не зависели — в том смысле, что всякое выражение, обладающее значением, всегда имело одно и то же значение. Вспомним, однако, пример со снятием денег со счета и просмотром получившегося баланса из начала раздела 3.1.1:

[33] Представление в виде выражений удобно, потому что при этом отпадает необходимость давать имена промежуточным выражениям в вычислении. Наша исходная формулировка языка ограничений громоздка по той же причине, по которой многие языки оказываются громоздкими при работе с составными данными. Например, если нам нужно вычислить произведение $(a+b)\cdot(c+d)$, где переменные представляют вектора, мы можем работать в «императивном» стиле, с процедурами, которые присваивают значения указанным векторным аргументам, но сами не возвращают вектора как значения:

```
(v-sum a b temp1)
(v-sum c d temp2)
(v-prod temp1 temp2 answer)
```

С другой стороны, мы можем работать с выражениями, используя процедуры, которые возвращают вектора как значения, и таким образом избежать прямого упоминания temp1 и temp2:

```
(define answer (v-prod (v-sum a b) (v-sum c d)))
```

Поскольку Лисп позволяет возвращать составные объекты как результаты процедур, мы можем преобразовать свой императивный язык ограничений в язык на основе выражений, как показано в этом упражнении. В языках, где средства работы с составными объектами бедны, как в Алголе, Бейсике и Паскале (если явно не использовать паскалевские переменные-указатели), обычно при решении таких задач программист ограничен императивным стилем. Поскольку формат выражений предпочтителен, можно спросить, есть ли причина строить систему императивно, как мы поступили в этом разделе. Одна из причин состоит в том, что язык ограничений, не ориентированный на выражения, дает нам возможность работать не только с объектами-соединителями, но и с объектами-ограничениями (например, значением, порождаемым процедурой adder). Это будет полезно, если мы захотим расширить систему новыми операциями, которые работают с ограничениями напрямую, а не только косвенным образом через операции над соединителями. Хотя реализовать работу с выражениями на основе императивной реализации просто, сделать обратное значительно труднее.

```
(withdraw 25)
75

(withdraw 25)
50
```

Здесь последовательное вычисление одного и того же выражения приводит к различным результатам. Такое поведение возникает из-за того, что выполнение предложений присваивания (в данном случае присваивания переменной `balance`) отмечает *моменты времени* (moments in time), когда значения меняются. Результат вычисления выражения зависит не только от самого выражения, но и от того, происходит ли вычисление до или после таких моментов. Построение моделей в терминах вычислительных объектов с внутренним состоянием заставляет нас рассматривать время как существенное для программирования понятие.

Можно пойти еще дальше в структурировании наших вычислительных объектов, чтобы точнее отразить наше восприятие физического мира. Объекты мира изменяются не последовательно один за другим. Мы воспринимаем их как действующие *параллельно* (concurrently) — все вместе. Так что зачастую бывает естественно моделировать системы как сообщества вычислительных процессов, работающих параллельно. Точно так же, как можно сделать программы модульными, организуя их в виде объектов с раздельным внутренним состоянием, часто имеет смысл разделять вычислительные модели на части, вычисляющиеся раздельно и одновременно. Даже если на самом деле предполагается выполнять программы на последовательном компьютере, практика написания программ так, как будто вычисление будет параллельным, заставляет программиста избегать несущественных временны́х ограничений, и таким образом повышает модульность программ.

Параллельное вычисление не только делает программы модульнее, оно к тому же может дать выигрыш в скорости перед последовательным. Последовательные компьютеры выполняют только одну операцию за раз, так что время, необходимое для решения задачи, пропорционально общему количеству выполняемых операций[34]. Однако если возможно разбить задачу на части, которые относительно независимы друг от друга и должны общаться между собой редко, может оказаться возможным раздать эти куски отдельным вычисляющим процессорам и получить выигрыш, пропорциональный числу имеющихся процессоров.

К несчастью, проблемы, связанные с присваиванием, становятся только тяжелее в присутствии параллелизма. Связано ли это с тем, что параллельно работает мир, или компьютер, но явление одновременных вычислений привносит дополнительную сложность в наше понимание времени.

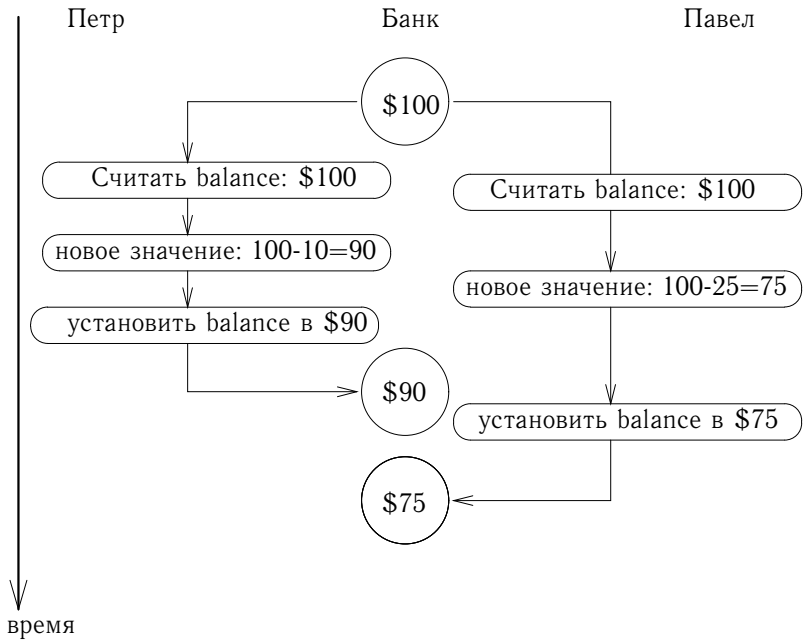


Рис. 3.29. Временная диаграмма, показывающая, как чередование действий при двух операциях со счетом может привести к неправильному балансу.

3.4.1. Природа времени в параллельных системах

На первый взгляд, время — вещь простая. Это порядок, накладываемый на события[35]. Для всяких двух событий A и B, либо A случается раньше B, либо A и B происходят одновременно, либо A случается позже B. Например, возвращаясь к примеру с банковским счетом, пусть Петр берет с общего счета 10 долларов, а Павел 25, притом, что сначала на счету 100 долларов. На счету останется 65 долларов. В зависимости от порядка двух событий, последовательность балансов на счету будет либо $100 → $90 → $65, либо $100 → $75 → $65. В компьютерной реализации банковской системы эта изменяющаяся последовательность балансов может моделироваться через последовательные присваивания переменной `balance`.

Однако в некоторых ситуациях такой взгляд может вести к проблемам. Допустим, что Петр и Павел, и еще другие люди помимо них, имеют доступ к совместному банковскому счету через сеть банкоматов, разбросанных по всему миру. Последовательность значений баланса будет критическим образом зависеть от точной хронологии доступа и деталей коммуникации между машинами.

Неопределенность порядка событий может приводить к серьезным проблемам в проектировании компьютерных систем. Например, предположим, что действия Петра и Павла реализованы как два отдельных процесса с общей переменной `balance`, и что каждый процесс определяется процедурой из раздела 3.1.1:

```
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Недостаточно денег на счете"))
```

Если два процесса работают одновременно, то Петр может проверить баланс и попытаться снять разрешенную сумму. Однако за промежуток времени между моментами, когда Петр проверяет баланс, и когда он завершает снятие денег, Павел может снять какую-то сумму и сделать результат Петровой проверки несостоятельным.

И это еще не самое худшее. Рассмотрим выражение

```
(set! balance (- balance amount))
```

которое выполняется во время каждого снятия денег. Выполнение происходит в три шага: (1) считывание значения переменной `balance`; (2) вычисление нового значения баланса; (3) присвоение переменной `balance` этого нового значения. Если процессы Петра и Павла выполняют это предложение параллельно, то в двух процессах снятия денег порядок чтения переменной `balance` и присваивания могут чередоваться.

Временна́я диаграмма на рисунке 3.29 показывает порядок событий, при котором `balance` сначала равен 100. Петр берет 10, Павел 25, и однако в итоге `balance` оказывается равен 75. Как показано на диаграмме, причина аномалии состоит в том,

[34]На самом деле большинство процессоров выполняют несколько операций за раз, используя стратегию, называемую *конвейеризация* (pipelining). Хотя этот метод значительно повышает степень использования аппаратных ресурсов, он используется только для ускорения выполнения последовательного потока вычислений, сохраняя поведение последовательной программы.

[35]Граффити на одной стене в Кембридже: «Время — это устройство для того, чтобы случалось не все сразу».

что у Павла присваивание переменной значения 75 основано на предположении, что значение `balance`, которое надо уменьшить, равно 100. Однако это предположение стало неверным, когда Петр сделал `balance` равным 90. Для банковской системы это катастрофическая ошибка, так как не сохраняется общее количество денег в системе. До транзакций общая сумма была 100 долларов. После же у Петра оказывается 10 долларов, у Павла 25, и у банка 75[36].

Общее явление, иллюстрируемое здесь, состоит в том, что различные процессы могут разделять одну и ту же переменную состояния. Сложность возникает оттого, что с этой переменной в одно и то же время может пытаться работать более одного процесса. В примере с банковским счетом во время каждой транзакции клиент должен иметь возможность действовать так, как будто остальных клиентов не существует. Когда клиент изменяет баланс, исходя из его предыдущего значения, ему надо обеспечить гарантии того, что прямо перед моментом изменения баланс все еще соответствует его, клиента, представлениям.

Правильное поведение параллельных программ

Вышеприведенный пример демонстрирует типичную неочевидную ошибку, которая может возникнуть в параллельной программе. Сложность здесь восходит к присваиванию переменным, разделяемым между различными процессами. Мы уже знаем, что при работе с `set!` требуется осторожность, потому что результаты вычислений зависят от порядка, в котором происходят присваивания[37]. При наличии параллелизма нужно быть острожным вдвойне, поскольку не всегда можно управлять порядком, в котором присваивания происходят в разных процессах. Если несколько таких изменений могут происходить одновременно (как в случае с двумя вкладчиками, имеющими доступ к общему счету), нам требуется способ обеспечить правильную работу системы. Например, в случае со снятием денег с общего счета, мы должны сделать так, чтобы общее количество денег оставалось неизменным. Чтобы заставить параллельные программы работать корректно, иногда требуется наложить некоторые ограничения на одновременное исполнение.

Одно из возможных ограничений на параллелизм может состоять в том, что никакие две операции, способные изменить разделяемые переменные состояния, не могут исполняться одновременно. Это очень серьезное ограничение. Для распределенной банковской системы это означало бы, что проектировщик системы должен сделать так, что в каждый момент происходит не более одной транзакции. Это требование чрезмерно консервативное и ведет к неэффективности. На рисунке 3.30 показан случай с совместным счетом Петра и Павла, причем у Павла есть еще и

[36] Еще худшая ошибка могла бы случиться, если бы две операции `set!` попытались одновременно изменить баланс. В результате содержимое памяти могло бы стать случайной комбинацией данных, записанных двумя процессами. В большинство компьютеров встроена блокировка элементарных операций записи в память, которая предохраняет от такого одновременного доступа. Однако даже такой, казалось бы, простой метод защиты придает дополнительную сложность проектированию многопроцессорных компьютеров, где требуются сложные протоколы *согласованности кэша* (cache coherence), чтобы у разных процессоров были непротиворечивые точки зрения на содержимое памяти, при том, что данные могут дублироваться («кэшироваться») в разных процессорах, чтобы увеличить скорость доступа к памяти.

[37] Программа подсчета факториала из раздела 3.1.3 демонстрирует это в рамках одного последовательного процесса.

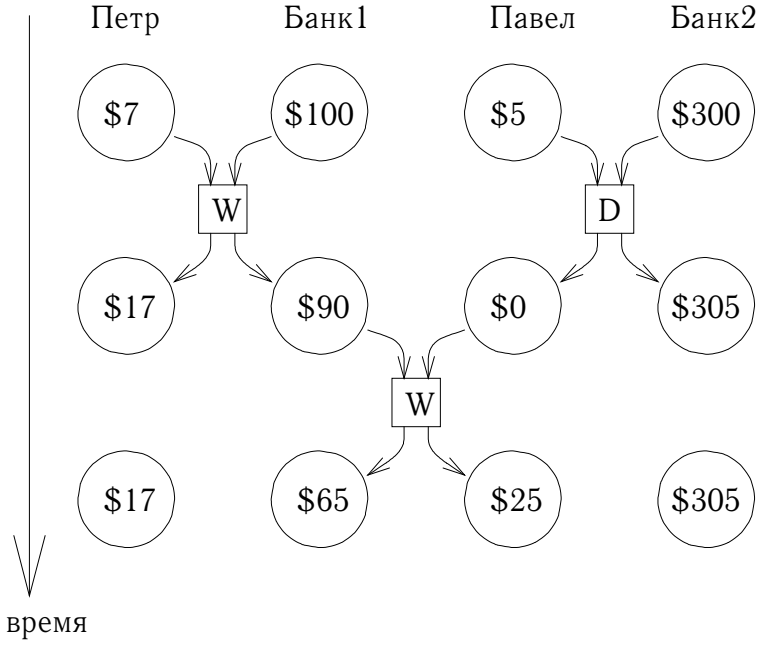


Рис. 3.30. Одновременные операции при работе с совместным счетом в Банке 1 и личным счетом в Банке 2.

собственный счет. Диаграмма показывает две операции снятия денег с совместного счета (одну проводит Петр, одну Павел), а также занесение Павлом денег на личный счет[38]. Два снятия денег с одного счета не должны происходить одновременно (поскольку оба работают с одним счетом), и Павел не может одновременно снять деньги и занести их в банк (поскольку и та, и другая операция касаются кошелька Павла). Однако не должно быть препятствий, мешающих Павлу заносить деньги на личный счет в то время, как Петр берет деньги с общего счета.

Менее драконовское ограничение на параллелизм могло бы состоять в том, чтобы параллельная система выдавала такие же результаты, как если бы процессы происходили последовательно. У этого ограничения две важных стороны. Во-первых, от процессов на самом деле не требуется последовательного исполнения, а только результаты, совпадающие с теми, которые *получались бы*, если бы они работали один за другим. В примере на рис. 3.30, проектировщик банковской системы спокойно может разрешить одновременное занесение денег Павлом и снятие их Петром, поскольку общий результат будет таков, как будто бы они шли последовательно. Во-вторых, у параллельной программы может быть более одного «правильного» результата, потому что мы требуем только, что-

[38]По столбцам: содержимое кошелька Петра, общий счет (в Банке 1), кошелек Павла и личный счет Павла (в Банке 2), до и после каждого снятия (W) и занесения денег на счет (D). Петр берет 10 долларов из Банка 1; Павел кладет 5 долларов в Банк 2, затем берет 25 долларов из Банка 1.

бы он совпадал с результатом при *каком-нибудь* последовательном порядке. Например, предположим, что общий счет Петра и Павла вначале равен 100 долларам, Петр кладет на него 40 долларов, а Павел снимает половину имеющихся там денег. При этом последовательное исполнение может привести к значению на счету либо в 70, либо в 90 долларов (см. упражнение 3.38)[39].

Можно найти и еще более слабые требования для корректного выполнения параллельных программ. Программа, имитирующая диффузию (например, поток тепла в объекте), может состоять из большого числа процессов, каждый из которых изображает маленький участок пространства, и которые параллельно обновляют свои значения. Каждый процесс в цикле изменяет свое значение на среднее между своим собственным значением и значениями соседей. Этот алгоритм сходится к правильному ответу независимо от порядка, в котором выполняются операции; нет никакой нужды в ограничениях на параллельное использование разделяемых значений.

Упражнение 3.38.
Пусть Петр, Павел и Мария имеют общий счет, на котором вначале лежит 100 долларов. Петр кладет на счет 10 долларов, одновременно с этим Павел берет 20, а Мария берет половину денег со счета. При этом они выполняют следующие операции:

Петр: `(set! balance (+ balance 10))`
Павел: `(set! balance (- balance 20))`
Мария: `(set! balance (- balance (/ balance 2)))`

а. Перечислите возможные значения `balance` после завершения операций, предполагая, что банковская система требует от транзакций исполняться последовательно в каком-то порядке.

б. Назовите какие-нибудь другие значения, которые могли бы получиться, если бы система разрешала операциям чередоваться. Нарисуйте временные диаграммы, подобные рис. 3.29, чтобы объяснить, как возникают такие результаты.

3.4.2. Механизмы управления параллелизмом

Мы убедились, что сложность работы с параллельными процессами происходит из необходимости учитывать порядок чередования событий в различных процессах. Предположим, к примеру, что у нас есть два процесса, один с упорядоченными событиями (a, b, c), а другой с упорядоченными событиями (x, y, z). Если эти два процесса исполняются параллельно, без каких-либо дополнительных ограничений на чередование событий, то возможно 20 различных порядков событий, соблюдающих упорядочение их внутри каждого из процессов:

$$\begin{array}{llll}
(a,b,c,x,y,z) & (a,x,b,y,c,z) & (x,a,b,c,y,z) & (x,a,y,z,b,c) \\
(a,b,x,c,y,z) & (a,x,b,y,z,c) & (x,a,b,y,c,z) & (x,y,a,b,c,z) \\
(a,b,x,y,c,z) & (a,x,y,b,c,z) & (x,a,b,y,z,c) & (x,y,a,b,z,c) \\
(a,b,x,y,z,c) & (a,x,y,b,z,c) & (x,a,y,b,c,z) & (x,y,a,x,b,c) \\
(a,x,b,c,y,z) & (a,x,y,z,b,c) & (x,a,y,b,z,c) & (x,y,x,a,b,c)
\end{array}$$

[39]Более формально это утверждение можно выразить, сказав, что поведение параллельных программ — *недетерминированное* (nondeterministic). То есть, они описываются не функциями с одним значением, а функциями, чьи результаты являются множествами возможных значений. В разделе 4.3 мы рассмотрим язык для выражения недетерминистских вычислений.

При разработке этой системы нам как программистам пришлось бы рассматривать результаты каждого из этих 20 упорядочений и проверять, что каждое из них допустимо. С ростом числа процессов и событий такой подход быстро становится нереалистичным.

Более практичный подход к проектированию параллельных систем состоит в том, чтобы придумать общие механизмы, которые бы ограничивали чередование событий в параллельных процессах и тем самым давали нам уверенность, что поведение программы верно. Для этой цели было разработано большое количество механизмов. В этом разделе мы опишем один из них — *сериализатор* (serializer).

Сериализация доступа к разделяемой памяти

Идея сериализации заключается в следующем: процессы выполняются параллельно, но при этом существуют определенные группы процедур, которые не могут выполняться одновременно. Выражаясь точнее, сериализация порождает выделенные множества процедур, такие, что в каждом сериализованном множестве в любой момент может происходить выполнение только одной процедуры из множества. Если какая-то процедура из множества уже выполняется, то процесс, который пытается выполнить любую процедуру из множества, будет приостановлен до тех пор, пока не закончится текущее вычисление процедуры.

С помощью сериализации можно управлять доступом к разделяемым переменным. Например, если мы хотим присвоить разделяемой переменной значение, зависящее от ее текущего значения, мы помещаем доступ к прежнему значению и присваивание нового в одну процедуру. Затем мы помещаем все такие процедуры в один сериализатор и тем самым добиваемся того, что никакая другая процедура, которая присваивает значения этой переменной, не может выполняться одновременно с нашей. Это гарантирует нам, что значение переменной не может измениться в промежутке между доступом к ней и соответствующим ему присваиванием.

Сериализаторы в Scheme

Чтобы сделать это описание более конкретным, предположим, что мы расширили язык Scheme, добавив в него процедуру `parallel-execute`:

(`parallel-execute` $\langle p_1 \rangle$ $\langle p_2 \rangle$ ... $\langle p_k \rangle$)

Каждый из $\langle p \rangle$ должен быть процедурой без аргументов. `Parallel-execute` создает для каждого $\langle p \rangle$ отдельный процесс, который выполняет $\langle p \rangle$ (с пустым набором аргументов). Все эти процессы выполняются параллельно[40].

Чтобы продемонстрировать, как эта процедура используется, рассмотрим

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (+ x 1))))
```

[40] `Parallel-execute` не входит в стандартную Scheme, но такая процедура может быть реализована в MIT Scheme. В нашей реализации новые процессы выполняются параллельно еще и с исходным Scheme-процессом. Кроме того, в нашей реализации значение, которое возвращает `parallel-execute`, представляет собой специальный управляющий объект, с помощью которого можно остановить все новосозданные процессы.

Здесь создаются два параллельных процесса — P_1, который присваивает `x` значение `x` умножить на `x`, и P_2, который увеличивает `x` на единицу. После того, как вычисление закончено, `x` может иметь одно из пяти значений, в зависимости от чередования событий в P_1 и P_2:

- 101: P_1 делает `x` равным 100, затем P_2 его увеличивает.
- 121: P_2 увеличивает `x`, делая его равным 11, затем P_1 присваивает ему значение `x` умножить на `x`.
- 110: P_2 изменяет `x` с 10 на 11 в промежутке между двумя обращениями к `x` из P_1 во время вычисления `(* x x)`.
- 11: P_2 читает `x`, затем P_1 присваивает ему значение 100, затем P_1 пишет `x`
- 100: P_1 читает `x` (дважды), затем P_2 присваивает ему значение 11, затем P_1 записывает значение `x`.

Мы можем ограничить параллелизм, используя сериализованные процедуры, которые создаются *сериализаторами* (serializers). Сериализаторы порождаются процедурой `make-serializer`, реализация которой дана ниже. Сериализатор принимает в качестве аргумента процедуру, и возвращает сериализованную процедуру с таким же поведением. Все вызовы сериализатора порождают сериализованные процедуры, принадлежащие одному множеству.

Таким образом, в отличие от предыдущего примера, выполнение

```
(define x 10)
(define s (make-serializer))
(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (+ x 1)))))
```

может иметь только два результата, 101 и 121. Остальные возможности отбрасываются, поскольку выполнение P_1 и P_2 не может чередоваться.

Ниже приведена версия процедуры `make-account` из раздела 3.1.1, в которой помещение денег на счет и снятие их со счета сериализованы:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance) balance)
            (else (error "Неизвестный запрос -- MAKE-ACCOUNT"
                         m))))
    dispatch))
```

В такой реализации два процесса не могут параллельно помещать деньги на счет или снимать их. Таким образом устраняется источник ошибки, показанной на рис. 3.29, где Петр изменяет баланс на счете в промежутке между моментами, когда Павел считывает значение баланса, и когда он производит присваивание. С другой стороны, у каждого счета свой собственный сериализатор, так что операции с различными счетами могут происходить параллельно.

Упражнение 3.39.
Какие из пяти возможных исходов параллельного выполнения сохраняются, если мы сериализуем выполнение таким образом:

```
(define x 10)
(define s (make-serializer))
(parallel-execute (lambda () (set! x ((s (lambda () (* x x))))))
                  (s (lambda () (set! x (+ x 1)))))
```

Упражнение 3.40.
Укажите все возможные значения `x` при выполнении

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (* x x x))))
```

Какие из них сохраняются, если вместо этого мы выполняем сериализованные процедуры:

```
(define x 10)
(define s (make-serializer))
(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (* x x x)))))
```

Упражнение 3.41.
Бен Битобор считает, что лучше было бы реализовать банковский счет таким образом (измененная строка отмечена комментарием):

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance)
             ((protected (lambda () balance))))  ; сериализовано
            (else (error "Неизвестный запрос -- MAKE-ACCOUNT"
                         m))))
    dispatch))
```

поскольку несериализованный доступ к банковскому счету может привести к неправильному поведению. Вы согласны? Существует ли сценарий, который демонстрирует обоснованность беспокойства Бена?

Упражнение 3.42.
Бен Битобор говорит, что слишком расточительно в ответ на каждое сообщение `withdraw` и `deposit` создавать по новой сериализованной процедуре. Он говорит, что можно изменить `make-account` так, чтобы все вызовы `protected` происходили вне процедуры `dispatch`. Таким образом, счет будет возвращать одну и ту же сериализованную процедуру (созданную тогда же, когда и сам счет) каждый раз, когда у него просят процедуру снятия денег:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (let ((protected-withdraw (protected withdraw))
          (protected-deposit (protected deposit)))
      (define (dispatch m)
        (cond ((eq? m 'withdraw) protected-withdraw)
              ((eq? m 'deposit) protected-deposit)
              ((eq? m 'balance) balance)
              (else (error "Неизвестный запрос -- MAKE-ACCOUNT"
                           m))))
      dispatch)))
```

Безопасно ли такое изменение? В частности, есть ли разница в том, в каком порядке может происходить параллельное выполнение в этих двух версиях `make-account`?

Сложности при использовании множественных разделяемых ресурсов

Сериализаторы предоставляют нам мощную абстракцию, которая позволяет изолировать сложности выполнения параллельных программ, так что мы получаем возможность работать с ними аккуратно (и, будем надеяться, без ошибок). Однако, хотя при работе только с одним разделяемым ресурсом (например, с одним банковским счетом) использовать сериализаторы относительно просто, при наличии множественных разделяемых ресурсов параллельное программирование может быть предательски сложным.

Чтобы проиллюстрировать одну из ряда трудностей, которые могут возникнуть, предположим, что нам требуется поменять местами балансы на двух банковских счетах. Мы читаем каждый счет, чтобы узнать баланс, вычисляем разницу между балансами, снимаем ее с одного счета и кладем на другой. Это можно реализовать следующим образом[41]:

[41] Мы упростили `exchange`, пользуясь тем, что наше сообщение `deposit` может принимать отрицательные суммы. (Для банковской системы это серьезная ошибка!)

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                       (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

Эта процедура работает правильно в том случае, когда только один процесс пытается осуществить обмен. Допустим, однако, что Петр и Павел имеют доступ к совместным счетам $a1$, $a2$ и $a3$, и что Петр меняет местами $a1$ и $a2$, а Павел в то же время обменивает $a1$ и $a3$. Даже если снятие и занесение денег на отдельные счета сериализованы (как в процедуре `make-account` из предыдущего раздела), `exchange` может привести к неверным результатам. Например, может оказаться, что Петр посчитает разницу между $a1$ и $a2$, но Павел изменит баланс на $a1$ прежде, чем Петр закончит обмен[42]. Чтобы добиться правильного поведения, мы должны устроить так, чтобы процедура `exchange` блокировала всякий параллельный доступ к счетам на все время обмена.

Один из способов этого достичь — сериализовать всю процедуру `exchange` сериализаторами обоих счетов. Ради этого мы откроем доступ к сериализаторам счетов. Обратите внимание, что, раскрывая сериализатор, мы намеренно ломаем модульное построение объекта-банковского счета. Следующая версия процедуры `make-account` идентична исходной версии из раздела 3.1.1, за исключением того, что имеется сериализатор для защиты переменной баланса, и он экспортируется через передачу сообщений:

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Неизвестный запрос -- MAKE-ACCOUNT"
                         m))))
    dispatch))
```

С помощью этой версии мы можем выполнять сериализованное занесение и снятие денег. Заметим, однако, что, в отличие от предыдущей версии сериализованного счета, теперь каждый пользователь объектов-банковских счетов должен явным образом управлять сериализацией, например, так[43]:

[42] Если балансы на счетах вначале равны 10, 20 и 30 долларам, то после любого количества параллельных обменов балансы должны по прежнему быть 10, 20 и 30, в каком-то порядке. Сериализации доступа к отдельным счетам недостаточно, чтобы это гарантировать. См. упр. 3.43.

[43] В упражнении 3.45 рассматривается вопрос, почему занесение и снятие денег теперь не сериализуются

```
(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit)))
    ((s d) amount)))
```

Экспорт сериализатора дает нам достаточно гибкости, чтобы реализовать сериализованную программу обмена. Мы просто-напросто сериализуем исходную процедуру `exchange` сериализаторами обоих счетов:

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))
```

Упражнение 3.43.
Предположим, что значения баланса на трех счетах вначале равны 10, 20 и 30 долларам, и что несколько процессов занимаются обменом значений баланса. Покажите, что если эти процессы выполняются последовательно, то после любого количества обменов значения баланса по-прежнему будут равны 10, 20 и 30 долларам, в каком-то порядке. Нарисуйте временную диаграмму вроде той, которая изображена на рис. 3.29, и покажите, что указанное условие может нарушаться, если работает первая версия процедуры обмена из этого раздела. Покажите, с другой стороны, что даже с первой программой `exchange` общая сумма балансов на счетах сохранится. Нарисуйте временную диаграмму, показывающую, что если бы мы не сериализовали транзакции по отдельным счетам, это условие тоже могло бы нарушаться.

Упражнение 3.44.
Рассмотрим задачу переноса денег с одного счета на другой. Бен Битобор утверждает, что ее можно решить с помощью следующей процедуры, даже в тех случаях, когда много людей одновременно перемещают деньги между различными счетами, если использовать при этом какой-то механизм, сериализующий операции занесения на счет и снятия со счета, например, версию `make-account` из нашего текста.

```
(define (transfer from-account to-account amount)
  ((from-account 'withdraw) amount)
  ((to-account 'deposit) amount))
```

Хьюго Дум считает, что с этой версией возникнут проблемы и что нужно использовать более сложный подход, вроде того, который требуется при решении задачи обмена. Прав ли он? Если нет, то в чем состоит существенная разница между задачей перевода денег и задачей обмена счетов? (Нужно предположить, что значение баланса на `from-account` по крайней мере равно `amount`.)

Упражнение 3.45.
Хьюго Дум полагает, что теперь, когда операции снятия денег со счета и занесения их на счет перестали сериализовываться автоматически, система банковских счетов стала неоправданно сложной

счетом автоматически.

и работать с ней правильным образом чересчур трудно. Он предлагает сделать так, чтобы `make-account-and-serializer` экспортировал сериализатор (для использования в процедурах вроде `serialized-exchange`), и вдобавок сам использовал его для сериализации простых операций со счетом, как это делал `make-account`. Он предлагает переопределить объект-счет так:

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Недостаточно денег на счете"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (balance-serializer withdraw))
            ((eq? m 'deposit) (balance-serializer deposit))
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Неизвестный запрос -- MAKE-ACCOUNT"
                         m))))
    dispatch))

(define (deposit account amount)
 ((account 'deposit) amount))
```

Объясните, в чем Хьюго ошибается. В частности, рассмотрите, что происходит при вызове `serialized-exchange`.

Реализация сериализаторов

Мы реализуем сериализаторы на основе более примитивного механизма синхронизации, называемого *мьютекс* (mutex). Мьютекс — это объект, который поддерживает две операции: его можно *захватить* (acquire), и его можно *освободить* (release). Когда мьютекс захвачен, никакая другая операция захвата того же самого мьютекса произойти не может, пока его не освободят[44]. В нашей реализации каждый сериализатор содержит по мьютексу. Получая процедуру `p`, сериализатор возвращает процедуру, которая захватывает мьютекс, выполняет `p`, и затем освобождает мьютекс. Благодаря этому, только одна из процедур, порожденных сериализатором, может исполняться в каждый момент времени. Именно такого поведения мы и хотели добиться от сериализации.

[44]Название «мьютекс» происходит от английского *mutual exclusion* «взаимное исключение». Общая проблема построения механизма, который позволил бы параллельным процессам безопасно разделять ресурсы, называется проблемой взаимного исключения. Наши мьютексы являются простым вариантом механизма *семафоров* (semaphores) (см. упражнение 3.47), которые впервые появились в Системе Мультипрограммирования THE, разработанной в Эйндховенском Техническом Университете и названной по первым буквам голландского названия этого учебного заведения (Dijkstra 1968a). Операции захвата и освобождения изначально назывались P и V, от голландских глаголов *passeren* (пройти) и *vrijgeven* (освободить), употребляемых по отношению к семафорам на железных дорогах. Классическое описание Дейкстры (Dijkstra 1968b) было одним из первых ясных изложений вопросов управления параллелизмом, и там было показано, как решаются при помощи семафоров различные задачи.

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
      serialized-p)))
```

Мьютекс — изменяемый объект (здесь мы используем одноэлементный список, который будем называть *ячейкой* (cell)), способный хранить значение истина или ложь. Когда значение ложно, мьютекс можно захватывать. Когда значение истинно, мьютекс недоступен, и процесс, который попытается его захватить, вынужден будет ждать.

Конструктор мьютекса `make-mutex` для начала присваивает содержимому ячейки значение ложь. Для захвата мьютекса мы проверяем значение ячейки. Если мьютекс доступен, мы делаем значение истинным и идем дальше. Если нет, мы входим в цикл ожидания, все время пытаясь захватить мьютекс, пока он не окажется свободным[45]. Чтобы освободить мьютекс, мы присваиваем значению ячейки ложь.

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire)))
            ((eq? m 'release) (clear! cell))))
    the-mutex))
```

`Test-and-set!` проверяет ячейку и возвращает результат проверки. Помимо того, если значение было ложным, `test-and-set!` устанавливает значение в истину, прежде чем вернуть ложь. Мы можем описать это поведение так:

```
(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

Однако эта реализация `test-and-set!`, как она есть, не годится. Здесь есть важная тонкость, и именно здесь управление параллелизмом становится частью системы: операция `test-and-set!` должна производиться *атомарно* (atomically). Это значит, что мы должны гарантировать, что когда процесс протестировал ячейку и убедился, что ее значение ложь, значение будет установлено в истину прежде, чем какой-либо еще процесс успеет проверить ячейку. Если мы такую гарантию не обеспечим, мьютекс может сломаться таким же образом, как банковский счет на рис. 3.29. (См. упражнение 3.46.)

[45] В большинстве систем разделения времени процессы, блокированные на мьютексе, не тратят время в «занятом ожидании», как это описано здесь. Вместо этого система назначает на исполнение другой процесс, пока первый ждет, а когда мьютекс освобождается, она будит заблокированный процесс.

Реализация `test-and-set!` зависит от того, как наша система на самом деле управляет параллельными процессами. Например, мы можем выполнять параллельные процессы на последовательном процессоре при помощи механизма разделения времени, который перебирает процессы по очереди, дает каждому из них выполняться в течение небольшого промежутка времени, а затем прерывает его и переходит к следующему процессу. В таком случае `test-and-set!` может запрещать смену процесса в момент между проверкой и присваиванием[46]. С другой стороны, в многопроцессорных компьютерах бывают команды, которые обеспечивают атомарные операции прямо на уровне аппаратуры[47].

Упражнение 3.46.
Допустим, что мы реализуем `test-and-set` в виде обыкновенной процедуры, как показано в тексте, не пытаясь сделать ее атомарной. Нарисуйте временную диаграмму, подобную диаграмме на рис. 3.29, и покажите, как реализация мьютекса может ошибиться и позволить двум процессам одновременно захватить мьютекс.

Упражнение 3.47.
Семафор (размера n) представляет собой обобщение мьютекса. Подобно мьютексу, семафор поддерживает операции захвата и освобождения, но захватить его одновременно могут до n процессов. Прочие процессы, которые попытаются захватить семафор, должны будут ждать освобождения. Дайте реализацию семафоров

а. в терминах мьютексов.

б. в терминах атомарных операций `test-and-set!`.

Тупик

Теперь, когда мы рассмотрели, как реализуются сериализаторы, мы убеждаемся, что с обменом счетов по-прежнему связаны проблемы, даже с вышеописанной процедурой `serialized-exchange`. Допустим, что Петр хочет обменять $a1$ и $a2$, а Павел в то

[46] В MIT Scheme на однопроцессорной системе можно реализовать `test-and-set!` следующим образом:

```
(define (test-and-set! cell)
  (without-interrupts
   (lambda ()
     (if (car cell)
         true
         (begin (set-car! cell true)
                false)))))
```

`Without-interrupts` запрещает прерывания по таймеру, пока выполняется его процедурный аргумент.

[47] Есть много вариантов таких команд — включая проверку-и-установку, проверку-и-сброс, обмен, сравнение-и-обмен, загрузку с резервированием и условную запись, — и их форма должна точно соответствовать интерфейсу между процессором и памятью в данной машине. Один из возникающих вопросов состоит в том, что происходит, когда два процесса пытаются получить один и тот же ресурс в точности одновременно при помощи такой команды. Тут требуется какой-то механизм, принимающий решение, который из процессов получает управление. Такой механизм называется *арбитром* (arbiter). Обычно арбитры представляют собой аппаратные устройства. К сожалению, можно доказать, что нельзя построить справедливого арбитра, работающего в 100% случаев, если не позволять арбитру принимать решение неопределенно долгое время. Сущность этого явления была открыта французским философом XIV века Жаном Буриданом в комментарии к *De caelo* Аристотеля. Буридан указал, что идеально разумная собака, помещенная между двумя одинаково привлекательными кусками еды, должна умереть от голода, поскольку она не сможет решить, к какому куску идти в первую очередь.

же время пытается обменять $a2$ и $a1$. Допустим, что процесс Петра доходит до некоторой точки внутри сериализованной процедуры, защищающей $a1$, и сразу вслед за этим процесс Павла входит в сериализованную процедуру, защищающую $a2$. Теперь Петр не может двигаться дальше (ему надо войти в сериализованную процедуру для $a2$), пока Павел не выйдет из сериализованной процедуры для $a2$. Точно так же Павел не может двигаться дальше, пока Петр не выйдет из сериализованной процедуры для $a1$. Оба процесса замирают навеки в ожидании друг друга. Такая ситуация называется *тупик* (deadlock). В любой системе, которая предоставляет доступ к множественным разделяемым ресурсам, существует опасность тупика.

В этой ситуации можно избежать тупика, если присвоить каждому счету уникальный идентификационный номер, и переписать `serialized-exchange` так, чтобы процесс всегда пытался сначала войти в процедуру, которая защищает счет с наименьшим номером. Хотя для задачи обмена это решение работает хорошо, бывают и другие ситуации, в которых требуются более развитые методы избежания тупиков, или где тупика нельзя избежать в принципе. (См. упражнения 3.48 и 3.49.)[48]

Упражнение 3.48.
Подробно объясните, почему метод избежания тупиков, описанный выше (т. е. счета нумеруются, и каждый процесс сначала пытается захватить счет с меньшим номером), в самом деле позволяет избежать тупика в задаче обмена балансов. Перепишите `serialized-exchange` с использованием этой идеи. (Придется также изменить `make-account`, так, чтобы каждый счет создавался вместе с номером, и чтобы этот номер можно было считать, послав соответствующее сообщение.)

Упражнение 3.49.
Опишите сценарий, в котором вышеописанный механизм избежания тупиков не работает. (Подсказка: в задаче обмена счетов каждый процесс заранее знает, к каким счетам ему нужен будет доступ. Рассмотрите ситуацию, в которой процессу нужно сначала получить доступ к каким-то разделяемым ресурсам, прежде чем он сможет определить, какие ресурсы ему потребуются дополнительно.)

Параллелизм, время и взаимодействие

Мы видели, что для программирования параллельных систем, когда различные процессы имеют доступ к разделяемому состоянию, необходимо управление порядком событий, и мы видели, как можно добиться нужного порядка с помощью надлежащего использования сериализаторов. Однако проблемы параллелизма лежат глубже, поскольку, с фундаментальной точки зрения, не всегда ясно, что имеется в виду под «разделяемым состоянием».

Механизмы вроде `test-and-set!` требуют, чтобы процессы в произвольные моменты времени имели доступ к глобальному разделяемому флагу. На современных высокоскоростных процессорах это реализуется сложно и неэффективно, поскольку, благодаря средствам оптимизации вроде конвейеров и кэширования памяти, содержимое памяти не

[48]Общий метод избежания тупиков путем нумерации разделяемых ресурсов и захвата их по порядку придумал Хейвендер (Havender 1968). В ситуациях, где тупика нельзя избежать, нужны меры по *выходу из тупика* (deadlock recovery), когда от процессов требуется «откатиться» из тупикового состояния и повторить попытку. Механизмы выхода из тупика широко используются в системах управления базами данных. Эта тема детально рассматривается у Грея и Рейтера (Gray and Reuter 1993).

обязательно должно в каждый момент находиться в непротиворечивом состоянии. Из-за этого в современных многопроцессорных системах идея сериализаторов вытесняется новыми подходами к управлению параллелизмом[49].

Кроме того, проблемы с разделяемым состоянием возникают в больших распределенных системах. Например, рассмотрим распределенную банковскую систему, в которой отдельные местные банки поддерживают собственные значения баланса счетов и время от времени сравнивают их со значениями, хранимыми в других местах. В такой системе значение «баланс счета» не будет определенным ни в какой момент, кроме как сразу после синхронизации. Если Петр вносит деньги на счет, который он делит с Павлом, когда мы должны считать, что баланс изменился, — когда меняется баланс в местном банке или только после синхронизации? А если Павел обращается к счету через другую ветвь системы, какие ограничения нужно наложить на банковскую систему, чтобы ее поведение считалось «правильным»? Единственное, что может иметь значение для определения «правильности», — это поведение, которое Павел и Петр наблюдают по отдельности, и состояние счета сразу после синхронизации. Вопросы о «настоящем» значении баланса или порядке событий между синхронизациями могут не иметь значения или даже смысла[50].

Общее в этих проблемах то, что синхронизация различных процессов, установление общего состояния и управление порядком событий требуют взаимодействия процессов. В сущности, любое понятие времени при управлении параллельными процессами должно быть прочно привязано к взаимодействию процессов[51]. Любопытно, что похожая связь между временем и обменом информацией возникает в теории относительности, где скорость света (самого быстрого сигнала, который можно использовать для синхронизации событий) служит универсальной константой, связывающей пространство и время. Сложности, с которыми мы сталкиваемся при работе с временем и состоянием в вычислительных моделях, могут на самом деле отражать фундаментальную сложность физического мира.

3.5. Потоки

Теперь у нас есть ясное понимание того, как присваивание может служить инструментом моделирования, а также понятие о сложности проблем, связанных с ним. Пора задать вопрос, нельзя ли организовать работу иначе и избежать части этих проблем.

[49] Один из подходов, альтернативных сериализации, называется *барьерная синхронизация* (barrier synchronization). Программист позволяет параллельным процессам выполняться как угодно, но устанавливает определенные точки синхронизации («барьеры»), так что ни один процесс не может продолжаться, пока все они не достигли барьера. Современные процессоры обладают машинными командами, которые позволяют программистам устанавливать точки синхронизации там, где требуется иметь непротиворечивое состояние. Например, в Power PC™ имеются две предназначенные для этого команды: SYNC и EIEIO (Enforced In-Order Execution of Input-Output, Гарантированно Последовательное Исполнение Ввода-Вывода).

[50] Такая точка зрения может казаться странной, но при этом существуют системы, которые именно так и работают. Изменения на счетах, связанных с кредитными картами, например, обычно поддерживаются отдельно в каждой стране, а изменения в различных странах согласовываются время от времени. Таким образом, баланс на счете может быть различным в различных странах.

[51] Для распределенных систем эта точка зрения исследовалась Лэмпортом (Lamport 1978). Он показал, как при помощи взаимодействия установить «глобальные часы», через которые можно управлять порядком событий в распределенных системах.

В этом разделе мы исследуем альтернативный подход к моделированию состояния, основанный на структурах данных, называемых *потоками* (streams). Как нам предстоит убедиться, потоки могут смягчить некоторые трудности в моделировании состояния.

Давайте сделаем шаг назад и рассмотрим еще раз, откуда происходят эти сложности. Пытаясь моделировать явления реального мира, мы приняли несколько, казалось бы, разумных решений: мы моделировали объекты внешнего мира, обладающие состоянием, при помощи вычислительных объектов с внутренними переменными. Мы отождествили течение времени в мире с течением времени в компьютере. Мы имитировали на компьютере изменение состояния моделируемых объектов при помощи присваивания внутренним переменным объектов-моделей.

Возможен ли другой подход? Можно ли избежать отождествления времени в компьютере с временем в моделируемом мире? Должны ли мы заставить модель изменяться во времени, чтобы смоделировать явления изменяющегося мира? Давайте подумаем об этом в терминах математических функций. Можно описать изменение во времени величины x с помощью функции $x(t)$, где время выступает как аргумент. Если мы сосредотачиваем внимание на x момент за моментом, мы думаем об изменяющейся величине. Однако если мы обращаем внимание на всю хронологию значений, мы не подчеркиваем изменение — функция сама по себе не изменяется[52].

Если время измеряется дискретными интервалами, мы можем смоделировать функцию времени как последовательность (возможно, бесконечную). В этом разделе мы увидим, как моделировать изменение в виде последовательностей, которые представляют картины изменения во времени систем, подвергаемых моделированию. С этой целью мы вводим новую структуру данных, называемую *поток* (stream). С абстрактной точки зрения, поток — это просто последовательность. Однако, как мы увидим, прямое представление потоков в виде списков (как в разделе 2.2.1) не полностью раскрывает мощь работы с потоками. В качестве альтернативы мы введем метод *задержанных вычислений* (delayed evaluation), который позволит нам представлять очень большие (даже бесконечные) последовательности в виде потоков.

Работа с потоками позволяет моделировать системы, обладающие состоянием, совершенно не используя присваивание и изменяемые данные. Отсюда есть важные следствия, как теоретические, так и практические, поскольку мы приобретаем возможность строить модели, лишенные недостатков, связанных с присваиванием. С другой стороны, парадигма потоков вызывает свои собственные трудности, и вопрос, какой из методов моделирования ведет к построению более модульных и легко поддерживаемых систем, остается открытым.

3.5.1. Потоки как задержанные списки

Как мы видели в разделе 2.2.3, последовательности можно использовать как стандартные интерфейсы для комбинирования программных модулей. Мы сформулировали мощные абстракции для работы с последовательностями, такие как `map`, `filter` и `accumulate`, с помощью которых можно описать широкий класс действий одновременно коротко и изящно.

[52] Физики иногда принимают эту точку зрения, вводя «мировые линии» частиц в рассуждениях о движении. Кроме того, мы уже упоминали (в разделе 2.2.3), что это естественный ход мысли при рассуждениях о системах обработки сигналов. Мы рассмотрим приложение потоков к обработке сигналов в разделе 3.5.3.

К сожалению, если представлять последовательности в виде списков, за это изящество приходится расплачиваться чрезвычайной неэффективностью как с точки зрения времени, так и с точки зрения объема памяти, который требуется нашим вычислениям. Когда мы представляем операции над последовательностями в виде трансформаций списков, программам приходится на каждом шагу строить и копировать структуры данных (которые могут быть громадными).

Чтобы понять, почему это так, сравним две программы для вычисления суммы всех простых чисел на интервале. Первая программа написана в стандартном итеративном стиле[53]:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

Вторая программа производит то же самое вычисление с помощью операций над последовательностями из раздела 2.2.3:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

Во время вычисления первая программа должна хранить только накапливаемую сумму. Напротив, фильтр во второй программе не может начать тестировать, пока `enumerate-interval` не создала полного списка чисел на интервале. Фильтр порождает еще один список, который, в свою очередь, передается `accumulate`, прежде, чем он сожмется в сумму. Первой программе не требуется такого количества промежуточной памяти, — мы можем считать, что она просто проходит интервал снизу вверх, добавляя к сумме каждое простое число, которое ей встретится.

Неэффективность использования списков становится болезненно очевидной, если мы воспользуемся парадигмой последовательностей для вычисления второго простого числа в интервале от 1000 до 1 000 000 при помощи следующего выражения:

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

Это выражение находит второе простое число, однако на это затрачивается возмутительное количество вычислительных ресурсов. Мы строим список из почти миллиона целых чисел, фильтруем этот список, проверяя каждый его элемент на простоту, а затем почти весь результат игнорируем. При более традиционном программистском подходе мы бы чередовали перечисление и фильтрацию, и остановились бы по достижении второго простого числа.

[53] Мы предполагаем, что у нас имеется предикат `prime?` (например, из раздела 1.2.6), который проверяет, является ли число простым.

Потоки представляют собой прием, который дает возможность работать с последовательностями и при этом ничего не терять на представлении последовательностей в виде списков. Потоки сочетают лучшее из обоих подходов: мы можем изящно формулировать программы в терминах операций с последовательностями и при этом сохранять эффективность пошагового вычисления. Основная идея состоит в том, чтобы строить список только частично и передавать частично построенный список программе, потребляющей поток. Если потребитель запросит доступ к той части потока, которая еще не сконструирована, поток автоматически достроит ровно такую часть себя самого, какая нужна, и сохранит таким образом иллюзию, что он существует целиком. Другими словами, хотя программы будут писаться так, как будто обрабатываются полные последовательности, мы так спроектируем реализацию потоков, что построение потока будет автоматически и незаметно для пользователя чередоваться с его использованием.

На первый взгляд, потоки — это просто списки, у которых процедуры работы с ними переименованы. Имеется конструктор, `cons-stream`, и два селектора, `stream-car` и `stream-cdr`, причем выполняются уравнения

```
(stream-car (cons-stream x y)) = x

(stream-cdr (cons-stream x y)) = y
```

Имеется специальный объект, `the-empty-stream`, который не может быть результатом никакой операции `cons-stream`, и который можно распознать процедурой `stream-null?`[54]. Таким образом, можно создавать и использовать потоки, точно так же, как списки, для представления составных данных, организованных в виде последовательности. В частности, можно построить потоковые аналоги операций со списками из главы 2, таких, как `list-ref`, `map` и `for-each`[55]:

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
```

[54] В реализации MIT `the-empty-stream` совпадает с пустым списком `'()`, а процедура `stream-null?` совпадает с `null?`.

[55] Здесь у Вас должно возникнуть беспокойство. То, что мы определяем столь сходные процедуры для потоков и списков, показывает, что мы упускаем некую глубинную абстракцию. К сожалению, чтобы использовать эту абстракцию, нам нужно более точное управление процессом вычисления, чем у нас сейчас есть. Мы подробнее обсудим этот вопрос в конце раздела 3.5.4. В разделе 4.2 мы разработаем среду, в которой списки и потоки объединяются.

С помощью stream-for-each потоки можно печатать:

```
(define (display-stream s)
  (stream-for-each display-line s))

(define (display-line x)
  (newline)
  (display x))
```

Чтобы заставить реализацию потоков автоматически и незаметно чередовать построение потока с его использованием, мы сделаем так, чтобы cdr потока вычислялся тогда, когда к нему обращается процедура stream-cdr, а не тогда, когда поток создается процедурой cons-stream. Такое проектное решение заставляет вспомнить обсуждение рациональных чисел в разделе 2.1.2, где мы увидели, что можно приводить рациональные числа к наименьшему знаменателю либо во время создания числа, либо во время обращения к нему. Две реализации рациональных чисел предоставляют одну и ту же абстракцию, однако наш выбор влияет на эффективность работы. Существует подобная связь и между потоками и обычными списками. В качестве абстракции данных потоки не отличаются от списков. Разница состоит в том, когда вычисляются их элементы. В обычных списках и car, и cdr вычисляются во время построения. У потоков cdr вычисляется при обращении.

Наша реализация потоков основана на особой форме под названием delay. Выполнение (delay ⟨*выражение*⟩) не вычисляет ⟨*выражение*⟩, а вместо этого возвращает так называемый *задержанный объект* (delayed object). Мы можем считать, что это «обещание» вычислить выражение когда-нибудь в будущем. В качестве пары к delay имеется процедура force, которая берет задержанный объект в качестве аргумента и вычисляет его — фактически, заставляя delay выполнить обещание. Ниже мы увидим, как можно реализовать delay и force, но сначала давайте посмотрим, как с их помощью строить потоки.

Cons-stream — это особая форма, такая, что

```
(cons-stream ⟨a⟩ ⟨b⟩)
```

эквивалентно

```
(cons ⟨a⟩ (delay ⟨b⟩))
```

Это означает, что мы строим потоки при помощи пар. Однако вместо того, чтобы поместить значение остатка потока в cdr пары, мы кладем туда обещание вычислить остаток, если нас об этом попросят. Теперь можно определить stream-car и stream-cdr как процедуры:

```
(define (stream-car stream) (car stream))

(define (stream-cdr stream) (force (cdr stream)))
```

Streams-car возвращает car пары. Stream-cdr берет cdr пары и вычисляет хранящееся там задержанное выражение, чтобы получить остаток потока[56].

[56] В отличие от stream-car и stream-cdr, которые можно определить в виде процедур, cons-stream

Реализация потоков в действии

Чтобы посмотреть, как ведет себя эта реализация, давайте проанализируем «возмутительное» вычисление с простыми числами, переформулированное через потоки:

```
(stream-car
  (stream-cdr
   (stream-filter prime?
                  (stream-enumerate-interval 10000 1000000))))
```

Мы увидим, что теперь вычисления происходят эффективно.

Вначале зовется процедура `stream-enumerate-interval` с аргументами 1000 и 1000000. `Stream-enumerate-interval` — это потоковый аналог процедуры `enumerateinterval` (раздел 2.2.3):

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))
```

и, таким образом, результат, возвращаемый `stream-enumerate-interval`, сформированный `cons-stream` внутри нее, равен[57]

```
(cons 10000
      (delay (stream-enumerate-interval 10001 1000000)))
```

А именно, `stream-enumerate-interval` возвращает поток, представленный в виде пары, `car` которой равен 10000, а `cdr` является обещанием вычислить остаток интервала, когда попросят. Теперь этот поток отфильтровывается на предмет поиска простых чисел с помощью потокового аналога процедуры `filter` (раздел 2.2.3):

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred
                                     (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

`Stream-filter` проверяет `stream-car` потока (то есть `car` пары, то есть 10000). Поскольку это не простое число, `stream-filter` смотрит на `streamcdr` своего входного потока. Вызов `stream-cdr` приводит к вычислению задержанного вызова `stream-enumerate-interval`, возвращающего

обязан быть особой формой. Если бы он был процедурой, то, согласно нашей модели вычислений, выполнение `(cons-stream` $\langle a\rangle$ $\langle b\rangle$`)` автоматически приводило бы к вычислению $\langle b\rangle$, а именно этого мы и не хотим. По этой же причине `delay` должен быть особой формой, хотя `force` может оставаться обычной процедурой.

[57] Показанные здесь числа на самом деле не появляются в возвращаемом выражении. Возвращается исходное выражение вместе с окружением, в котором переменным присвоены соответствующие значения. Например, там, где напечатано число 10001, стоит `(+ low 1)`, и переменная `low` связана со значением 10000.

```
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
```

Теперь `stream-filter` смотрит на `stream-car` этого потока, 10001, видит, что и это не простое число, снова зовет `stream-cdr` и так далее, пока `stream-enumerate-interval` не выдаст простое число 10007. Тогда `streamfilter`, в соответствии со своим определением, вернет

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

что в данном случае равняется

```
(cons 10007
      (delay
        (stream-filter
         prime?
         (cons 10008
               (delay
                 (stream-enumerate-interval 10009
                                            1000000))))))
```

Теперь этот результат передается в `stream-cdr` из нашего исходного выражения. При этом вызывается задержанный `stream-filter`, который, в свою очередь, вынуждает задержанные вызовы `stream-enumerate-interval`, пока не доберется до следующего простого числа, а именно 10009. Наконец, результат, передаваемый в `stream-car` нашего исходного выражения, равен

```
(cons 10009
      (delay
        (stream-filter
         prime?
         (cons 10010
               (delay
                 (stream-enumerate-interval 10011
                                            1000000))))))
```

`Stream-car` возвращает 10009, и вычисление закончено. На простоту было проверено ровно столько чисел, сколько было необходимо, чтобы найти второе простое число на интервале, и сам интервал был перебран только до того места, которое было нужно фильтру простых чисел.

В общем, мы можем считать задержанные вычисления программированием, «управляемым потребностями», в котором каждый шаг вычислений в потоковом процессе активизируется лишь настолько, насколько это нужно для следующего шага. Таким образом, нам удалось отделить реальный порядок событий при вычислении от внешней структуры процедур. Мы пишем процедуры так, как будто потоки существуют «все целиком», а на самом деле вычисление происходит пошагово, как и при программировании в традиционном стиле.

Реализация `delay` **и** `force`

`Delay` и `force` могут казаться таинственными операциями, но на самом деле их реализация весьма проста. `Delay` должно упаковать выражение так, чтобы потом его можно было выполнить по требованию, и мы добиваемся этого, просто рассматривая выражение как тело процедуры. Можно сделать `delay` особой формой, такой, чтобы

```
(delay ⟨выражение⟩)
```

было синтаксическим сахаром для

```
(lambda () ⟨выражение⟩)
```

`Force` просто вызывает (безаргументную) процедуру, порожденную `delay`, так что она может быть реализована как процедура

```
(define (force delayed-object)
  (delayed-object))
```

При такой реализации `delay` и `force` работают согласно описанию, однако к ней можно добавить важную оптимизацию. Во многих приложениях мы вынуждаем один и тот же задержанный объект по многу раз. В рекурсивных программах с использованием потоков это может привести к существенной неэффективности (см. упражнение 3.57). Решение состоит в том, чтобы строить задержанные объекты так, чтобы при первом вынуждении они сохраняли вычисленное значение. Последующие обращения будут просто возвращать сохраненное значение без повторения вычислений. Другими словами, мы реализуем `delay` как особого рода мемоизированную процедуру, подобную описанным в упражнении 3.27. Один из способов этого добиться — использовать следующую процедуру, которая принимает процедуру (без аргументов) и возвращает ее мемоизированную версию. При первом вызове мемоизированная процедура сохраняет результат. При последующих вызовах она просто его возвращает.

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

Теперь можно определить `delay` таким образом, что `(delay` ⟨*выражение*⟩`)` равносильно

```
(memo-proc (lambda () ⟨выражение⟩))
```

а определение `force` не меняется[58].

[58]Есть много возможных реализаций потоков помимо описанной в этом разделе. Задержанное вычисление, ключевой элемент, который делает потоки практически полезными, было частью метода передачи параметров

Упражнение 3.50.
Закончите следующее определение, которое обобщает процедуру `stream-map`, чтобы она позволяла использовать процедуры от нескольких аргументов, подобно `map` из раздела 2.2.1, сноска 12.

```
(define (stream-map proc . argstreams)
  (if (⟨??⟩ (car argstreams))
      the-empty-stream
      (⟨??⟩
       (apply proc (map ⟨??⟩ argstreams))
       (apply stream-map
              (cons proc (map ⟨??⟩ argstreams))))))
```

Упражнение 3.51.
Чтобы внимательнее изучить задержанные вычисления, мы воспользуемся следующей процедурой, которая печатает свой аргумент, а затем возвращает его:

```
(define (show x)
  (display-line x)
  x)
```

Что печатает интерпретатор в ответ на каждое выражение из следующей последовательности[59]?

```
(define x (stream-map show (stream-enumerate-interval 0 10)))

(stream-ref x 5)

(stream-ref x 7)
```

Упражнение 3.52.
Рассмотрим последовательность выражений

```
(define sum 0)

(define (accum x)
  (set! sum (+ x sum))
  sum)
```

по имени (by name) в языке Алгол-60. Использование этого механизма для реализации потоков впервые было описано Ландином (Landin 1965). Задержанное вычисление для потоков ввели в Лисп Фридман и Уайз (Friedman and Wise 1976). В их реализации `cons` всегда задерживает вычисление своих аргументов, так что списки автоматически ведут себя как потоки. Мемоизирующая оптимизация известна также как *вызов по необходимости* (by need). В сообществе программистов на Алголе задержанные объекты из нашей первой реализации назывались бы *санками вызова по имени* (call-by-name thunks), а оптимизированный вариант *санками вызова по необходимости* (call-by-need thunks).

[59]Упражнения типа 3.51 и 3.52 помогают понять, как работает `delay`. С другой стороны, смешение задержанного вычисления с печатью — или, хуже того, с присваиванием, — ужасно запутывает, и преподаватели, читающие курсы по языкам программирования, часто пытают студентов экзаменационными вопросами вроде упражнений из этого раздела. Незачем и говорить, что писать программы, зависящие от таких тонкостей, — показатель чрезвычайно плохого стиля. Отчасти мощность потокового программирования в том и заключается, что можно игнорировать порядок, в котором на самом деле происходят события в программах. К сожалению, ровно этого мы и не можем себе позволить в присутствии присваивания, заставляющего нас думать о времени и изменении.

```
(define seq (stream-map accum (stream-enumerate-interval 1 20)))
(define y (stream-filter even? seq))
(define z (stream-filter (lambda (x) (= (remainder x 5) 0))
                         seq))

(stream-ref y 7)

(display-stream z)
```

Каково значение `sum` после вычисления каждого из этих выражений? Что печатается при вычислении выражений `stream-ref` и `display-stream`? Изменился бы этот результат, если бы мы реализовали `(delay` ⟨*выражение*⟩`)` просто как `(lambda ()` ⟨*выражение*⟩`)`, не применяя оптимизацию через `memo-proc`? Объясните свой ответ.

3.5.2. Бесконечные потоки

Мы видели, как можно поддерживать иллюзию работы с потоками как с цельными объектами, хотя на самом деле мы вычисляем только ту часть потока, к которой нам требуется доступ. Этот метод можно использовать, чтобы эффективно представлять последовательности в виде потоков, даже если эти последовательности весьма длинны. Еще удивительнее то, что при помощи потоков можно представлять последовательности бесконечной длины. Рассмотрим, например, следующее определение потока положительных целых чисел:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1))
```

Такая запись имеет смысл, потому что описывает `sequence` как пару, у которой `car` равен 1, а `cdr` является обещанием породить целые числа, начиная с 2. Такой поток бесконечен, но в любой данный момент мы можем работать только с конечной его частью. Таким образом, наши программы никогда не узнают, что целиком бесконечного потока не существует.

При помощи `integers` можно определять другие бесконечные потоки, например, поток чисел, не делящихся на 7:

```
(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))
```

Теперь мы можем искать числа, не делящиеся на 7, просто обращаясь к элементам этого потока:

```
(stream-ref no-sevens 100)
```

По аналогии с `integers`, можно определить бесконечный поток чисел Фибоначчи:

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

`Fibs` представляет собой пару, `car` которой равен 0, а `cdr` является обещанием вычислить `(fibgen 1 1)`. Когда мы выполняем это задержанное `(fibgen 1 1)`, оно порождает пару, где `car` равен 1, а в `cdr` лежит обещание вычислить `(fibgen 1 2)`, и так далее.

Чтобы продемонстрировать пример более интересного потока, можно обобщить `no-sevens` и построить бесконечный поток простых чисел, используя метод, известный как *решето Эратосфена* (sieve of Eratosthenes)[60]. Сначала мы строим поток чисел, начиная с 2, первого простого числа. Для того, чтобы найти остальные простые числа, мы фильтруем кратные двойки из потока остальных чисел. Получается поток, который начинается с 3, следующего простого числа. Теперь из остатка потока мы фильтруем числа, кратные 3. Получается поток, начинающийся с 5, следующего простого, и так далее. Другими словами, мы строим простые числа с помощью просеивающего процесса, описываемого так: чтобы просеять поток S, нужно сформировать поток, в котором первый элемент совпадает с первым элементом S, а остаток получается фильтрацией множителей первого элемента из оставшейся части S и просеивания того, что получится. Такой процесс нетрудно описать в терминах операций над потоками:

```
(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))

(define primes (sieve (integers-starting-from 2)))
```

Теперь, чтобы найти определенное простое число, надо только попросить:

```
(stream-ref primes 50)
233
```

Интересно представить себе систему обработки сигналов, соответствующую `sieve`, показанную на «хендерсоновской диаграмме» на рисунке 3.31[61]. Входной поток попадает в «рас`cons`ер», который отделяет первый элемент потока от его хвоста. При помощи

[60] Эратосфен, греческий философ третьего века до н. э. из Александрии, знаменит тем, что он дал первую верную оценку длины окружности Земли, которую он вычислил, наблюдая тени, отбрасываемые в полдень летнего солнцестояния. Метод решета Эратосфена, несмотря на свою древность, лежал в основе специальных аппаратных устройств-«решет», которые до недавних пор были самыми мощными устройствами для поиска простых чисел. Однако начиная с 70-х годов такие устройства были вытеснены развитием вероятностных методик, обсуждаемых в разделе 1.2.6.

[61] Мы назвали этот способ изображения потоков в честь Питера Хендерсона, который первым показал нам диаграммы такого вида как способ рассуждений об обработке потоков. Сплошные линии представляют потоки передаваемых сигналов. Прерывистая линия от `car` к `cons` и `filter` указывает, что здесь передается не поток, а единичное значение.

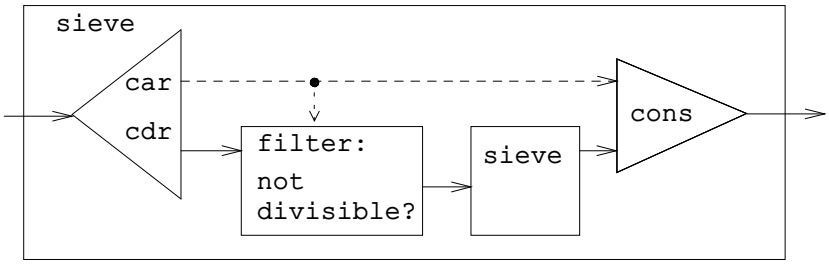


Рис. 3.31. Решето для поиска простых чисел в виде системы обработки сигналов.

первого элемента строится фильтр на делимость, и через него пропускается остаток входного потока, а выход запускается в еще один элемент `sieve`. Затем исходный первый элемент сочетается при помощи `cons` с выходом внутреннего `sieve`, и получается выходной поток. Таким образом, не только входной поток бесконечен, но и обработчик сигналов также бесконечен, поскольку одно решето содержит в себе другое.

Неявное определение потоков

Потоки `integers` и `fibs` были определены при помощи «порождающих» процедур, которые явным образом вычисляют элементы потока один за другим. Однако можно определять потоки неявно, пользуясь задержанным вычислением. Например, следующее выражение определяет `ones` как бесконечный поток, состоящий из одних единиц:

```
(define ones (cons-stream 1 ones))
```

Это выражение работает примерно так же, как рекурсивная процедура: `ones` является парой, чей `car` есть 1, а `cdr` представляет собой обещание вычислить `ones`. Обращение к `cdr` дает нам снова 1 и обещание вычислить `ones`, и так далее.

Можно делать и более интересные вещи с помощью операций вроде `addstreams`, которая порождает поэлементную сумму двух данных потоков[62]:

```
(define (add-streams s1 s2)
  (stream-map + s1 s2))
```

Теперь можно определить поток целых чисел следующим образом:

```
(define integers (cons-stream 1 (add-streams ones integers)))
```

Здесь `integers` определяются как поток, в котором первый элемент 1, а остаток равен сумме `ones` и `integers`. Таким образом, второй элемент `integers` равен 1 плюс первый элемент `integers`, то есть 2; третий элемент равен 1 плюс второй элемент `integers`, то есть 3, и так далее. Это определение работает потому, что в любой момент сгенерировано достаточно элементов потока `integers`, чтобы мы могли обратиться к ним в определении и породить следующий элемент.

В том же стиле можно определить числа Фибоначчи:

[62] Здесь используется обобщенная версия `stream-map` из упражнения 3.50.

```
(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (stream-cdr fibs)
                                         fibs))))
```

Это определение говорит, что `fibs` есть поток, начинающийся с 0 и 1, такой, что остаток потока порождается сложением `fibs` с собой самим, сдвинутым на одну позицию:

```
        1  1  2  3  5   8  13  21  ... = (stream-cdr fibs)
        0  1  1  2  3   5   8  13  ... = fibs
  ------------------------------------------------------

  0  1  1  2  3  5  8  13  21  34  ... = fibs
```

Еще одна полезная процедура для подобных определений потоков — `scalestream`. Она умножает каждый элемент потока на данную константу:

```
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor)) stream))
```

Например,

```
(define double (cons-stream 1 (scale-stream double 2)))
```

порождает поток степеней двойки: 1, 2, 4, 8, 16, 32 ...

Можно дать альтернативное определение потока простых чисел, начав с потока целых чисел, и фильтруя его через проверку на простоту. Вначале нам потребуется первое простое число, 2:

```
(define primes
  (cons-stream
   2
   (stream-filter prime? (integers-starting-from 3))))
```

Это определение не столь тривиально, как кажется, поскольку мы будем проверять число n на простоту, проверяя, делится ли n на простые числа (а не на все целые), меньшие или равные $\sqrt{n}$:

```
(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) true)
          ((divisible? n (stream-car ps)) false)
          (else (iter (stream-cdr ps)))))
  (iter primes))
```

Это рекурсивное определение, поскольку `primes` определяются посредством предиката `prime?`, а он сам использует поток `primes`. Работает эта процедура потому, что в любой момент имеется достаточно элементов потока `primes` для проверки на простоту следующего требуемого числа. А именно, при проверке `n` либо оказывается не простым (а в таком случае имеется уже сгенерированное простое число, на которое оно делится),

либо оно простое (а в таком случае, имеется уже сгенерированное простое число — то есть, простое число меньше n, — большее $\sqrt{n}$[63].

Упражнение 3.53.
Не запуская программу, опишите элементы потока, порождаемого

```
(define s (cons-stream 1 (add-streams s s)))
```

Упражнение 3.54.
Определите процедуру `mul-streams`, аналогичную `add-streams`, которая порождает поэлементное произведение двух входных потоков. С помощью нее и потока `integers` закончите следующее определение потока, n-й элемент которого (начиная с 0) равен факториалу $n+1$:

```
(define factorials (cons-stream 1 (mul-streams ⟨??⟩ ⟨??⟩)))
```

Упражнение 3.55.
Определите процедуру `partial-sums`, которая в качестве аргумента берет поток S, а возвращает поток, элементы которого равны $S_0, S_0+S_1, S_0+S_1+S_2, \ldots$. Например, `(partial-sums integers)` должно давать поток 1, 3, 6, 10, 15 ...

Упражнение 3.56.
Существует знаменитая задача, впервые сформулированная Р. Хэммингом: породить в возрастающем порядке и без повторений все положительные целые числа, у которых нет других простых делителей, кроме 2, 3 и 5. Очевидное решение состоит в том, чтобы перебирать все натуральные числа по очереди и проверять, есть ли у них простые множители помимо 2, 3 и 5. Однако эта процедура весьма неэффективна, поскольку чем больше числа, тем меньшая их доля соответствует условию. Применим альтернативный подход: назовем искомый поток чисел `S` и обратим внимание на следующие факты:

- `S` начинается с 1.
- Элементы `(scale-streams 2)` также принадлежат `S`
- То же верно и для `(scale-stream S 3)` и `(scale-stream S 5)`.
- Других элементов `S` нет.

Теперь требуется только соединить элементы из этих источников. Для этого мы определяем процедуру `merge`, которая сливает два упорядоченных потока в один упорядоченный поток, убирая при этом повторения:

```
(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (let ((s1car (stream-car s1))
               (s2car (stream-car s2)))
```

[63]Это тонкая деталь, которая основана на том, что $p_{n+1} \le p_n^2$ (Здесь p_k обозначает k-е простое число.) Такие оценки достаточно трудно доказать. Античное доказательство Евклида показывает, что имеется бесконечное количество простых чисел, и что $p_{n+1} \le p_1 p_2 \cdots p_n + 1$. Никакого существенно лучшего результата не было найдено до 1851 года, когда русский математик П. Л. Чебышев доказал, что для всех n, $p_{n+1} \le 2p_n$. Предположение, что это так, было высказано в 1845 году и известно как *гипотеза Бертрана* (Bertrand's hypothesis). Доказательство можно найти в разделе 22.3 в книге Hardy and Wright 1960.

```
(cond ((< s1car s2car)
       (cons-stream s1car (merge (stream-cdr s1) s2)))
      ((> s1car s2car)
       (cons-stream s2car (merge s1 (stream-cdr s2))))
      (else
       (cons-stream s1car
                    (merge (stream-cdr s1)
                           (stream-cdr s2)))))))))
```

Тогда требуемый поток можно получить с помощью `merge` таким образом:

```
(define S (cons-stream 1 (merge ⟨??⟩ ⟨??⟩)))
```

Заполните пропуски в местах, обозначенных знаком ⟨*??*⟩.

Упражнение 3.57.
Сколько сложений происходит при вычислении n-го числа Фибоначчи, в случае, когда мы используем определение $fibs$ через процедуру `add-streams`? Покажите, что число сложений выросло бы экспоненциально, если бы мы реализовали `(delay` ⟨*выражение*⟩`)` просто как `(lambda ()` ⟨*выражение*⟩`)`, без оптимизации через процедуру `memo-proc` из раздела 3.5.1[64].

Упражнение 3.58.
Дайте интерпретацию потоку, порождаемому следующей процедурой:

```
(define (expand num den radix)
  (cons-stream
   (quotient (* num radix) den)
   (expand (remainder (* num radix) den) den radix)))
```

(Элементарная процедура `quotient` возвращает целую часть частного двух целых чисел.) Каковы последовательные элементы потока, порожденного выражением `(expand 1 7 10)`? Что дает вычисление `(expand 3 8 10)`?

Упражнение 3.59.
В разделе 2.5.3 мы увидели, как реализовать систему арифметики многочленов, используя представление многочленов в виде списка термов. Подобным же образом можно работать со *степенными рядами* (power series), например

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3\cdot 2} + \frac{x^4}{4\cdot 3\cdot 2} + \cdots,$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4\cdot 3\cdot 2} - \cdots,$$

$$\sin x = x - \frac{x^3}{3\cdot 2} + \frac{x^5}{5\cdot 4\cdot 3\cdot 2} - \cdots,$$

представленными в виде бесконечных потоков. Будем представлять последовательность $a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots$ как поток, элементами которого являются коэффициенты $a_0, a_1, a_2, a_3 \ldots$

[64] Это упражнение показывает, как близко связан вызов по необходимости с обычной мемоизацией, описанной в упражнении 3.27. В этом упражнении мы при помощи присваивания явным образом создавали локальную таблицу. Наша оптимизация с вызовом по необходимости, в сущности, автоматически создает такую же таблицу, сохраняя значения в уже размороженных частях потока.

а. Интеграл последовательности $a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots$ есть последовательность

$$c + a_0x + \frac{1}{2}a_1x^2 + \frac{1}{3}a_2x^3 + \frac{1}{4}a_3x^4 + \cdots$$

где c — произвольная константа. Определите процедуру `integrate-series`, которая на входе принимает поток $a_0, a_1, a_2, \ldots$, представляющую степенной ряд, и возвращает поток $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \ldots$ коэффициентов при неконстантных членах интеграла последовательности. (Поскольку в результате отсутствует постоянный член, он не представляет собой степенной ряд; при использовании `integrate-series` мы через `cons` будем присоединять к началу соответствующую константу.)

б. Функция $x \mapsto e^x$ равна своей собственной производной. Отсюда следует, что e^x и интеграл e^x суть одна и та же последовательность, с точностью до постоянного члена, который равен $e^0 = 1$. Соответственно, можно породить последовательность для e^x через

```
(define exp-series
  (cons-stream 1 (integrate-series exp-series)))
```

Покажите, как породить последовательности для синуса и косинуса, опираясь на то, что производная синуса равна косинусу, а производная косинуса равна минус синусу:

```
(define cosine-stream
  (cons-stream 1 ⟨??⟩))

(define sine-series
  (cons-stream 0 ⟨??⟩))
```

Упражнение 3.60.
Если степенной ряд представляется в виде потока своих коэффициентов, как в упражнении 3.59, то сумма последовательностей реализуется посредством `add-streams`. Завершите определение следующей процедуры для перемножения последовательностей:

```
(define (mul-series s1 s2)
   (cons-stream ⟨??⟩ (add-streams ⟨??⟩ ⟨??⟩)))
```

Можете проверить свою процедуру, убедившись, что $\sin^2 x + \cos^2 x = 1$ с помощью последовательностей из упражнения 3.59.

Упражнение 3.61.
Пусть S будет степенным рядом (упражнение 3.59 с постоянным членом 1. Предположим, что мы хотим найти степенной ряд $1/S$, то есть такой ряд X, что $S \cdot X = 1$. Запишем $S = 1 + S_R$, где S_R — часть S после постоянного члена. Тогда мы можем решить уравнение для X так:

$$\begin{aligned} S \cdot X &= 1 \\ (1 + S_R) \cdot X &= 1 \\ X + S_R \cdot X &= 1 \\ X &= 1 - S_R \cdot X \end{aligned}$$

Другими словами, X есть степенной ряд с постоянным членом 1, чьи члены с более высокими степенями определяются как минус произведение S_R и X. Воспользовавшись этим, напишите процедуру `invert-unit-series`, которая вычисляет $1/S$ для степенного ряда S с постоянным членом 1. Вам потребуется `mul-series` из упражнения 3.60.

Упражнение 3.62.
При помощи результатов упражнений 3.60 и 3.61 определите процедуру `div-series`, которая делит один степенной ряд на другой. `Div-series` должна работать для любых двух рядов, при условии, что ряд в знаменателе начинается с ненулевого постоянного члена. (Если в знаменателе постоянный член равен нулю, `div-series` должна сообщать об ошибке.) Покажите, как при помощи `div-series` и результата упражнения 3.59 получить степенной ряд для тангенса.

3.5.3. Использование парадигмы потоков

Потоки с задержкой вычисления могут служить мощным инструментом моделирования. Они дают многие из преимуществ, обычно предоставляемых внутренним состоянием и присваиванием. Более того, они избегают некоторых из теоретических неудобств, связанных с введением присваивания в язык программирования.

Потоковый метод может изменять взгляд на вещи, так как он позволяет строить системы с другими границами модулей, не такими, как в системах, основанных на присваивании переменным состояния. Например, можно сосредоточивать внимание на всей временной последовательности (или сигнале), а не на значении переменных состояния в отдельные моменты. Оказывается удобно сочетать и сравнивать параметры состояния в различные моменты времени.

Итерация как потоковый процесс

В разделе 1.2.1 мы ввели понятие итеративного процесса, по мере исполнения изменяющего переменные состояния. Теперь мы узнали, что можно представлять состояние в виде «вневременного» потока значений, а не набора обновляемых переменных. Давайте примем этот взгляд и заново рассмотрим процедуру поиска квадратного корня из раздела 1.1.7. Напомним, что идея процедуры состояла в том, чтобы порождать последовательность все лучших и лучших приближений к квадратному корню x, снова и снова применяя процедуру улучшения гипотезы:

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
```

В исходной процедуре `sqrt` эти гипотезы были последовательными значениями переменной состояния. Вместо этого можно породить бесконечный поток гипотез, в голове которого стоит начальная гипотеза 1[65]:

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
                 (stream-map (lambda (guess)
                               (sqrt-improve guess x))
                             guesses)))
  guesses)

(display-stream (sqrt-stream 2))
```

[65] Внутреннюю переменную `guesses` нельзя связать с помощью `let`, поскольку значение `guesses` зависит от нее самой. В упражнении 3.63 рассматривается вопрос, зачем здесь нужна внутренняя переменная.

```
1
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
...
```

Можно порождать все больше элементов потока, получая все лучшие приближения. Если нужно, можно написать процедуру, которая бы порождала гипотезы до тех пор, пока ответ не окажется достаточно хорош. (См. упражнение 3.64.)

Еще один итеративный процесс, который можно рассматривать подобным образом — аппроксимация числа π, основанная на знакочередующемся ряде, упомянутом в разделе 1.3.1:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Сначала мы порождаем поток элементов ряда (числа, обратные нечетным натуральным, с чередующимся знаком). Затем мы берем поток сумм все большего количества элементов (при помощи процедуры `partial-sums` из упражнения 3.55) и домножаем результат на 4:

```
(define (pi-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (pi-summands (+ n 2)))))

(define pi-stream
  (scale-stream (partial-sums (pi-summands 1)) 4))

(display-stream pi-stream)
4.
2.666666666666667
3.466666666666667
2.8952380952380956
3.3396825396825403
2.9760461760461765
3.2837384837384844
3.017071817071818
...
```

Получается поток все более точных приближений к π, но сходятся эти приближения довольно медленно. Восемь членов последовательности поместили π между 3.284 и 3.017.

Пока что подход с потоком состояний не слишком отличается от потока с переменными состояния. Однако потоки дают нам возможность проделывать некоторые интересные трюки. Например, поток можно преобразовать с помощью *ускорителя последовательности* (sequence accelerator), преобразующего последовательность приближений в новую последовательность, которая сходится к тому же значению, что и исходная, но быстрее.

Один такой ускоритель, открытый швейцарским математиком восемнадцатого века Леонардом Эйлером, хорошо работает с последовательностями частичных сумм знакочередующихся рядов (рядов, знаки элементов которых чередуются). По методу Эйлера,

если S_n есть n-й член исходного ряда, то ускоренная последовательность имеет элементы

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Таким образом, если исходная последовательность представлена как поток значений, преобразованная последовательность дается процедурой

```
(define (euler-transform s)
  (let ((s0 (stream-ref s 0))    ; S_{n-1}
        (s1 (stream-ref s 1))    ; S_n
        (s2 (stream-ref s 2)))   ; S_{n+1}
    (cons-stream (- s2 (/ (square (- s2 s1))
                          (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))
```

Можно продемонстрировать ускорение Эйлера на нашей последовательности приближений к π:

```
(display-stream (euler-transform pi-stream))
3.166666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

Более того, можно ускорить ускоренную последовательность, рекурсивно ускорить результат, и так далее. То есть, можно создать поток потоков (структуру, которую мы будем называть *табло* (tableau)), в котором каждый поток есть результат преобразования предыдущего:

```
(define (make-tableau transform s)
  (cons-stream s
               (make-tableau transform
                             (transform s))))
```

Табло имеет вид

$$\begin{array}{cccccc} s_{00} & s_{01} & s_{02} & s_{03} & s_{04} & \dots \\ & s_{10} & s_{11} & s_{12} & s_{13} & \dots \\ & & s_{20} & s_{21} & s_{22} & \dots \\ & & & & \dots & \end{array}$$

Наконец, можно построить последовательность, членами которой будут первые элементы каждой строки табло:

```
(define (accelerated-sequence transform s)
  (stream-map stream-car
              (make-tableau transform s)))
```

Можно показать, как работает такое «сверхускорение» на последовательности приближений к π:

```
(display-stream (accelerated-sequence euler-transform
                                      pi-stream))
4.
3.166666666666667
3.142105263157895
3.141599357319005
3.1415927140337785
3.1415926539752927
3.1415926535911765
3.141592653589778
...
```

Результат впечатляет. Восемь членов последовательности дают нам верное значение π с точностью до 14 десятичных знаков. Если бы у нас была только исходная последовательность приближений к π, то пришлось бы вычислить порядка 10^{13} ее элементов (то есть довести последовательность до такого места, где ее элементы становятся меньше 10^{-13}), чтобы добиться такой точности!

Все эти методы ускорения можно было бы реализовать и без помощи потоков. Однако формулировка в терминах потоков обладает особым удобством и изяществом, поскольку мы имеем доступ ко всей последовательности состояний в виде структуры данных, с которой можно работать при помощи единого набора операций.

Упражнение 3.63.
Хьюго Дум спрашивает, почему нельзя было написать `sqrt-stream` более простым способом, без внутренней переменной `guesses`:

```
(define (sqrt-stream x)
  (cons-stream 1.0
               (stream-map (lambda (guess)
                             (sqrt-improve guess x))
                           (sqrt-stream x))))
```

Лиза П. Хакер отвечает, что эта версия процедуры значительно менее эффективна, поскольку производит избыточные вычисления. Объясните Лизин ответ. Сохранилось бы отличие в эффективности, если бы реализация `delay` использовала только `(lambda ()` ⟨*выражение*⟩`)`, без оптимизации через `memo-proc` (см. раздел 3.5.1)?

Упражнение 3.64.
Напишите процедуру `stream-limit`, которая в качестве аргумента принимает поток и число (погрешность). Она должна просматривать поток, пока не найдется два элемента подряд, различающихся меньше, чем на погрешность, и возвращать второй из этих элементов. При помощи этой процедуры можно будет вычислять квадратные корни с заданной точностью так:

```
(define (sqrt x tolerance)
  (stream-limit (sqrt-stream x) tolerance))
```

Упражнение 3.65.
С помощью ряда

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$$

породите три последовательности приближений к натуральному логарифму 2, так же, как мы выше сделали это для π. Как быстро сходятся эти последовательности?

Бесконечные потоки пар

В разделе 2.2.3 мы видели, как парадигма работы с последовательностями рассматривает вложенные циклы традиционной парадигмы в виде процессов, определенных на последовательности пар. Если мы обобщим этот метод на бесконечные потоки, то сможем писать программы, которые трудно воспроизвести с помощью обычных циклов, поскольку «цикл» охватывает бесконечное множество.

Например, пусть нам хочется обобщить процедуру `sum-of-primes` из раздела 2.2.3 так, чтобы получился поток из *всех* пар натуральных чисел (i, j), таких, что $i \leq j$ и $i + j$ простое. Если `int-pairs` есть последовательность всех пар натуральных чисел (i, j), где $i \leq j$, то необходимый нам поток таков[66]:

```
(stream-filter (lambda (pair)
                 (prime? (+ (car pair) (cadr pair))))
               int-pairs)
```

Задача, следовательно, состоит в том, чтобы породить поток `int-pairs`. В более общем случае допустим, что у нас есть два потока $S = (S_i)$ и $T = (T_j)$, и представим себе бесконечную матрицу

$$\begin{array}{cccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ (S_1, T_0) & (S_1, T_1) & (S_1, T_2) & \ldots \\ (S_2, T_0) & (S_2, T_1) & (S_2, T_2) & \ldots \\ \ldots & & & \end{array}$$

Нам хочется породить поток, который содержит все пары из этой матрицы, лежащие на диагонали или выше, а именно пары

$$\begin{array}{cccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ & (S_1, T_1) & (S_1, T_2) & \ldots \\ & & (S_2, T_2) & \ldots \\ & & & \ldots \end{array}$$

(Если мы возьмем и S, и T равными потоку натуральных чисел, то получим как раз необходимый нам поток `int-pairs`.)

Назовем общий поток пар `(pairs S T)`, и будем считать, что он состоит из трех частей: пары (S_0, T_0), остатка пар в первом ряду, и всех остальных пар[67]:

$$\begin{array}{c|ccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ \hline & (S_1, T_1) & (S_1, T_2) & \ldots \\ & & (S_2, T_2) & \ldots \\ & & & \ldots \end{array}$$

[66] Как и в разделе 2.2.3, мы представляем пару натуральных чисел в виде списка, а не лисповской пары.
[67] В упражнении 3.68 объясняется, почему мы выбрали именно такую декомпозицию.

Заметим, что третья часть этой декомпозиции (пары, не лежащие в первом ряду) суть пары, получаемые (рекурсивно) из `(stream-cdr S)` и `(stream-cdr T)`. Заметим также, что вторая часть (остаток первого ряда) есть

```
(stream-map (lambda (x) (list (stream-car s) x))
            (stream-cdr t))
```

Таким образом, мы можем сформировать наш поток пар так:

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (⟨как-нибудь смешать⟩
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

Чтобы закончить определение процедуры, нужно выбрать какой-нибудь способ смешать два внутренних потока. В голову приходит воспользоваться потоковым аналогом процедуры `append` из раздела 2.2.1:

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

Однако эта идея не срабатывает с бесконечными потоками, поскольку, прежде чем перейти ко второму потоку, нужно пройти весь первый поток до конца. В частности, если мы попробуем породить все пары натуральных чисел при помощи

```
(pairs integers integers)
```

то получившийся поток сначала попытается перечислить все пары, где первый элемент равен 1, а следовательно, никогда не породит ни одной пары с другим значением первого члена.

Для работы с бесконечными потоками требуется придумать способ смешения, который гарантировал бы, что каждый элемент будет достигнут, если программе дать достаточно времени. Изящный способ добиться этого состоит в том, чтобы воспользоваться следующей процедурой `interleave`[68]:

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

[68]Точная формулировка требования, которому должен удовлетворять порядок слияния, выглядит так: должна существовать функция от двух аргументов f, такая, что пара, соответствующая i-му элементу первого потока и j-му элементу второго, появится в качестве элемента выходного потока под номером $f(i, j)$. Трюк с чередованием через `interleave` нам показал Дэвид Тёрнер, который использовал его в языке KRC (Turner 1981).

Поскольку `interleave` чередует элементы из двух потоков, всякий элемент второго потока рано или поздно попадет в смешанный поток, даже если первый поток бесконечен.

Таким образом, мы можем породить требуемый поток пар так:

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

Упражнение 3.66.
Рассмотрим поток `(pairs integers integers)` Можете ли Вы что-то сказать о порядке, в котором пары попадают в поток? Например, сколько приблизительно пар предшествуют паре (1, 100)? Паре (99, 100)? (100, 100)? (Если Вы способны предоставить точные математические утверждения, — прекрасно. Однако если Вы увязаете в деталях, достаточно качественных оценок.)

Упражнение 3.67.
Измените процедуру так, чтобы `(pairs integers integers)` порождало поток из *всех* пар натуральных чисел (i, j), без дополнительного условия $i \leq j$. Подсказка: потребуется примешать еще один поток.

Упражнение 3.68.
Хьюго Дум считает, что построение потока пар из трех частей — процедура слишком сложная. Он предлагает вместо того, чтобы отделять пару (S_0, T_0), работать с первой строкой целиком:

```
(define (pairs s t)
  (interleave
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (pairs (stream-cdr s) (stream-cdr t))))
```

Будет ли такой код работать? Посмотрите, что произойдет, если мы попытаемся вычислить `(pairs integers integers)`, используя определение Хьюго.

Упражнение 3.69.
Напишите процедуру `triples`, которая берет три бесконечных потока S, T и U, и порождает поток троек (S_i, T_j, U_k), таких, что $i \leq j \leq k$. С помощью `triples` породите поток всех Пифагоровых троек натуральных чисел, т. е. таких троек (i, j, k), что $i \leq j$ и $i^2 + j^2 = k^2$

Упражнение 3.70.
Интересно было бы уметь порождать потоки в каком-либо полезном порядке, а не в порядке, задаваемом к случаю придуманным процессом чередования. Можно воспользоваться методом, подобным процедуре `merge` из упражнения 3.56, если мы определим способ сказать, что одна пара целых чисел «меньше» другой. Один из способов состоит в том, чтобы определить «функцию взвешивания» $W(i, j)$ и постановить, что (i_1, j_1) меньше, чем (i_2, j_2), если $W(i_1, j_1) \leq W(i_2, j_2)$. Напишите процедуру `merge-weighted`, которая во всем подобна `merge`, но только в качестве

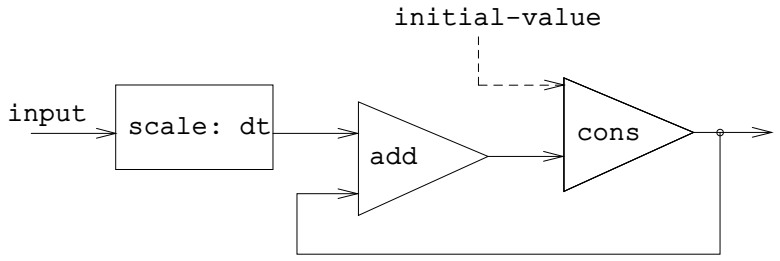


Рис. 3.32. Процедура `integral` в виде системы преобразования сигналов

дополнительного аргумента принимает процедуру `weight`, которая вычисляет вес пары, и используется для определения порядка, в котором элементы должны появляться в получающемся смешанном потоке[69]. При помощи `merge-weighted` напишите процедуру `weighted-pairs`, обобщающую `pairs`. Она должна принимать два потока и процедуру, вычисляющую функцию взвешивания, и порождать поток пар, упорядоченных по весу. Породите, используя эту процедуру:

а. Поток всех пар натуральных чисел (i, j) где $i \leq j$, упорядоченных по сумме $i + j$.

б. поток всех пар натуральных чисел (i, j), где $i \leq j$, ни i, ни j не делится ни на 2, ни на 3, ни на 5, и пары упорядочены по значению суммы $2i + 3j + 5ij$.

Упражнение 3.71.
Числа, которые можно выразить в виде суммы двух кубов более, чем одним способом, иногда называют *числами Рамануджана* (Ramanujan numbers), в честь математика Шринивасы Рамануджана[70]. Упорядоченные потоки пар предлагают изящное решение для задачи порождения таких чисел. Чтобы найти число, которое можно двумя разными способами записать в виде суммы двух кубов, требуется только породить поток пар натуральных чисел (i, j), взвешенных согласно сумме $i^3 + j^3$ (см. упражнение 3.70), и искать в этом потоке две пары подряд с одинаковым весом. Напишите процедуру для порождения чисел Рамануджана. Первое такое число 1729. Каковы следующие пять?

Упражнение 3.72.
Используя метод, подобный описанному в упражнении 3.71, породите поток всех чисел, которые можно записать как сумму двух квадратов тремя различными способами (и покажите, каковы эти способы).

Потоки как сигналы

Мы начали обсуждение потоков с того, что описали их как вычислительные аналоги «сигналов» в системах обработки сигналов. На самом деле с помощью потоков такие

[69] Мы будем требовать от функции взвешивания, чтобы вес пары возрастал при движении вправо по строке или вниз по столбцу в матрице пар.

[70] Цитата из некролога на смерть Рамануджана, написанного Г. Х. Харди (Hardy 1921): «Кажется, это мистер Литлвуд заметил, что «каждое натуральное число было ему другом». Я помню, как однажды навестил его, когда он лежал больной в Путни. Я приехал в такси номер 1729, сказал, что число показалось мне скучным, и выразил надежду, что это не было несчастливым знаком. «Нет, — ответил он, — это очень интересное число; это наименьшее число, которое можно двумя различными способами выразить как сумму двух кубов». Трюк с использованием взвешенных пар для порождения чисел Рамануджана нам показал Чарльз Лейзерсон.

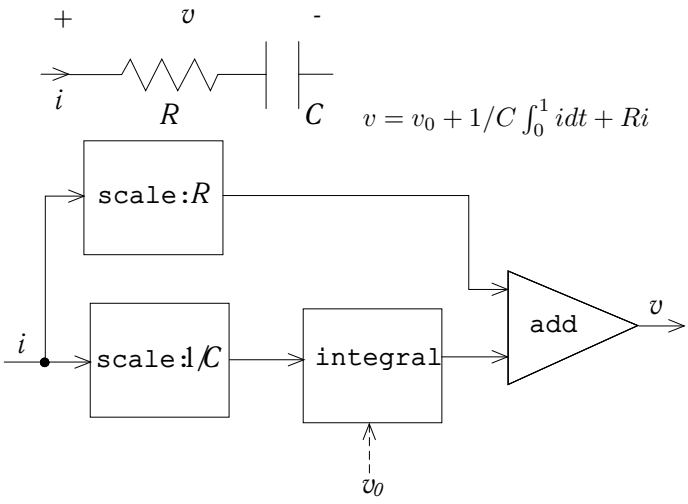


Рис. 3.33. RC-цепь и связанная с ней диаграмма потока сигналов.

системы можно моделировать самым непосредственным образом, представляя значения сигнала в последовательные моменты времени как последовательные элементы потока. Например, можно реализовать *интегратор* (integrator), или *сумматор* (summer), который, для входного потока $x = (x_i)$, начального значения C и малого приращения времени dt, собирает сумму

$$S_i = C + \sum_{j=1}^{i} x_j dt$$

и возвращает поток значений $S = (S_i)$. Следующая процедура `integral` напоминает «неявное» определение потока целых (раздел 3.5.2):

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int)))
  int)
```

На рисунке 3.32 показана система преобразования сигналов, соответствующая процедуре `integral`. Входной поток делится на отрезки dt и пропускается через сумматор, а вывод сумматора опять направляется на его вход. Ссылка на самого себя в определении `int` отражена на диаграмме в виде цикла обратной связи, соединяющего выход сумматора с одним из его входов.

Упражнение 3.73.
Можно моделировать электрические цепи с помощью потоков, представляющих значения тока или напряжения в определенные моменты времени. Допустим, например, что у нас имеется *цепь RC*

(RC circuit), состоящая из резистора с сопротивлением R и конденсатора емкостью C, соединенных последовательно. Значение напряжения v в зависимости от заданного тока i определяется формулой, показанной на рис. 3.33. Структура формулы показана на прилагаемой диаграмме потока сигналов.

Напишите процедуру `RC`, моделирующую эту цепь. На входе `RC` должна получать значения R, C и dt, и выдавать процедуру, которая принимает на входе поток значений тока i и начальное значение напряжения v_0, а на выходе выдает поток значений напряжения v. Например, у Вас должна быть возможность смоделировать при помощи `RC` RC-цепь с $R = 5$ ом, $C = 1$ фараде, и временным шагом в 0,5 секунды, вычислив `(define RC1 (RC 5 1 0.5))`. Здесь `RC1` определяется как процедура, которая принимает на входе поток, представляющий временную последовательность токов, и исходное напряжение на конденсаторе, а на выходе дает временной поток напряжений.

Упражнение 3.74.
Лиза П. Хакер разрабатывает систему для обработки сигналов, приходящих от физических сенсоров. Один из важных инструментов, который она хочет построить, — это сигнал, описывающий *переходы входного сигнала через ноль* (zero crossings). Выходной сигнал должен равняться +1, когда сигнал на входе меняется с отрицательного на положительный, -1, когда сигнал меняется с положительного на отрицательный, и 0 в остальных случаях. (Допустим, что знак нулевого входа положителен). Например, типичный входной сигнал и связанный с ним сигнал перехода через ноль могут выглядеть так:

$$\begin{array}{l} \ldots 1 \;\; 2 \;\; 1.5 \;\; 1 \;\; 0.5 \;\; -0.1 \;\;\; -2 \;\; -3 \;\; -2 \;\; -0.5 \;\; 0.2 \;\; 3 \;\; 4 \;\; \ldots \\ \ldots \;\; 0 \;\; 0 \;\;\; 0 \;\; 0 \;\;\; 0 \;\;\;\; -1 \;\;\; 0 \;\;\; 0 \;\;\; 0 \;\;\;\; 0 \;\;\; 1 \;\; 0 \;\; 0 \ldots \end{array}$$

В Лизиной системе сигнал от сенсора представляется как поток `sense-data`, а `zero-crossings` представляет соответствующий поток пересечений нуля. Для начала Лиза пишет процедуру `sign-change-detector`, которая берет два значения в качестве аргументов и, сравнив их знаки, выдает 0, 1 или -1. Затем она строит поток переходов через ноль следующим образом:

```
(define (make-zero-crossings input-stream last-value)
  (cons-stream
   (sign-change-detector (stream-car input-stream) last-value)
   (make-zero-crossings (stream-cdr input-stream)
                        (stream-car input-stream))))

(define zero-crossings (make-zero-crossings sense-data 0))
```

Мимо проходит Лизина начальница Ева Лу Атор и замечает, что программа приблизительно равносильна следующей, написанной с использованием обобщенной версии `stream-map` из упражнения 3.50:

```
(define zero-crossings
  (stream-map sign-change-detector sense-data ⟨выражение⟩))
```

Завершите программу, вставив необходимое ⟨*выражение*⟩.

Упражнение 3.75.
К сожалению, Лизин детектор перехода через ноль из упражнения 3.74 оказывается недостаточным, потому что зашумленный сигнал от сенсоров приводит к ложным срабатываниям. Инженер-электронщик Дайко Поправич предлагает Лизе сгладить сигнал, чтобы отфильтровать шум, прежде, чем отлавливать пересечение нуля. Лиза принимает его совет и решает извлечь переходы

через ноль из сигнала, полученного взятием среднего арифметического каждого значения входных данных с предыдущим значением. Она объясняет задачу своему помощнику Хьюго Думу, и тот пытается реализовать идею, поправив Лизин текст следующим образом:

```
(define (make-zero-crossings input-stream last-value)
  (let ((avpt (/ (+ (stream-car input-stream) last-value) 2)))
    (cons-stream (sign-change-detector avpt last-value)
                 (make-zero-crossings (stream-cdr input-stream)
                                      avpt))))
```

Этот код неверно реализует замысел Лизы. Найдите ошибку, внесенную Хьюго, и исправьте ее, не меняя структуру программы. (Подсказка: придется увеличить число аргументов `make-zero-crossings`.)

Упражнение 3.76.
Ева Лу Атор недовольна подходом Хьюго из упражнения 3.75. Написанная им программа не модульна, поскольку смешивает операции сглаживания и отлова пересечений ноля. Например, тест на пересечение не должен изменяться, если Лизе удастся найти другой способ улучшить качество входного сигнала. Помогите Хьюго и напишите процедуру `smooth`, которая берет на входе поток, а на выходе выдает поток, элементы которого получены усреднением каждых двух последовательных элементов входного потока. Затем используйте `smooth` как компоненту и реализуйте детектор перехода через ноль в более модульном стиле.

3.5.4. Потоки и задержанное вычисление

Процедура `integral` в конце предыдущего раздела показывает, как с помощью потоков можно моделировать системы обработки сигналов, которые содержат циклы обратной связи. Цикл обратной связи для сумматора, показанный на рис. 3.32, моделируется тем, что внутренний поток `int` в процедуре `integral` определяется с использованием себя самого:

```
(define int
  (cons-stream initial-value
               (add-streams (scale-stream integrand dt)
                            int)))
```

Способность интерпретатора работать с таким косвенным определением зависит от `delay`, встроенного в `cons-stream`. Без этой задержки интерпретатор не мог бы построить `int`, не вычислив оба аргумента `cons-stream`, а для этого нужно, чтобы `int` уже был определен. В общем случае, `delay` играет ключевую роль, когда мы моделируем системы обработки сигналов с обратной связью при помощи потоков. В отсутствие задержки нам приходилось бы формулировать модели так, чтобы вход всякого обрабатывающего блока полностью вычислялся, прежде чем блок выдает что-либо на выходе. Такое условие исключает циклы.

К сожалению, потоковые модели систем с циклами могут требовать применения задержек помимо той, которая «спрятана» в `cons-stream`. Например, на рисунке 3.34 показана система обработки сигналов, решающая дифференциальное уравнение $dy/dt = f(y)$, где f — заданная функция. На рисунке показан отображающий блок, который применяет f ко входному сигналу, связанный в цикл обратной связи с интегратором. Это

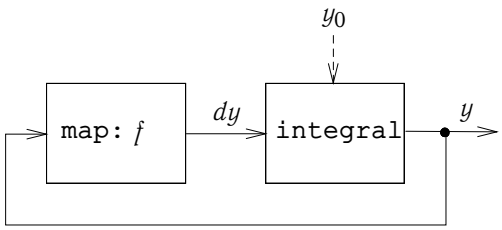


Рис. 3.34. «Аналоговая компьютерная цепь», которая решает уравнение $dy/dt = f(y)$.

очень похоже на работу аналоговых схем, действительно используемых для решения такого рода уравнений.

Если нам дано начальное значение y_0, мы могли бы попытаться смоделировать эту систему с помощью процедуры

```
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y))
  y)
```

Эта процедура не работает, потому что вызов `integral` в первой строке `solve` требует, чтобы был определен входной поток `dy`, а это происходит только во второй строке процедуры `solve`.

С другой стороны, замысл, заключенный в этом определении, вполне здрав, поскольку мы можем, в принципе, начать порождать поток `y` и не зная `dy`. Действительно, `integral` и многие другие операции над потоками обладают свойствами, подобными `cons-stream`, а именно, мы можем породить часть ответа, даже если нам дана только частичная информация об аргументах. В случае `integral`, первый элемент выходного потока есть указанное начальное значение `initial-value`. Таким образом, можно породить первый элемент выходного потока и не вычисляя интегрируемую величину `dy`. А раз мы знаем первый элемент `y`, то `stream-map` во второй строке `solve` может начать работать и породить первый элемент `dy`, а с его помощью мы получим второй элемент `y`, и так далее.

Чтобы воспользоваться этой идеей, переопределим `integral` так, чтобы он ожидал интегрируемый поток в виде *задержанного аргумента* (delayed argument). `Integral` будет размораживать вычисление входного потока через `force` только тогда, когда ему нужно породить элементы входного потока помимо первого:

```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (let ((integrand (force delayed-integrand)))
                   (add-streams (scale-stream integrand dt)
                                int))))
   int)
```

Теперь можно реализовать процедуру `solve`, задержав вычисление `dy` внутри определения `y`[71]:

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Теперь при любом вызове `integral` необходимо задерживать интегрируемый аргумент. Можно показать, что процедура `solve` работает, аппроксимируя $e \approx 2.718$ вычислением в точке $y = 1$ решения дифференциального уравнения $dy/dt = y$ с начальным условием $y(0) = 1$:

```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000)
2.716924
```

Упражнение 3.77.
Вышеприведенная процедура `integral` была аналогична «непрямому» определению бесконечного потока натуральных чисел из раздела 3.5.2. В виде альтернативы можно дать определение `integral`, более похожее на `integers-starting-from` (также в разделе 3.5.2):

```
(define (integral integrand initial-value dt)
  (cons-stream initial-value
               (if (stream-null? integrand)
                   the-empty-stream
                   (integral (stream-cdr integrand)
                             (+ (* dt (stream-car integrand))
                                initial-value)
                             dt))))
```

В системах с циклами эта реализациея порождает такие же проблемы, как и наша исходная версия `integral`. Модифицируйте процедуру так, чтобы она ожидала `integrand` как задержанный аргумент, а следовательно, могла быть использована в процедуре `solve`.

Упражнение 3.78.
Рассмотрим задачу проектирования системы обработки сигналов для решения гомогенных линейных дифференциальных уравнений второго порядка

$$\frac{d^2y}{dt^2} - a\frac{dy}{dt} - by = 0$$

Выходной поток, моделирующий y, порождается сетью, содержащей цикл. Этот цикл возникает потому, что значение d^2y/dt^2 зависит от значений y и dy/dt, а они оба получаются интегрированием d^2y/dt^2. Диаграмма, которую нам хотелось бы закодировать, показана на рис. 3.35. Напишите процедуру `solve-2nd`, которая в качестве аргументов берет константы a, b и dt и начальные значения y_0 и dy_0 для y и dy, и порождает поток последовательных значений y.

[71] Не гарантируется, что эта процедура будет работать во всех реализациях Scheme, но для любой реализации должен найтись простой способ заставить подобную процедуру работать. Проблемы связаны с тонкими различиями в том, как реализации Scheme обрабатывают внутренние определения. (См. раздел 4.1.6.)

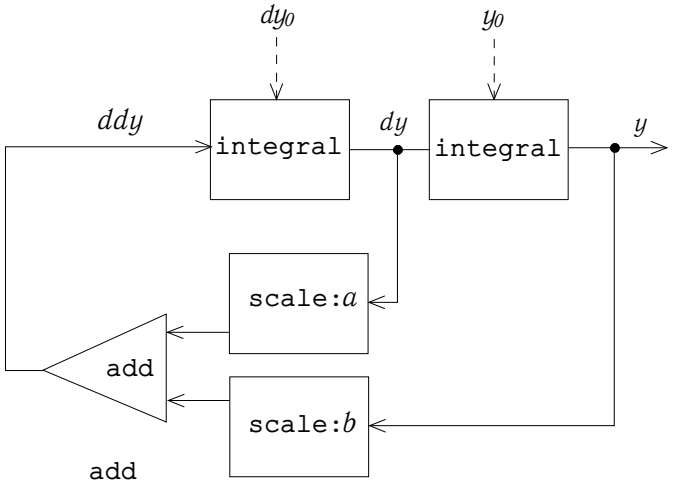


Рис. 3.35. Диаграмма потока сигналов для решения линейного дифференциального уравнения второго порядка.

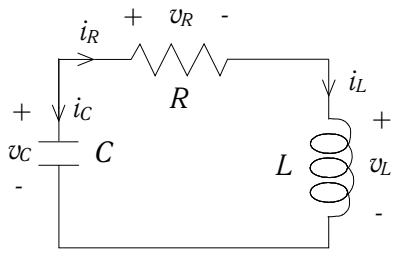


Рис. 3.36. Последовательная RLC-цепь

Упражнение 3.79.
Обобщите процедуру `solve-2nd` из упражнения 3.78 так, чтобы с ее помощью можно было решать дифференциальные уравнения второго порядка общего вида $d^2y/dy^2 = f(dy/dt, y)$.

Упражнение 3.80.
Последовательная RLC-цепь (series RLC circuit) состоит из резистора, конденсатора и катушки индуктивности, соединенных последовательно, как показано на рис. 3.36. Если сопротивление, индуктивность и емкость равны, соответственно, R, L и C, то отношения между напряжением v и током i на трех элементах описываются уравнениями

$$v_R = i_R R$$

$$v_L = L\frac{di_L}{dt}$$

$$i_C = C\frac{dv_C}{dt}$$

а цепь диктует соотношения

$$i_R = i_L = -i_C$$

$$v_C = v_L + v_R$$

Сочетание этих условий показывает, что состояние цепи (характеризуемое через v_C, напряжение на конденсаторе, и i_L, ток через катушку) описывается парой дифференциальных уравнений

$$\frac{dv_C}{dt} = -\frac{i_L}{C}$$

$$\frac{di_L}{dt} = \frac{1}{L}v_C - \frac{R}{L}i_L$$

Диаграмма потока сигналов, представляющая эту систему дифференциальных уравнений, показана на рисунке 3.37.

Напишите процедуру `RLC`, которая в качестве аргументов берет параметры цепи R, L и C и точность по времени dt. Подобно процедуре `RC` из упражнения 3.73, `RLC` должна порождать процедуру, которая берет начальные значения переменных состояния v_{C_0} и i_{L_0} и порождает (через `cons`) пару потоков состояния v_C и i_L. С помощью `RLC` породите пару потоков, которая моделирует поведение RLC-цепи с $K = 1$ ом, $C = 0.2$ фарад, $L = 1$ генри, $dt = 0.1$ секунды, и начальными значениями $i_{L_0} = 0$ ампер и $v_{C_0} = 10$ вольт.

Нормальный порядок вычислений

Примеры из этого раздела показывают, как явное использование `delay` и `force` сообщает программированию большую гибкость, однако те же самые примеры показывают, как наши программы от этого могут стать сложнее и запутаннее. Например, новая процедура `integral` позволяет моделировать системы с циклами, но теперь нам приходится помнить, что звать ее надо с задержанным аргументом, и все процедуры, которые пользуются `integral`, должны это знать. В результате мы создали два класса процедур: обычные и те, которым требуются задержанные аргументы. В общем случае создание новых классов процедур требует от нас еще и создания новых классов процедур высших порядков[72].

[72]Здесь мы получаем в Лиспе слабое отражение тех сложностей, которые возникают при работе с процедурами высших порядков в обыкновенных сильно типизированных языках вроде Паскаля. В таких языках

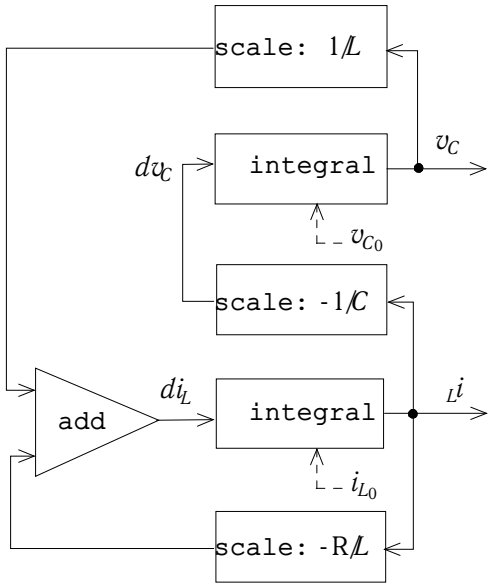


Рис. 3.37. Диаграмма потока сигналов для решения уравнений последовательной RLC–цепи.

Один из способов избежать необходимости вводить два класса процедур состоит в том, чтобы заставить все процедуры принимать задержанные аргументы. Можно принять модель вычислений, в которой все аргументы процедур автоматически задерживаются, и вынуждение происходит только тогда, когда их значения реально нужны (например, для выполнения элементарной операции). Таким образом наш язык станет использовать нормальный порядок вычислений, который мы впервые описали, когда разговор шел о подстановочной модели вычислений в разделе 1.1.5. Переход к нормальному порядку вычислений предоставляет нам изящный и единообразный способ упростить использование задержанных вычислений, и если бы нас интересовала только обработка потоков, было бы естественно принять эту стратегию. В разделе 4.2, после того, как мы изучим устройство вычислителя, мы увидим, как можно преобразовать язык именно таким способом. К сожалению, добавив задержки в вызовы процедур, мы совершенно лишили себя возможности строить программы, работа которых зависит от порядка событий,

---

программисту нужно указывать типы данных для аргументов и результата каждой процедуры: число, логическое значение, последовательность и т. д. Следовательно, мы не можем выразить такую абстракцию, как «применить данную процедуру `proc` ко всем элементам последовательности» в виде единой процедуры высшего порядка вроде `stream-map`. Вместо этого нам потребуется отдельная процедура для каждой комбинации типов аргументов и результата, которые можно указать для `proc`. Практическая поддержка понятия «тип данных» при наличии процедур высших порядков приводит ко многим интересным проблемам. Один из способов работы с ними иллюстрирует язык ML (Gordon, Milner, and Wadsworth 1979), в котором «полиморфные типы данных» включают шаблоны для преобразований между типами данных высшего уровня. Более того, для большинства процедур в ML типы данных явно не определяются программистом. Вместо этого в ML встроен механизм *вывода типов* (type inference), который при помощи контекстной информации вычисляет типы данных для вновь определяемых процедур.

то есть программы, использующие присваивание, изменяющие свои данные или производящие ввод-вывод. Одно-единственное использование `delay` в форме `cons-stream` уже может привести к неразберихе, как показано в упражнениях 3.51 и 3.52. Насколько известно, в языках программирования изменение состояния и задержанные вычисления плохо совместимы, и поиск возможностей использовать одновременно и то, и другое является активной областью исследований.

3.5.5. Модульность функциональных программ и модульность объектов

Как мы видели в разделе 3.1.2, одно из основных преимуществ от введения присваивания состоит в том, что мы можем повысить модульность своих систем при помощи инкапсуляции, или «сокрытия», частей большой системы во внутренних переменных. Потоковые модели могут предоставить нам такой же уровень модульности без использования присваивания. В качестве примера мы можем заново реализовать аппроксимацию π методом Монте-Карло, которую мы рассматривали в разделе 3.1.2, с точки зрения обработки потоков.

Главная задача при обеспечении модульности состояла в том, что нам хотелось спрятать внутреннее состояние генератора случайных чисел от программ, которые пользуются случайными числами. Мы начали с процедуры `rand-update`, последовательные значения которой служили для нас источником случайных чисел, и уже с ее помощью построили генератор случайных чисел:

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

При формулировке посредством потоков генератора случайных чисел как такового не существует, имеется только поток случайных чисел, полученных вызовами `rand-update`:

```
(define random-numbers
  (cons-stream random-init
               (stream-map rand-update random-numbers)))
```

С его помощью мы порождаем поток результатов испытаний Чезаро, проведенных на последовательных парах потока случайных чисел:

```
(define cesaro-stream
  (map-successive-pairs (lambda (r1 r2) (= (gcd r1 r2) 1))
                        random-numbers))

(define (map-successive-pairs f s)
  (cons-stream
   (f (stream-car s) (stream-car (stream-cdr s)))
   (map-successive-pairs f (stream-cdr (stream-cdr s)))))
```

Поток `cesaro-stream` подается на вход процедуре `monte-carlo`, которая порождает поток оценок вероятности. Затем этот результат преобразуется, и получается поток оценок значения π. В этой версии программы не требуется параметра, указывающего, сколько испытаний требуется проводить. Более точные оценки π (полученные при большем количестве испытаний) можно получить, дальше заглянув в поток `pi`:

```
(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
     (/ passed (+ passed failed))
     (monte-carlo
      (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1))))

(define pi
  (stream-map (lambda (p) (sqrt (/ 6 p)))
              (monte-carlo cesaro-stream 0 0)))
```

Такой подход достаточно модулен, поскольку мы по-прежнему имеем возможность сформулировать общую процедуру `monte-carlo`, работающую с произвольными испытаниями. Однако здесь нет ни присваивания, ни внутреннего состояния.

Упражнение 3.81.
В упражнении 3.6 обсуждалась возможность обобщить генератор случайных чисел и позволить пользователю сбрасывать последовательность случайных чисел, так, чтобы можно было порождать воспроизводимые «случайные» последовательности. Постройте потоковый вариант такой же процедуры-генератора, которая работает со входным потоком запросов вида `generate` — породить новое число, либо `reset` — сбросить последовательность в нужную точку, и которая порождает требуемый поток случайных чисел. Не используйте в своем решении присваивание.

Упражнение 3.82.
Переделайте на основе потоков упражнение 3.5 на интегрирование методом Монте-Карло. Потоковая версия процедуры `estimate-integral` не требует аргумента, который говорит, сколько проводить испытаний. Вместо этого она порождает поток оценок, основанных на все большем количестве испытаний.

Взгляд на время в функциональном программировании

Вернемся теперь к вопросам об объектах и изменении, поднятым в начале этой главы, и рассмотрим их в новом свете. Мы ввели присваивание и изменяемые объекты, чтобы иметь механизм для модульного построения программ, которые моделируют обладающие состоянием системы. Мы порождали вычислительные объекты с внутренними переменными состояния и изменяли эти объекты при помощи присваивания. Мы моделировали временнóе поведение объектов мира через временное поведение соответствующих вычислительных объектов.

Теперь мы видим, что потоки дают альтернативный способ моделирования объектов, обладающих внутренним состоянием. Можно моделировать изменяющуюся величину, например, внутреннее состояние какого-либо объекта, через поток, который представляет динамику изменений состояния. В сущности, с помощью потоков мы представляем время явно, так что время в моделируемом мире оказывается отделено от последовательности событий, происходящих во время вычисления. Действительно, благодаря наличию `delay` между имитируемым временем модели и последовательностью событий при вычислении может быть весьма мало общего.

Чтобы сопоставить эти два подхода к моделированию, рассмотрим еще раз «обработчик снятия денег», следящий за значением баланса на банковском счету. В разделе 3.1.3 мы реализовали упрощенную версию такой программы обработки:

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

Вызовы `make-simplified-withdraw` порождают вычислительные объекты, и каждый из них содержит внутреннюю переменную `balance`, которая уменьшается при каждом обращении к объекту. Этот объект принимает в качестве аргумента количество денег `amount`, а возвращает новый баланс. Можно представить себе, как пользователь банковского счета печатает последовательность входных данных для такого объекта и рассматривает на экране дисплея последовательность возвращаемых данных.

С другой стороны, можно смоделировать обработчик снятия денег и в виде процедуры, которая принимает на входе баланс и поток снимаемых сумм, а порождает поток последовательных балансов на счету:

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
   balance
   (stream-withdraw (- balance (stream-car amount-stream))
                    (stream-cdr amount-stream))))
```

`Stream-withdraw` реализует хорошо определенную математическую функцию, выход которой полностью определяется входом. Однако предположим, что вход `amount-stream` есть поток последовательных значений, вводимых пользователем, и что получающийся поток балансов выводится на печать. В таком случае, с точки зрения пользователя, который печатает значения и смотрит на результаты, потоковый процесс обладает тем же поведением, что и объект, созданный при помощи `make-simplified-withdraw`. Однако в потоковой версии нет ни присваивания, ни внутренней переменной состояния, и, следовательно, она не вызывает никаких теоретических сложностей из описанных в разделе 3.1.3. И все-таки система обладает состоянием!

Это достижение достойно внимания. Несмотря на то, что `stream-withdraw` реализует хорошо определенную математическую функцию, поведение которой не меняется, у пользователя создается впечатление, что он взаимодействует с системой, обладающей изменяющимся состоянием. Один из способов разрешить парадокс заключается в том, чтобы понять, что именно существование пользователя во времени навязывает системе состояние. Если бы пользователь мог принять более отстраненную точку зрения и

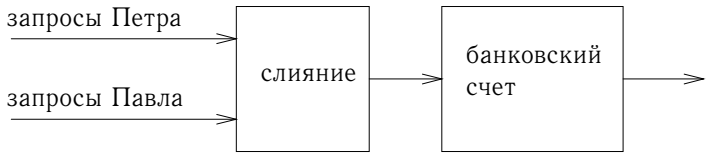


Рис. 3.38. Совместный банковский счет, смоделированный через слияние двух потоков событий-транзакций.

думать в терминах потоков и балансов, а не отдельных актов взаимодействия, система выглядела бы как объект без состояния[73].

С точки зрения одной части сложного процесса кажется, что другие его части меняются со временем. Они содержат скрытое изменчивое внутреннее состояние. Если мы хотим писать программы, моделирующие такой тип естественной декомпозиции нашего мира (как мы видим его со своей точки зрения, будучи частицами этого мира) при помощи структур в нашем компьютере, мы строим вычислительные объекты, не являющиеся функциональными, — они обязаны меняться со временем. Мы моделируем состояние при помощи внутренних переменных, и изменение состояния мы моделируем через присваивание этим переменным. Пойдя по этому пути, мы делаем время выполнения вычислительной модели временем мира, частью которого мы являемся, и так в нашем компьютере возникают «объекты».

Моделирование при помощи объектов — мощная и интуитивно понятная техника, во многом потому, что она соответствует восприятию взаимодействия с миром, частью которого мы являемся. Однако, как мы неоднократно видели в этой главе, в таких моделях возникают неудобные вопросы управления порядком событий и синхронизации множественных процессов. Возможность избежать этих проблем стимулировала развитие *функциональных языков программирования* (functional programming languages), в которых нет понятий присваивания и изменяемых данных. В таком языке все процедуры реализуют точно определенные математические функции, поведение которых не меняется. Функциональный подход весьма привлекателен при работе с параллельными системами[74].

С другой стороны, при более внимательном взгляде мы обнаружим, что и функциональные модели не избавляют от проблем, связанных со временем. Одна из самых болезненных возникает, когда нам нужно проектировать интерактивные системы, особенно такие, которые моделируют взаимодействие между независимыми сущностями. К примеру, рассмотрим еще раз реализацию банковской системы, которая позволяет иметь совместные счета. В традиционной системе с присваиванием и объектами информация о том, что у Петра и Павла есть общий счет, моделировалась бы тем, что и Петр, и

[73] Подобным образом в физике, когда мы наблюдаем за движением частицы, мы говорим, что позиция (состояние) частицы изменяется. Однако с точки зрения мировой линии частицы в пространстве-времени никакого изменения нет.

[74] Джон Бэкус, изобретатель Фортрана, привлек внимание к функциональному программированию, когда в 1978 году получил премию Тьюринга Американской Ассоциации по Вычислительным Машинам (ACM). В своей инаугурационной речи (Backus 1978) он горячо отстаивал функциональный подход. Хороший обзор функционального программирования дается в книгах Henderson 1980 и Darlington, Henderson, and Turner 1982.

Павел посылали бы заказы на транзакции одному и тому же объекту-банковскому счету, как мы видели в разделе 3.1.3. С точки зрения потоков, где «объекты» сами по себе не существуют, банковский счет, как мы уже указывали, может моделироваться в виде процесса, работающего с потоком заказов на транзакции и порождающего поток реакций. Соответственно, информация о том, что Петр и Павел совместно владеют счетом, может моделироваться путем смешения потока заказов Петра на транзакции с потоком Павла и направления слитого потока в процесс-поток банковского счета, как показано на рисунке 3.38.

Проблему в этой формулировке вызывает понятие *слияния* (merge). Неверным решением будет просто брать по очереди один заказ от Петра и один от Павла. Допустим, что Павел очень редко обращается к счету. Не следует заставлять Петра ждать, пока Павел обратится к счету, прежде чем он сможет осуществить вторую транзакцию. Как бы ни было реализовано слияние, оно должно чередовать потоки транзакций так, чтобы соответствовать «реальному времени» с точки зрения Петра и Павла, в том смысле, что если Петр и Павел встретятся, то они могут согласиться, что определенные транзакции произошли до встречи, а определенные после[75] Это в точности то же самое ограничение, с которым нам приходилось сталкиваться в разделе 3.4.1, где у нас возникла необходимость ввести явную синхронизацию, чтобы добиться «правильного» порядка событий при параллельной обработке объектов, обладающих состоянием. Таким образом, при попытке поддержать функциональный стиль необходимость сливать потоки ввода от различных агентов опять привносит те самые проблемы, от которых функциональный стиль должен был нас избавить.

В начале этой главы мы поставили цель научиться строить вычислительные модели, чья структура соответствует нашему восприятию реального мира, который мы моделируем. Мы можем моделировать мир либо как собрание ограниченных во времени взаимодействующих объектов, обладающих состоянием, либо же как единую, вневременную, лишенную состояния сущность. Каждая из этих точек зрения имеет свои преимущества, но ни одна из них не удовлетворяет нас полностью. Время великого объединения пока не настало[76].

---

[75]Заметим, что для любых двух потоков в принципе существует более одного возможного способа чередования. Так что с технической точки зрения «слияние» не функция, а отношение — ответ не является детерминистской функцией аргументов. Мы уже упоминали (в примечании 39), что недетерминизм имеет существенное значение при работе с параллельными процессами. Отношение слияния показывает тот же самый недетерминизм с функциональной точки зрения. В разделе 4.3 мы рассмотрим еще одну точку зрения на недетерминизм.

[76]Объектная модель строит приближенное описание мира, разделяя его на отдельные фрагменты. Функциональная модель не проводит границ модулей по границам объектов. Объектная модель полезна тогда, когда раздельное состояние «объектов» намного больше, чем состояние, общее для всех или некоторых из них. Примером области, где объектный взгляд не работает, является квантовая механика, где попытки думать об объектах как отдельных частицах ведут к парадоксам и недоразумениям. Объединение объектного взгляда с функциональным может иметь отношение не столько к программированию, сколько к фундаментальным вопросам эпистемологии.

Глава 4

Метаязыковая абстракция

> . . . Именно в словах кроется магия — в таких, как «абракадабра», «Сезам, откройся» и проч., — но магические слова из одной истории перестают быть таковыми в следующей. Настоящая магия состоит в том, чтобы понять, когда и для чего слово сработает; трюк в том, чтобы выучить трюк.
> . . . А слова эти состоят из букв нашего алфавита: пара дюжин закорючек, которые мы способны черкнуть пером. Вот где ключ. . . ! И сокровище тоже, если только мы сумеем его заполучить! Как будто. . . как будто ключ к сокровищу и *есть* само сокровище!
>
> Джон Барт.
> «Химера»
> (Перевод Виктора Лапицкого)

Исследуя науку проектирования программ, мы видели, что программисты-эксперты управляют сложностью своих программ при помощи тех же общих методик, какими пользуются проектировщики всех сложных систем. Они сочетают элементарные единицы, получая при этом составные объекты, с помощью абстракции составных объектов формируют строительные блоки высших порядков, и при этом с целью сохранения модульности выбирают наиболее удобный общий взгляд на структуру системы. Демонстрируя эти методы, мы использовали Лисп как язык для описания процессов и для построения вычислительных объектов данных, и процессы — для моделирования сложных явлений реального мира. Однако по мере того, как мы сталкиваемся со все более сложными задачами, мы обнаруживаем, что Лиспа, да и любого заранее заданного языка программирования, недостаточно для наших нужд. Чтобы эффективнее выражать свои мысли, постоянно приходится обращаться к новым языкам. Построение новых языков является мощной стратегией управления сложностью в инженерном проектировании; часто оказывается, что можно расширить свои возможности работы над сложной задачей, приняв новый язык, позволяющий нам описывать (а следовательно, и обдумывать) задачу новым способом, используя элементы, методы их сочетания и механизмы абстракции, специально подогнанные под стоящие перед нами проблемы[1].

[1] Та же самая идея встречается во всех областях техники. Например, у инженеров-электронщиков существует множество языков для описания схем. Два из них — это язык электрических *сетей* и язык электрических *систем*. Язык сетей делает акцент на физическом моделировании устройств в терминах дискретных электриче-

Программирование изобилует языками. Есть физические языки, например, языки машинных кодов для конкретных компьютеров. Основным вопросом для них является представление данных и управления через отдельные биты памяти и машинные команды. Пишущий программы на машинном языке озабочен тем, чтобы при помощи данной аппаратуры создать системы и инструменты для эффективной реализации вычисления при ограниченных ресурсах. Языки высокого уровня, возводимые поверх машинных, скрывают вопросы конкретной реализации данных в виде набора битов и представления программ как последовательности машинных команд. В этих языках присутствуют средства комбинации и абстракции, например определения функций, которые подходят для более крупномасштабной организации систем.

Метаязыковая абстракция (metalinguistic abstraction), то есть построение новых языков, играет важную роль во всех отраслях инженерного проектирования. Для компьютерного программирования она особенно важна, поскольку в программировании мы можем не только формулировать новые языки, но и реализовывать их через построение вычислителей. *Вычислитель* (evaluator) (или *интерпретатор* (interpreter)) для языка программирования — это процедура, которая, будучи примененной к выражению языка, производит действия, необходимые для вычисления этого выражения.

Без преувеличения можно сказать, что самая основополагающая идея в программировании такова:

> Вычислитель, который определяет значение выражений в языке программирования — всего лишь обычная программа.

С этой мыслью приходит и новое представление о себе самих: мы начинаем видеть в себе разработчиков языков, а не просто пользователей языков, придуманных другими.

На самом деле, почти любую программу можно рассматривать как вычислитель для какого-то языка. Например, система работы с многочленами из раздела 2.5.3 заключает в себе правила арифметики многочленов и реализует их в терминах операций над данными в списочной форме. Если мы дополним эту систему процедурами для чтения и печати многочленов, то перед нами окажется ядро специализированного языка для решения задач символьной математики. И программа моделирования цифровой логики из раздела 3.3.4, и программа распространения ограничений из раздела 3.3.5 содержат свои собственные языки, со своими примитивами, средствами их комбинирования и абстракции. С этой точки зрения, техника работы с крупномасштабными компьютерными системами сливается с техникой создания новых компьютерных языков, и вся информатика — не более (но и не менее), чем наука о построении подходящих языков описания.

Сейчас мы начинаем обзор методов, которые позволяют создавать одни языки на

ских элементов. Элементарными объектами этого языка являются элементарные электрические компоненты — резисторы, конденсаторы, катушки индуктивности и транзисторы, задаваемые через физические переменные: напряжение и ток. Описывая схемы на языке сетей, инженер озабочен физическими характеристиками своего проекта. Элементами системного языка, напротив, являются модули обработки сигнала, например, фильтры и усилители. Существенно только функциональное поведение модулей, и сигналами манипулируют безотносительно к тому, в виде какого напряжения или тока они реализуются физически. Язык систем построен на языке сетей, в том смысле, что элементы систем обработки сигнала состоят из электрических схем. Однако здесь инженера интересует крупномасштабная организация электрических устройств, решающая определенную задачу; их физическая совместимость подразумевается. Такая послойная организация языков служит еще одним примером уровневого метода проектирования, проиллюстрированного в разделе 2.2.4 на примере языка описания изображений.

основе других. В этой главе в качестве основы мы будем использовать Лисп, и вычислители будем реализовывать как процедуры на Лиспе. Лисп особенно хорошо подходит для этой задачи благодаря своей способности представлять символические выражения и обрабатывать их. Первый шаг к пониманию того, как реализуются языки, мы сделаем, построив вычислитель для самого Лиспа. Язык, реализуемый нашим интерпретатором, будет подмножеством диалекта Лиспа Scheme, которым мы пользуемся в этой книге. Несмотря на то, что интерпретатор, описанный в этой главе, написан для конкретного диалекта Лиспа, он содержит основную структуру вычислителя для любого языка, ориентированного на выражения и предназначенного для написания программ для последовательной машины. (На самом деле, глубоко внутри большинства языковых процессоров содержится маленький интерпретатор «Лиспа».) Этот интерпретатор несколько упрощен для удобства и наглядности обсуждения, и некоторые детали, которые важно было бы включить в Лисп-систему промышленного качества, здесь были оставлены за рамками изложения. Тем не менее, этот простой интерпретатор способен выполнить большинство программ из данной книги.[2]

Важное преимущество, которое нам дает вычислитель, доступный в виде программы на Лиспе, состоит в том, что мы можем реализовывать альтернативные правила вычисления, описывая их как модификации программы вычислителя. В частности, мы можем извлечь из этой способности немалую выгоду, добиваясь более полного контроля над тем, как в вычислительных моделях реализуется понятие времени. Этому вопросу была специально посвящена глава 3. Там мы смягчили некоторые сложности работы с состоянием и присваиваниями, при помощи потоков отделив представление времени во внешнем мире от времени внутри компьютера. Однако программы, работающие с потоками, иногда бывали излишне громоздки, поскольку их ограничивал аппликативный порядок вычисления, принятый в Scheme. В разделе 4.2 мы изменим язык и получим более изящный подход в виде интерпретатора с *нормальным порядком вычисления* (normal-order evaluation).

В разделе 4.3 язык меняется более радикально, и выражения получают не одно единственное значение, а множество. В этом языке *недетерминистских вычислений* (nondeterministic computing) становится естественным порождать все возможные значения выражения, а затем искать среди них те, которые удовлетворяют определенным ограничениям. Если описывать это в терминах вычисления и времени, то время как будто разветвляется на множество «возможных будущих», и мы ищем среди них подходящие временные линии. При работе с недетерминистским интерпретатором отслеживание множества значений и поиск осуществляются автоматически встроенными механизмами языка.

В разделе 4.4 мы реализуем язык *логического программирования* (logic programming), в котором знание выражается в терминах отношений, а не в терминах вычислений со входами и выходами. Несмотря на то, что язык при этом оказывается сильно отличным от Лиспа, как, впрочем, и от любого привычного языка, мы увидим, что интерпретатор для языка логического программирования имеет, в сущности, ту же структуру, что и интерпретатор Лиспа.

---

[2]Самое важное, чего не хватает в нашем интерпретаторе, — это механизмов, обрабатывающих ошибки и поддерживающих отладку. Более подробное обсуждение вычислителей можно найти в книге Friedman, Wand, and Haynes 1992, которая содержит обзор языков программирования на примере последовательности интерпретаторов, написанных на Scheme.

4.1. Метациклический интерпретатор

Наш интерпретатор Лиспа будет реализован как программа на Лиспе. Может показаться, что размышления о выполнении Лисп-программ при помощи интерпретатора, который сам написан на Лиспе, составляют порочный круг. Однако вычисление есть процесс, так что вполне логично описывать процесс вычисления с помощью Лиспа — в конце концов, это наш инструмент для описания процессов[3]. Интерпретатор, написанный на языке, который он сам реализует, называется *метациклическим* (metacircular).

В сущности, метациклический интерпретатор является формулировкой на языке Scheme модели вычислений с окружениями, описанной в разделе 3.2. Напомним, что в этой модели было две основные части:

- Чтобы выполнить комбинацию (составное выражение, не являющееся особой формой), нужно вычислить его подвыражения и затем применить значение подвыражения-оператора к значениям подвыражений-операндов.

- Чтобы применить составную процедуру к набору аргументов, нужно выполнить тело процедуры в новом окружении. Для того, чтобы построить это окружение, нужно расширить окружение объекта-процедуры кадром, в котором формальные параметры процедуры связаны с аргументами, к которым процедура применяется.

Эти два правила описывают сущность процесса вычисления, основной цикл, в котором выражения, которые требуется выполнить в окружении, сводятся к процедурам, которые нужно применить к аргументам, а те, в свою очередь, сводятся к новым выражениям, которые нужно выполнить в новых окружениях, и так далее, пока мы не доберемся до символов, чьи значения достаточно найти в окружении, и элементарных процедур, которые применяются напрямую (см. рис. 4.1)[4]. Этот цикл вычисления будет построен в виде взаимодействия двух основных процедур интерпретатора, `eval` и `apply`, описанных в разделе 4.1.1 (см. рис. 4.1).

[3] Даже с учетом этого, остаются важные стороны процесса вычисления, которые в нашем интерпретаторе не проясняются. Самая важная из них — точные механизмы того, как одни процедуры вызывают другие и возвращают значения процедурам, которые их вызвали. Эти вопросы мы рассмотрим в главе 5, где мы исследуем процесс вычисления более внимательно, реализуя вычислитель как простую регистровую машину.

[4] Если нам дается возможность применять примитивы, то что остается сделать для реализации интерпретатора? Задача интерпретатора состоит не в том, чтобы определить примитивы языка, а в том, чтобы обеспечить связующие элементы — средства комбинирования и абстракции, — которые превращают набор примитивов в язык. А именно:

- Интерпретатор позволяет работать с вложенными выражениями. Например, чтобы вычислить значение выражения `(+ 1 6)`, достаточно применения примитивов, но этого недостаточно для работы с выражением `(+ 1 (* 2 3))`. Сама по себе элементарная процедура `+` способна работать только с числами, и если передать ей аргумент — выражение `(* 2 3)`, она сломается. Одна из важных задач интерпретатора — устроить вычисление так, чтобы `(* 2 3)` свелось к значению 6, прежде чем оно будет передано `+` как аргумент.

- Интерпретатор позволяет использовать переменные. Например, элементарная процедура сложения не знает, как работать с выражениями вроде `(+ x 1)`. Нам нужен интерпретатор, чтобы следить за переменными и получать их значения, прежде чем запускать элементарные процедуры.

- Интерпретатор позволяет определять составные процедуры. При этом нужно хранить определения процедур, знать, как эти определения используются при вычислении выражений, и обеспечивать механизм, который позволяет процедурам принимать аргументы.

- Интерпретатор дает особые формы, вычисляющиеся иначе, чем вызовы процедур.

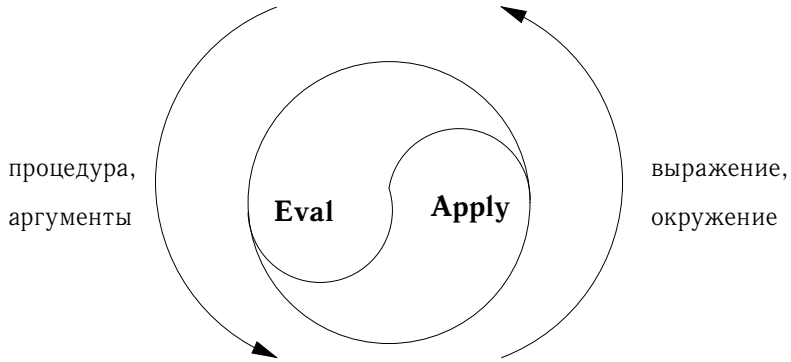


Рис. 4.1. Цикл `eval-apply` раскрывает сущность компьютерного языка.

Реализация интерпретатора будет зависеть от процедур, определяющих *синтаксис* (syntax) выполняемых выражений. При помощи абстракции данных мы сделаем интерпретатор независимым от представления языка. К примеру, вместо того, чтобы окончательно решать, что присваивание выражается в виде списка, в котором первым элементом стоит символ `set!`, мы пользуемся абстрактным предикатом `assignment?`, чтобы распознавать присваивание, и абстрактными селекторами `assignment-variable` и `assignment-value`, чтобы обращаться к его частям. Реализация выражений будет подробно рассмотрена в разделе 4.1.2. Имеются также операции, описанные в разделе 4.1.3, которые определяют представление процедур и окружений. Например, `make-procedure` порождает составные процедуры, `lookup-variable-value` извлекает значения переменных, а `apply-primitive-procedure` применяет элементарную процедуру к указанному списку аргументов.

4.1.1. Ядро интерпретатора

Процесс вычисления можно описать как взаимодействие двух процедур: `eval` и `apply`.

Eval

Процедура `eval` в качестве аргументов принимает выражение и окружение. Она относит выражение к одному из возможных классов и управляет его выполнением. `Eval` построена как разбор случаев в зависимости от синтаксического типа выполняемого выражения. Для того, чтобы процедура была достаточно общей, определение типа выражения мы формулируем абстрактно, не связывая себя никакой конкретной реализацией различных типов выражений. Для каждого типа выражений имеется предикат, который распознает этот тип, и абстрактные средства для выбора его частей. Такой *абстрактный синтаксис* (abstract syntax) позволяет легко видеть, как можно изменить синтаксис языка и использовать тот же самый интерпретатор, но только с другим набором синтаксических процедур.

Элементарные выражения

• Для самовычисляющихся выражений, например, чисел, `eval` возвращает само выражение.

• `Eval` должен находить значения переменных, просматривая окружение.

Особые формы

• Для выражений с кавычкой (`quote`), `eval` возвращает само закавыченное выражение.

• Присваивание переменной (или ее определение) должно вызвать `eval` рекурсивно, чтобы вычислить новое значение, которое требуется связать с переменной. Окружение нужно модифицировать, изменяя (или создавая) связывание для переменной.

• Выражение `if` требует специальной обработки своих частей: если предикат истинен, нужно выполнить следствие; если нет, альтернативу.

• Выражение `lambda` требуется преобразовать в процедуру, пригодную к применению. Для этого нужно упаковать параметры и тело `lambda`-выражения вместе с окружением, в котором оно вычисляется.

• Выражение `begin` требует выполнения своих подвыражений в том порядке, как они появляются.

• Разбор случаев (`cond`) преобразуется во вложенные выражения `if` и затем вычисляется.

Комбинации

• Для применения процедуры `eval` должна рекурсивно вычислить операцию и операнды комбинации. Получившиеся процедура и аргументы передаются `apply`, которая распоряжается собственно применением процедуры.

Вот определение `eval`:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
```

```
        (else
         (error "Неизвестный тип выражения -- EVAL" exp))))
```

Ясности ради, `eval` реализована как перебор альтернатив через `cond`. Недостаток этой реализации — наша процедура обрабатывает только несколько указанных типов выражений, и, не меняя определение `eval`, новые типы добавить нельзя. В большинстве реализаций Лиспа распределение выражений по типам сделано в стиле, управляемом данными. Это дает пользователю возможность добавлять новые типы выражений, которые `eval` будет способен распознать, не меняя само определение `eval`. (См. упражнение 4.3.)

Apply

Процедура `apply` принимает два аргумента: процедуру и список аргументов, к которым ее надо применить. `Apply` делит процедуры на два класса: для применения примитивов она зовет `apply-primitive-procedure`; составные процедуры она применяет, по очереди вычисляя выражения, составляющие тело процедуры. Окружение, в котором вычисляется тело составной процедуры, получается из базового окружения, хранящегося в процедуре, добалением кадра, где параметры процедуры связываются с аргументами, к которым процедура применяется. Вот определение `apply`:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Неизвестный тип процедуры -- APPLY" procedure))))
```

Аргументы процедур

Обрабатывая применение процедуры, `eval` получает список аргументов, к которым процедуру надо применить, при помощи `list-of-values`. Процедура `list-of-values` в качестве аргумента берет список операндов комбинации. Она вычисляет каждый аргумент и возвращает список соответствующих значений[5].

```
(define (list-of-values exps env)
  (if (no-operands? exps)
```

[5]Ветку `application?` в `eval` можно было бы упростить, используя `map` (и постановив, что `operands` возвращает список) вместо того, чтобы писать явным образом процедуру `list-of-values`. Мы решили не использовать здесь `map`, чтобы подчеркнуть, что интерпретатор можно написать без обращения к процедурам высших порядков (а следовательно, его можно написать на языке, в котором нет таких процедур), притом, что язык, поддерживаемый интерпретатором, содержит процедуры высших порядков.

```
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

Условные выражения

Процедура `eval-if` вычисляет предикатную часть выражения `if` в данном окружении. Если результат истинен, `eval-if` выполняет следствие, если нет, — альтернативу:

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Использование `true?` в `eval-if` подчеркивает вопрос о связи между реализуемым языком и языком реализации. Выражение `if-predicate` выполняется в реализуемом языке, и, следовательно, результат его является значением этого языка. Предикат интерпретатора `true?` переводит это значение в значение, которое может быть проверено выражением `if` в языке реализации: метациклическое представление истины может не совпадать с ее представлением в нижележащей Scheme[6].

Последовательности

Процедура `eval-sequence` вызывается из `apply` для выполнения последовательности выражений в теле процедуры, а также из `eval` для обработки последовательности выражений в выражении `begin`. Она принимает в виде аргументов последовательность выражений и окружение, и выполняет выражения в том порядке, в котором они ей даны. Возвращаемое значение совпадает со значением последнего выражения.

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Присваивания и определения

Следующая процедура обрабатывает присваивание переменным. При помощи `eval` она находит значение, которое требуется присвоить, и передает переменную и получившееся значение в процедуру `set-variable-value!` для включения в текущее окружение.

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

[6] В нашем случае, язык реализации и реализуемый язык совпадают. Размышления о значении `true?` расширяют наше сознание безотносительно к материальной сущности истины.

Определения переменных обрабатываются сходным образом[7]:

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

В качестве возвращаемого значения для присваивания или определения мы выбрали символ `ok`[8].

Упражнение 4.1.
Заметим, что мы не можем сказать, вычисляет ли метациклический интерпретатор операнды слева направо или справа налево. Порядок вычисления наследуется от нижележащего Лиспа: если аргументы `cons` в процедуре `list-of-values` вычисляются слева направо, то и операнды в `list-of-values` будут вычисляться слева направо. Если же вычисление аргументов `cons` происходит справа налево, то и `list-of-values` будет вычислять операнды справа налево.

Напишите версию `list-of-values`, которая вычисляет операнды слева направо, вне зависимости от порядка вычислений в нижележащем Лиспе. Напишите также версию, которая вычисляет операнды справа налево.

4.1.2. Представление выражений

Интерпретатор напоминает программу символьного дифференцирования, описанную в разделе 2.3.2. Обе программы работают с символьными выражениями. В обеих результат работы с составным выражением определяется рекурсивной обработкой частей выражения и сочетанием частичных результатов, причем способ сочетания зависит от типа выражения. И там, и там мы использовали абстракцию данных, чтобы отделить общие правила работы от деталей того, как представлены выражения. Для программы дифференцирования это означало, что одна и та же процедура взятия производной могла работать с алгебраическими выражениями в префиксной, инфиксной или какой-либо другой записи. Для интерпретатора это означает, что синтаксис языка определяется исключительно процедурами, которые классифицируют выражения и выделяют их части.

Вот описание синтаксиса нашего языка:

- К самовычисляющимся объектам относятся только числа и строки:

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

- Переменные представляются в виде символов:

```
(define (variable? exp) (symbol? exp))
```

[7]Эта реализация `define` не учитывает один тонкий вопрос в обработке внутренних определений, хотя в большинстве случаев работает правильно. В чем состоит проблема и как ее решить, мы увидим в разделе 4.1.6.

[8]Как мы упоминали при введении `define` и `set!`, их значения в Scheme зависят от реализации — то есть автор реализации имеет право выбрать такое значение, какое он хочет.

- Выражения с кавычкой имеют форму `(quote` ⟨*закавыченное-выражение*⟩`)`:

```
(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))
```

`Quoted?` определена посредством процедуры `tagged-list?`, которая распознает списки, начинающиеся с указанного символа[9]:

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

- Присваивания имеют форму `(set!` ⟨*переменная*⟩ ⟨*выражение*⟩`)`:

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

- Определения имеют вид

```
(define ⟨переменная⟩ ⟨значение⟩)
```

или

```
(define (⟨переменная⟩ ⟨параметр1⟩ ... ⟨параметрn⟩)
  ⟨тело⟩)
```

Вторая форма (стандартное определение процедуры) является синтаксическим сахаром для

```
(define ⟨переменная⟩
  (lambda (⟨параметр1⟩ ... ⟨параметрn⟩)
    ⟨тело⟩))
```

Соответствующие синтаксические процедуры выглядят так:

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
```

[9] В разделе 2.3.1 упоминается, что интерпретатор рассматривает закавыченное выражение как список, начинающийся с `quote`, даже если выражение напечатано через знак кавычки. Например, выражение `'a` будет выглядеть для интерпретатора как `(quote a)`. См. упражнение 2.55.

```
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                   (cddr exp))))
```

• `Lambda`-выражения являются списками, которые начинаются с символа `lambda`:

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))
```

Мы приводим также конструктор для `lambda`-выражений. Он используется в вышеприведенной процедуре `definition-value`:

```
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

• Условные выражения начинаются с `if` и имеют предикат, следствие и (необязательную) альтернативу. Если в выражении нет части-альтернативы, мы указываем в ее качестве `false`[10].

```
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

Мы предоставляем также конструктор для `if`-выражений. Его будет использовать процедура `cond->if` для преобразования выражений `cond` в выражения `if`:

```
(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

[10]Значение выражения `if` в случае, когда предикат ложен, а альтернатива отсутствует, в Scheme не определено; здесь мы решили сделать его ложным. Мы будем поддерживать переменные `true` и `false` в выполняемых выражениях путем связывания их в глобальном окружении. См. раздел 4.1.4.

• `Begin` упаковывает последовательность выражений в одно выражение. В синтаксические операции над выражениями `begin` мы включаем извлечение самой последовательности из выражения `begin`, а также селекторы, которые возвращают первое выражение и остаток выражений в последовательности[11].

```
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))
```

Кроме того, мы даем конструктор `sequence->exp` (для использования в процедуре `cond->if`), который преобразует последовательность в единое выражение, используя, если надо, `begin`:

```
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons 'begin seq))
```

• Вызов процедуры — это любое составное выражение, не попадающее ни в один из перечисленных типов. Его `car` — это оператор, а `cdr` — список операндов:

```
(define (application? exp) (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))
```

Производные выражения

Некоторые особые формы языка можно определить через выражения, включающие другие особые формы, вместо того, чтобы задавать их напрямую. Как пример рассмотрим `cond`, который можно реализовать как гнездо выражений `if`. Например, задачу вычисления выражения

[11] Эти селекторы для списка выражений, а также соответствующие им селекторы для списка операндов, не предназначаются для абстракции данных. Они введены в качестве мнемонических имен для основных списковых операций, чтобы легче было понимать вычислитель с явным управлением из раздела 5.4.

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

можно свести к задаче вычисления следующего выражения, состоящего из форм `if` и `begin`:

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero)
               0)
        (- x)))
```

Такая реализация обработки `cond` упрощает интерпретатор, поскольку она уменьшает количество особых форм, для которых требуется явно описывать процесс вычисления.

Мы включаем в интерпретатор синтаксические процедуры, которые определяют доступ к частям выражения `cond`, а также процедуру `cond->if`, которая преобразует выражения `cond` в выражения `if`. Анализ случаев начинается с `cond` и состоит из списка ветвей-вариантов вида предикат-действие. Вариант считается умолчательным, если его предикатом является символ `else`[12].

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; нет ветви else
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "Ветвь ELSE не последняя -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
```

[12]Значение выражения `cond`, когда все предикаты ложны, а вариант по умолчанию `else` отсутствует, в языке Scheme не определено; здесь мы решили сделать его ложным.

```
                    (sequence->exp (cond-actions first))
                    (expand-clauses rest))))))
```

Выражения (вроде `cond`), которые мы желаем реализовать через синтаксические преобразования, называются *производными* (derived expressions). Выражения `let` также являются производными (см. упражнение 4.6)[13].

Упражнение 4.2.
Хьюго Дум хочет переупорядочить ветви `cond` так, чтобы ветвь для вызова процедур располагалась перед веткой для присваивания. Он утверждает, что при этом интерпретатор станет эффективнее: поскольку в программах обычно больше вызовов процедур, чем присваиваний, определений и т. д., его усовершенствованный `eval` обычно будет рассматривать меньше вариантов, чем исходный, при распознавании типа выражения.

а. Что за ошибка содержится в плане Хьюго? (Подсказка: что сделает его интерпретатор с выражением `(define x 3)`?)

б. Хьюго расстроен, что его план не сработал. Он готов пойти на любые жертвы, чтобы позволить интерпретатору распознавать вызовы процедур до того, как он проверяет все остальные типы выражений. Помогите ему, изменив синтаксис интерпретируемого языка так, чтобы вызовы процедур начинались с символа `call`. Например, вместо `(factorial 3)` нам теперь придется писать `(call factorial 3)`, а вместо `(+ 1 2)` — `(call + 1 2)`.

Упражнение 4.3.
Перепишите `eval` так, чтобы диспетчеризация происходила в стиле, управляемом данными. Сравните результат с дифференцированием, управляемым данными, из упражнения 2.73. (Можно использовать `car` составного выражения в качестве типа этого выражения, так как это хорошо сочетается с синтаксисом, реализованным в этом разделе.)

Упражнение 4.4.
Вспомним определения особых форм `and` и `or` из главы 1:

- `and:` выражения вычисляются слева направо. Если значение какого-то из них оказывается ложным, возвращается ложь; оставшиеся выражения не вычисляются. Если все выражения оказываются истинными, возвращается значение последнего из них. Если нет ни одного выражения, возвращается истина.
- `or:` выражения вычисляются слева направо. Если значение какого-то из них оказывается истинным, возвращается это значение; оставшиеся выражения не вычисляются. Если все выражения оказываются ложными, или нет ни одного выражения, возвращается ложь.

Введите `and` и `or` в качестве новых особых форм интерпретатора, определив соответствующие синтаксические процедуры и процедуры выполнения `eval-and` и `eval-or`. В качестве альтернативы покажите, как можно реализовать `and` и `or` в виде производных выражений.

[13] Практические Лисп-системы предоставляют механизм, который дает пользователю возможность добавлять новые производные выражения и определять их значения через синтаксические преобразования, не внося изменений в вычислитель. Такое преобразование, определяемое пользователем, называется *макрос* (macro). Добавить простой механизм для определения макросов легко, однако в получающемся языке возникают сложные проблемы конфликта имен. Множество исследований посвящено поиску механизмов определения макросов, в которых такие проблемы не возникают. См., например, Kohlbecker 1986, Clinger and Rees 1991 и Hanson 1991.

Упражнение 4.5.
В языке Scheme есть дополнительная разновидность синтаксиса вариантов `cond`, `(`⟨*проверка*⟩ `=>` ⟨*потребитель*⟩`)`. Если результат вычисления ⟨*проверки*⟩ оказывается истинным значением, то вычисляется ⟨*потребитель*⟩. Его значение должно быть одноместной процедурой; эта процедура вызывается со значением ⟨*проверки*⟩ в качестве аргумента, и результат этого вызова возвращается как значение выражения `cond`. Например,

```
(cond ((assoc 'b '((a 1) (b 2))) => cadr)
      (else false))
```

имеет значение 2. Измените обработку `cond` так, чтобы она поддерживала этот расширенный синтаксис.

Упражнение 4.6.
Выражения `let` производны, поскольку

```
(let ((⟨пер1⟩⟨выр1⟩) ... (⟨перn⟩⟨вырn⟩))
  ⟨тело⟩)
```

эквивалентно

```
((lambda (⟨пер1⟩ ... ⟨перn⟩)
   ⟨тело⟩)
 ⟨выр1⟩
 ...
 ⟨вырn⟩)
```

Напишите синтаксическое преобразование `let->combination`, которое сводит вычисление `let`-выражений к вычислению комбинаций указанного вида, и добавьте соответствующую ветку для обработки `let` к `eval`.

Упражнение 4.7.
Особая форма `let*` подобна `let`, но только связывания переменных в `let*` происходят последовательно, и каждое следующее связывание происходит в окружении, где видны все предыдущие. Например,

```
(let* ((x 3)
       (y (+ x 2))
       (z (+ x y 5)))
  (* x z))
```

возвращает значение 39. Объясните, каким образом можно переписать выражение `let*` в виде набора вложенных выражений `let`, и напишите процедуру `let*->nested-lets`, которая проделывает это преобразование. Если мы уже реализовали `let` (упражнение 4.6) и хотим теперь расширить интерпретатор так, чтобы он обрабатывал `let*`, достаточно ли будет добавить в `eval` ветвь, в которой действием записано

```
(eval (let*->nested-lets exp) env)
```

или нужно явным образом преобразовывать `let*` в набор непроизводных выражений?

Упражнение 4.8.
«Именованный `let`» — это вариант `let`, который имеет вид

```
(let ⟨переменная⟩ ⟨связывание⟩ ⟨тело⟩)
```

⟨*Связывание*⟩ и ⟨*тело*⟩ такие же, как и в обычном `let`, но только ⟨*переменная*⟩ связана в ⟨*теле*⟩ с процедурой, у которой тело ⟨*тело*⟩, а имена параметров — переменные в ⟨*связываниях*⟩. Таким образом, можно неоднократно выполнять ⟨*тело*⟩, вызывая процедуру по имени ⟨*переменная*⟩. Например, итеративную процедуру для порождения чисел Фибоначчи (раздел 1.2.2) можно переписать при помощи именованного `let` как

```
(define (fib n)
  (let fib-iter ((a 1)
                 (b 0)
                 (count n))
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1)))))
```

Измените преобразование `let->combination` из упражнения 4.6 так, чтобы оно поддерживало именованный `let`.

Упражнение 4.9.
Во многих языках имеются различные конструкции для построения циклов, например, `do`, `for`, `while` и `until`. В Scheme итеративные процессы можно выразить через обычные вызовы процедур, так что особые конструкции не дают никакого существенного выигрыша в вычислительной мощности. С другой стороны, часто они удобны. Придумайте какие-нибудь конструкции для итерации, дайте примеры их использования и покажите, как их реализовать в виде производных выражений.

Упражнение 4.10.
При помощи абстракции данных мы смогли написать процедуру `eval` так, что она не зависит от конкретного синтаксиса интерпретируемого языка. Чтобы проиллюстрировать это свойство, разработайте новый синтаксис для Scheme, изменив процедуры из этого раздела и ничего не трогая в `eval` и `apply`.

4.1.3. Структуры данных интерпретатора

Помимо внешнего синтаксиса выражений, реализация интерпретатора должна определить также внутренние структуры данных, с которыми она работает во время выполнения программы, в частности, представление процедур и окружений, а также истинных и ложных значений.

Проверка предикатов

В условных выражениях мы воспринимаем в качестве истины все, кроме специального ложного объекта `false`.

```
(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))
```

Представление процедур

Работая с примитивами, мы предполагаем, что у нас есть следующие процедуры:

- (apply-primitive-procedure ⟨*процедура*⟩ ⟨*аргументы*⟩)

применяет данную элементарную процедуру к значениям аргументов из списка ⟨*аргументы*⟩ и возвращает результат вызова.

- (primitive-procedure? ⟨*процедура*⟩)

проверяет, является ли ⟨*процедура*⟩ элементарной.

Эти механизмы работы с элементарными процедурами подробнее описаны в разделе 4.1.4.

Составная процедура строится из параметров, тéла процедуры и окружения при помощи конструктора `make-procedure`:

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))
```

Действия над окружениями

Интерпретатору нужно иметь несколько операций, действующих над окружениями. Как объясняется в разделе 3.2, окружение представляет собой последовательность кадров, а каждый кадр является таблицей связываний, соотносящих переменные с их значениями. Для работы с окружениями мы используем следующие операции:

- `(lookup-variable-value` ⟨*переменная*⟩ ⟨*окружение*⟩`)` возвращает значение, связанное с символом ⟨*переменная*⟩ в ⟨*окружении*⟩, либо сообщает об ошибке, если переменная не связана.

- `(extend-environment` ⟨*переменные*⟩ ⟨*значения*⟩ ⟨*исх-окр*⟩`)` возвращает новое окружение, состоящее из нового кадра, в котором символы из списка ⟨*переменные*⟩ связаны с соответствующими элементами списка ⟨*значения*⟩, а объемлющим окружением является окружение ⟨*исх-окр*⟩.

- `(define-variable!` ⟨*переменная*⟩ ⟨*значение*⟩ ⟨*окружение*⟩`)` добавляет к первому кадру ⟨*окружения*⟩ новое связывание, которое сопоставляет ⟨*переменной*⟩ ⟨*значение*⟩.

- `(set-variable-value!` ⟨*переменная*⟩ ⟨*значение*⟩ ⟨*окружение*⟩`)` изменяет связывание ⟨*переменной*⟩ в ⟨*окружении*⟩ так, что в дальнейшем ей будет соответствовать ⟨*значение*⟩, либо сообщает об ошибке, если переменная не связана.

Чтобы реализовать все эти операции, мы представляем окружение в виде списка кадров. Объемлющее окружение живет в `cdr` этого списка. Пустое окружение — это просто пустой список.

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())
```

Каждый кадр в окружении представляется в виде пары списков: список переменных, связанных в кадре, и список значений[14].

```
(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

Чтобы расширить окружение новым кадром, который связывает переменные со значениями, мы порождаем кадр, который состоит из списка переменных и списка значений, и присоединяем его к окружению. Если количество переменных и количество значений не совпадают, сообщаем об ошибке.

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Получено слишком много аргументов" vars vals)
          (error "Получено слишком мало аргументов" vars vals))))
```

Чтобы найти переменную в окружении, мы просматриваем список переменных в первом кадре. Если находим нужную переменную, то возвращаем соответствующий элемент списка значений. Если мы не находим переменную в текущем кадре, то ищем в объемлющем окружении, и так далее. Если мы добираемся до пустого окружения, нужно сообщить об ошибке «неопределенная переменная».

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
```

[14] В нижеследующем коде кадры не являются настоящей абстракцией данных: `set-variable-value!` и `define-variable!` явным образом изменяют значения в кадре при помощи `set-car!`. Назначение процедур работы с кадрами — сделать код операций над окружениями простым для чтения.

```
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Несвязанная переменная" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Чтобы присвоить переменной новое значение в указанном окружении, мы ищем переменную, точно так же, как в `lookup-variable-value`, и изменяем соответствующее значение, когда его находим.

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Несвязанная переменная -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Чтобы определить переменную, мы просматриваем первый кадр в поисках связывания для нее, и изменяем связывание, если его удается найти (так же, как в `set-variable-value!`). Если связывания не существует, мы присоединяем его к первому кадру.

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

Описанный здесь метод — только один из многих способов представления окружений. Поскольку мы при помощи абстракции данных отделили конкретную реализацию от остальных частей интерпретатора, при желании мы можем сменить представление окружений. (См. упражнение 4.11.) В Лисп-системе промышленного качества быстрота операций над окружениями — особенно обращения к переменной — очень сильно влияет на общую производительность. Представление, описанное здесь, при всей своей

концептуальной простоте неэффективно и, скорее всего, его не стали бы использовать в рабочей системе[15].

Упражнение 4.11.
Вместо того, чтобы представлять кадр в виде списка списков, его можно представить как список связываний, где каждое связывание является парой из имени и значения. Перепишите операции с окружениями в соответствии с этим альтернативным представлением.

Упражнение 4.12.
Процедуры `set-variable-value!`, `define-variable!` и `lookup-variable-value` можно выразить посредством более абстрактных процедур для просмотра структуры окружений. Определите абстракции, которые фиксируют общую схему поведения, и с их помощью перепишите эти три процедуры.

Упражнение 4.13.
Scheme позволяет создавать новые связывания через `define`, но не дает никакого способа избавиться от связывания. Реализуйте в интерпретаторе особую форму `make-unbound!`, которая изымает связывание данного символа из окружения, в котором `make-unbound!` выполняется. Задача определена не до конца. Например, нужно ли удалять связывания в других кадрах, кроме первого? Дополните спецификацию и объясните свой выбор вариантов.

4.1.4. Выполнение интерпретатора как программы

Написав интерпретатор, мы получаем в руки описание (выраженное на Лиспе) процесса вычисления лисповских выражений. Одно из преимуществ наличия описания в виде программы в том, что эту программу можно запустить. У нас внутри Лиспа есть работающая модель того, как сам Лисп вычисляет выражения. Она может служить средой для экспериментов с правилами вычисления, и дальше в этой главе мы такими экспериментами и займемся.

Программа-вычислитель в конце концов сводит выражения к применению элементарных процедур. Следовательно, единственное, что нам требуется для запуска интерпретатора, — создать механизм, который обращается к нижележащей Лисп-системе и моделирует вызовы элементарных процедур.

Нам нужно иметь связывание для каждого имени элементарной процедуры, чтобы, когда `eval` выполняет вызов примитива, у него был объект, который можно передать в `apply`. Поэтому мы выстраиваем глобальное окружение, связывающее особые объекты с именами элементарных процедур, которые могут появляться в вычисляемых нами выражениях. Кроме того, глобальное окружение включает связывания для символов `true` и `false`, так что их можно использовать как переменные в вычисляемых выражениях.

```
(define (setup-environment)
  (let ((initial-env
```

[15]Недостаток этого представления (как и варианта из упражнения 4.11) состоит в том, что вычислителю может понадобиться просматривать слишком много кадров, чтобы найти связывание конкретной переменной. (Такой подход называется *глубокое связывание* (deep binding).) Один из способов избежать такой потери производительности — использовать стратегию под названием *лексическая адресация* (lexical addressing), которая обсуждается в разделе 5.5.6.

```
        (extend-environment (primitive-procedure-names)
                            (primitive-procedure-objects)
                            the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))

(define the-global-environment (setup-environment))
```

Как именно мы представляем объекты-элементарные процедуры, не имеет значения. Требуется только, чтобы их можно было распознавать и применять, вызывая процедуры `primitive-procedure?` и `apply-primitive-procedure`. Мы решили представлять примитивы в виде списка, начинающегося с символа `primitive` и содержащего процедуру нижележащего Лиспа, которая реализует данный примитив.

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))
```

`Setup-environment` получит имена и реализации элементарных процедур из списка[16].

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
         другие примитивы
        ))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```

Чтобы вызвать элементарную процедуру, мы просто применяем процедуру-реализацию к аргументам, используя нижележащую Лисп-систему[17].

[16] Любую процедуру, определенную в нижележащем Лиспе, можно использовать как примитив для метациклического интерпретатора. Имя примитива, установленного в интерпретаторе, не обязательно должно совпадать с именем его реализации в нижележащем Лиспе; здесь имена одни и те же потому, что метациклический интерпретатор реализует саму Scheme. Так, например, мы могли бы написать в списке `primitive-procedures` что-нибудь вроде `(list 'first car)` или `(list 'square (lambda (x) (* x x)))`.

[17] `Apply-in-underlying-scheme` — это процедура `apply`, которой мы пользовались в предыдущих главах. Процедура `apply` метациклического интерпретатора (раздел 4.1.1) имитирует работу этого примитива. Наличие двух процедур с одинаковым именем ведет к технической проблеме при запуске интерпретатора,

```
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

Для удобства работы с метациклическим интерпретатором мы организуем *управляющий цикл* (driver loop), который моделирует цикл чтения-выполнения-печати нижележащей Лисп-системы. Этот цикл печатает *подсказку* (prompt), считывает входное выражение, вычисляет это выражение в глобальном окружении и распечатывает результат. Перед каждым результатом мы помещаем *подсказку вывода* (output prompt), чтобы отличить значение выражения от всего прочего, что может быть напечатано[18].

```
(define input-prompt ";;; Ввод M-Eval:")
(define output-prompt ";;; Значение M-Eval:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))
```

Мы пользуемся специальной процедурой вывода `user-print`, чтобы не печатать окружение составных процедур, которое может быть очень длинным списком, и даже может содержать циклы.

```
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

поскольку определение `apply` метациклического интерпретатора загородит определение примитива. Можно избежать этого, переименовав метациклический `apply`, и избавиться таким образом от конфликта с именем элементарной процедуры. Мы же вместо этого приняли решение сохранить ссылку на исходный `apply`, выполнив

```
(define apply-in-underlying-scheme apply)
```

прежде, чем определили `apply` в интерпретаторе. Теперь мы можем обращаться к исходной версии `apply` под другим именем.

[18] Элементарная процедура `read` ожидает ввода от пользователя и возвращает ближайшее полное выражение, которое он напечатает. Например, если пользователь напечатает `(+ 23 x)`, результатом `read` будет трехэлементный список из символа `+`, числа 23 и символа `x`. Если пользователь введет `'x`, результатом `read` будет двухэлементный список из символа `quote` и символа `x`.

Теперь для запуска интерпретатора нам остается только проинициализировать глобальное окружение и войти в управляющий цикл. Вот пример работы интерпретатора:

```
(define the-global-environment (setup-environment))

(driver-loop)

;;; Ввод M-Eval:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))

;;; Значение M-Eval:
ok

;;; Ввод M-Eval:
(append '(a b c) '(d e f))
;;; Значение M-Eval:
(a b c d e f)
```

Упражнение 4.14.
Ева Лу Атор и Хьюго Дум экспериментируют с метациклическим интерпретатором каждый по отдельности. Ева вводит определение `map` и запускает несколько тестовых программ с его использованием. Они замечательно работают. Хьюго, со своей стороны, ввел системную версию `map` как примитив метациклического интерпретатора. Когда он пытается его выполнить, все ломается самым ужасным образом. Объясните, почему у Хьюго `map` не работает, а у Евы работает.

4.1.5. Данные как программы

При рассмотрении программы на Лиспе, вычисляющей лисповские выражения, может быть полезна аналогия. Одна из возможных точек зрения на значение программы состоит в том, что программа описывает абстрактную (возможно, бесконечно большую) машину. Рассмотрим, например, знакомую нам программу для вычисления факториалов:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

Можно считать эту программу описанием машины, которая содержит узлы для вычитания, умножения и проверки на равенство, двухпозиционный переключатель и еще одну факториал-машину. (Факториал-машина получается бесконечной, поскольку она содержит другую факториал-машину внутри себя.) На рисунке 4.2 изображена потоковая диаграмма факториал-машины, которая показывает, как спаяны ее части.

Подобным образом, мы можем рассматривать вычислитель как особого рода машину, которой подается в виде сырья описание другой машины. Обработав свои входные данные, вычислитель перестраивает себя так, чтобы моделировать описываемую машину.

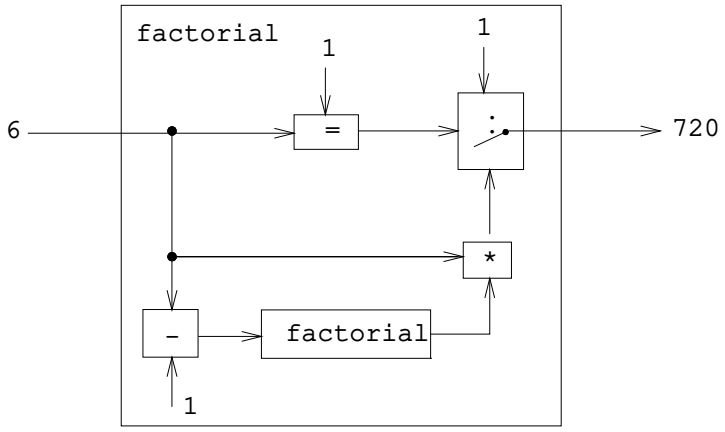


Рис. 4.2. Программа вычисления факториала, изображенная в виде абстрактной машины.

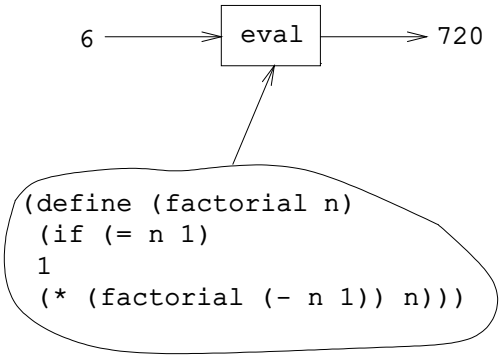


Рис. 4.3. Вычислитель, моделирующий факториальную машину.

Например, если мы скормим вычислителю определение `factorial`, как показано на рисунке 4.3, он сможет считать факториалы.

С этой точки зрения, наш вычислитель-интерпретатор выглядит как *универсальная машина* (universal machine). Она имитирует другие машины, представленные в виде Лисп-программ[19]. Это замечательное устройство. Попробуйте представить себе аналогичный вычислитель для электрических схем. Это была бы схема, которой на вход поступает сигнал, кодирующий устройство какой-то другой схемы, например, фильтра. Восприняв этот вход, наша схема-вычислитель стала бы работать как фильтр, соответствующий описанию. Такая универсальная электрическая схема имеет почти невообразимую сложность. Удивительно, что интерпретатор программ — сам по себе программа довольно простая[20].

Еще одна замечательная черта интерпретатора заключается в том, что он служит мостом между объектами данных, которыми манипулирует язык программирования, и самим языком. Представим себе, что работает программа интерпретатора (реализованная на Лиспе), и что пользователь вводит выражения в интерпретатор и рассматривает результаты. С точки зрения пользователя, входное выражение вроде `(* x x)` является выражением языка программирования, которое интерпретатор должен выполнить. Однако с точки зрения интерпретатора это всего лишь список (в данном случае, список из трех символов: `*`, `x` и `x`), с которым нужно работать по ясно очерченным правилам.

Нас не должно смущать, что программы пользователя являются данными для интерпретатора. На самом деле, иногда бывает удобно игнорировать это различие и, предоставляя пользовательским программам доступ к `eval`, давать пользователю возможность явным образом вычислить объект данных как выражение Лиспа. Во многих диалектах Лиспа имеется элементарная процедура `eval`, которая в виде аргументов берет выражение и окружение, и вычисляет выражение в указанном окружении[21]. Таким образом,

---

[19]То, что машины описаны на языке Лисп, несущественно. Если дать нашему интерпретатору программу на Лиспе, которая ведет себя как вычислитель для какого-нибудь другого языка, скажем, Си, то вычислитель для Лиспа будет имитировать вычислитель для Си, который, в свою очередь, способен сымитировать любую машину, описанную в виде программы на Си. Подобным образом, написание интерпретатора Лиспа на Си порождает программу на Си, способную выполнить любую программу на Лиспе. Главная идея здесь состоит в том, что любой вычислитель способен имитировать любой другой. Таким образом, понятие «того, что в принципе можно вычислить» (если не принимать во внимание практические вопросы времени и памяти, потребной для вычисления), независимо от языка компьютера и выражает глубинное понятие *вычислимости* (computability). Это впервые было ясно показано Аланом М. Тьюрингом (1912-1954), чья статья 1936 года заложила основы теоретической информатики. В этой статье Тьюринг представил простую модель вычислений, — теперь известную как *машина Тьюринга* (Turing machine), — и утверждал, что любой «эффективный процесс» выразим в виде программы для такой машины. (Этот аргумент известен как *тезис Чёрча-Тьюринга* (Church-Turing thesis).) Затем Тьюринг реализовал универсальную машину, т. е. машину Тьюринга, которая работает как вычислитель для программ машин Тьюринга. При помощи этой схемы рассуждений он показал, что существуют коррекно поставленные задачи, которые не могут быть решены машиной Тьюринга (см. упражнение 4.15), а следовательно не могут быть сформулированы в виде «эффективного процесса». Позднее Тьюринг внес фундаментальный вклад и в развитие практической информатики. Например, ему принадлежит идея структурирования программ с помощью подпрограмм общего назначения. Биографию Тьюринга можно найти в Hodges 1983.

[20]Некоторые считают странным, что вычислитель, реализованный с помощью относительно простой процедуры, способен имитировать программы, более сложные, чем он сам. Существование универсальной машины-вычислителя — глубокое и важное свойство вычисления. *Теория рекурсии* (recursion theory), отрасль математической логики, занимается логическими пределами вычислимости. В прекрасной книге Дугласа Хофштадтера «Гёдель, Эшер, Бах» (Hofstadter 1979) исследуются некоторые из этих идей.

[21]Предупреждение: эта процедура `eval` — не то же самое, что процедура `eval`, реализованная нами в разделе 4.1.1, потому что она работает с *настоящими* окружениями, а не с искусственными структурами

как

```
(eval '(* 5 5) user-initial-environment)
```

так и

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

возвращают результат 25[22].

Упражнение 4.15.
Если даны одноаргументная процедура `p` и объект `a`, то говорят, что `p` «останавливается» на `a`, если выражение `(p a)` возвращает значение (а не печатает сообщение об ошибке или выполняется вечно). Покажите, что невозможно написать процедуру `halts?`, которая бы точно определяла для любой процедуры `p` и любого объекта `a`, останавливается ли `p` на `a`. Используйте следующее рассуждение: если бы имелась такая процедура `halts?`, можно было бы написать следующую программу:

```
(define (run-forever) (run-forever))

(define (try p)
  (if (halts? p p)
      (run-forever)
      'halted))
```

Теперь рассмотрите выражение `(try try)` и покажите, что любое возможное завершение (остановка или вечное выполнение) нарушает требуемое поведение `halts?`[23].

4.1.6. Внутренние определения

Наша модель вычислений с окружениями и метациклический интерпретатор выполняют определения по очереди, расширяя кадр окружения на одно определение за раз. Это особенно удобно для диалоговой разработки программы, когда программисту нужно свободно смешивать вызовы процедур с определениями новых процедур. Однако если мы внимательно поразмыслим над внутренними определениями, с помощью которых реализуется блочная структура (введенная в разделе 1.1.8), то мы увидим, что пошаговое расширение окружения — одно имя за другим — может оказаться не лучшим способом определения локальных переменных.

Рассмотрим процедуру с внутренними определениями, например

окружений, которые мы построили в разделе 4.1.3. С этими настоящими окружениями пользователь не может работать, как с обычными списками; к ним нужно обращаться через `eval` или другие специальные операции. Подобным образом, элементарная процедура `apply`, упомянутая раньше, не то же самое, что метациклическая `apply`, поскольку она использует настоящие процедуры Scheme, а не объекты-процедуры, которые мы конструировали в разделах 4.1.3 и 4.1.4.

[22] Реализация MIT Scheme имеет процедуру `eval`, а также символ `user-initial-environment`, связанный с исходным окружением, в котором вычисляются выражения.

[23] Хотя здесь мы предположили, что `halts?` получает процедурный объект, заметим, что рассуждение остается в силе даже в том случае, когда на вход подается текст процедуры и ее окружение. В этом и состоит знаменитая *теорема об остановке* (Halting Theorem) Тьюринга, в которой был дан первый пример *невычислимой* (non-computable) задачи, т. е. корректно поставленного задания, которое невозможно выполнить с помощью вычислительной процедуры.

```
(define (f x)
  (define (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
   ⟨остаток тела  f⟩)
```

Здесь нам хочется, чтобы имя `odd?` в теле процедуры `even?` ссылалось на процедуру `odd?`, которая определена позже, чем `even?`. Область действия имени `odd?` — это все тело `f`, а не только та его часть, которая лежит за точкой внутри `f`, где определяется `odd?`. В самом деле, ели заметить, что сама `odd?` определена с помощью `even?` — так что `even?` и `odd?` являются взаимно рекурсивными процедурами, — то становится ясно, что единственная удовлетворительная интерпретация двух `define` — рассматривать их так, как будто `even?` и `odd?` были добавлены в окружение одновременно. В общем случае, сферой действия локального имени является целиком тело процедуры, в котором вычисляется `define`.

В нынешнем виде интерпретатор будет вычислять вызовы `f` правильно, но причина этого «чисто случайная»: поскольку определения внутренних процедур расположены в начале, никакие их вызовы не вычисляются, пока они все не определены. Следовательно, к тому времени, когда выполняется `even?`, `odd?` уже определена. Фактически, последовательный механизм вычисления дает те же результаты, что и механизм, непосредственно реализующий одновременное определение, для всякой процедуры, где внутренние определения стоят в начале тела, а вычисление выражений для определяемых переменных не использует ни одну из этих переменных. (Пример процедуры, которая не удовлетворяет этим требованиям, так что последовательное определение не равносильно одновременному, можно найти в упражнении 4.19.)[24]

Однако имеется простой способ обрабатывать определения так, чтобы у локально определенных имен оказалась действительно общая сфера действия, — достаточно лишь создать все будущие внутренние переменные текущего окружения, прежде чем начнется вычисление какого-либо из выражений, возвращающих значение. Можно это сделать, например, путем синтаксического преобразования `lambda`-выражений. Прежде чем выполнять тело выражения `lambda`, мы «прочесываем» его и уничтожаем все внутренние определения. Локально определенные переменные будут созданы через `let`, а затем получат значения посредством присваивания. Например, процедура

```
(lambda ⟨переменные⟩
  (define u ⟨e1⟩)
```

[24]Нежелание зависеть в программах от этого механизма вычисления побудило нас написать «администрация ответственности не несет» в примечании 28 в главе 1. Настаивая на том, чтобы внутренние определения стояли в начале тела и не использовали друг друга во время вычисления самих определений, стандарт IEEE Scheme дает авторам реализаций некоторую свободу при выборе механизма вычисления этих определений. Выбор того или иного правила вычисления может показаться мелочью, которая влияет только на интерпретацию «плохих» программ. Однако в разделе 5.5.6 мы увидим, что через переход к модели с одновременным определением внутренних переменных можно избежать некоторых досадных трудностей, которые бы в противном случае возникли при написании компилятора.

```
  (define v ⟨e2⟩)
  ⟨e3⟩)
```

преобразуется в

```
(lambda ⟨переменные⟩
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (set! u ⟨e1⟩)
    (set! v ⟨e2⟩)
    ⟨e3⟩)))
```

где `*unassigned*` — специальный символ, который при поиске переменной вызывает сообщение об ошибке, если программа пытается использовать значение переменной, которой ничего еще не присвоено.

Альтернативная стратегия поиска внутренних определений показана в упражнении 4.18. В отличие от преобразования, продемонстрированного только что, она навязывает программисту следующее ограничение: значение каждой определяемой переменной должно вычисляться без обращения к значениям других определяемых переменных[25].

Упражнение 4.16.
В этом упражнении мы реализуем только что описанный метод обработки внутренних определений. Мы предполагаем, что интерпретатор поддерживает `let` (см. упражнение 4.6).

а. Измените процедуру `lookup-variable-value` (раздел 4.1.3) так, чтобы она, обнаруживая в качестве значения символ `*unassigned*`, сообщала об ошибке.

б. Напишите процедуру `scan-out-defines`, которая берет тело процедуры и возвращает его эквивалент без внутренних определений, выполняя описанное нами преобразование.

в. Вставьте `scan-out-defines` в интерпретатор, либо в `make-procedure`, либо в `procedure-body`. Какое из этих мест лучше? Почему?

Упражнение 4.17.
Нарисуйте диаграммы окружения, которое находится в силе в момент выполнения выражения ⟨*e3*⟩ из процедуры выше по тексту, и сравните его устройство при последовательной обработке определений и при описанном выше преобразовании. Откуда в преобразованной программе берется дополнительный кадр? Объясните, почему это различие никогда не отражается на поведении корректных программ. Придумайте, как заставить интерпретатор реализовать правило «одновременной» сферы действия для внутренних определений без создания дополнительного кадра.

Упражнение 4.18.
Рассмотрим альтернативную стратегию обработки определений, которая переводит пример из текста в

```
(lambda ⟨переменные⟩
  (let ((u '*unassigned*)
```

[25]Стандарт IEEE Scheme допускает различные стратегии реализации. В нем говорится, что программист обязан подчиняться этому ограничению, но реализация может его не проверять. Некоторые реализации Scheme, включая MIT Scheme, используют преобразование, показанное выше. В таких реализациях будут работать некоторые из программ, которые это ограничение нарушают.

```
        (v '*unassigned*))
    (let ((a ⟨e1⟩)
          (b ⟨e2⟩))
      (set! u a)
      (set! v b))
    ⟨e3⟩))
```

Здесь `a` и `b` представляют новые имена переменных, созданные интерпретатором, которые не встречаются в пользовательской программе. Рассмотрим процедуру `solve` из раздела 3.5.4:

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Будет ли эта процедура работать, если внутренние определения преобразуются так, как предлагается в этом упражнении? А если так, как в тексте раздела? Объясните.

Упражнение 4.19.
Бен Битобор, Лиза П. Хакер и Ева Лу Атор спорят о том, каким должен быть результат выражения

```
(let ((a 1))
  (define (f x)
    (define b (+ a x))
    (define a 5)
    (+ a b))
  (f 10))
```

Бен говорит, что следует действовать согласно последовательному правилу для `define`: `b` равно 11, затем `a` определяется как 5, так что общий результат равен 16. Лиза возражает, что взаимная рекурсия требует правила одновременной сферы действия для внутренних определений и нет причин рассматривать имена процедур отдельно от прочих имен. То есть она выступает за механизм, реализованный в упражнении 4.16. При этом `a` оказывается не определено в момент, когда вычисляется `b`. Следовательно, по мнению Лизы, процедура должна выдавать ошибку. Ева не согласна с обоими. Она говорит, что если определения вправду должны считаться одновременными, то 5 как значение `a` должно использоваться при вычислении `b`. Следовательно, по мнению Евы, `a` должно равняться 5, `b` должно быть 15, а общий результат 20. Какую из этих точек зрения Вы поддерживаете (если у Вас нет своей четвертой)? Можете ли Вы придумать способ реализации внутренних определений, который бы работал так, как предлагает Ева[26]?

Упражнение 4.20.
Поскольку внутренние определения выглядят последовательными, а на самом деле параллельны, некоторые предпочитают их вовсе избегать и вместо этого пользуются особой формой `letrec`. `Letrec` выглядит так же, как `let`, поэтому неудивительно, что переменные в нем связываются одновременно и имеют одинаковую для всех сферу действия. Можно переписать процедуру-пример `f` из текста без внутренних определений, но при этом в точности с тем же значением, так:

[26] Авторы MIT Scheme согласны с Лизой, и вот почему: в принципе права Ева — определения следует рассматривать как одновременные. Однако придумать универсальный эффективный механизм, который вел бы себя так, как она требует, кажется трудным. Если же такого механизма нет, то лучше порождать ошибку в сложных случаях параллельных определений (мнение Лизы), чем выдавать неверный ответ (как хочет Бен).

```
(define (f x)
  (letrec ((even?
            (lambda (n)
              (if (= n 0)
                  true
                  (odd? (- n 1)))))
           (odd?
            (lambda (n)
              (if (= n 0)
                  false
                  (even? (- n 1))))))
    ⟨остаток тела  f⟩))
```

Выражение `letrec` имеет вид

```
(letrec ((⟨пер1⟩ ⟨выр1⟩) ... (⟨перn⟩ ⟨вырn⟩))
  ⟨тело⟩)
```

и является вариантом `let`, в котором выражения $\langle выр_k \rangle$, устанавливающие начальные значения для переменных $\langle пер_k \rangle$, вычисляются в окружении, которое включает все связывания `letrec`. Это делает возможным рекурсию между связываниями, к примеру, взаимную рекурсию `even?` и `odd?` в последнем примере, или вычисление факториала 10 через

```
(letrec ((fact
          (lambda (n)
            (if (= n 1)
                1
                (* n (fact (- n 1)))))))
  (fact 10))
```

а. Реализуйте `letrec` как производное выражение, переводя выражение `letrec` в выражение `let`, как показано в тексте раздела или в упражнении 4.18. То есть переменные `letrec` должны создаваться в `let`, а затем получать значение через `set!`.

б. Хьюго Дум совсем запутался во всех этих внутренних определениях. Ему кажется, что если кому-то не нравятся `define` внутри процедуры, то пусть пользуются обычным `let`. Покажите, что́ в его рассуждениях неверно. Нарисуйте диаграмму, показывающую окружение, в котором выполняется ⟨*остаток тела* `f`⟩ во время вычисления выражения `(f 5)`, если `f` определена как в этом упражнении. Нарисуйте диаграмму окружений для того же вычисления, но только с `let` на месте `letrec` в определении `f`.

Упражнение 4.21.
Как ни удивительно, интуитивная догадка Хьюго (в упражнении 4.20) оказывается верной. Действительно, можно строить рекурсивные процедуры без использования `letrec` (и даже без `define`), только способ это сделать намного тоньше, чем казалось Хьюго. Следующее выражение вычисляет факториал 10 с помощью рекурсивной процедуры[27]:

```
((lambda (n)
   ((lambda (fact)
```

[27] В этом примере показан программистский трюк, позволяющий формулировать рекурсивные процедуры без помощи `define`. Самый общий прием такого рода называется *Y-оператор* (Y operator), и с его помощью можно реализовать рекурсию в «чистом λ-исчислении». (Подробности о лямбда-исчислении можно найти в Stoy 1977, а демонстрацию Y-оператора на Scheme в Gabriel 1988.)

```
      (fact fact n))
    (lambda (ft k)
      (if (= k 1)
          1
          (* k (ft ft (- k 1)))))))
 10)
```

а. Проверьте, что это выражение на самом деле считает факториалы (вычисляя его). Постройте аналогичное выражение для вычисления чисел Фибоначчи.

б. Рассмотрим следующую процедуру, включающую взаимно рекурсивные внутренние определения:

```
(define (f x)
  (define (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
  (even? x))
```

Восстановите пропущенные фрагменты так, чтобы получилось альтернативное определение `f`, где нет ни внутренних определений, ни `letrec`:

```
(define (f x)
  ((lambda (even? odd?)
     (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? ⟨??⟩ ⟨??⟩ ⟨??⟩)))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? ⟨??⟩ ⟨??⟩ ⟨??⟩))))))
```

4.1.7. Отделение синтаксического анализа от выполнения

Написанный нами интерпретатор прост, но очень неэффективен, потому что синтаксический анализ выражений перемешан в нем с их выполнением. Таким образом, сколько раз выполняется программа, столько же раз анализируется ее синтаксис. Рассмотрим, например, вычисление `(factorial 4)`, если дано следующее определение факториала:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

Каждый раз, когда вызывается `factorial`, интерпретатор должен определить, что тело процедуры является условным выражением, и извлечь его предикат. Только после этого он может вычислить предикат и поступить в соответствии с его значением. Каждый раз, когда вычисляется выражение `(* (factorial (- n 1)) n)` или подвыражения `(factorial (- n 1))` и `(- n 1)`, интерпретатор должен произвести анализ

случаев внутри `eval`, выяснить, что выражение является вызовом процедуры, а также извлечь его оператор и операнды. Такой анализ недёшев. Проделывать его многократно — неразумно.

Можно преобразовать интерпретатор так, чтобы синтаксический анализ проводился только один раз, и повысить таким образом эффективность работы[28]. Мы разбиваем процедуру `eval`, которая принимает выражение и окружение, на две части. `Analyze` берет только выражение. Она выполняет синтаксический анализ и возвращает новую *исполнительную* процедуру (execution procedure). В этой процедуре упакована работа, которую придется проделать при выполнении выражения. Исполнительная процедура берет в качестве аргумента окружение и завершает вычисление. При этом экономится работа, потому что `analyze` будет для каждого выражения вызываться только один раз, а исполнительная процедура, возможно, многократно.

После разделения анализа и выполнения `eval` превращается в

```
(define (eval exp env)
  ((analyze exp) env))
```

Результатом вызова `analyze` является исполнительная процедура, которая применяется к окружению. `Analyze` содержит тот же самый анализ, который делал исходный `eval` из раздела 4.1.1, однако процедуры, между которыми мы выбираем, только анализируют, а не окончательно выполняют выражение.

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Неизвестный тип выражения -- ANALYZE" exp))))
```

Вот самая простая из процедур анализа, которая обрабатывает самовычисляющиеся выражения. Ее результатом является исполнительная процедура, которая игнорирует свой аргумент-окружение и просто возвращает само выражение:

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))
```

В случае кавычки мы можем добиться некоторого выигрыша, извлекая закавыченное выражение только один раз на стадии анализа, а не на стадии выполнения.

[28]Такое преобразование является неотъемлемой частью процесса компиляции, который мы рассмотрим в главе 5. Джонатан Рис написал для проекта T интерпретатор Scheme с похожей структурой приблизительно в 1982 голу (Rees and Adams 1982). Марк Фили (Feeley 1986, см. также Feeley and Lapalme 1987) независимо изобрел этот метод в своей дипломной работе.

```
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
```

Поиск переменной нужно проводить на стадии выполнения, поскольку при этом требуется знать окружение[29].

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))
```

Анализ присваивания, `analyze-assignment`, также должен отложить само присваивание до времени выполнения, когда будет в наличии окружение. Однако возможность (рекурсивно) проанализировать выражение `assignmentvalue` сразу, на стадии анализа, — это большой выигрыш в эффективности, поскольку теперь это выражение будет анализироваться только однажды. То же верно и для определений:

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))
```

Для условных выражений мы извлекаем и анализируем предикат, следствие и альтернативу на стадии анализа.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

При анализе выражения `lambda` также достигается значительный выигрыш в эффективности: тело `lambda` анализируется только один раз, а процедура, получающаяся в результате выполнения `lambda`, может применяться многократно.

[29]Есть, впрочем, важная часть поиска переменной, которую все-таки *можно* осуществить во время синтаксического анализа. Как мы покажем в разделе 5.5.6, можно определить позицию в структуре окружения, где будет находиться нужное значение, и таким образом избежать необходимости искать в окружении элемент, который соответствует переменной.

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

Анализ последовательности выражений (в `begin` или в теле `lambda`-выражения) более сложен[30]. Каждое выражение в последовательности анализируется, и для каждого получается исполнительная процедура. Эти исполнительные процедуры комбинируются в одну общую исполнительную процедуру, которая принимает в качестве аргумента окружение и последовательно вызывает каждую из частичных исполнительных процедур, передавая ей окружение как аргумент.

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Пустая последовательность -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

Для вызова процедуры мы анализируем оператор и операнды и строим исполнительную процедуру, которая вызывает исполнительную процедуру оператора (получая при этом объект-процедуру, которую следует применить) и исполнительные процедуры операндов (получая аргументы). Затем мы все это передаем в `execute-application`, аналог `apply` из раздела 4.1.1. `Execute-application` отличается от `apply` тем, что тело составной процедуры уже проанализировано, так что нет нужды в дальнейшем анализе. Вместо этого мы просто вызываем исполнительную процедуру для тела, передавая ей расширенное окружение.

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env))
                                aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
```

[30]См. упражнение 4.23, в котором объясняются некоторые подробности обработки последовательностей.

```
                                 args
                                 (procedure-environment proc))))
        (else
         (error
          "Неизвестный тип процедуры -- EXECUTE-APPLICATION"
          proc))))
```

В нашем новом интерпретаторе используются те же структуры данных, синтаксические процедуры и вспомогательные процедуры времени выполнения, что и в разделах 4.1.2, 4.1.3 и 4.1.4.

Упражнение 4.22.
Расширьте интерпретатор из этого раздела так, чтобы он поддерживал `let`. (См. упражнение 4.6.)

Упражнение 4.23.
Лиза П. Хакер не понимает, зачем делать `analyze-sequence` такой сложной. Все остальные процедуры анализа — простые трансформации соответствующих вычисляющих процедур (или ветвей `eval`) из раздела 4.1.1. Лиза ожидала, что `analyze-sequence` будет выглядеть так:

```
(define (analyze-sequence exps)
  (define (execute-sequence procs env)
    (cond ((null? (cdr procs)) ((car procs) env))
          (else ((car procs) env)
                (execute-sequence (cdr procs) env))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Пустая последовательность -- ANALYZE"))
    (lambda (env) (execute-sequence procs env))))
```

Ева Лу Атор объясняет Лизе, что версия в тексте проделывает больше работы по вычислению последовательности во время анализа. В Лизиной исполнительной процедуре вызовы частичных исполнительных процедур, вместо того, чтобы быть встроенными, перебираются в цикле. В результате, хотя отдельные выражения в последовательности оказываются проанализированы, сама последовательность анализируется во время выполнения.

Сравните две версии `analyze-sequence`. Рассмотрите, например, обычный случай (типичный для тел процедур), когда в последовательности только одно выражение. Какую работу будет делать исполнительная процедура, предложенная Лизой? А процедура из текста раздела? Как соотносятся эти две процедуры в случае последовательности из двух выражений?

Упражнение 4.24.
Спроектируйте и проведите несколько экспериментов, чтобы сравнить скорость исходного метациклического вычислителя и его версии из этого раздела. С помощью результатов этих опытов оцените долю времени, которая тратится на анализ и на собственно выполнение в различных процедурах.

4.2. Scheme с вариациями: ленивый интерпретатор

Теперь, имея в своем распоряжении интерпретатор, выраженный в виде программы на Лиспе, мы можем экспериментировать с различными вариантами строения языка, просто

модифицируя этот интерпретатор. В самом деле, часто изобретение нового языка начинается с того, что пишут интерпретатор, который встраивает новый язык в существующий язык высокого уровня. Например, если нам хочется обсудить какую-то деталь предлагаемой модификации Лиспа с другим членом Лисп-сообщества, мы можем предъявить ему интерпретатор, в котором эта модификация реализована. Тогда наш адресат может поэкспериментировать с новым интерпретатором и послать в ответ свои замечания в виде дальнейших модификаций. Реализация на высокоуровневой основе не только упрощает проверку и отладку вычислителя; такое встраивание к тому же позволяет разработчику слизывать[31] черты языка-основы, как наш встроенный интерпретатор Лиспа использовал примитивы и структуру управления нижележащего Лиспа. Только позже (да и то не всегда) разработчику приходится брать на себя труд построения полной реализации на низкоуровневом языке или в аппаратуре. В этом разделе и следующем мы изучаем некоторые вариации на тему Scheme, которые значительно увеличивают ее выразительную силу.

4.2.1. Нормальный порядок вычислений и аппликативный порядок вычислений

В разделе 1.1, где мы начали обсуждение моделей вычисления, мы указали, что Scheme — язык с *аппликативным порядком вычисления* (applicative-order language), а именно, что все аргументы процедур в Scheme вычисляются в момент вызова. Напротив, в языках с *нормальным порядком вычисления* (normal-order language) вычисление аргументов процедур задерживается до момента, когда действительно возникает нужда в их значениях. Если вычисление аргументов процедур откладывается как можно дольше (например, до того момента, когда они требуются какой-либо элементарной процедуре), то говорят о *ленивом вычислении* (lazy evaluation)[32]. Рассмотрим процедуру

```
(define (try a b)
  (if (= a 0) 1 b))
```

Выполнение `(try 0 (/ 1 0))` в Scheme приводит к ошибке. При ленивых вычислениях никакой ошибки не возникнет. Вычисление выражения даст результат 1, поскольку к аргументу `(/ 1 0)` обращаться не понадобится.

Примером использования ленивых вычислений может служить процедура `unless`:

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

которую можно использовать в выражениях вроде

[31] Слизывать (*snarf*): «Брать, в особенности большой документ или файл, с целью использовать с разрешения владельца или без оного». Пролизывать (*snarf down*): «Слизывать, иногда с дополнительным значением восприятия, переработки или понимания». (Эти определения были слизаны из Steele et al. 1983. См. также Raymond 1993.)

[32] Терминологическая разница между выражениями «ленивый» и «нормальный порядок вычислений» несколько размыта. Обычно «ленивый» относится к механизмам конкретных интерпретаторов, а «нормальный порядок» к семантике языков независимо от способа реализации. Однако разделение здесь не жесткое, и часто эти термины употребляются как синонимы.

```
(unless (= b 0)
        (/ a b)
        (begin (display "exception: returning 0")
               0))
```

В аппликативном языке это не будет работать, потому что и обычное значение, и значение исключения будут выполнены еще до вызова `unless` (См. упражнение 1.6). Преимущество ленивых вычислений в том, что некоторые процедуры, например, та же `unless`, могут выполнять полезные действия, даже если вычисление некоторых их аргументов способно привести к ошибке или бесконечному циклу.

Если тело процедуры начинает выполняться прежде, чем вычисляется ее аргумент, то процедура называется *нестрогой* (non-strict) по этому аргументу. Если же аргумент вычисляется прежде, чем происходит вход в процедуру, то процедура называется *строгой* (strict) по этому аргументу[33]. В чисто аппликативном языке все процедуры строги по всем своим аргументам. В языке с чисто нормальным порядком вычислений все составные процедуры нестроги по всем своим аргументам, а элементарные процедуры могут быть и такими, и такими. Бывают также языки (см. упражнение 4.31), которые дают программисту возможность явно обозначать строгость определяемых им процедур.

Яркий пример процедуры, которой может быть полезно оказаться нестрогой, — это `cons` (и вообще почти любой конструктор структур данных). Можно производить полезные вычисления, составлять из элементов структуры данных и работать с ними, даже если значения элементов неизвестны. Вполне имеет смысл задача, например, посчитать длину списка, не зная значений его отдельных элементов. В разделе 4.2.3 мы развиваем эту идею и реализуем потоки из главы 3 в виде списков, составленных из нестрогих `cons`-пар.

Упражнение 4.25.
Предположим, что мы (в обычной Scheme с аппликативным порядком вычислений) определяем `unless` как показано выше, а затем определяем `factorial` через `unless`:

```
(define (factorial n)
  (unless (= n 1)
          (* n (factorial (- n 1)))
          1))
```

Что произойдет, если мы попытаемся вычислить `(factorial 5)`? Будут ли наши определения работать в языке с нормальным порядком вычислений?

Упражнение 4.26.
Бен Битобор и Лиза П. Хакер расходятся во мнениях о важности ленивых вычислений для реализации конструкций вроде `unless`. Бен указывает, что при аппликативном порядке `unless` можно реализовать как особую форму. Лиза отвечает, что в таком случае `unless` будет просто синтаксисом, а не процедурой, которую можно использовать в сочетании с процедурами высших

[33] Термины «строгий» и «нестрогий» означают, в сущности, то же самое, что «аппликативный» и «нормальный» порядок вычислений, но только они относятся к отдельным процедурам и их аргументам, а не к языку в целом. На конференциях по языкам программирования можно услышать, как кто-нибудь говорит: «В языке Hassle с нормальным порядком вычислений есть несколько строгих примитивов. Остальные процедуры принимают аргументы через ленивое вычисление».

порядков. Проясните детали в обеих позициях. Покажите, как реализовать `unless` в виде производного выражения (вроде `cond` или `let`), и приведите пример ситуации, когда имеет смысл, чтобы `unless` была процедурой, а не особой формой.

4.2.2. Интерпретатор с ленивым вычислением

В этом разделе мы реализуем язык с нормальным порядком вычислений, который отличается от Scheme только тем, что все составные процедуры по всем аргументам нестроги. Элементарные процедуры по-прежнему будут строгими. Совсем несложно, модифицируя интерпретатор из раздела 4.1.1, добиться, чтобы интерпретируемый язык вел себя таким образом. Почти что все требуемые изменения сосредоточены вокруг механизма процедурного вызова.

Основная идея состоит в том, что при вызове процедуры интерпретатор должен определить, какие аргументы требуется вычислить, а какие задержать. Задержанные аргументы не вычисляются, а преобразуются в объекты, называемые *санками* (thunks)[34*]. В санке должна содержаться информация, необходимая, чтобы вычислить значение аргумента, когда оно потребуется, и сделать это так, как будто оно вычислено во время вызова. Таким образом, санк должен содержать выражение-аргумент и окружение, в котором вычисляется вызов процедуры.

Процесс вычисления санка называется *вынуждением* (forcing a thunk)[35] Вообще говоря, санк вынуждается только тогда, когда требуется его значение: когда он передается в элементарную процедуру, использующую его значение; когда он служит предикатом в условном выражении; или когда он является значением оператора, который нужно применить как процедуру. Мы должны решить, будем ли мы *мемоизировать* (memoize) санки, как мы делали с задержанными объектами в разделе 3.5.1. При использовании мемоизации, когда санк вынуждается в первый раз, он запоминает вычисленное значение. Последующие вызовы только возвращают запомненное значение, не вычисляя его заново. Мы делаем выбор в пользу мемоизации, поскольку для многих приложений это эффективнее. Здесь, однако, имеются тонкости[36].

---

[34]Название «санк» было придумано в неформальной группе, которая обсуждала реализацию вызова по имени в Алголе 60. Было замечено, что большую часть анализа («обдумывания», *thinking about*) выражения можно производить во время компиляции; таким образом, во время выполнения выражение будет уже большей частью «обдумано» (*thunk about* — намеренно неверно образованная английская форма) (Ingerman et al. 1960).

*В русскоязычной литературе слово *thunk* иногда переводится как «переходник». Нам кажется, что в данном случае такой перевод мешал бы пониманию текста. — *прим. перев.*

[35]Это аналогично использованию слова `force` («вынудить», «заставить») для задержанных объектов, при помощи которых в главе 3 представлялись потоки. Основная разница между тем, что мы делаем здесь, и тем, чем мы занимались в главе 3, состоит в том, что теперь мы встраиваем задержку и вынуждение в интерпретатор, и они применяются автоматически и единообразно во всем языке.

[36]Ленивые вычисления, совмещенные с мемоизацией, иногда называют методом передачи аргументов с *вызовом по необходимости* (call by need), в отличие от *вызова по имени* (call by name). (Вызов по имени, введенный в Алголе 60, аналогичен немемоизированному ленивому вычислению.) Как проектировщики языка мы можем сделать интерпретатор мемоизирующим или немемоизирующим, или же оставить это на усмотрение программистов (упражнение 4.31). Как можно было ожидать из главы 3, этот выбор вызывает к жизни вопросы, особенно тонкие и запутанные в присутствии присваивания. (См. упражнения 4.27 и 4.29.) В замечательной статье Клингера (Clinger 1982) делается попытка прояснить многомерную путаницу, которая здесь возникает.

Преобразование интерпретатора

Основная разница между ленивым интерпретатором и интерпретатором из раздела 4.1.1 состоит в обработке вызовов процедур внутри `eval` и `apply`.

Ветка `application?` в `eval` принимает вид

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

Это почти тот же код, что был в ветке `application?` в `eval` из раздела 4.1.1. Однако при ленивом вычислении мы зовем `apply` с выражениями операндов, а не с аргументами, которые получаются при их вычислении. Мы также передаем `apply` окружение, поскольку оно понадобится для построения санков, если нам хочется, чтобы аргуметы вычислялись с задержкой. Оператор мы по-прежнему вычисляем, потому что сама применяемая процедура нужна `apply`, чтобы выбрать действие на основании ее типа (элементарная или составная) и применить ее.

Всякий раз, когда нам требуется собственно значение выражения, мы вместо простого `eval` пользуемся процедурой

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

чтобы, если значение выражения является санком, оно было вынуждено.

Наша новая версия `apply` также почти совпадает с версией из раздела 4.1.1. Разница состоит в том, что `eval` передает ей невычисленные выражения: для элементарных процедур (они строгие) мы вычисляем все аргументы и затем вызываем примитив; для составных процедур (они нестрогие) мы прежде применения процедуры замораживаем все аргументы.

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))  ; изменение
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)  ; изменение
           (procedure-environment procedure))))
        (else
         (error
          "Неизвестный тип процедуры -- APPLY" procedure))))
```

Процедуры, обрабатывающие аргументы, почти такие же, как `list-of-values` из раздела 4.1.1, но только `list-of-delayed-args` замораживает аргументы, вместо того, чтобы их вычислять, а в `list-of-arg-values` вместо `eval` используется `actual-value`:

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                                env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

Кроме того, нам требуется изменить в интерпретаторе обработку `if`, где вместо `eval` мы должны вызывать `actual-value`, чтобы значение предикатного выражения вычислялось прежде, чем мы проверим, истинно оно или ложно:

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Наконец, нужно изменить процедуру `driver-loop` (раздел 4.1.4), чтобы она звала `actual-value` вместо `eval`. Таким образом, если задержанное значение добирается до цикла чтение-вычисление-печать, то оно, прежде чем печататься, будет разморожено. Кроме того, чтобы показать, что работа идет с ленивым интерпретатором, мы изменим подсказки:

```
(define input-prompt ";;; Ввод L-Eval:")
(define output-prompt ";;; Значение L-Eval:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

Внеся эти изменения, мы можем запустить интерпретатор и протестировать его. Успешное вычисление выражения `try`, описанного в разделе 4.2.1, показывает, что интерпретатор проводит ленивое вычисление:

```
(define the-global-environment (setup-environment))

(driver-loop)

;;; Ввод L-eval:
```

```
(define (try a b)
  (if (= a 0) 1 b))
;;; Значение L-Eval:
ok

;;; Ввод L-eval:
(try 0 (/ 1 0))
;;; Значение L-Eval:
1
```

Представление санков

Наш интерпретатор должен устроить работу так, чтобы при применении процедур к аргументам порождались санки, и чтобы потом они вынуждались. Выражение в санке должно запаковываться вместе с окружением, так, чтобы потом можно было по ним вычислить аргумент. Чтобы вынудить санк, мы просто извлекаем из него выражение и окружение, и вычисляем выражение в окружении. Мы используем при этом не `eval`, а `actual-value`, так что если результат выражения сам окажется санком, мы и его вынудим, и так пока не доберемся до не-санка.

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

Простой способ упаковать выражение вместе с окружением — создать список из выражения и окружения. Таким образом, мы порождаем санк так:

```
(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))
```

Однако на самом деле нам в интерпретаторе нужны не такие санки, а мемоизированные. Мы сделаем так, чтобы санк при вынуждении превращался в вычисленный санк. Для этого мы будем заменять хранимое в нем выражение на значение и менять метку санка, чтобы можно было понять, что он уже вычислен[37].

[37] Заметим, что, вычислив выражение, мы еще и стираем из санка окружение. Это не влияет на то, какие значения возвращает интерпретатор. Однако при этом экономится память, поскольку стирание ссылки из санка на `env`, когда она становится больше не нужна, позволяет подвергнуть эту структуру *сборке мусора* (garbage collection) и заново использовать ее память. Мы обсудим это в разделе 5.3.

Подобным образом можно было бы разрешить собирать как мусор ненужные окружения в мемоизированных задержанных объектах из раздела 3.5.1: `memo-proc`, сохранив значение процедуры `proc`, делала бы что-нибудь вроде `(set! proc '())`, чтобы забыть саму процедуру (включающую окружение, где было вычислено `delay`).

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
                                  ; заменить  exp на его значение
           (set-cdr! (cdr obj) '()) ; забыть ненужное  env
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

Заметим, что одна и та же процедура `delay-it` работает и с мемоизацией, и без нее.

Упражнение 4.27.
Допустим, мы вводим в ленивый интерпретатор следующее выражение:

```
(define count 0)

(define (id x)
  (set! count (+ count 1))
  x)
```

Вставьте пропущенные значения в данной ниже последовательности действий и объясните свои ответы[38]:

```
(define w (id (id 10)))

;;; Ввод L-Eval:
count
;;; Значение L-Eval:
⟨вывод⟩

;;; Ввод L-Eval:
w
;;; Значение L-Eval:
⟨вывод⟩

;;; Ввод L-Eval:
count
;;; Значение L-Eval:
⟨вывод⟩
```

[38]Это упражнение показывает, что взаимодействие между ленивыми вычислениями и побочными эффектами может быть весьма запутанным. Ровно этого можно было ожидать после обсуждения в главе 3.

Упражнение 4.28.
`Eval`, передавая оператор в `apply`, вычисляет его не при помощи `eval`, а через `actual-value`, чтобы вынудить. Приведите пример, который показывает, что такое вынуждение необходимо.

Упражнение 4.29.
Придумайте пример программы, которая, по Вашему мнению, будет работать намного медленнее без мемоизации, чем с мемоизацией. Рассмотрим, помимо этого, следующую последовательность действий, в которой процедура `id` определена как в упражнении 4.27, а счетчик `count` начинает с 0:

```
(define (square x)
  (* x x))

;;; Ввод L-Eval:
(square (id 10))
;;; Значение L-Eval:
⟨вывод⟩

;;; Ввод L-Eval:
count
;;; Значение L-Eval:
⟨вывод⟩
```

Укажите, как будет выглядеть вывод в случае с мемоизирующим интерпретатором и с немемоизирующим.

Упражнение 4.30.
Пабло Э. Фект, бывший программист на языке C, беспокоится, что ленивый интерпретатор не вынуждает выражения в последовательности, и оттого некоторые побочные эффекты могут никогда не произойти. Поскольку ни у одного выражения в последовательности, помимо конечного, значение не используется (выражение стоит там только ради своего эффекта, например, чтобы присвоить значение переменной или что-нибудь напечатать), у значения такого выражения не может впоследствии быть применения, для которого его потребуется вынудить (например, в качестве аргумента элементарной процедуры). Поэтому П.Э. Фект считает, что при выполнении последовательности нужно все выражения, кроме последнего, вынуждать. Он предлагает изменить `eval-sequence` из раздела 4.1.1 так, чтобы она вместо `eval` использовала `actual-value`:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (actual-value (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

а. Бен Битобор считает, что Пабло неправ. Он показывает ему процедуру `for-each` из упражнения 2.23 — важный пример последовательности с побочными эффектами:

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))
```

Он утверждает, что интерпретатор из текста (с исходным `eval-sequence`) правильно работает с этой процедурой:

```
;;; Ввод L-Eval:
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
;;; Значение L-Eval:
done
```

Объясните, почему Бен прав насчет поведения `for-each`.

б. Пабло соглашается с Беном по поводу примера с `for-each`, но говорит, что, предлагая изменить `eval-sequence`, он имел в виду другой тип программ. Он определяет в ленивом интерпретаторе следующие две процедуры:

```
(define (p1 x)
  (set! x (cons x '(2)))
  x)

(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))
```

Какие значения вернут `(p1 1)` и `(p2 1)` с исходной `eval-sequence`? Каковы будут значения с изменением, которое предлагает Пабло?

в. Пабло указывает также, что изменение `eval-sequence`, которое он предлагает, не влияет на поведение примера из части а. Объясните, почему это так.

г. Как, по-Вашему, нужно работать с последовательностями в ленивом интерпретаторе? Что Вам нравится больше: подход Пабло, подход, приведенный в тексте, или что-нибудь третье?

Упражнение 4.31.

Подход, принятый в этом разделе, нехорош тем, что вносит изменение в Scheme, не сохраняя ее семантику. Было бы приятнее реализовать ленивые вычисления как *совместимое расширение* (upward-compatible extension), то есть так, чтобы обычные программы на Scheme работали как прежде. Этого можно добиться, расширив синтаксис определений процедур, так, чтобы пользователь мог решать, нужно ли задерживать аргументы. При этом можно еще предоставить пользователю выбор между задержкой с мемоизацией и без нее. Например, определение

```
(define (f a (b lazy) c (d lazy-memo))
   ...)
```

делало бы `f` процедурой от четырех аргументов, причем первый и третий вычисляются при вызове процедуры, второй задерживается, а четвертый задерживается и мемоизируется. Таким образом, обыкновенные определения процедур будут задавать такое же поведение, как в обычной Scheme, а добавление декларации `lazy-memo` к каждому параметру каждой составной процедуры приведет к поведению, как у ленивого интерпретатора, описанного в этом разделе. Разработайте и реализуйте изменения, с помощью которых можно получить такое расширение Scheme. Вам придется реализовать новые синтаксические процедуры для нового синтаксиса `define`. Кроме того, надо будет добиться, чтобы `eval` и `apply` определяли, когда надо задерживать аргументы, и соответствующим образом задерживали и вынуждали их. Наконец, придется обеспечить,чтобы вынуждение было с мемоизацией или без оной, смотря по обстоятельствам.

4.2.3. Потоки как ленивые списки

В разделе 3.5.1 мы показали, как реализовать потоки в виде задержанных списков. Мы ввели особые формы `delay` и `cons-stream`, которые позволили нам строить «обещания» вычислить `cdr` потока, не выполняя эти обещания до более позднего времени. Можно было бы использовать этот же метод и вводить новые особые формы всякий раз, когда нам требуется детальное управление процессом вычисления, но это было бы весьма неуклюже. Прежде всего, особая форма, в отличие от процедуры, не является полноправным объектом, и ее нельзя использовать в сочетании с процедурами высших порядков[39]. Кроме того, нам пришлось ввести потоки как новый тип объектов данных, похожий на списки, но отличный от них, и из-за этого потребовалось заново переписать для работы с потоками множество обычных операций над списками (`map`, `append` и тому подобное).

Когда у нас есть ленивое вычисление, списки и потоки можно считать одним и тем же типом, так что не возникает нужды в особых формах и в отдельных наборах операций для списков и потоков. Все, что нам требуется, — это так устроить дела, чтобы `cons` оказалась нестрогой. Можно сделать это, расширив интерпретатор и разрешив нестрогие элементарные процедуры, а затем реализовать `cons` как одну из таких процедур. Однако проще вспомнить (из раздела 2.1.3), что вообще не существует особой нужды реализовывать `cons` как примитив. Вместо этого можно представлять пары в виде процедур[40].

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
```

Выраженные через эти базовые операции, стандартные определения операций над списками будут работать как с бесконечными списками (потоками), так и с конечными, а потоковые операции можно определить как операции над списками. Вот несколько примеров:

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))

(define (map proc items)
  (if (null? items)
```

[39] Это как раз тот вопрос, который возник по отношению к процедуре `unless` в упражнении 4.26.

[40] Это процедурное представление, описанное в упражнении 2.4. В сущности, подошла бы и любая другая процедурная реализация (например, на основе передачи сообщений). Обратите внимание, что внести эти определения в ленивый интерпретатор можно, просто набрав их в управляющем цикле. Если мы изначально включили `cons`, `car` и `cdr` как примитивы в глобальное окружение, они будут переопределены. (См. также упражнения 4.33 и 4.34.)

```
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))

(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))

(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))

(define ones (cons 1 ones))

(define integers (cons 1 (add-lists ones integers)))

;;; Ввод L-Eval:
(list-ref integers 17)
;;; Значение L-Eval:
18
```

Заметим, что ленивые списки еще ленивее, чем потоки в главе 3: задерживается не только `cdr` списка, но и `car`[41]. На самом деле, даже доступ к `car` или `cdr` ленивой пары не обязательно вынуждает значение элемента списка. Значение будет вынуждено только тогда, когда это действительно нужно — например, чтобы использовать его в качестве аргумента примитива или напечатать в качестве ответа.

Ленивые пары также помогают с решением проблемы, которая возникла в разделе 3.5.4, где мы обнаружили, что формулировка потоковых моделей систем с циклами может потребовать оснащения программы явными операциями `delay`, помимо тех, что встроены в `cons-stream`. При ленивом вычислении все аргументы процедур единообразно задерживаются. Например, можно реализовать процедуры для интегрирования списка и решения дифференциальных уравнений так, как мы изначально намеревались в разделе 3.5.4:

```
(define (integral integrand initial-value dt)
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt)
                     int)))
  int)

(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)
```

[41] Благодаря этому можно реализовать задержанные версии не только последовательностей, но и более общих видов списковых структур. В Hughes 1990 обсуждаются некоторые применения «ленивых деревьев».

```
;;; Ввод L-Eval:
;: (list-ref (solve (lambda (x) x) 1 .001) 1000)
;;; Значение L-Eval:
2.176924
```

Упражнение 4.32.
Приведите несколько примеров, которые показывают разницу между потоками из главы 3.5.4 и «более ленивыми» списками, описанными в этом разделе. Как можно воспользоваться этой дополнительной ленивостью?

Упражнение 4.33.
Бен Битобор проверяет вышеописанную реализацию при помощи выражения

```
(car '(a b c))
```

К его большому удивлению, в ответ выдается ошибка. После некоторого размышления он понимает, что «списки». которые получаются при чтении кавычек, отличаются от списков, управляемых новыми определениями `cons`, `car` и `cdr`. Измените работу интерпретатора с закавыченными выражениями так, чтобы при вводе списковых выражений в цикле управления получались настоящие ленивые списки.

Упражнение 4.34.
Измените управляющий цикл интерпретатора так, чтобы ленивые пары и списки печатались каким-либо разумным образом. (Как Вы собираетесь работать с бесконечными списками)? Вероятно, понадобится также изменить представление ленивых пар, чтобы при печати интерпретатор их распознавал и печатал особым образом.

4.3. Scheme с вариациями — недетерминистское вычисление

В этом разделе мы расширяем интерпретатор Scheme так, чтобы он поддерживал парадигму программирования, называемую *недетерминистское вычисление* (nondeterministic computing), встраивая в интерпретатор средства поддержки автоматического поиска. Это значительно более глубокое изменение в языке, чем введение ленивых вычислений в разделе 4.2.

Подобно обработке потоков, недетерминистское вычисление полезно в задачах типа «порождение и проверка». Рассмотрим такую задачу: даются два списка натуральных чисел, и требуется найти пару чисел — одно из первого списка, другое из второго, — сумма которых есть простое число. В разделе 2.2.3 мы уже рассмотрели, как это можно сделать при помощи операций над конечными последовательностями, а в разделе 3.5.3 — при помощи бесконечных потоков. Наш подход состоял в том, чтобы породить последовательность всех возможных пар и отфильтровать ее, выбирая пары, в которых сумма есть простое число. Порождаем ли мы на самом деле сначала всю последовательность, как в главе 2, или чередуем порождение и фильтрацию, как в главе 3, несущественно для общей картины того, как организовано вычисление.

При недетерминистском подходе используется другой образ. Просто представим себе, что мы (каким-то образом) выбираем число из первого списка и число из второго списка, а затем предъявляем (при помощи какого-то механизма) требование, чтобы их сумма была простым числом. Это выражается следующей процедурой:

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

Может показаться, что эта процедура просто переформулирует задачу, а не указывает способ ее решить. Однако это законная недетерминистская программа[42].

Основная идея здесь состоит в том, что выражениям в недетерминистском языке разрешается иметь более одного возможного значения. Например, `an-element-of` может вернуть любой элемент данного списка. Наш интерпретатор недетерминистских программ будет автоматически выбирать возможное значение и запоминать, что он выбрал. Если впоследствии какое-либо требование не будет выполнено, интерпретатор попробует другой вариант выбора и будет перебирать варианты, пока вычисление не закончится успешно или пока варианты не иссякнут. Подобно тому, как ленивый интерпретатор освобождал программиста от заботы о деталях задержки и вынуждения значений, недетерминистский интерпретатор позволяет ему не заботиться о том, как происходит выбор.

Поучительно будет сравнить различные понятия времени, складывающиеся при недетерминистских вычислениях и обработке потоков. При обработке потоков ленивые вычисления используются для того, чтобы устранить связь между временем, когда строится поток возможных ответов, и временем, когда порождаются собственно ответы. Интерпретатор создает иллюзию, что все возможные ответы предоставлены нам во вневременной последовательности. При недетерминистских вычислениях выражение представляет собой исследование множества возможных миров, каждый из которых определяется множеством выбранных вариантов. Некоторые возможные миры приводят в тупик, другие дают полезные ответы. Вычислитель недетерминистских программ создает иллюзию, что время разветвляется, и что у наших программ есть различные возможные истории исполнения. Если мы оказываемся в тупике, мы можем вернуться к последней точке выбора и продолжить путь по другой ветке.

Описываемый в этом разделе интерпретатор недетерминистских программ называется `amb`-интерпретатор, потому что он основан на новой особой форме `amb`. Мы можем ввести вышеприведенное определение `prime-sum-pair` в управляющем цикле `amb`-интерпретатора (наряду с определениями `prime?`, `an-element-of` и `require`) и запустить процедуру:

```
;;; Ввод Amb-Eval:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
```

[42] Мы предполагаем, что уже заранее определена процедура `prime?`, которая проверяет числа на простоту. Даже если такая процедура определена, `prime-sum-pair` может подозрительно напоминать бестолковую попытку определения квадратного корня на псевдо-Лиспе из начала раздела 1.1.7. На самом деле, подобного рода процедура вычисления квадратного корня может быть сформулирована в виде недетерминистской программы. Вводя в интерпретатор механизм поиска, мы размываем границу между чисто декларативными описаниями и императивными спецификациями способов вычислить ответ. В разделе 4.4 мы пойдем еще дальше в этом направлении.

```
;;; Начало новой задачи
;;; Значение Amb-Eval:
(3 20)
```

Возвращенное значение было получено после того, как интерпретатор сделал несколько попыток выбора из каждого списка, последняя из которых оказалась успешной.

В разделе 4.3.1 вводится форма amb и показывается, как она поддерживает недетерминизм через механизм поиска, встроенный в интерпретатор. В разделе 4.3.2 приводятся примеры недетерминистских программ, а раздел 4.3.3 содержит подробности того, как реализовать amb-интерпретатор путем модификации обычного интерпретатора Scheme.

4.3.1. Amb и search

Чтобы расширить Scheme и поддержать недетерминистское программирование, мы вводим новую особую форму amb[43]. Выражение (amb $\langle e_1 \rangle$ $\langle e_2 \rangle$ ... $\langle e_n \rangle$) возвращает «произвольным образом» значение одного из n выражений $\langle e_i \rangle$. Например, выражение

```
(list (amb 1 2 3) (amb 'a 'b))
```

имеет шесть возможных значений:

```
(1 a)  (1 b)  (2 a)  (2 b)  (3 a)  (3 b)
```

Amb с одним вариантом возвращает обыкновенное (одно) значение.

Amb без вариантов — выражение (amb) — является выражением без приемлемых значений. С операционной точки зрения, выполнение выражения (amb) приводит к «неудаче» в вычислении: выполнение обрывается, и никакого значения не возвращается. При помощи этого выражения можно следующим образом выразить требование, чтобы выполнялось предикатное выражение p:

```
(define (require p)
  (if (not p) (amb)))
```

Через amb и require можно реализовать процедуру an-element-of, используемую выше:

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

Если список пуст, an-element-of терпит неудачу. В противном случае он произвольным образом возвращает либо первый элемент списка, либо элемент, выбранный из хвоста списка.

Можно также выразить выбор из бесконечного множества. Следующая процедура произвольным образом возвращает целое число, большее или равное некоторому данному n:

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

[43] Идея недетерминистского программирования с помощью amb-выражений впервые была описана Джоном Маккарти в 1961 году (см. McCarthy 1967).

Это похоже на потоковую процедуру `integers-starting-from`, описанную в разделе 3.5.2, но есть важное различие: потоковая процедура возвращает поток, который представляет последовательность всех целых чисел, начиная с `n`, а процедура, написанная через `amb`, выдает одно целое число[44].

Мысля абстрактно, мы можем представить, что выполнение выражения `amb` заставляет время разветвиться, и на каждой ветке оно продолжается с одним из возможных значений выбора. Мы говорим, что `amb` представляет собой *точку недетерминистского выбора* (nondeterministic choice point). Если бы у нас была машина с достаточным числом процессоров, которые можно было бы динамически выделять, то поиск можно было бы реализовать напрямую. Выполнение происходило бы, как в последовательной машине, пока не встретится выражение `amb`. В этот момент выделялись и инициализировались бы дополнительные процессоры, которые продолжали бы все параллельные потоки выполнения, обусловленные выбором. Каждый процессор продолжал бы последовательное выполнение одного из потоков, как если бы он был единственным, пока поток не оборвется, потерпев неудачу, не разделится сам или не завершится[45].

С другой стороны, если у нас есть машина, которая способна выполнять только один процесс (или небольшое число параллельных процессов), альтернативы приходится рассматривать последовательно. Можно представить себе интерпретатор, который в каждой точке выбора произвольным образом выбирает, по какой ветке продолжить выполнение. Однако случайный выбор может легко привести к неудачам. Можно было бы запускать такой интерпретатор многократно, делая случайный выбор и надеясь, что в конце концов мы получим требуемое значение, но лучше проводить *систематический поиск* (systematic search) среди всех возможных путей выполнения. `Amb`-интерпретатор, который мы разработаем в этом разделе, реализует систематический поиск следующим образом: когда интерпретатор встречает выражение `amb`, он сначала выбирает первый вариант. Такой выбор может в дальнейшем привести к другим точкам выбора. В каждой точке выбора интерпретатор сначала будет выбирать первый вариант. Если выбор приводит к неудаче, интерпретатор автомагически[46] *возвращается* (backtracks) к последней точке выбора и пробует следующий вариант. Если в какой-то точке выбора варианты исчерпаны, интерпретатор возвращается к предыдущей точке выбора и продолжает оттуда. Такой процесс реализует стратегию поиска, которую называют *поиск в глубину* (depth-first search) или *хронологический поиск с возвратом* (chronological backtracking)[47].

---

[44] На самом деле, различие между произвольным выбором с возвратом единственного значения и возвратом всех возможных значений выбора определяется в некоторой степени точкой зрения. С точки зрения того кода, который использует значение, недетерминистский выбор возвращает одно значение. С точки зрения программиста, проектирующего код, недетерминистский выбор потенциально возвращает все возможные значения, а вычисление разветвляется, вследствие чего каждое значение исследуется отдельно.

[45] Можно возразить, что этот механизм безнадежно неэффективен. Чтобы решить какую-нибудь просто сформулированную задачу таким образом, могут потребоваться миллионы процессоров, и бóльшую часть времени бóльшая часть этих процессоров будет ничем не занята. Это возражение нужно воспринимать в контексте истории. Память раньше точно так же считалась дорогим ресурсом. В 1964 году мегабайт памяти стоил 400 000 долларов. Сейчас в каждом персональном компьютере имеется много мегабайтов памяти, и бóльшую часть времени бóльшая часть этой памяти не используется. Трудно недооценить стоимость электроники при массовом производстве.

[46] Автомагически: «Автоматически, но при этом таким способом, который говорящий почему-либо (обычно либо из-за его сложности, либо уродливости, или даже тривиальности) не склонен объяснять». (Steele 1983; Raymond 1993)

[47] У встраивания стратегий автоматического поиска в языки программирования долгая и пестрая история. Первые предположения, что недетерминистские алгоритмы можно изящно реализовать в языке программиро-

Управляющий цикл

Управляющий цикл amb-интерпретатора не совсем обычен. Он считывает выражение и печатает значение первого успешного вычисления, как в примере с prime-sum-pair в начале раздела. Если нам хочется увидеть значение следующего успешного выполнения, мы можем попросить интерпретатор вернуться и попробовать породить значение следующего успешного выполнения. Для этого нужно ввести символ try-again. Если вводится какое-то другое выражение, а не try-again, интерпретатор начнет решать новую задачу, отбрасывая неисследованные варианты предыдущей. Вот пример работы с интерпретатором:

```
;;; Ввод Amb-Eval:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Начало новой задачи
;;; Значение Amb-Eval:
(3 20)

;;; Ввод Amb-Eval:
try-again
;;; Значение Amb-Eval:
(3 110)

;;; Ввод Amb-Eval:
try-again
;;; Значение Amb-Eval:
(8 35)

;;; Ввод Amb-Eval:
try-again
```

вания с поиском и автоматическим возвратом, высказывались Робертом Флойдом (Floyd 1967). Карл Хьюитт (Hewitt 1969) изобрел язык программирования Плэнер (Planner), который явным образом поддерживал автоматический хронологический поиск в возвратом, обеспечивая встроенную стратегию поиска в глубину. Сассман, Виноград и Чарняк (Sussman, Winograd, and Charniak 1971) реализовали подмножество этого языка, названное ими МикроПлэнер (MicroPlanner), которое использовалось в работе по автоматическому решению задач и планированию действий роботов. Похожие идеи, основанные на логике и доказательстве теорем, привели к созданию в Эдинбурге и Марселе изящного языка Пролог (Prolog) (который мы обсудим в разделе 4.4). Разочаровавшись в автоматическом поиске, Макдермот и Сассман (McDermott and Sussman 1972) разработали язык Коннивер (Conniver), в котором имелись механизмы, позволявшие программисту управлять стратегией поиска. Однако это оказалось слишком громоздким, и Сассман и Столлман (Sussman and Stallman 1975) нашли более удобный в обращении подход, когда исследовали методы символьного анализа электрических цепей. Они разработали схему нехронологического поиска с возвратом, которая была основана на отслеживании логических зависимостей, связывающих факты, и стала известна как метод *поиска с возвратом, управляемого зависимостями* (dependency-directed backtracking). При всей своей сложности, их метод позволял строить достаточно эффективные программы, так как почти не проводилось излишнего поиска. Дойл (Doyle 1979) и Макаллестер (McAllester 1978; McAllester 1980) обобщили и сделали более ясными идеи Столлмана и Сассмана, разработав новую парадигму для формулирования поиска, называемую сейчас *поддержание истины* (truth maintenance). Все современные системы решения задач основаны на какой-либо форме поддержания истины. У Форбуса и де Клеера (Forbus and deKleer 1993) можно найти обсуждение изящных способов строить системы с поддержанием истины и приложения, в которых используется поддержание истины. Заби, Макаллестер и Чепман (Zabih, McAllester, and Chapman 1987) описывают недетерминистское расширение Scheme, основанное на amb; оно похоже на интерпретатор, обсуждаемый в этом разделе, но более сложно, поскольку использует поиск с возвратом, управляемый зависимостями, а не хронологический. Уинстон (Winston 1992) дает введение в обе разновидности поиска с возвратом.

```
;;; Нет больше значений
(prime-sum-pair (quote (1 3 5 8))  (quote (20 35 110)))

;;; Ввод Amb-Eval:
(prime-sum-pair '(19 27 30) '(11 36 58))
;;; Начало новой задачи
;;; Значение Amb-Eval:
(30 11)
```

Упражнение 4.35.
Напишите процедуру `an-integer-between`, которая возвращает целое число, лежащее между двумя заданными границами. С ее помощью можно следующим образом реализовать процедуру для поиска Пифагоровых троек, то есть троек чисел (i, j, k) между заданными границами, таких, что $i \leq j$ и $i^2 + j^2 = k^2$:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

Упражнение 4.36.
В упражнении 3.69 рассматривалась задача порождения потока *всех* Пифагоровых троек, без всякой верхней границы диапазона целых чисел, в котором надо искать. Объясните, почему простая замена `an-integer-between` на `an-integer-startingfrom` в процедуре из упражнения 4.35 не является адекватным способом порождения произвольных Пифагоровых троек. Напишите процедуру, которая решает эту задачу. (Это значит, что Вам нужно написать процедуру, для которой многократный запрос `try-again` в принципе способен породить все Пифагоровы тройки.)

Упражнение 4.37.
Бен Битобор утверждает, что следующий метод порождения Пифагоровых троек эффективнее, чем приведенный в упражнении 4.35. Прав ли он? (Подсказка: найдите, сколько вариантов требуется рассмотреть.)

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high))
        (hsq (* high high)))
    (let ((j (an-integer-between i high)))
      (let ((ksq (+ (* i i) (* j j))))
        (require (>= hsq ksq))
        (let ((k (sqrt ksq)))
          (require (integer? k))
          (list i j k))))))
```

4.3.2. Примеры недетерминистских программ

В разделе 4.3.3 описывается реализация `amb`-интерпретатора. Однако для начала мы приведем несколько примеров его использования. Преимущество недетерминистского

программирования состоит в том, что можно отвлечься от деталей процесса поиска, а следовательно, выражать программы на более высоком уровне абстракции.

Логические загадки

Следующая задача (взятая из Dinesman 1968) — типичный представитель большого класса простых логических загадок.

> Бейкер, Купер, Флетчер, Миллер и Смит живут на разных этажах пятиэтажного дома. Бейкер живет не на верхнем этаже. Купер живет не на первом этаже. Флетчер не живет ни на верхнем, ни на нижнем этаже. Миллер живет выше Купера. Смит живет не на соседнем с Флетчером этаже. Флетчер живет не на соседнем с Купером этаже. Кто где живет?

Можно впрямую определить, кто на каком этаже живет, перечислив все возможности и наложив данные нам ограничения[48].

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Выполнение выражения `(multiple-dwelling)` дает следующий результат:

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

[48] В нашей программе используется следующая процедура, определяющая, все ли элементы списка отличны друг от друга:

```
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
```

Процедура `member` подобна `memq`, но на равенство проверяет с помощью `equal?`, а не `eq?`.

Эта простая процедура работает, но работает очень медленно. В упражнениях 4.39 и 4.40 обсуждаются возможные улучшения.

Упражнение 4.38.
Измените процедуру `multiple-dwelling`, отказавшись от требования, что Смит и Флетчер живут не на соседних этажах. Сколько решений имеется у измененной загадки?

Упражнение 4.39.
Влияет ли порядок ограничений в процедуре `multiple-dwelling` на ответ? Влияет ли он на время, необходимое для поиска ответа? Если Вы считаете, что он имеет значение, то покажите, как можно ускорить программу, переупорядочив ограничения. Если Вы считаете, что порядок значения не имеет, объясните, почему.

Упражнение 4.40.
Сколько возможных соответствий между людьми и этажами имеется в задаче о проживании, если учитывать требование, что все живут на разных этажах, и если его не учитывать? Крайне неэффективно порождать все возможные соответствия между людьми и этажами, а затем полагаться на то, что поиск с возвратом отсечет лишнее. Например, большая часть ограничений зависит только от одной или двух переменных, соответствующих людям, и их можно было бы проверять раньше, чем этажи выбраны для всех действующих лиц. Напишите и продемонстрируйте значительно более эффективную недетерминистскую процедуру, которая бы решала задачу, порождая только те варианты, которые еще не исключены благодаря предыдущим ограничениям. (Подсказка: потребуется набор вложенных выражений `let`.)

Упражнение 4.41.
Напишите процедуру для решения задачи о проживании на обычной Scheme.

Упражнение 4.42.
Решите задачу «Лгуньи» (из Phillips 1934):

> Пять школьниц писали экзаменационную работу. Им показалось, что их родители чересчур интересовались результатом, и поэтому они решили, что каждая девочка должна написать домой о результатах экзамена и при этом сделать одно верное и одно неверное утверждение. Вот соответствующие выдержки из их писем:
>
> Бетти: «Китти была на экзамене второй, а я только третьей».
>
> Этель: «Вам будет приятно узнать, что я написала лучше всех. Второй была Джоан».
>
> Джоан: «Я была третьей, а бедная Этель последней».
>
> Китти: «Я оказалась второй. Мэри была только четвертой».
>
> Мэри: «Я была четвертой. Первое место заняла Бетти».
>
> В каком порядке на самом деле расположились отметки девочек?

Упражнение 4.43.
Решите с помощью `amb`-интерпретатора следующую задачу[49].

[49] Задача взята из книжки «Занимательные загадки», опубликованной в 60-е годы издательством Литтон Индастриз. Книжка приписывает задачу газете «Кэнзас стейт энджинир».

> У отца Мэри Энн Мур есть яхта, и у каждого из четверых его друзей тоже. Эти четверо друзей — полковник Даунинг, мистер Холл, сэр Барнакл Худ и доктор Паркер. У каждого из них тоже есть по дочери, и каждый из них назвал свою яхту в честь дочери одного из своих друзей. Яхта сэра Барнакла называется Габриэлла, яхта мистера Мура — Лорна, а у мистера Холла яхта Розалинда. Мелисса, яхта полковника Даунинга, названа в честь дочери сэра Барнакла. Отец Габриэллы владеет яхтой, названной в честь дочери доктора Паркера. Кто отец Лорны?

Попытайтесь написать программу так, чтобы она работала эффективно (см. упражнение 4.40). Кроме того, определите, сколько будет решений, если не указывается, что фамилия Мэри Энн — Мур.

Упражнение 4.44.
В упражнении 2.42 описывалась «задача о восьми ферзях», в которой требуется расставить на шахматной доске восемь ферзей так, чтобы ни один не бил другого. Напишите недетерминистскую программу для решения этой задачи.

Синтаксический анализ естественного языка

Программы, которые должны принимать на входе естественный язык, обычно прежде всего пытаются провести *синтаксический анализ* (parsing) ввода, то есть сопоставить входному тексту какую-то грамматическую структуру. Например, мы могли бы попытаться распознавать простые предложения, состоящие из артикля, за которым идет существительное, а вслед за ними глагол, например *The cat eats* («Кошка ест»). Чтобы выполнять такой анализ, нам нужно уметь определять части речи, к которым относятся отдельные слова. Мы можем для начала составить несколько списков, которые задают классы слов[50]:

```
(define nouns '(noun student professor cat class))

(define verbs '(verb studies lectures eats sleeps))

(define articles '(article the a))
```

Нам также нужна *грамматика* (grammar), то есть набор правил, которые описывают, как элементы грамматической структуры составляются из меньших элементов. Простейшая грамматика может постановить, что предложение всегда состоит из двух частей — именной группы, за которой следует глагол, — и что именная группа состоит из артикля и имени существительного. С такой грамматикой предложение *The cat eats* разбирается так:

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))
```

Мы можем породить такой разбор при помощи простой программы, в которой для каждого грамматического правила имеется своя процедура. Чтобы разобрать предложение, мы определяем две его составные части и возвращаем список из этих элементов, помеченный символом `sentence`:

[50] Здесь мы используем соглашение, что первый элемент списка обозначает часть речи, к которой относятся остальные слова списка.

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

Подобным образом, разбор именной группы состоит в поиске артикля и существительного:

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

На самом нижнем уровне разбор сводится к многократной проверке, является ли следующее неразобранное слово элементом списка слов для данной части речи. Чтобы реализовать это, мы заводим глобальную переменную `*unparsed*`, содержащую еще неразобранный ввод. Каждый раз, проверяя слово, мы требуем, чтобы `*unparsed*` не была пустым списком и чтобы ее значение начиналось со слова из указанного списка. Если это так, мы убираем слово из `*unparsed*` и возвращаем его вместе с частью речи (которую можно найти в голове списка)[51].

```
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))
```

Чтобы запустить разбор, нужно только присвоить переменной `*unparsed*` весь имеющийся ввод, попытаться проанализировать предложение и убедиться, что ничего не осталось в конце:

```
(define *unparsed* '())

(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))
```

Теперь мы можем опробовать анализатор и убедиться, что он работает на нашем простом примере:

```
;;; Ввод Amb-Eval:
(parse '(the cat eats))
;;; Начало новой задачи
;;; Значение Amb-Eval:
(sentence (noun-phrase (article the) (noun cat)) (verb eats))
```

[51] Обратите внимание, что `parse-word` изменяет список необработанных слов при помощи `set!`. Для того, чтобы это работало, `amb`-интерпретатор при возврате должен отменять действия операций `set!`.

Amb-интерпретатор здесь удобно использовать потому, что ограничения на разбор легко выражаются при помощи `require`. Однако по-настоящему достоинства автоматического поиска с возвратом проявляются тогда, когда мы обращаемся к более сложным грамматикам, где имеются варианты декомпозиции единиц.

Добавим к грамматике список предлогов:

```
(define prepositions '(prep for to in by with))
```

и определим предложную группу (например, *for the cat*, «для кошки») как последовательность из предлога и именной группы:

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
        (parse-word prepositions)
        (parse-noun-phrase)))
```

Теперь мы можем сказать, что предложение — это именная группа, за которой следует глагольная группа, а глагольная группа — это либо глагол, либо глагольная группа, дополненная предложной группой[52]:

```
(define (parse-sentence)
  (list 'sentence
         (parse-noun-phrase)
         (parse-verb-phrase)))

(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                             verb-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
```

Раз уж мы занялись этим делом, можно также уточнить определение именной группы и разрешить выражения вроде *a cat in the class* («кошка в аудитории»). То, что раньше называлось именной группой, теперь мы будем называть простой именной группой, а именная группа теперь может быть либо простой именной группой, либо именной группой, которая дополняется предложной группой:

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))

(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
```

[52]Заметим, что это определение рекурсивно — за глаголом может следовать любое число предложных групп.

```
                              noun-phrase
                              (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
```

Обновленная грамматика позволяет разбирать более сложные предложения. Например,

```
(parse '(the student with the cat sleeps in the class))
```

(«студент с кошкой спит в аудитории») дает

```
(sentence
 (noun-phrase
  (simple-noun-phrase (article the) (noun student))
  (prep-phrase (prep with)
               (simple-noun-phrase
                (article the) (noun cat))))
 (verb-phrase
  (verb sleeps)
  (prep-phrase (prep in)
               (simple-noun-phrase
                (article the) (noun class)))))
```

Заметим, что входное предложение может иметь более одного законного анализа. В предложении *The professor lectures to the student with the cat* («Профессор читает лекцию студенту с кошкой») может иметься в виду, что профессор вместе с кошкой читают лекцию, или что кошка — у студента. Наша недетерминистская программа находит оба варианта:

```
(parse '(the professor lectures to the student with the cat))
```

дает

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb-phrase
   (verb lectures)
   (prep-phrase (prep to)
                (simple-noun-phrase
                 (article the) (noun student))))
  (prep-phrase (prep with)
               (simple-noun-phrase
                (article the) (noun cat)))))
```

Если попросить интерпретатор поискать еще, получится

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb lectures)
```

```
(prep-phrase (prep to)
             (noun-phrase
              (simple-noun-phrase
               (article the) (noun student))
              (prep-phrase (prep with)
                           (simple-noun-phrase
                            (article the) (noun cat)))))))
```

Упражнение 4.45.
Согласно заданной выше грамматике, следующее предложение можно проанализировать пятью различными способами: *The professor lectures to the student in the class with the cat* («Профессор читает лекцию студенту в аудитории с кошкой»). Покажите эти пять разборов и объясните разницу в оттенках значения между ними.

Упражнение 4.46.
Интерпретаторы в разделах 4.1 и 4.2 не определяют, в каком порядке вычисляются операнды при вызове процедуры. Мы увидим, что `amb`-интерпретатор вычисляет их слева направо. Объясните, почему программа разбора не стала бы работать, если бы операнды вычислялись в каком-нибудь другом порядке.

Упражнение 4.47.
Хьюго Дум говорит, что поскольку глагольная группа — это либо глагол, либо глагольная группа плюс предложная группа, было бы намного естественнее определить процедуру `parse-verb-phrase` так (и то же сделать для именных групп):

```
(define (parse-verb-phrase)
  (amb (parse-word verbs)
       (list 'verb-phrase
             (parse-verb-phrase)
             (parse-prepositional-phrase))))
```

Работает ли этот вариант? Изменится ли поведение программы, если мы поменяем местами выражения в `amb`?

Упражнение 4.48.
Дополните описанную выше грамматику так, чтобы она могла работать с более сложными предложениями. Например, можно позволить именным и глагольным группам включать прилагательные и наречия, или же можно обрабатывать сложные предложения[53].

Упражнение 4.49.
Лизу П. Хакер больше интересует не анализ предложений, а их порождение. Она замечает, что если изменить процедуру `parse-word` так, чтобы она игнорировала «входное предложение», всегда

[53]Грамматики такого рода могут быть сколь угодно сложными, но по сравнению с настоящей обработкой естественного языка они остаются игрушкой. Настоящее понимание естественного языка компьютером требует сложного сочетания синтаксического анализа с интерпретацией значения. С другой стороны, даже простые анализаторы могут быть полезны для поддержки гибких командных языков в программах вроде систем поиска информации. Уинстон (Winston 1992) описывает вычислительные подходы к пониманию настоящего естественного языка, а также применение простых грамматик в командных языках.

заканчивалась успехом и порождала подходящее слово, мы сможем использовать те же программы, которые мы написали для анализа, для порождения предложений. Реализуйте идею Лизы и покажите первые пять-шесть порожденных предложений[54].

4.3.3. Реализация amb-интерпретатора

Выполнение выражения в обыкновенной Scheme может вернуть результат, может вообще не завершиться, и, наконец, может закончиться сообщением об ошибке. В недетерминистской Scheme при выполнении выражения, в дополнение ко всему этому, может еще обнаружиться тупик, и в этом случае вычисление должно откатиться к предыдущей точке выбора. Интерпретация недетерминистской Scheme осложняется из-за этой дополнительной возможности.

Мы построим amb-интерпретатор для недетерминистской Scheme, модифицировав анализирующий интерпретатор из раздела 4.1.7[55]. Как и в анализирующем интерпретаторе, вычисление выражения происходит путем вызова исполнительной процедуры, которая получается при анализе этого выражения. Разница между интерпретацией обыкновенной Scheme и недетерминистской Scheme будет полностью сводиться к исполнительным процедурам.

Исполнительные процедуры и продолжения

Напомним, что исполнительные процедуры обыкновенного интерпретатора принимают один аргумент: окружение, в котором происходит вычисление выражения. В противоположность этому, исполнительные процедуры amb-интерпретатора принимают три аргумента: окружение и две процедуры, называемые *процедурами продолжения* (continuation procedures). Вычисление выражения будет заканчиваться вызовом одного из этих продолжений: если результатом вычисления является значение, то зовется *продолжение успеха* (success continuation) с этим значением в качестве аргумента; если вычисление натыкается на тупик, вызывается *продолжение неудачи* (failure continuation). Построение и вызов соответствующих продолжений служит механизмом, с помощью которого в недетерминистском интерпретаторе реализуется поиск с возвратом.

Задача продолжения успеха — принять значение и продолжить вычисление. Помимо этого значения, продолжение успеха получает дополнительное продолжение неудачи, которое нужно будет вызвать, если использование значения приведет в тупик.

Задача продолжения неудачи — попробовать другую ветвь недетерминистского процесса. Главная особенность недетерминистского языка состоит в том, что выражения могут представлять собой точки выбора между вариантами. Выполнение такого выражения должно продолжиться согласно одному из указанных взаимоисключающих вариантов, несмотря на то, что заранее неизвестно, какие варианты приведут к приемлемым

[54] Несмотря на то, что идея Лизы (будучи удивительно простой) дает результат, порождаемые предложения оказываются довольно скучными — они не отображают возможные предложения нашего языка никаким интересным образом. Дело в том, что грамматика рекурсивна во многих местах, а метод Лизы «проваливается» в одну из рекурсий и там застревает. Как с этим можно бороться, Вы увидите в упражнении 4.50.

[55] В разделе 4.2 мы решили реализовать ленивый интерпретатор как модификацию обыкновенного метациклического интерпретатора из раздела 4.1.1. Напротив, здесь в основу amb-интерпретатора мы кладем анализирующий интерпретатор из раздела 4.1.7, поскольку исполнительные процедуры этого интерпретатора служат удобной базой для реализации поиска с возвратом.

результатам. При обработке такой ситуации интерпретатор выбирает один из вариантов и передает его значение продолжению успеха. Кроме того, он строит и передает вместе со значением продолжение неудачи, которое потом можно вызвать, чтобы рассмотреть другой вариант.

Неудача возникает во время вычисления (то есть, зовется продолжение неудачи), когда пользовательская программа явным образом отказывается от текущего рассматриваемого варианта (например, вызов `require` может привести к выполнению `(amb)`, а это выражение всегда терпит неудачу — см. раздел 4.3.1). В этом месте продолжение неудачи вернет нас к последней по времени точке и оттуда направит по другому варианту. Если же в этой точке выбора больше не осталось вариантов, то запускается неудача в предыдущей точке выбора, и так далее. Кроме того, продолжения неудачи запускаются управляющим циклом в ответ на запрос `try-again`, чтобы найти еще одно значение последнего выражения.

Помимо того, если на какой-то ветке процесса, возникшей в результате выбора, происходит операция с побочным эффектом (например, присваивание переменной), то может понадобиться отменить побочный эффект, если процесс наткнется на неудачу, прежде чем будет сделан новый выбор. Этого мы добиваемся, заставив операцию с побочным эффектом порождать продолжение неудачи, которое отменяет эффект и отправляет неудачу дальше.

Итак, продолжения неудачи порождаются

- в выражениях `amb` — чтобы обеспечить механизм выбора альтернативных вариантов, если текущий выбор, сделанный внутри `amb`, приведет к тупику;
- в управляющем цикле верхнего уровня — чтобы иметь возможность сообщить о неудаче, когда перебраны все альтернативы;
- в присваиваниях — чтобы во время отката перехватывать неудачи и отменять присваивания.

Неудачи возбуждаются только тогда, когда программа заходит в тупик. Это происходит

- если пользовательская программа выполняет выражение `(amb)`;
- если пользователь печатает `try-again` в управляющем цикле.

Кроме того, продолжения неудачи вызываются при обработке неудачи:

- Когда продолжение неудачи, порожденное присваиванием, заканчивает отмену побочного эффекта, оно вызывает то предложение неудачи, которое оно само перехватило, и посредством его отправляет неудачу назад до точки выбора, которая привела к присваиванию, либо до верхнего уровня.
- Когда продолжение неудачи для `amb` исчерпывает все варианты выбора, оно вызывает продолжение неудачи, которое изначально было дано `amb`, и посредством его распространяет неудачу до предыдущей точки выбора, либо до верхнего уровня.

Структура интерпретатора

Процедуры представления синтаксиса и данных в `amb`-интерпретаторе, а также базовая процедура `analyze`, совпадают с соответствующими процедурами в интерпретаторе

из раздела 4.1.7, только здесь требуются дополнительные синтаксические процедуры для анализа особой формы amb[56]:

```
(define (amb? exp) (tagged-list? exp 'amb))

(define (amb-choices exp) (cdr exp))
```

Кроме того, требуется добавить в процедуру разбора analyze ветку, которая будет распознавать эту особую форму и порождать соответствующую исполнительную процедуру:

```
((amb? exp) (analyze-amb exp))
```

Процедура верхнего уровня ambeval (сходная с версией eval?, приведенной в разделе 4.1.7) анализирует данное выражение и применяет полученную исполнительную процедуру к данному окружению и двум данным продолжениям:

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

Продолжение успеха представляет собой процедуру от двух аргументов: только что полученного значения и продолжения неудачи, которое нужно будет применить, если обработка значения впоследствии приведет к неудаче. Продолжение неудачи представляет собой процедуру без аргументов. Таким образом, общая форма исполнительной процедуры такова:

```
(lambda (env succeed fail)
    ;;  succeed выглядит как  (lambda (value fail) ...)
    ;;  fail выглядит как  (lambda () ...)
   ...)
```

Например,

```
(ambeval ⟨выражение⟩
         the-global-environment
         (lambda (value fail) value)
         (lambda () 'failed))
```

попытается вычислить данное выражение, и вернет либо его значение (если вычисление будет успешным), либо символ failed (если вычисление потерпит неудачу). Вызов ambeval в нижеприведенном управляющем цикле использует намного более сложные процедуры продолжения, которые возвращаются к выполнению цикла и поддерживают запрос try-again.

Сложность amb-интерпретатора по большей части заключается в механизмах передачи продолжений, когда исполнительные процедуры вызывают друг друга. Читая код в этом разделе, следует сравнивать каждую исполнительную процедуру с соответствующей процедурой обыкновенного интерпретатора из раздела 4.1.7.

[56] Мы предполагаем, что интерпретатор поддерживает let (см. упражнение 4.22), который мы использовали в недетерминистских программах.

Простые выражения

Исполнительные процедуры для простейших видов выражений здесь, в сущности, такие же, как и в обычном интерпретаторе, не считая того, что здесь надо уметь управлять продолжениями. Исполнительные процедуры просто зовут продолжение успеха, давая ему значение выражения, и передают дальше продолжение неудачи, которое получили сами.

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env succeed fail)
      (succeed qval fail))))

(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env)
             fail)))

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env)
               fail))))
```

Заметим, что поиск переменной всегда «успешен». Если процедуре `lookup-variable-value` не удается найти значение, она, как обычно, сообщает об ошибке. Такая «неудача» означает ошибку в программе: ссылку на несвязанную переменную; это не означает, что нам нужно пробовать какой-либо другой вариант недетерминистского выбора вместо того, который исполняется сейчас.

Условные выражения и последовательности

Обработка условных выражений также похожа на соответствующий процесс в обычном интерпретаторе. Исполнительная процедура, порождаемая в `analyze-if`, зовет исполнительную процедуру предиката `pproc` с продолжением успеха, которое, проверив, истинно ли значение предиката, в соответствии с этим выполняет либо следствие, либо альтернативу. Если выполнение `pproc` терпит неудачу, вызывается исходное продолжение неудачи, переданное в выражение `if`.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
```

```
            ;; продолжение успеха при вычислении предиката
            ;; и получении  pred-value
           (lambda (pred-value fail2)
             (if (true? pred-value)
                 (cproc env succeed fail2)
                 (aproc env succeed fail2)))
            ;; продолжение неудачи при вычислении предиката
           fail))))
```

Последовательности тоже обрабатываются так же, как и в предыдущем интерпретаторе, если не считать махинаций в подпроцедуре `sequentially`, которые требуются для передачи продолжений. А именно, чтобы выполнить последовательно `a` и `b`, мы вызываем `a` с продолжением успеха, вызывающим `b`.

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
          ;; продолжение успеха при вызове  a
         (lambda (a-value fail2)
           (b env succeed fail2))
          ;; продолжение неудачи при вызове  a
         fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Пустая последовательность -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

Определения и присваивания

Определения — еще один случай, когда обработка продолжений сопряжена с известными трудностями, поскольку требуется сначала вычислить выражение, которое будет значением определяемой переменной, а затем уже ее собственно определить. Ради этого процедура вычисления значения `vproc` вызывается со следующими аргументами: окружение, продолжение успеха и продолжение неудачи. Если вычисление `vproc` происходит успешно и дает значение `val` для определяемой переменной, то переменная определяется и успех распространяется далее:

```
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)
               (define-variable! var val env)
```

```
            (succeed 'ok fail2))
          fail))))
```

Присваивания устроены интереснее. Это первый случай, когда мы действительно используем продолжения, а не просто передаем их из процедуры в процедуру. Исполнительная процедура для присваивания начинается так же, как и процедура для определения. Сначала она пытается получить новое значение, которое надо присвоить переменной. Если вычисление `vproc` терпит неудачу, неудачно и все присваивание.

Однако если `vproc` выполняется удачно, и мы действительно выполняем присваивание, то нам нужно рассмотреть возможность, что текущая ветка вычисления позже, может быть, приведет к неудаче. Тогда нам понадобится откатиться к моменту до присваивания. Таким образом, нам нужно уметь отменить присваивание в процессе возврата[57].

Этого мы добиваемся, передавая `vproc` продолжение успеха (отмеченное ниже комментарием «**1**»), которое сохраняет старое значение переменной, прежде чем присвоить ей новое значение и продолжить вычисление. Продолжение неудачи, которое передается вместе со значением присваивания (и отмечено ниже комментарием «**2**»), восстанавливает старое значение переменной, прежде чем продолжить откат. То есть, успешное присваивание дает продолжение неудачи, которое перехватит последующую неудачу; неудача, которая в противном случае вызвала бы `fail2`, вместо этого зовет эту процедуру, а она отменяет присваивание и уже затем зовет `fail2`.

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)          ; *1*
               (let ((old-value
                      (lookup-variable-value var env)))
                 (set-variable-value! var val env)
                 (succeed 'ok
                          (lambda ()     ; *2*
                            (set-variable-value! var
                                                 old-value
                                                 env)
                            (fail2)))))
             fail))))
```

Вызов процедур

Исполнительная процедура для вызовов не содержит никаких новшеств, кроме сложных технических деталей работы с продолжениями. Сложность возникает внутри `analyze-application` и обусловлена необходимостью следить за продолжениями успеха и неудачи при вычислении операндов. Мы вычисляем операнды с помощью процедуры `get-args`, а не простого `map`, как в обыкновенном интерпретаторе.

[57] Мы не заботились об отмене определений, поскольку можно предположить, что внутренние определения изымаются (раздел 4.1.6).

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail2)
               (get-args aprocs
                         env
                         (lambda (args fail3)
                           (execute-application
                            proc args succeed fail3))
                         fail2))
             fail))))
```

Заметьте, как в `get-args` для движения через `cdr` по списку исполнительных процедур `aproc` и сборки через `cons` получающегося списка аргументов каждая `aproc` в списке вызывается с продолжением успеха, которое рекурсивно зовет `get-args`. Каждый из этих рекурсивных вызовов `get-args` имеет продолжение успеха, значение которого — `cons` свежеполученного аргумента со списком уже собранных аргументов:

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
                    ;; продолжение успеха для этой  aproc
                    (lambda (arg fail2)
                      (get-args (cdr aprocs)
                                env
                                 ;; продолжение успеха для
                                 ;; рекурсивного вызова  get-args
                                (lambda (args fail3)
                                  (succeed (cons arg args)
                                           fail3))
                                fail2))
                    fail)))
```

Собственно вызов процедуры, который выполняет `execute-application`, осуществляется так же, как и в обыкновенном интерпретаторе, не считая того, что необходимо управлять продолжениями.

```
(define (execute-application proc args succeed fail)
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))
          succeed
```

```
          fail))
        (else
         (error
          "Неизвестный тип процедуры -- EXECUTE-APPLICATION"
          proc))))
```

Выполнение выражений amb

Особая форма amb — ключевой элемент недетерминистского языка. Здесь лежит сущность процесса интерпретации и обоснование необходимости отслеживать продолжения. Исполнительная процедура для amb определяет цикл try-next, который перебирает исполнительные процедуры для всех возможных значений выражения amb. Каждая из исполнительных процедур вызывается с продолжением неудачи, которое попробует выполнить следующий вариант. Когда вариантов больше не остается, все выражение amb терпит неудачу.

```
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
                           succeed
                           (lambda ()
                             (try-next (cdr choices))))))
      (try-next cprocs))))
```

Управляющий цикл

Управляющий цикл amb-интерпретатора сложен из-за наличия механизма, позволяющего пользователю заново попытаться выполнить выражение. Цикл использует процедуру internal-loop, которая в качестве аргумента принимает процедуру try-again. Наш замысел состоит в том, чтобы вызов try-again переходил к следующему нерассмотренному варианту в недетерминистском вычислении. Процедура internal-loop либо зовет try-again, если пользователь набирает try-again в управляющем цикле, либо запускает новое вычисление, вызывая ambeval.

Продолжение неудачи в этом вызове ambeval сообщает пользователю, что значений больше нет, и перезапускает управляющий цикл.

Продолжение успеха для вызова ambeval устроено тоньше. Мы печатаем вычисленное значение, а потом заново запускаем внутренний цикл с процедурой try-again, которая сможет попробовать следующий вариант. Этот переход к следующему варианту выражается процедурой next-alternative, которая передана вторым аргументом в продолжение успеха. Обычно мы считаем этот второй аргумент продолжением неудачи, которое придется использовать, если текущая ветвь исполнения потерпит неудачу. Однако в данном случае мы завершили успешное вычисление, так что «неудачный» вариант можно позвать для того, чтобы найти дополнительные успешные варианты вычисления.

```
(define input-prompt ";;; Ввод Amb-Eval:")
```

```
(define output-prompt ";;; Значение Amb-Eval:")

(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline)
            (display ";;; Начало новой задачи ")
            (ambeval input
                     the-global-environment
                      ;; успех  ambeval
                     (lambda (val next-alternative)
                       (announce-output output-prompt)
                       (user-print val)
                       (internal-loop next-alternative))
                      ;; неудача  ambeval
                     (lambda ()
                       (announce-output
                        ";;; Нет больше значений")
                       (user-print input)
                       (driver-loop)))))))
  (internal-loop
   (lambda ()
     (newline)
     (display ";;; Задача не задана")
     (driver-loop))))
```

Самый первый вызов `internal-loop` использует процедуру `try-again`, которая жалуется, что не было дано никакой задачи, и возобновляет управляющий цикл. Такое поведение требуется, если пользователь набирает `try-again`, еще не задав выражение для вычисления.

Упражнение 4.50.
Реализуйте новую особую форму `ramb`, которая подобна `amb`, однако перебирает варианты не слева направо, а в случайном порядке. Покажите, как такая форма может пригодиться в Лизиной задаче из упражнения 4.49

Упражнение 4.51.
Реализуйте новую разновидность присваивания `permanent-set!` — присваивание, которое не отменяется при неудачах. Например, можно выбрать два различных элемента в списке и посчитать, сколько для этого потребовалось попыток, следующим образом:

```
(define count 0)

(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b c))))
  (permanent-set! count (+ count 1))
```

```
  (require (not (eq? x y)))
  (list x y count))
;;; Начало новой задачи
;;; Значение Amb-Eval:
(a b 2)

;;; Ввод Amb-Eval:
try-again
;;; Значение Amb-Eval:
(a c 3)
```

Какие значения были бы напечатаны, если бы мы вместо `permanent-set!` использовали здесь обычный `set!`?

Упражнение 4.52.
Реализуйте новую конструкцию `if-fail`, которая позволяет пользователю перехватить неудачу при выполнении выражения. `If-fail` принимает два выражения. Первое она выполняет как обычно и, если вычисление успешно, возвращает его результат. Однако если вычисление неудачно, то возвращается значение второго выражения, как в следующем примере:

```
;;; Ввод Amb-Eval:
(if-fail (let ((x (an-element-of '(1 3 5))))
           (require (even? x))
           x)
         'all-odd)
;;; Начало новой задачи
;;; Значение Amb-Eval:
all-odd

;;; Ввод Amb-Eval:
(if-fail (let ((x (an-element-of '(1 3 5 8))))
           (require (even? x))
           x)
         'all-odd)
;;; Начало новой задачи
;;; Значение Amb-Eval:
8
```

Упражнение 4.53.
Если у нас есть `permanent-set!`, описанное в упражнении 4.51, и `if-fail` из упражнения 4.52, то каков будет результат вычисления

```
(let ((pairs '()))
  (if-fail (let ((p (prime-sum-pair '(1 3 5 8) '(20 35 110))))
             (permanent-set! pairs (cons p pairs))
             (amb))
           pairs))
```

Упражнение 4.54.
Если бы мы не догадались, что конструкцию `require` можно реализовать как обычную процедуру с помощью `amb`, так что пользователь сам может определить ее в своей недетерминистской

программе, то нам пришлось бы задать эту конструкцию в виде особой формы. Потребовались бы синтаксические процедуры

```
(define (require? exp) (tagged-list? exp 'require))

(define (require-predicate exp) (cadr exp))
```

новая ветвь разбора в `analyze`:

```
((require? exp) (analyze-require exp))
```

а также процедура `analyze-require`, которая обрабатывает выражения `require`. Допишите следующее определение `analyze-require`:

```
(define (analyze-require exp)
  (let ((pproc (analyze (require-predicate exp))))
    (lambda (env succeed fail)
      (pproc env
             (lambda (pred-value fail2)
               (if ⟨??⟩
                   ⟨??⟩
                   (succeed 'ok fail2)))
             fail))))
```

4.4. Логическое программирование

В главе 1 мы подчеркивали, что информатика имеет дело с императивным знанием («как сделать»), в то время как математика имеет дело с декларативным знанием («что такое»). Действительно, языки программирования требуют, чтобы программист, выражая свои знания, указывал методы пошагового решения определенных задач. С другой стороны, языки высокого уровня обеспечивают в рамках своих реализаций существенный объем методологических знаний, которые освобождает пользователя от забот о многих деталях того, как проходит описываемое вычисление.

Большинство языков программирования, включая Лисп, построены вокруг вычисления значений математических функций. Языки, ориентированные на выражения, (такие, как Лисп, Фортран и Алгол) пользуются тем, что выражение, описывающее значение функции, можно интерпретировать и как способ вычислить это значение. По этой причине большинство языков программирования имеют уклон в однонаправленные вычисления (вычисления со строго определенными входом и выходом). Имеются, однако, совсем другие языки программирования, в которых этот уклон ослаблен. Пример такого языка мы видели в разделе 3.3.5, где объектами вычисления были арифметические ограничения. В системе ограничений направление и порядок вычислений определены не столь четко; стало быть, чтобы провести вычисление, система должна содержать в себе более детальное знание «как сделать», чем в случае с обычным арифметическим вычислением. Однако это не значит, что пользователь вовсе не отвечает за то, чтобы обеспечить систему императивным знанием. Существует множество сетей, которые задают одно и то же множество ограничений, и пользователю нужно выбрать из множества математически эквивалентных сетей одну подходящую, чтобы описать нужное вычисление.

Недетерминистский интерпретатор программ из раздела 4.3 тоже представляет собой отход от представления, что программирование связано с построением алгоритмов для

вычисления однонаправленных функций. В недетерминистском языке у выражений может быть более одного значения, и оттого вычисление работает с отношениями, а не с функциями, у которых значение только одно. Логическое программирование расширяет эту идею, сочетая реляционный взгляд на программирование с мощной разновидностью символьного сопоставления с образцом, которую называют *унификация* (unification)[58].

Когда этот подход работает, он служит весьма мощным способом написания программ. Отчасти эта мощь проистекает из того, что один факт вида «что такое» можно использовать для решения нескольких различных задач с разными компонентами «как сделать». Для примера рассмотрим операцию `append`, которая в качестве аргументов принимает два списка и объединяет их элементы в один список. В процедурном языке вроде Лиспа можно определить `append` через базовый конструктор списков `cons`, как в разделе 2.2.1:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

Эту процедуру можно рассматривать как перевод на Лисп следующих двух правил; первое покрывает случай, когда первый список пуст, а второе — случай непустого списка, представляющего собой `cons` двух частей:

- Для любого списка `y`, `append` пустого списка и `y` дает `y`.
- Для любых `u`, `v`, `y` и `z`, `append` от `(cons u v)` и `y` дает `(cons u z)`, если `append` от `v` и `y` дает `z`[59].

С помощью процедуры `append` мы можем решать задачи типа

Найти `append` от `(a b)` и `(c d)`.

[58] Логическое программирование выросло из долгой традиции исследований по автоматическому доказательству теорем. Ранние программы доказательства теорем достигали лишь скромных результатов, так как они полностью перебирали пространство возможных доказательств. Крупный прорыв, который сделал такой поиск осмысленным, случился в начале 1960х годов, когда были открыты *алгоритм унификации* (unification algorithm) и *принцип резолюции* (resolution principle) (Robinson 1965). Резолюцию использовали, например, Грин и Рафаэль (Green and Raphael 1968, см. также Green 1969) как основу дедуктивной системы вопрос-ответ. Большую часть этого периода исследователи сосредотачивались на алгоритмах, которые гарантированно находят решение, если оно существует. Такими алгоритмами было трудно управлять, и трудно было указать им направление доказательства. Хьюитт (Hewitt 1969) нашел возможность сочетать управляющую структуру языка программирования с операциями системы логического манипулирования, и это привело к появлению работы по автоматическому поиску, упомянутой в разделе 4.3.1 (примечание 47). В то же самое время в Марселе Кольмероэ разрабатывал системы обработки естественного языка, основанные на правилах (см. Colmerauer et al. 1973). Для представления этих правил он изобрел язык Пролог. Ковальски (Kowalski 1973; Kowalski 1979) в Эдинбурге обнаружил, что выполнение программы на Прологе можно интерпретировать как доказательство теорем (с использованием метода доказательства, называемого линейной резолюцией Хорновских форм). Слияние этих двух линий привело к возникновению традиции логического программирования. Таким образом, в споре о приоритетах в области логического программирования французы могут указать на корни Пролога в Марсельском университете, а британцы на работы, сделанные в университете Эдинбурга. А по мнению исследователей из MIT, обе эти группы разработали логическое программирование, когда пытались понять, что же хотел сказать Хьюитт в своей блистательной, но трудночитаемой диссертации. Историю логического программирования можно найти в Robinson 1983.

[59] Соответствие между правилами и процедурой такое: пусть `x` из процедуры (когда `x` непустой) соответствует `(cons u v)` из правила. Тогда `z` из правила соответствует `append` от `(cdr x)` и `y`.

Однако тех же двух правил достаточно для решения следующих типов вопросов, на которые процедура ответить не может:

Найти список `y`, такой, что `append (a b)` и `y` дает `(a b c d)`.

Найти все такие `x` и `y`, что `append` от них дает `(a b c d)`.

В языке логического программирования, когда программист пишет «процедуру» `append`, он формулирует два правила, приведенные выше. Знание «как сделать» автоматически обеспечивается интерпретатором, что позволяет использовать одну эту пару правил для ответа на все три типа вопросов об `append`[60].

У современных языков логического программирования (включая тот, который мы сейчас реализуем) есть существенные недостатки, а именно: их общие методы «как сделать» порой заводят в ненужные бесконечные циклы или вызывают нежелательное поведение другого рода. Логическое программирование сейчас активно исследуется в информатике[61].

Ранее в этой главе мы изучили технологию реализации интерпретаторов и описали те ее элементы, которые необходимы в интерпретаторе Лисп-подобного языка (в сущности, любого традиционного языка). Теперь мы воспользуемся этими идеями при рассмотрении интерпретатора для языка логического программирования. Мы называем этот язык *языком запросов* (query language), поскольку он весьма удобен для извлечения информации из баз данных при помощи *запросов* (queries), то есть выраженных на нашем языке вопросов. Несмотря на то, что язык запросов сильно отличается от Лиспа, его удобно обсуждать в терминах той же самой общей схемы, которую мы использовали до сих пор: как набор элементарных составляющих, дополненных средствами комбинирования, которые позволяют нам сочетать простые составляющие и получать при этом сложные, и средствами абстракции, которые позволяют нам рассматривать сложные составляющие как единые концептуальные единицы. Интерпретатор языка логического программирования существенно сложнее, чем интерпретатор языка типа Лиспа. Тем не менее, нам предстоит убедиться, что наш интерпретатор языка запросов содержит многие из тех же элементов, которые были в интерпретаторе из раздела 4.1. В частности, у нас будет часть «eval», которая классифицирует выражения в соответствии с типом, и часть «apply», которая реализует механизм абстракции языка (процедуры в случае

[60] Это ни в коем случае не освобождает программиста полностью от решения задачи, как вычислить ответ. Существует множество математически эквивалентных наборов правил для отношения `append`, и только некоторые из них можно превратить в эффективное средство для вычисления в каком-либо направлении. Вдобавок, иногда информация «что такое» ничего не говорит о том, *как* вычислить ответ, — возьмем, например, задачу найти такое y, что $y^2 = x$.

[61] Пик интереса к логическому программированию пришелся на начало 80-х, когда японское правительство инициировало амбициозный проект, целью которого было построение сверхбыстрых компьютеров, оптимизированных для логических языков программирования. Скорость таких компьютеров предполагалось измерять в LIPS (Logical Inferences Per Second — число логических выводов в секунду), а не в обычных FLOPS (FLoating-point Operations Per Second — число операций с плавающей точкой в секунду). Несмотря на то, что в рамках проекта удалось создать аппаратное и программное обеспечение, которое изначально планировалось, интересы международной компьютерной промышленности сместились в другом направлении. Обзор и оценку японского проекта можно найти в Feigenbaum and Shrobe 1993. К тому же и в сообществе логических программистов возник интерес к реляционному программированию на основе других методов, помимо простого сопоставления с образцом, например, к работе с численными ограничениями — вроде тех, которые присутствуют в системе распространения ограничений из раздела 3.3.5.

Лиспа и *правила* (rules) в случае логического программирования). Кроме того, в реализации центральную роль будет играть структура данных, построенная из кадров и определяющая соотношение между символами и связанными с ними значениями. Еще одна интересная сторона нашей реализации языка запросов — то, что мы существенным образом используем потоки, введенные в главе 3.

4.4.1. Дедуктивный поиск информации

Логическое программирование хорошо приспособлено для построения интерфейсов к базам данных, служащих для поиска информации. Язык запросов, который мы реализуем в этой главе, спроектирован именно для такого использования.

Чтобы показать, чем занимается система запросов, мы покажем, как с ее помощью управлять базой данных персонала для «Микрошафт», процветающей компании из окрестностей Бостона со специализацией в области высоких технологий. Язык предоставляет возможность поиска информации о сотрудниках, производимого с помощью образцов; он также может осуществлять логический вывод на основании общих правил.

База данных

База данных персонала «Микрошафт» содержит *утверждения* (assertions) о сотрудниках компании. Вот информация о Бене Битоборе, местном компьютерном гуру:

```
(адрес (Битобор Бен) (Сламервилл (Ридж Роуд) 10))
(должность (Битобор Бен) (компьютеры гуру))
(зарплата (Битобор Бен) 60000)
```

Каждое утверждение представляет собой список (в данном случае тройку). элементы которого сами могут быть списками.

В качестве местного гуру Бен отвечает за компьютерный отдел компании и руководит двумя программистами и одним техником. Вот информация о них:

```
(адрес (Хакер Лиза П) (Кембридж (Массачусетс Авеню) 78))
(должность (Хакер Лиза П) (компьютеры программист))
(зарплата (Хакер Лиза П) 40000)
(начальник (Хакер Лиза П) (Битобор Бен))

(адрес (Фект Пабло Э) (Кембридж (Эймс Стрит) 3))
(должность (Фект Пабло Э) (компьютеры программист))
(зарплата (Фект Пабло Э) 35000)
(начальник (Фект Пабло Э) (Битобор Бен))

(адрес (Поправич Дайко) (Бостон (Бэй Стейт Роуд) 22))
(должность (Поправич Дайко) (компьютеры техник))
(зарплата (Поправич Дайко) 25000)
(начальник (Поправич Дайко) (Битобор Бен))
```

Имеется также программист-стажер, над которым начальствует Лиза:

```
(адрес (Дум Хьюго) (Сламервилл (Пайн Три Роуд) 80))
(должность (Дум Хьюго) (компьютеры программист стажер))
```

```
(зарплата (Дум Хьюго) 30000)
(начальник (Дум Хьюго) (Хакер Лиза П))
```

Все эти служащие работают в компьютерном отделе, на что указывает слово **компьютеры** в начале описания их должностей.

Бен — служащий высокого ранга. Его начальник — сам глава компании:

```
(начальник (Битобор Бен) (Уорбак Оливер))

(адрес (Уорбак Оливер) (Суэлсли (Топ Хип Роуд)))
(должность (Уорбак Оливер) (администрация большая шишка))
(зарплата (Уорбак Оливер) 150000)
```

Помимо компьютерного отдела, руководимого Беном, в компании имеется бухгалтерия, где работает главный бухгалтер со своим помощником:

```
(адрес (Скрудж Эбин) (Уэстон (Шейди Лейн) 10))
(должность (Скрудж Эбин) (бухгалтерия главный бухгалтер))
(зарплата (Скрудж Эбин) 75000)
(начальник (Скрудж Эбин) (Уорбак Оливер))

(адрес (Крэтчит Роберт) (Олстон (Норт Гарвард Стрит) 16))
(должность (Крэтчит Роберт) (бухгалтерия писец))
(зарплата (Крэтчит Роберт) 18000)
(начальник (Крэтчит Роберт) (Скрудж Эбин))
```

Есть еще секретарь главы компании:

```
(адрес (Фиден Кон) (Сламервилл (Онион Сквер) 5))
(должность (Фиден Кон) (администрация секретарь))
(зарплата (Фиден Кон) 25000)
(начальник (Фиден Кон) (Уорбак Оливер))
```

Данные содержат также утверждения о том, какой род работы могут выполнять сотрудники, имеющие другую должность. Например, компьютерный гуру способен выполнять работу как компьютерного программиста, так и компьютерного техника:

```
(может-замещать (компьютеры гуру) (компьютеры программист))
(может-замещать (компьютеры гуру) (компьютеры техник))
```

Программист может выполнять работу стажера:

```
(может-замещать (компьютеры программист)
                (компьютеры программист стажер))
```

Кроме того, как всем известно,

```
(может-замещать (администрация секретарь)
                (администрация большая шишка))
```

Простые запросы

Язык запросов дает пользователям возможность извлекать информацию из базы данных, формулируя запросы в ответ на приглашение системы. Например, чтобы найти всех программистов, можно сказать

```
;;; Ввод запроса:
(должность ?x (компьютеры программист))
```

Система выведет следующие результаты:

```
;;; Результаты запроса:
(должность (Хакер Лиза П) (компьютеры программист))
(должность (Фект Пабло Э) (компьютеры программист))
```

Входной запрос указывает, что мы ищем в базе данных записи, соответствующие некоторому *образцу* (pattern). В этом примере образец указывает, что запись должна состоять из трех элементов, из которых первый является символом `должность`, второй может быть чем угодно, а третий представляет собой список `(компьютеры программист)`. «Что угодно», которое может стоять на второй позиции в искомом списке, изображается *переменной образца* (pattern variable) `?x`. В общем случае переменная образца — это символ, который мы считаем именем переменной, предваряемый знаком вопроса. Несколько позже мы увидим, почему имеет смысл давать переменным образца имена, а не просто ставить в образцы `?`, означающее «что угодно». Система отвечает на простой запрос, выводя все записи в базе данных, соответствующие введенному образцу.

В образце может содержаться более одной переменной. Например,

```
(адрес ?x ?y)
```

выводит адреса всех служащих.

В образце может совсем не быть переменных. В этом случае запрос просто проверяет, содержится ли запись в базе. Если да, будет одна подходящая под образец запись; если нет, ни одной.

Одна и та же переменная может встречаться в образце в нескольких местах, и это означает, что одинаковое «что угодно» должно встретиться в каждом из этих мест. Ради этого переменным и даются имена. Например,

```
(начальник ?x ?x)
```

находит всех сотрудников, которые сами себе начальники (впрочем, в нашей пробной базе таковых не имеется).

Запросу

```
(должность ?x (компьютеры ?type))
```

соответствуют все записи о должностях, в которых третий элемент является двухэлементным списком, а первый элемент в нем `компьютеры`:

```
(должность (Битобор Бен) (компьютеры гуру))
(должность (Хакер Лиза П) (компьютеры программист))
(должность (Фект Пабло Э) (компьютеры программист))
(должность (Поправич Дайко) (компьютеры техник))
```

Этому образцу *не* соответствует запись

```
(должность (Дум Хьюго) (компьютеры программист стажер))
```

поскольку третий элемент здесь является списком из трех элементов, а третий элемент образца указывает, что элементов должно быть два. Если бы нам захотелось изменить образец так, чтобы третий элемент мог быть любым списком, который начинается с `компьютеры`, мы могли бы написать[62]

```
(должность ?x (компьютеры . ?type))
```

Например,

```
(компьютеры . ?type)
```

соответствуют данные

```
(компьютеры программист стажер)
```

причем `?type` равняется списку `(программист стажер)`. Тому же образцу соответствуют данные

```
(компьютеры программист)
```

где `?type` равняется списку `(программист)`, и данные

```
(компьютеры)
```

где `?type` равняется пустому списку `()`.

Можно следующим образом описать обработку простых запросов в нашем языке:

• Система находит все присваивания переменным в образце запроса, которые *удовлетворяют* (satisfy) запросу — то есть, все наборы значений переменных, такие, что если переменные образца *конкретизуются* (are instantiated), то есть замещаются, своими значениями, то результат находится в базе данных.

• Система отвечает на запрос, перечисляя все конкретизации образца с удовлетворяющими ему присваиваниями переменным.

Заметим, что в случае, когда образец не содержит переменных, запрос сводится к выяснению, находится ли образец в базе. Если да, то нулевое присваивание, не сообщающее значений никаким переменным, удовлетворяет запросу к текущей базе данных.

Упражнение 4.55.
Постройте простые запросы, которые извлекают из базы данных следующую информацию:

а. Все сотрудники, начальником которых является Бен Битобор.

б. Имена и должности всех работников бухгалтерии.

в. Имена и адреса всех сотрудников, живущих в Сламервилле.

[62] Здесь используется точечная запись, введенная в упражнении 2.20.

Составные запросы

Простые запросы являются элементарными операциями языка запросов. Чтобы порождать составные операции, язык предоставляет средства комбинирования. Один из элементов, превращающих язык запросов в язык логического программирования — то, что средства комбинирования запросов отражают средства комбинирования, используемые при построении логических выражений: `and` (и), `or` (или) и `not` (не). (Здесь `and`, `or` и `not` — это не элементарные выражения Лиспа, а операции, встроенные в язык запросов.)

Мы можем найти адреса всех программистов с помощью `and` так:

```
(and (должность ?person (компьютеры программист))
     (адрес ?person ?where))
```

Получаем на выводе

```
(and (должность (Хакер Лиза П) (компьютеры программист))
     (адрес (Хакер Лиза П) (Кембридж (Массачусетс Авеню) 78)))

(and (должность (Фект Пабло Э) (компьютеры программист))
     (адрес (Фект Пабло Э) (Кембридж (Эймс Стрит) 3)))
```

В общем случае, запросу

(and ⟨*запрос*$_1$⟩ ... ⟨*запросn*⟩)

удовлетворяют все наборы значений переменных образца, которые одновременно удовлетворяют ⟨*запросу*$_1$⟩ ... ⟨*запросу*$_n$⟩.

Как и в случае с простыми запросами, система при обработке составного запроса находит все присваивания переменным образца, удовлетворяющие запросу, и затем печатает все конкретизации запроса этими значениями.

Другой метод построения составных запросов — через `or`. Например,

```
(or (начальник ?x (Битобор Бен))
    (начальник ?x (Хакер Лиза П)))
```

найдет всех сотрудников, над которыми начальствует Бен Битобор или Лиза П. Хакер:

```
(or (начальник (Хакер Лиза П) (Битобор Бен))
    (начальник (Хакер Лиза П) (Хакер Лиза П)))

(or (начальник (Фект Пабло Э) (Битобор Бен))
    (начальник (Фект Пабло Э) (Хакер Лиза П)))

(or (начальник (Поправич Дайко) (Битобор Бен))
    (начальник (Поправич Дайко) (Хакер Лиза П)))

(or (начальник (Дум Хьюго) (Битобор Бен))
    (начальник (Дум Хьюго) (Хакер Лиза П)))
```

В общем случае, запросу

```
(or ⟨запрос1⟩ ... ⟨запросn⟩)
```

удовлетворяют все наборы значений переменных образца, которые удовлетворяют по крайней мере одному из ⟨*запроса*$_1$⟩ ... ⟨*запроса*$_n$⟩.

Кроме того, составные запросы можно порождать при помощи `not`. Например,

```
(and (начальник ?x (Битобор Бен))
     (not (должность ?x (компьютеры программист))))
```

ищет всех сотрудников, для которых начальник Бен Битобор, не являющихся программистами. В общем случае, запросу

```
(not ⟨запрос1⟩)
```

удовлетворяют все присваивания переменным образца, которые не удовлетворяют ⟨*запросу*$_1$⟩[63].

Последняя комбинирующая форма называется `lisp-value`. Когда она стоит в начале образца, она указывает, что следующий элемент является предикатом Лиспа, который требуется применить к остальным (конкретизированным) элементам как к аргументам. В общем случае, образец

```
(lisp-value ⟨предикат⟩ ⟨арг1⟩ ... ⟨argn⟩)
```

удовлетворяется теми присваиваниями переменным образца, для которых применение ⟨*предиката*⟩ к конкретизированным ⟨*арг*$_1$⟩ ... ⟨*арг*$_n$⟩ дает истину. Например, чтобы найти всех сотрудников с зарплатой выше 30000 долларов, можно написать[64]

```
(and (зарплата ?person ?amount)
     (lisp-value > ?amount 30000))
```

Упражнение 4.56.
Сформулируйте составные запросы для получения следующей информации:

а. имена всех сотрудников, у которых начальником Бен Битобор, и их адреса;

б. все сотрудники, чья зарплата ниже, чем у Бена Битобора, вместе с их зарплатой и зарплатой Бена;

в. все сотрудники, у которых начальник не относится к компьютерному отделу, вместе с именем и должностью их начальника.

Правила

В дополнение к элементарным и составным запросам, наш язык обладает средством абстракции запросов. Это *правила* (rules). Правило

[63] Это описание `not` верно только для простых случаев. На самом деле поведение этой конструкции более сложное. Мы исследуем тонкости `not` в разделах 4.4.2 и 4.4.3.

[64] `Lisp-value` имеет смысл использовать только для тех операций, которых нет в языке запросов. В частности, с его помощью не следует проверять равенство (так как для этого предназначено сопоставление в языке запросов) и неравенство (так как это можно сделать посредством правила `same`, приведенного ниже).

```
(rule (живет-около ?person-1 ?person-2)
      (and (адрес ?person-1 (?town . ?rest-1))
           (адрес ?person-1 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

говорит, что двое людей живут рядом друг с другом, если они живут в одном городе. Выражение `not` в конце необходимо для того, чтобы правило не говорило про всех людей, что они живут сами около себя. Отношение `same` (тот же самый) определяется очень простым правилом[65]:

```
(rule (same ?x ?x))
```

Следующее правило объявляет, что сотрудник является «шишкой», если он начальствует над кем-нибудь, кто сам является начальником:

```
(rule (шишка ?person)
      (and (начальник ?middle-manager ?person)
           (начальник ?x ?middle-manager)))
```

В общем случае правило выглядит как

(rule ⟨*заключение*⟩ ⟨*тело*⟩)

где ⟨*заключение*⟩ — это образец, а ⟨*тело*⟩ — произвольный запрос[66]. Можно считать, что правило представляет собой большое (даже бесконечное) множество утверждений, а именно, все конкретизации заключения при помощи присваиваний переменным, удовлетворяющих телу правила. Когда мы описывали простые запросы (образцы), мы сказали, что присваивание переменным удовлетворяет образцу в том случае, когда конкретизированный образец имеется в базе данных. Однако образец не обязательно должен явно присутствовать в базе данных как утверждение. Это может быть неявное утверждение, следующее из правила. Например, запрос

```
(живет-около ?x (Битобор Бен))
```

выдает

```
(живет-около (Дум Хьюго) (Битобор Бен))
(живет-около (Фиден Кон) (Битобор Бен))
```

Чтобы найти всех программистов, живущих около Бена Битобора, можно спросить

[65]Заметим, что правило `same` не нужно для того, чтобы сделать два объекта одинаковыми: достаточно просто использовать одну и ту же переменную образца — тогда у нас с самого начала будет иметься только один объект, а не два. Например, обратите внимание на `?town` в правиле `живет-около` или `?middle-manager` в правиле `шишка` ниже. `Same` оказывается полезным, когда нам хочется, чтобы два объекта различались, как `?person-1` и `?person-2` в правиле `живет-около`. При том, что использование одной переменной в двух местах в запросе требует, чтобы в обоих местах присутствовало одно значение, использование разных переменных не означает, что значения различаются. (Значения, присваиваемые различным переменным образца, могут быть как разными, так и одинаковыми.)

[66]Кроме того, мы разрешаем иметь правила без тела, вроде `same`, и будем полагать, что такое правило означает, что заключение удовлетворяется любыми значениями переменных.

```
(and (должность ?x (компьютеры программист))
     (живет-около ?x (Битобор Бен)))
```

Как и в случае с составными процедурами, правила можно использовать внутри других правил (как мы видели в `живет-около`), и они даже могут быть рекурсивными. Например, правило

```
(rule (одчиняется ?staff-person ?boss)
      (or (начальник ?staff-prerson ?boss)
          (and (начальник ?staff-person ?middle-manager)
               (подчиняется ?middle-manager ?boss))))
```

говорит, что служащий подчиняется руководителю, если руководитель командует им непосредственно или (рекурсивно) непосредственный начальник служащего подчиняется руководителю.

Упражнение 4.57.
Определите правило, которое говорит, что служащий 1 может заменить служащего 2, если либо служащий 1 имеет ту же должность, что и служащий 2, либо человек с должностью служащего 1 может выполнять работу служащего 2, и при этом служащие 1 и 2 — разные люди. Используя это правило, составьте запросы, которые находят следующую информацию:

а. все служащие, которые могут заменить П.Э. Фекта.

б. все служащие, которые могут заменить кого-то, кто получает больше их самих, с указанием двух зарплат.

Упражнение 4.58.
Определите правило, которое говорит, что человек «независим» в своем отделе, если он работает в этом отделе, но у него нет начальника, который работает в том же отделе.

Упражнение 4.59.
Бен Битобор пропускает слишком много совещаний. Опасаясь потерять из-за этой глупой привычки работу, он решает, что с ней надо что-то делать. Он добавляет данные обо всех еженедельных совещаниях в базу данных «Микрошафт» в виде следующих утверждений:

```
(совещание бухгалтерия (понедельник 9))
(совещание администрация (понедельник 10))
(совещание компьютеры (среда 15))
(совещание администрация (пятница 13))
```

Все эти утверждения сообщают о совещаниях отделов. Кроме того, Бен вводит утверждение о совещании всей компании, которое относится ко всем отделам. Все сотрудники компании должны ходить на это совещание.

```
(совещание вся-компания (среда 16))
```

а. В пятницу утром Бен хочет спросить у базы данных, какие совещания происходят в этот день. Как ему надо составить запрос?

б. Лиза П. Хакер недовольна. Она считает, что намного полезнее было бы, если бы можно было спрашивать о совещаниях, указывая свое имя. Она пишет правило, гласящее, что совещания, куда служащему надо ходить, это совещания всей компании и совещания отдела, где он работает. Допишите тело Лизиного правила.

```
(rule (время-совещания ?person ?day-and-time)
      ⟨тело⟩)
```

в. Лиза приходит на работу в среду и хочет узнать, на какие совещания ей нужно идти в этот день. Если имеется правило `время-совещания`, то какой запрос ей надо подать?

Упражнение 4.60.
Подав запрос

```
(живет-около ?person (Хакер Лиза П))
```

Лиза П. Хакер может найти людей, которые живут с ней рядом, и с которыми она вместе может ездить на работу. С другой стороны, когда она пытается найти все пары людей, живущих друг около друга, при помощи запроса

```
(живет-около ?person-1 ?person-2)
```

она видит, что каждая подходящая пара людей попадается в выводе дважды, например

```
(живет-около (Хакер Лиза П) (Фект Пабло Э))
(живет-около (Фект Пабло Э) (Хакер Лиза П))
```

Почему так происходит? Можно ли получить список людей, живущих рядом друг с другом, в котором каждая пара появлялась бы по одному разу? Ответ объясните.

Логика как программы

Можно рассматривать правило как своего рода логическую импликацию: *если* присваивание значений переменным образца удовлетворяет телу, *то* оно удовлетворяет заключению. Следовательно, можно считать, что язык запросов способен производить *логический вывод* (logical deduction) на основании правил. В качестве примера рассмотрим операцию `append`, описанную в начале раздела 4.4. Как мы уже сказали, `append` характеризуется следующими двумя правилами:

- Для любого списка `y`, `append` пустого списка и `y` дает `y`
- Для любых `u`, `v`, `y` и `z`, `append` от `(cons u v)` и `y` дает `(cons u z)`, если `append` от `v` и `y` дает `z`.

Чтобы выразить это в нашем языке запросов, мы определяем два правила для отношения

```
(append-to-form x y z)
```

которое можно интерпретировать как «`append` от `x` и `y` дает `z`»:

```
(rule (append-to-form () ?y ?y))

(rule (append-to-form (?u . ?v) ?y (?u . ?z))
      (append-to-form ?v ?y ?z))
```

В первом правиле нет тела, и это означает, что следствие выполняется для любого значения `?y`. Обратите также внимание, как во втором правиле `car` и `cdr` списка изображаются с использованием точечной записи.

При помощи этих двух правил мы можем формулировать запросы, которые вычисляют `append` от двух списков:

```
;;; Ввод запроса:
(append-to-form (a b) (c d) ?z)
;;; Результаты запроса:
(append-to-form (a b) (c d) (a b c d))
```

Удивительнее то, что мы с помощью тех же правил можем задать вопрос «Какой список, будучи добавлен к `(a b)`, дает `(a b c d)`?» Это делается так:

```
;;; Ввод запроса:
(append-to-form (a b) ?y (a b c d))
;;; Результаты запроса:
(append-to-form (a b) (c d) (a b c d))
```

Можно также запросить все пары списков, `append` от которых дает `(a b c d)`:

```
;;; Ввод запроса:
(append-to-form ?x ?y (a b c d))
;;; Результаты запроса:
(append-to-form () (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) () (a b c d))
```

Может показаться, что, используя правила для вывода ответов на перечисленные запросы, система демонстрирует немалый интеллект. На самом же деле, как мы увидим в следующем разделе, при разборе правил она следует хорошо определенному алгоритму. К сожалению, хотя в случае с `append` результаты впечатляют, в более сложных ситуациях общие методы могут не сработать, как мы увидим в разделе 4.4.3.

Упражнение 4.61.
Следующие правила определяют отношение `next-to`, которое находит в списке соседние элементы:

```
(rule (?x next-to ?y in (?x ?y . ?u)))

(rule (?x next-to ?y in (?v . ?z))
      (?x next-to ?y in ?z))
```

Каков будет ответ на следующие запросы?

```
(?x next-to ?y in (1 (2 3) 4))

(?x next-to 1 in (2 1 3 1))
```

Упражнение 4.62.
Определите правила, которые реализуют операцию `last-pair` из упражнения 2.17, которая возвращает последнюю пару непустого списка. Проверьте Ваши правила на таких запросах, как `(last-pair (3) ?x)`, `(last-pair (1 2 3) ?x)` и `(last-pair (2 ?x) (3))`. Правильно ли Ваши правила работают с запросами вида `(last-pair ?x (3))`?

Упражнение 4.63.
Следующая база данных (см. книгу Бытия, 4) содержит генеалогию сыновей Ады вплоть до Адама, через Каина:

```
(сын Адам Каин)
(сын Каин Енох)
(сын Енох Ирад)
(сын Ирад Мехиаель)
(сын Мехиаель Мафусал)
(сын Мафусал Ламех)
(жена Ламех Ада)
(сын Ада Иавал)
(сын Ада Иувал)
```

Сформулируйте правила, такие как «Если S сын F, а F сын G, то S внук G» и «Если W жена M, а S сын W, то S также сын M» (предполагается, что в библейские времена это в большей степени соответствовало истине, чем теперь). Эти правила должны позволить системе найти внука Каина; сыновей Ламеха; внуков Мафусала. (В упражнении 4.69 можно найти правила, с помощью которых выводятся более сложные родственные связи.)

4.4.2. Как действует система обработки запросов

В разделе 4.4.4 мы представим реализацию интерпретатора запросов в виде набора процедур. В этом разделе дается обзор системы и объясняется ее общая структура, без низкоуровневых деталей реализации. После того, как мы опишем интерпретатор, нам легче будет понять его ограничения и некоторые тонкости, в которых логические операции языка запросов отличаются от операций математической логики.

Должно быть очевидно, что вычислителю запросов требуется какая-то разновидность поиска, чтобы сопоставлять запросы с фактами и правилами в базе данных. Одним из способов сделать это была бы реализация системы запросов в виде недетерминистской программы с использованием `amb`-интерпретатора из раздела 4.3 (см. упражнение 4.78). Другая возможность состоит в том, чтобы управлять поиском при помощи потоков. Наша реализация использует этот второй подход.

Запросная система организована вокруг двух основных операций, которые называются *сопоставление с образцом* (pattern matching) и *унификация* (unification). Сначала мы опишем сопоставление с образцом и объясним, как эта операция, вместе с организацией информации в виде потоков кадров, позволяет нам реализовывать как простые, так и составные запросы. Затем мы обсудим унификацию — обобщение сопоставления с образцом, которое требуется для реализации правил. Наконец, мы покажем, как части интерпретатора связываются воедино процедурой, классифицирующей выражения, подобно тому, как `eval` разбирает выражения в интерпретаторе, описанном в разделе 4.1.

Сопоставление с образцом

Сопоставитель (pattern matcher) — это программа, которая проверяет, соответствует ли некоторая структура данных указанному образцу. Например, список `((a b) c (a b))` соответствует образцу `(?x c ?x)` при значении переменной `?x`, равном `(a b)`. Этот же список соответствует образцу `(?x ?y ?z)` при значениях переменных `?x` и

`?z`, равных `(a b)` и значении `?y`, равном `b`. Однако он не соответствует образцу `(?x a ?y)`, поскольку этот образец требует, чтобы вторым элементом списка был символ `a`.

Сопоставитель, который используется в запросной системе, принимает на входе образец, структуру данных и *кадр* (frame), в котором указываются связывания для различных переменных образца. Он проверяет, соответствует ли структура данных образцу без противоречия со связываниями переменных, уже находящимися в кадре. Если да, то сопоставитель возвращает кадр, дополнив его связываниями, определенными во время сопоставления. Если нет, он указывает, что сопоставление неудачно.

Например, сопоставление образца `(?x ?y ?x)` со списком `(a b a)` при пустом начальном кадре вернет кадр, в котором переменная `?x` связана со значением `a`, а `?y` со значением `b`. Попытка сопоставления того же списка с тем же образцом при начальном кадре, в котором указывается, что `?y` связывается с `a`, окажется неудачной. Попытка сопоставления с теми же данными и образцом, при начальном кадре, в котором `?y` связана со значением `b`, а `?x` несвязана, вернет исходный кадр, дополненный связыванием `a` для `?x`.

Сопоставитель — единственный механизм, необходимый для обработки простых образцов, не содержащих правил. Например, чтобы обработать запрос:

```
(должность ?x (компьютеры программист))
```

— мы просматриваем все утверждения в базе данных и выбираем те, которые соответствуют образцу при пустом начальном кадре. Для каждого найденного утверждения мы подставляем в образец значение `?x` из кадра, полученного при сопоставлении.

Потоки кадров

Проверка образцов по отношению к кадрам организована посредством потоков. Получив кадр, процесс сопоставления просматривает элементы базы данных один за другим. Для каждого входа базы данных сопоставитель порождает либо специальный символ, указывающий, что сопоставление оказалось неудачным, либо расширение кадра. Из результатов сопоставления всей базы данных собирается поток, и этот поток пропускается через фильтр, отбрасывающий неудачи. Получается поток всех кадров, которые расширяют данный кадр за счет сопоставления с каким-либо утверждением из базы[67].

В нашей системе запрос принимает входной поток кадров и для каждого кадра применяет вышеописанную операцию сопоставления, как показано на рис. 4.4. А именно, для каждого кадра во входном потоке запрос генерирует новый поток, содержащий все расширения этого кадра, порожденные сопоставлением с утверждениями из базы. Затем все эти потоки собираются в один громадный поток, который содержит все возможные расширения всех кадров входного потока. Этот поток и есть результат запроса.

[67] Поскольку сопоставление — в общем случае весьма дорогая операция, нам хотелось бы избежать применения полного сопоставителя к каждому элементу базы данных. Обычное решение этой проблемы — разбить процесс на грубое (быстрое) сопоставление и окончательное сопоставление. Грубое сопоставление отфильтровывает базу и находит кандидатуры на окончательное сопоставление. Если действовать аккуратно, можно построить базу данных таким образом, что часть работы грубого сопоставителя проделывается при построении базы, а не в момент отбора кандидатов. Это называется *индексированием* (indexing) базы данных. Существует множество приемов и схем индексирования баз данных. Наша реализация, которую мы описываем разделе 4.4.4, содержит простейший вариант такой оптимизации.

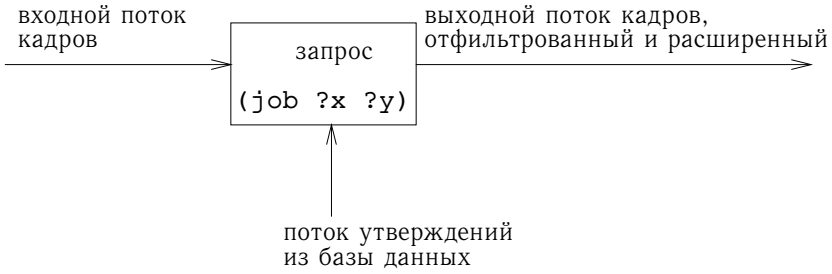


Рис. 4.4. Запрос обрабатывает поток кадров

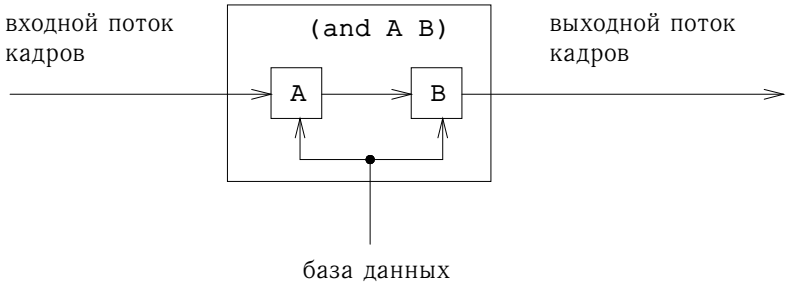


Рис. 4.5. Комбинация двух запросов через `and` осуществляется последовательной обработкой потока кадров.

Чтобы ответить на простой запрос, мы применяем его к потоку, который состоит из одного пустого кадра. Поток на выходе содержит все расширения пустого кадра (то есть, все ответы на наш запрос). Затем на основе этого потока кадров создается поток копий исходного образца запроса, в которых переменные конкретизированы значениями из всех кадров, и этот поток в конце концов печатается.

Составные запросы

Изящество реализации с потоками кадров по-настоящему проявляется при работе с составными запросами. При обработке составных запросов мы пользуемся тем, что наш сопоставитель умеет требовать, чтобы сопоставление не противоречило указанному кадру. Например, чтобы обработать `and` от двух запросов, скажем,

```
(and (может-замещать ?x (компьютеры программист стажер))
     (должность ?person ?x))
```

(неформально, «найти всех сотрудников, способных выполнять работу программиста-стажера»), сначала мы находим все записи базы, отвечающие образцу

```
(может-замещать ?x (компьютеры программист стажер))
```

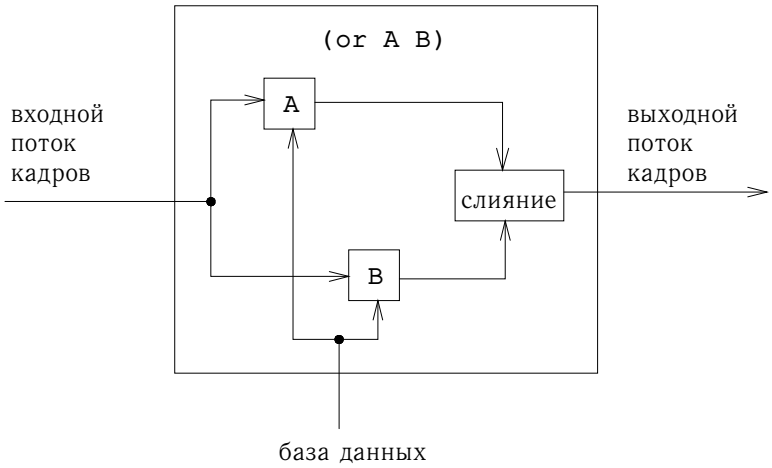


Рис. 4.6. Комбинация двух запросов через `or` осуществляется путем параллельной обработки потока кадров и слияния результатов.

Получается поток кадров, каждый из которых содержит связывание для `?x`. Затем для всех кадров этого потока мы находим записи, соответствующие образцу

```
(должность ?person ?x)
```

таким образом, чтобы не менять уже известное связывание переменной `?x`. Каждое новое сопоставление породит кадр, в котором будут содержаться связывания для `?x` и `?person`. `And` от двух запросов можно рассматривать как последовательное применение двух составляющих запросов, как показано на рис. 4.5. Кадры, прошедшие через первый запрос, фильтруются и расширяются вторым запросом.

На рисунке 4.6 показан аналогичный метод для вычисления `or` от двух запросов через параллельное выполнение двух составляющих запросов. Каждый запрос отдельно расширяет входной поток кадров. Затем два получившихся потока сливаются и образуют окончательный поток-результат.

Даже из этого высокоуровневого описания ясно, что обработка составных запросов может занимать много времени. Например, поскольку запрос может породить более одного выходного кадра для каждого входного, а каждый подзапрос в `and` принимает входные кадры от предыдущего подзапроса, в наихудшем случае число сопоставлений, которые должен произвести запрос `and`, растет экспоненциально с числом подзапросов (см. упражнение 4.76)[68]. Несмотря на то, что системы для обработки простых запросов вполне могут быть практически полезны, обработка сложных запросов чрезвычайно трудоемка[69].

[68] Впрочем, такой экспоненциальный взрыв в запросах `and` происходит редко, поскольку, как правило, дополнительные условия не увеличивают, а уменьшают число порождаемых кадров.

[69] Имеется обширная литература по системам управления базами данных, в которой основной темой является эффективная обработка сложных запросов.

С точки зрения потока кадров, запрос `not` работает как фильтр, уничтожающий все кадры, для которых его подзапрос можно удовлетворить. Например, имея образец

```
(not (должность ?x (компьютеры программист)))
```

мы для каждого кадра во входном потоке пытаемся породить расширенные кадры, которые удовлетворяют образцу `(должность ?x (компьютеры программист))`. Все кадры, для которых такие расширения существуют, мы изымаем из входного потока. В результате получается поток, состоящий только из тех кадров, в которых связывание для `?x` не удовлетворяет `(должность ?x (компьютеры программист))`. Например, при обработке запроса

```
(and (начальник ?x ?y)
     (not (должность ?x (компьютеры программист))))
```

первый подзапрос породит кадры со связанными значениями `?x` и `?y`. Затем выражение `not` отфильтрует этот поток, удалив все кадры, в которых значение `?x` удовлетворяет ограничению, что `?x` является программистом[70].

Особая форма `lisp-value` реализуется при помощи подобного же фильтра для потоков кадров. При помощи каждого кадра из потока мы конкретизируем все переменные образца, а затем применяем лисповский предикат. Все кадры, для которых предикат оказывается ложным, мы удаляем из входного потока.

Унификация

Чтобы обрабатывать правила языка запросов, нам нужно уметь находить правила, в которых заключения соответствуют данному входному образцу. Заключения правил подобны утверждениям, но только в них могут содержаться переменные, так что нам требуется обобщенный вариант сопоставления с образцом, — называемый *унификация* (unification), — в котором как «образец», так и «данные» могут содержать переменные.

Унификатор берет два образца, в каждом из которых могут быть константы и переменные, и определяет, возможно ли присвоить переменным значения, которые сделают два образца одинаковыми. Если да, то он возвращает кадр, содержащий эти значения. Например, при унификации `(?x a ?y)` и `(?y ?z a)` получится кадр, в котором все три переменные `?x`, `?y` и `?z` связаны со значением `a`. С другой стороны, унификация `(?x ?y a)` и `(?x b ?y)` потерпит неудачу, поскольку не имеется такого значения для `?y`, которое бы сделало два образца одинаковыми. (Чтобы вторые элементы образцов оказались равными, `?y` должно равняться `b`; однако, чтобы совпали третьи элементы, `?y` обязан быть `a`.) Подобно сопоставителю, унификатор, используемый в системе запросов, принимает на входе кадр и проводит унификации, не противоречащие содержимому этого кадра.

Алгоритм унификации — самая технически сложная часть запросной системы. При наличии сложных образцов может показаться, что для унификации требуются дедуктивные способности. Например, чтобы унифицировать `(?x ?x)` и `((a ?y c) (a b ?z))`, алгоритм обязан вычислить, что `?x` должен быть равен `(a b c)`, `?y` должен

[70]Существует тонкое различие между реализацией `not` в виде фильтра и значением отрицания в математической логике. См. раздел 4.4.3.

быть ?b, а ?z должен быть равен c. Можно считать, что этот процесс решает систему уравнений, описывающую компоненты образцов. В общем случае это будут взаимозависимые уравнения, для решения которых требуются существенные преобразования[71]. К примеру, унификацию (?x ?x) и ((a ?y c) (a b ?z)) можно рассматривать как систему уравнений

```
?x = (a ?y c)
?x = (a b ?z)
```

Из этих уравнений следует, что

```
(a ?y c) = (a b ?z)
```

а отсюда, в свою очередь, что

```
a = a, ?y = b, c = ?z
```

и, следовательно,

```
?x = (a b c)
```

При успешном сопоставлении с образцом все переменные оказываются связанными, и значения, с которыми они связаны, содержат только константы. Это верно и для всех примеров унификации, которые мы до сих пор рассмотрели. Однако в общем случае успешная унификация может не полностью определить значения переменных; какие-то переменные могут остаться неопределенными, а значения других сами могут содержать переменные.

Рассмотрим унификацию (?x a) и ((b ?y) ?z). Можно вычислить, что ?x = (b ?y), а a = ?z, но ничего больше нельзя сказать о значениях ?x и ?y. Унификация заканчивается успешно, поскольку, естественно, можно сделать образцы одинаковыми, присвоив значения ?x и ?y. Поскольку сопоставление никак не ограничивает значение, которое может принимать переменная ?y, никакого ее значения не оказывается в кадре-результате. Однако результат ограничивает значение ?x. Какое бы значение не имела переменная ?y, ?x должен равняться (b ?y). Таким образом, в кадр помещается связывание ?x со значением (b ?y). Если позже значение ?y оказывается определенным (путем сопоставления с образцом или унификации, которая должна соответствовать этому кадру) и добавляется в кадр, значение, связанное с ?x, будет ссылаться на него[72].

Применение правил

Главной операцией в компоненте запросной системы, который производит логический вывод на основе правил, является унификация. Чтобы увидеть, как этот компонент работает, рассмотрим обработку запроса, содержащего обращение к правилу, например:

[71] В одностороннем сопоставлении с образцом все уравнения, которые содержат переменные, заданы явно и уже решены относительно неизвестного (переменной образца).

[72] Можно считать, что унификация находит наиболее общий образец, который является специализацией двух входных образцов. А именно, унификация (?x a) и ((b ?y) ?z) равна ((b ?y) a), а унификация (?x a ?y) и (?y ?z a), описанная выше, равна (a a a). Однако в нашей реализации удобнее считать, что результатом унификации является не образец, а кадр.

```
(живет-около ?x (Хакер Лиза П))
```

Обрабатывая этот запрос, сначала мы при помощи описанной ранее обыкновенной процедуры сопоставления смотрим, имеются ли в базе данных утверждения, которые сопоставляются с данным образцом. (В данном случае таковых не окажется, поскольку в нашей базе данных нет никаких прямых утверждений о том, кто около кого живет.) На следующем шаге мы пытаемся унифицировать образец-запрос с заключением каждого правила. Мы обнаруживаем, что образец унифицируется с заключением правила

```
(rule (живет-около ?person-1 ?person-2)
      (and (адрес ?person-1 (?town . ?rest-1))
           (адрес ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

и получается кадр, в котором переменная `?person-2` связана со значением `(Хакер Лиза П)`, а переменная `?x` связана с (должна иметь то же значение, что и) `?person-1`. Теперь по отношению к этому кадру мы вычисляем составной запрос, содержащийся в теле правила. Успешные сопоставления расширят кадр, сообщив значение переменной `?person-1`, а соответственно, и `?x`, которую мы можем использовать при конкретизации исходного образца-запроса.

В общем случае обработчик запросов при применении правила, когда он пытается распознать образец-запрос в кадре, который содержит связывания для некоторых переменных образца, использует следующий метод:

- Унифицировать запрос с заключением правила и получить (если унификация успешна) расширение исходного кадра.
- По отношению к расширенному кадру вычислить запрос, который является телом правила.

Обратите внимание, насколько это похоже на метод применения процедуры в интерпретаторе `eval/apply` для Лиспа:

- Связать параметры процедуры с ее аргументами и получить кадр, расширяющий исходное окружение процедуры.
- По отношению к расширенному окружению вычислить выражение, которое является телом процедуры.

Подобие двух вычислителей неудивительно. Точно так же, как в Лиспе средством абстракции являются определения процедур, в языке запросов средством абстракции являются определения правил. В каждом случае мы развертываем абстракцию, создавая соответствующие связывания и вычисляя тело правила либо процедуры по отношению к расширенной среде.

Простые запросы

В этом разделе мы уже рассматривали, как вычислять простые запросы при отсутствии правил. Теперь, разобравшись, как применяются правила, мы можем описать, как простые запросы вычисляются с помощью как правил, так и утверждений.

Получая запрос-образец и поток кадров, мы порождаем для каждого входного кадра два новых потока:

• поток расширенных кадров, получаемых сопоставлением образца со всеми утверждениями базы данных (при помощи сопоставителя), а также

• поток расширенных кадров, полученных применением всех возможных правил (при помощи унификатора)[73].

Соединение двух этих потоков порождает поток, который состоит изо всех способов, которыми данный образец можно удовлетворить в соответствии с исходным кадром. Эти потоки (по одному на каждый кадр входного потока) соединяются, и получается единый большой поток. Окончательный поток, таким образом, состоит изо всех способов, которыми какой-либо кадр входного потока может быть расширен так, чтобы получалось сопоставление с данным запросом.

Вычислитель запросов и управляющий цикл

Несмотря на сложность встроенных операций сопоставления, система организована подобно интерпретатору любого языка. Процедура, координирующая операции сопоставления, называется `qeval` и играет роль, аналогичную процедуре `eval` для Лиспа. `Qeval` принимает на входе запрос и поток кадров. Ее выходом служит поток кадров, соответствующих успешным сопоставлениям с запросом, которые расширяют какой-либо кадр во входном потоке, как показано на рис. 4.4. Подобно `eval`, `qeval` распознает различные типы выражений (запросов) и для каждого из них вызывает соответствующую процедуру. Имеется по процедуре для каждой особой формы (`and`, `or`, `not` и `lisp-value`) и еще одна для простых запросов.

Управляющий цикл, аналогичный процедуре `driver-loop` из других интерпретаторов этой главы, считывает запросы с терминала. Для каждого запроса он вызывает `qeval` с запросом и потоком, состоящим из одного пустого кадра. Получается поток всех возможных сопоставлений (всех возможных расширений пустого кадра). Для каждого кадра в выходном потоке управляющий цикл конкретизирует входной запрос с использованием значений переменных, имеющихся в кадре. Затем этот поток конкретизированных запросов печатается[74].

Кроме того, управляющий цикл распознает особую команду `assert!`, которая говорит, что на вход поступает не запрос, а новое утверждение или правило, которое следует добавить в базу данных. Например,

```
(assert! (должность (Битобор Бен) (компьютеры гуру)))

(assert! (rule (шишка ?person)
               (and (начальник ?middle-manager ?person)
                    (начальник ?x ?middle-manager))))
```

[73] Поскольку унификация является обобщением сопоставления, можно было бы упростить систему и порождать оба потока с помощью унификатора. Однако обработка простого случая с помощью обычного сопоставителя показывает, как сопоставление (а не полноразмерная унификация) может само по себе быть полезным.

[74] Мы используем потоки (а не списки) кадров потому, что рекурсивное применение правил может порождать бесконечное число значений, удовлетворяющих запросу. Здесь существенно задержанное вычисление, осуществляемое потоками: система будет печатать ответы один за другим по мере их порождения, независимо от того, получается ли конечное или бесконечное количество ответов.

4.4.3. Является ли логическое программирование математической логикой?

На первый взгляд может показаться, что средства комбинирования, используемые в языке запросов, совпадают с операторами математической логики — *и* (`and`), *или* (`or`) и отрицанием (`not`), а при применении правил языка запросов производится корректный логический вывод[75]. Однако такая идентификация языка запросов с математической логикой неверна, поскольку язык запросов обладает *структурой управления* (control structure), которая интерпретирует логические утверждения процедурным образом. Часто из этой структуры управления можно извлечь пользу. Например, чтобы найти начальников всех программистов, можно сформулировать запрос двумя логически эквивалентными способами:

```
(and (должность ?x (компьютеры программист))
     (начальник ?x ?y))
```

и

```
(and (начальник ?x ?y)
     (должность ?x (компьютеры программист)))
```

Если в компании намного больше начальников, чем программистов (обычный случай), то первую форму использовать выгоднее, чем вторую, поскольку для каждого промежуточного результата (кадра), порождаемого первым подзапросом `and`, требуется просмотреть базу данных.

Цель логического программирования состоит в том, чтобы дать программисту способ разбить вычислительную задачу на две отдельные подзадачи: «что» требуется посчитать и «как» это сделать. Этого добиваются, выделив подмножество утверждений математической логики — достаточно мощное, чтобы описать все, что захочется вычислить, но при этом достаточно слабое, чтобы иметь управляемую процедурную реализацию. Идея состоит в том, чтобы, с одной стороны, программа, выраженная на языке логического программирования, была эффективной, и компьютер мог бы ее исполнить. Управление («как» считать) определяется порядком вычислений языка. У нас должна быть возможность определять порядок выражений и порядок подвыражений в них так, чтобы вычисление проходило правильным и эффективным способом. В то же самое время мы должны быть способны рассматривать результат вычислений («что» считать) как простое следствие законов логики.

Наш язык запросов можно рассматривать в качестве именно такого процедурно интерпретируемого подмножества математической логики. Утверждение представляет простой факт (атомарную пропозицию). Правило представляет импликацию, говорящую, что заключение истинно в случаях, когда истинно тело правила. Правило обладает естественной процедурной интерпретацией: чтобы доказать заключение правила, требуется доказать его тело. Следовательно, правила описывают вычисления. Однако поскольку

[75] То, что конкретный метод логического вывода корректен — утверждение не тривиальное. Требуется доказать, что исходя из истинных посылок, можно придти только к истинным заключениям. В применении правил используется *modus ponens*, известный метод вывода, который говорит, что если истинны утверждения A и *из* A *следует* B, то можно заключить истинность утверждения B.

правила можно рассматривать и как формулы математической логики, мы можем оправдать любой «вывод», производимый логической программой, показав, что того же результата можно достичь, работая строго в рамках логики[76].

Бесконечные циклы

Вследствие процедурной интерпретации логических программ для решения некоторых задач можно построить безнадежно неэффективные программы. Частным случаем неэффективности является ситуация, когда программа при работе над выводом впадает в бесконечный цикл. Возьмем простой пример: предположим, что мы строим базу данных знаменитых супружеских пар, в том числе

```
(assert! (супруг Минни Микки))
```

Если теперь мы спросим

```
(супруг Микки ?who)
```

мы не получим ответа, поскольку система не знает, что если A является супругом B, то B является супругом A. Поэтому мы вводим правило

```
(assert! (rule (супруг ?x ?y)
               (супруг ?y ?x)))
```

и снова делаем запрос

```
(супруг Микки ?who)
```

К сожалению, это вводит систему в бесконечный цикл следующим образом:

- Система обнаруживает, что применимо правило `супруг`; а именно, заключение `(супруг ?x ?y)` успешно унифицируется с образцом-запросом `(супруг Микки ?who)` и получается кадр, в котором переменная `?x` связана со значением `Микки`, а переменная `?y` со значением `?who`. Интерпретатор должен, таким образом, выполнить в этом кадре запрос `(супруг ?y ?x)` — в сущности, выполнить запрос `(супруг ?who Микки)`.
- Один ответ находится как утверждение в базе данных: `(супруг Минни Микки)`.
- Применимо также и правило `супруг`, так что интерпретатор снова выполняет его тело, которое теперь равно `(супруг Микки ?who)`.

[76] Это утверждение нужно ограничить соглашением: говоря о «выводе», производимом логической программой, мы предполагаем, что вычисление имеет конец. К сожалению, даже это ограниченное утверждение оказывается ложным для нашей реализации языка запросов (а также для программ на Прологе и большинстве других современных математических языков) из-за использования `not` и `lisp-value`. Как будет описано ниже, примитив `not`, реализованный в языке запросов, не всегда имеет то же значение, что отрицание в математической логике, а использование `lisp-value` вызывает дополнительные сложности. Можно было бы реализовать язык, согласованный с математической логикой, если просто убрать из него `not` и `lisp-value` и согласиться писать программы с использованием исключительно простых запросов, `and` и `or`. Однако при этом оказалась бы ограничена выразительная сила языка. Одна из основных тем исследований в логическом программировании — поиск способов более тесного согласования с математической логикой без чрезмерной потери выразительной силы.

Теперь система оказалась в бесконечном цикле. В сущности, найдет ли система простой ответ `(супруг Минни Микки)` прежде, чем окажется в цикле, зависит от деталей реализации, связанных с порядком, в котором система проверяет записи базы данных. Это простой пример циклов, которые могут возникнуть. Наборы взаимосвязанных правил могут привести к циклам, которые значительно труднее предвидеть, а возникновение цикла может зависеть от порядка подвыражений в `and` (см. упражнение 4.64) или от низкоуровневых деталей, связанных с порядком обработки запросов в системе[77].

Проблемы с `not`

Еще одна особенность запросной системы связана с `not`. Рассмотрим следующие два запроса к базе данных из раздела 4.4.1:

```
(and (начальник ?x ?y)
     (not (должность ?x (компьютеры программист))))

(and (not (должность ?x (компьютеры программист)))
     (начальник ?x ?y))
```

Эти два запроса приводят к различным результатам. Первый запрос сначала находит все записи в базе данных, соответствующие образцу `(начальник ?x ?y)`, затем фильтрует полученные кадры, удаляя те, в которых значение `?x` удовлетворяет образцу `(должность ?x (компьютеры программист))`. Второй запрос сначала фильтрует входные кадры, пытаясь удалить те, которые удовлетворяют образцу `(должность ?x (компьютеры программист))`. Поскольку единственный входной кадр пуст, он проверяет базу данных и смотрит, есть ли там записи, соответствующие `(должность ?x (компьютеры программист))`. Поскольку, как правило, такие записи имеются, выражение `not` удаляет пустой кадр, и остается пустой поток кадров. Следовательно, весь составной запрос также возвращает пустой поток.

Сложность состоит в том, что наша реализация `not` предназначена только для того, чтобы служить фильтром для значений переменных. Если выражение `not` обрабатывается с кадром, в котором часть переменных остается несвязанными (как `?x` в нашем примере), система выдаст неверный результат. Подобные сложности возникают и с использованием `lisp-value` — предикат Лиспа не сможет работать, если часть из его аргументов несвязана. См. упражнение 4.77.

Есть еще один, значительно более серьезный аспект, в котором `not` языка запросов отличается от отрицания в математической логике. В логике мы считаем, что выражение «не P» означает, что P ложно. Однако в системе запросов «не P» означает, что P невозможно доказать на основе информации из базы данных. Например, имея базу данных

[77] Это проблема не собственно логики, а процедурной интерпретации логики, которую дает наш интерпретатор. В данном случае можно написать интерпретатор, который не попадет в цикл. Например, можно пронумеровать доказательства, выводимые из наших утверждений и правил, по ширине, а не по глубине. Однако в такой системе оказывается труднее использовать порядок правил в программах. Одна из попыток встроить в такую программу тонкое управление вычислениями описана в deKleer et al. 1977. Еще один метод, который не ведет к столь же сложным проблемам с управлением, состоит в добавлении специальных знаний, например, детекторов для каких-то типов циклов (см. упражнение 4.67). Однако общую схему надежного предотвращения бесконечных путей в рассуждениях построить невозможно. Представьте себе дьявольское правило вида «чтобы доказать истинность $P(x)$, докажите истинность $P(f(x))$» для какой-нибудь хитро выбранной функции f.

из раздела 4.4.1, система радостно выведет разнообразные отрицательные утверждения, например, что Бен Битобор не любитель бейсбола, что на улице нет дождя, и что $2 + 2$ не равно 4[78]. Иными словами, операция `not` в языках логического программирования отражает так называемую *гипотезу о замкнутости мира* (closed world assumption) и считает, что вся релевантная информация включена в базу данных[79].

Упражнение 4.64.
Хьюго Дум по ошибке уничтожил в базе данных правило `подчиняется` (раздел 4.4.1). Обнаружив это, он быстро набивает правило заново, только, к сожалению, по ходу дела вносит небольшое изменение:

```
(rule (подчиняется ?staff-person ?boss)
      (or (начальник ?staff-person ?boss)
          (and (подчиняется ?middle-manager ?boss)
               (начальник ?staff-person ?middle-manager))))
```

Сразу после того, как Хьюго ввел информацию в систему, Кон Фиден хочет посмотреть, кому подчиняется Бен Битобор. Он вводит запрос

```
(подчиняется (Битобор Бен) ?who)
```

После ответа система проваливается в бесконечный цикл. Объясните, почему.

Упражнение 4.65.
П.Э. Фект, ожидая собственного продвижения по иерархии, дает запрос, который находит всех шишек (используя правило `шишка` из раздела 4.4.1):

```
(шишка ?who)
```

К его удивлению, система отвечает

```
;;; Результаты запроса:
(шишка (Уорбак Оливер))
(шишка (Битобор Бен))
(шишка (Уорбак Оливер))
(шишка (Уорбак Оливер))
(шишка (Уорбак Оливер))
```

Почему система упоминает Оливера Уорбака четыре раза?

Упражнение 4.66.
Бен работал над обобщением системы запросов так, чтобы можно было собирать статистику о компании. Например, чтобы найти сумму зарплат всех программистов, можно было бы сказать

```
(sum ?amount
     (and (должность ?x (компьютеры программист))
          (зарплата ?x ?amount)))
```

В общем случае новая система Бена допускает запросы вида

[78] Рассмотрим запрос `(not (любитель-бейсбола (Битобор Бен)))`. Система обнаруживает, что записи `(любитель-бейсбола (Битобор Бен))` в базе нет, так что пустой кадр образцу не соответствует и не удаляется из исходного потока кадров. Таким образом, результатом запроса является пустой кадр, он используется для конкретизации запроса, и выходит `(not (любитель-бейсбола (Битобор Бен)))`.

[79] Обсуждение и защита такой интерпретации `not` содержится в статье Кларка (Clark 1978).

```
(accumulation-function ⟨переменная⟩
                       ⟨запрос-образец⟩)
```

где в виде accumulation-function могут выступать sum (сумма), average (среднее) или maximum (максимум). Бен думает, что реализовать это расширение будет проще простого. Он просто скормит образец-запрос функции qeval и получит поток кадров. Затем он пропустит поток через функцию-отображение, которая из каждого кадра извлечет значение указанной переменной, и получившийся поток значений отдаст функции-накопителю. Когда Бен заканчивает свою реализацию и собирается ее опробовать, мимо проходит Пабло, все еще смущенный результатом запроса из упражнения 4.65. Когда Пабло показывает Бену полученный им от системы ответ, Бен хватается за голову: «Моя простая схема накопления не будет работать!»

Что понял Бен? Опишите, как он мог бы исправить ситуацию.

Упражнение 4.67.

Придумайте, как в запросную систему можно вставить детектор циклов, который избегает простых зацикливаний, вроде показанных в тексте и в упражнении 4.64. Общая идея состоит в том, что система должна хранить в каком-то виде историю текущей цепи рассуждений и не начинать обработку запроса, если она уже над ним работает. Опишите, информация какого вида (образцы и кадры) включается в историю и как можно проводить проверку. (После того, как Вы изучите в деталях реализацию запросной системы из раздела 4.4.4, Вы можете захотеть изменить систему и включить в нее свой детектор циклов.)

Упражнение 4.68.

Определите правила, с помощью которых реализуется операция reverse из упражнения 2.18, возвращающая список, элементы которого те же, что и в исходном, но идут в обратном порядке. (Подсказка: используйте append-to-form.) Могут ли Ваши правила ответить и на запрос (reverse (1 2 3) ?x), и на (reverse ?x (1 2 3))?

Упражнение 4.69.

Начав с базы данных и правил, сформулированных Вами в упражнении 4.63, постройте правила для добавления приставок «пра» в отношение внук. Система должна уметь понять, что Ирад — правнук Адама, а Иавал и Иувал приходятся Адаму прапрапрапраправнуками. (Подсказка: представляйте, например, утверждение об Ираде как ((пра внук) Адам Ирад). Напишите правила, которые определяют, заканчивается ли список словом внук. С помощью этого определите правило, которое позволяет вывести отношение ((пра . ?rel) ?x ?y), где список ?rel оканчивается на внук.) Проверьте свои правила на запросах ((пра внук) ?g ?ggs) и (?relationship Адам Ирад).

4.4.4. Реализация запросной системы

В разделе 4.4.2 описывалось, как работает запросная система. Теперь мы представляем полную реализацию системы во всех деталях.

4.4.4.1. Управляющий цикл и конкретизация

Управляющий цикл запросной системы читает входные выражения. Если выражение является правилом или утверждением, которое требуется добавить в базу данных, то происходит добавление. В противном случае предполагается, что выражение является

запросом. Управляющий цикл передает запрос вычислителю `qeval` вместе с начальным потоком, состоящим из одного пустого кадра. Результатом вычисления является поток кадров, порожденных заполнением переменных запроса значениями, найденными в базе данных. С помощью этих кадров порождается новый поток, состоящий из копий исходного запроса, в которых переменные конкретизированы значениями из потока кадров. Этот последний поток печатается на терминале:

```
(define input-prompt ";;; Ввод запроса:")
(define output-prompt ";;; Результаты запроса:")

(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline)
           (display "Утверждение добавлено в базу данных.")
           (query-driver-loop))
          (else
           (newline)
           (display output-prompt)
           (display-stream
            (stream-map
             (lambda (frame)
               (instantiate q
                            frame
                            (lambda (v f)
                              (contract-question-mark v))))
             (qeval q (singleton-stream '()))))
           (query-driver-loop)))))
```

Здесь, как и в других интерпретаторах из этой главы, мы пользуемся абстрактным синтаксисом языка запросов. Реализация синтаксиса выражений, включая предикат `assertion-to-be-added?` и селектор `add-assertion-body`, дается в разделе 4.4.4.7. Процедура `add-rule-or-assertion!` определяется в разделе 4.4.4.5.

Прежде чем обрабатывать входное выражение, управляющий цикл преобразует его синтаксис в форму, которая делает обработку эффективнее. При этом меняется представление переменных образца. Когда запрос конкретизируется, то все переменные, которые остались несвязанными, преобразуются, прежде чем печататься, обратно во входное представление. Эти преобразования производятся процедурами `query-syntax-process` и `contract-questionmark` (раздел 4.4.4.7).

Чтобы конкретизировать выражение, мы его копируем, заменяя при этом все переменные выражения их значениями из данного кадра. Значения сами по себе конкретизируются, поскольку и они могут содержать переменные (например, если `?x` внутри `exp` связано в результате унификации со значением `?y`, а уже `?y` связано со значением 5). Действие, которое требуется предпринять, если переменную не удается конкретизировать, задается процедурным аргументом `instantiate`.

```
(define (instantiate exp frame unbound-var-handler)
  (define (copy exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding
                 (copy (binding-value binding))
                 (unbound-var-handler exp frame))))
          ((pair? exp)
           (cons (copy (car exp)) (copy (cdr exp))))
          (else exp)))
  (copy exp))
```

Процедуры, управляющие связываниями, определяются в разделе 4.4.4.8.

4.4.4.2. Вычислитель

Процедура `qeval`, вызываемая из `query-driver-loop`, является основным вычислителем запросной системы. Она принимает на входе запрос и поток кадров и возвращает поток расширенных кадров. Особые формы она распознает через диспетчеризацию, управляемую данными, при помощи `get` и `put`, в точности так же, как мы реализовывали обобщенные операции в главе 2. Все запросы, которые не распознаются как особая форма, считаются простыми запросами и обрабатываются процедурой `simple-query`.

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
        (qproc (contents query) frame-stream)
        (simple-query query frame-stream))))
```

Селекторы `type` и `contents`, определяемые в разделе 4.4.4.7, реализуют абстрактный синтаксис особых форм.

Простые запросы

Процедура `simple-query` обрабатывает простые запросы. В качестве аргументов она принимает простой запрос (образец) и поток кадров, а возвращает поток, порожденный путем расширения каждого кадра всеми результатами успешного сопоставления записей базы данных с запросом.

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed
      (find-assertions query-pattern frame)
      (delay (apply-rules query-pattern frame))))
   frame-stream))
```

Для каждого кадра из входного потока мы с помощью `find-assertions` (раздел 4.4.4.3) сопоставляем образец со всеми утверждениями из базы данных, получая при

этом поток расширенных кадров. Кроме того, с помощью `apply-rules` (раздел 4.4.4.4) мы применяем все подходящие правила и получаем при этом еще один поток расширенных кадров. Два этих потока сливаются (при помощи `stream-append-delayed` из раздела 4.4.4.6) и дают на выходе поток, перечисляющий все способы, которыми исходный запрос можно удовлетворить в соответствии с исходным кадром (см. упражнение 4.71). Потоки от отдельных входных кадров соединяются через `stream-flatmap` (раздел 4.4.4.6) в один большой поток, содержащий все способы, которыми можно расширить кадры из входного потока и получить сопоставление с исходным запросом.

Составные запросы

Запросы с операцией `and` обрабатываются так, как показано на рис. 4.5, процедурой `conjoin`. `Conjoin` принимает в качестве аргументов конъюнкты и поток кадров, а возвращает поток расширенных кадров. Сначала она обрабатывает поток кадров и получает поток всех их возможных расширений, удовлетворяющих первому запросу конъюнкции. Затем, используя этот новый поток кадров, она рекурсивно применяет `conjoin` к остальным конъюнктам.

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts)
                      frame-stream))))
```

Выражение

```
(put 'and 'qeval conjoin)
```

настраивает процедуру `qeval` так, чтобы она при обнаружении формы `and` вызывала `conjoin`.

Запросы `or` обрабатываются подобным же образом, как показано на рис. 4.6. Выходные потоки отдельных дизъюнктов `or` вычисляются раздельно и смешиваются при помощи процедуры `interleave-delayed` из раздела 4.4.4.6. (См. упражнения 4.71 и 4.72.)

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
       (qeval (first-disjunct disjuncts) frame-stream)
       (delay (disjoin (rest-disjuncts disjuncts)
                       frame-stream)))))

(put 'or 'qeval disjoin)
```

Предикаты и селекторы для синтаксиса конъюнктов и дизъюнктов даны в разделе 4.4.4.7.

Фильтры

Запросы `not` обрабатываются так, как описано в разделе 4.4.2. Мы пытаемся расширить каждый кадр входного потока так, чтобы удовлетворялся отрицаемый запрос, и включаем данный кадр в поток-результат только в том случае, если расширить его нельзя.

```
(define (negate operands frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (stream-null? (qeval (negated-query operands)
                              (singleton-stream frame)))
         (singleton-stream frame)
         the-empty-stream))
   frame-stream))

(put 'not 'qeval negate)
```

`Lisp-value` — фильтр, подобный `not`. Образец расширяется с помощью каждого кадра из входного потока, применяется указанный предикат, и кадры, для которых он возвращает ложное значение, исключаются из входного потока. Если остаются несвязанные переменные запроса, возникает ошибка.

```
(define (lisp-value call frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (execute
          (instantiate
           call
           frame
           (lambda (v f)
             (error "Неизвестная переменная -- LISP-VALUE" v))))
         (singleton-stream frame)
         the-empty-stream))
   frame-stream))

(put 'lisp-value 'qeval lisp-value)
```

Процедура `execute`, которая применяет предикат к аргументам, должна вызвать `eval` от предикатного выражения, чтобы получить применяемую процедуру. Однако она не должна вычислять аргументы, поскольку это сами аргументы и есть, а не выражения, вычисление которых (на Лиспе) даст нам аргументы. Обратите внимание, что `execute` реализована с помощью `eval` и `apply` из нижележащей Лисп-системы.

```
(define (execute exp)
  (apply (eval (predicate exp) user-initial-environment)
         (args exp)))
```

Особая форма `always-true` порождает запрос, который всегда удовлетворяется. Она игнорирует свое подвыражение (обычно пустое) и попросту пропускает через себя все кадры входного потока. `Always-true` используется в селекторе `rule-body`

(раздел 4.4.4.7) чтобы дать тела правилам, для которых тела не определены (то есть правилам, заключения которых всегда удовлетворяются).

```
(define (always-true ignore frame-stream) frame-stream)

(put 'always-true 'qeval always-true)
```

Селекторы, которые определяют синтаксис `not` и `lisp-value`, определены в разделе 4.4.4.7.

4.4.4.3. Поиск утверждений с помощью сопоставления с образцом

Процедура `find-assertions`, вызываемая из `simple-query` (раздел 4.4.4.2), принимает на входе образец и кадр. Она возвращает поток кадров, каждый из которых расширяет исходный кадр сопоставлением данного образца с записью базы данных. Она пользуется `fetch-assertions` (раздел 4.4.4.5), чтобы найти поток всех утверждений базы, которые следует проверять на сопоставление с данными образцом и кадром. Мы используем `fetch-assertions` потому, что часто можно с помощью простых тестов исключить множество записей в базе данных из числа кандидатов на успешное сопоставление. Система продолжала бы работать, если бы мы исключили `fetch-assertions` и попросту проверяли поток всех утверждений базы, но при этом вычисление было бы менее эффективным, поскольку пришлось бы делать намного больше вызовов сопоставителя.

```
(define (find-assertions pattern frame)
  (stream-flatmap (lambda (datum)
                    (check-an-assertion datum pattern frame))
                  (fetch-assertions pattern frame)))
```

Процедура `check-an-assertion` принимает в качестве аргументов образец, объект данных (утверждение) и кадр, и возвращает либо одноэлементный поток с расширенным кадром, либо, если сопоставление неудачно, `the-emptystream`.

```
(define (check-an-assertion assertion query-pat query-frame)
  (let ((match-result
         (pattern-match query-pat assertion query-frame)))
    (if (eq? match-result 'failed)
        the-empty-stream
        (singleton-stream match-result))))
```

Сопоставитель как таковой возвращает либо символ `failed`, либо расширение данного кадра. Основная идея сопоставителя состоит в том, чтобы сравнивать образец с данными, элемент за элементом, и собирать при этом связывания переменных образца. Если образец и объект данных совпадают, то сопоставление оказывается успешным, и мы возвращаем поток собранных связываний. В противном случае, если образец является переменной, мы расширяем имеющийся кадр, связывая переменную с данными, если это не противоречит уже имеющимся в кадре связываниям. Если и образец, и данные являются парами, мы (рекурсивно) сопоставляем `car` образца с `car` данных и получаем кадр; затем с этим кадром мы сопоставляем `cdr` образца с `cdr` данных. Если ни один

из этих случаев не применим, сопоставление терпит неудачу, и мы возвращаем символ `failed`.

```
(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat)
                        (cdr dat)
                        (pattern-match (car pat)
                                       (car dat)
                                       frame)))
        (else 'failed)))
```

Вот процедура, которая расширяет кадр, добавляя к нему новое связывание, если это не противоречит уже имеющимся в кадре связываниям:

```
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
        (pattern-match (binding-value binding) dat frame)
        (extend var dat frame))))
```

Если для переменной в кадре нет связывания, мы просто добавляем к нему новое связывание этой переменной с элементом данных. В противном случае мы вызываем сопоставитель в данном кадре от элемента данных и имеющегося значения переменной в кадре. Если хранимое значение содержит только константы, (а это всегда так будет, если оно само было создано процедурой `extend-if-consistent` во время сопоставления с образцом), то это сопоставление просто проверит, совпадает ли хранимое значение с новым. Если да, то кадр вернется неизменным; если нет, вернется символ неудачи. Однако если хранимое в кадре значение было создано при унификации (см. раздел 4.4.4.4), то оно может содержать переменные образца. Рекурсивное сопоставление хранимого образца с новым элементом данных добавит или проверит связывания переменных в этом образце. Предположим, к примеру, что у нас есть кадр, в котором переменная `?x` связана с выражением `(f ?y)`, а `?y` несвязана, и что теперь мы хотим расширить этот кадр, связав `?x` со значением `(f b)`. Мы ищем в кадре `?x` и видим, что она связана с `(f ?y)`. Теперь нам нужно сопоставить `(f ?y)` с предлагаемым новым значением `(f b)` в том же самом кадре. В конце концов это сопоставление расширяет кадр, добавив связывание `?y` с `b`. `?X` по-прежнему связано с `(f ?y)`. Мы никогда не изменяем хранимое связывание и никогда не храним более одного связывания для одной и той же переменной.

Процедуры, при помощи которых `extend-if-consistent` работает со связываниями, определены в разделе 4.4.4.8.

Образцы с точечными хвостами

Если в образце содержится точка, за которой следует переменная образца, то переменная сопоставляется с остатком списка (а не со следующим его элементом), как

и следовало ожидать от точечной записи, описанной в упражнении 2.20. Несмотря на то, что реализованный нами сопоставитель на занимается специально поиском точек, работает он в этом случае так, как ему следует. Это происходит потому, что лисповский примитив `read`, с помощью которого `query-driver-loop` считывает запрос и представляет его в виде списковой структуры, обрабатывает точки особым образом.

Когда `read` встречает точку, вместо того, чтобы сделать следующее выражение очередным элементом списка (`car` в ячейке `cons`, `cdr` которой будет остатком списка), он делает его `cdr`ом списковой структуры. Например, списковая структура, которую `read` порождает при чтении образца `(компьютеры ?type)` могла бы быть построена с помощью выражения `(cons 'компьютеры (cons '?type '()))`, а та, которая получается при чтении `(компьютеры . ?type)`, могла бы получиться при вычислении `(cons 'компьютеры '?type)`.

Таким образом, когда `pattern-match` рекурсивно сравнивает `car`ы и `cdr`ы списка данных и образца, содержащего точку, он в конце концов сопоставляет переменную после точки (она служит `cdr` образца) с подсписком списка данных, и связывает переменную с этим списком. Например, сопоставление образца `(компьютеры . ?type)` со списком `(компьютеры программист стажер)` сопоставит переменную `?type` с подсписком `(программист стажер)`.

4.4.4.4. Правила и унификация

Процедура `apply-rules` — это аналог `find-assertion` (раздел 4.4.4.3). Она принимает на входе образец и кадр, а порождает поток расширенных кадров, применяя правила из базы данных. `Stream-flatmap` отображает через `apply-rule` поток возможно применимых правил (отобранных процедурой `fetch-rules` из раздела 4.4.4.5) и склеивает получившиеся потоки кадров.

```
(define (apply-rules pattern frame)
  (stream-flatmap (lambda (rule)
                    (apply-a-rule rule pattern frame))
                  (fetch-rules pattern frame)))
```

Процедура `apply-a-rule` применяет правила способом, описанным в разделе 4.4.2. Сначала она дополняет кадр-аргумент, унифицируя в его рамках заключение правила с образцом. Если это удается, она выполняет в получившемся кадре тело правила.

Однако прежде всего программа переименовывает все переменные в правиле и дает им уникальные новые имена. Это делается потому, что мы не хотим, чтобы переменные из различных применений правил смешивались друг с другом. К примеру, если в двух правилах используется переменная `?x`, то каждое из них может добавить связывание этой переменной к кадру, в котором оно применяется. Однако эти два `?x` не имеют друг к другу никакого отношения, и мы не должны обманываться и считать, что два связывания этих переменных обязаны соответствовать друг другу. Вместо переименования переменных мы могли бы придумать более хитрую структуру окружений; однако выбранный здесь подход с переименованиями — самый простой, хотя и не самый эффективный. (См. упражнение 4.79.) Вот процедура `apply-a-rule`:

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
```

```
    (let ((unify-result
           (unify-match query-pattern
                        (conclusion clean-rule)
                        query-frame)))
      (if (eq? unify-result 'failed)
          the-empty-stream
          (qeval (rule-body clean-rule)
                 (singleton-stream unify-result))))))
```

Селекторы `rule-body` и `conclusion`, извлекающие части правил, описаны в разделе 4.4.4.7.

Чтобы породить уникальные имена переменных, мы связываем с каждым применением правила уникальный идентификатор (например, число) и цепляем его к исходным именам переменных. Например, если идентификатор применения правила равен 7, мы можем заменить все `?x` в правиле на `?x-7`, а все `?y` на `?y-7`. (Процедуры `make-new-variable` и `new-rule-application-id` содержатся среди синтаксических процедур в разделе 4.4.4.7.)

```
(define (rename-variables-in rule)
  (let ((rule-application-id (new-rule-application-id)))
    (define (tree-walk exp)
      (cond ((var? exp)
             (make-new-variable exp rule-application-id))
            ((pair? exp)
             (cons (tree-walk (car exp))
                   (tree-walk (cdr exp))))
            (else exp)))
    (tree-walk rule)))
```

Алгоритм унификации реализуется в виде процедуры, которая принимает на входе два образца и кадр, а возвращает либо расширенный кадр, либо символ `failed`. Унификатор в основном подобен сопоставителю, но только он симметричен — переменные разрешаются с обеих сторон сопоставления. Процедура `unify-match` подобна `pattern-match`, за исключением нового отрезка кода (отмеченного знаком «***»), где обрабатывается случай, когда объект на правой стороне сопоставления является переменной.

```
(define (unify-match p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((var? p2) (extend-if-possible p2 p1 frame))  ; ***
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
                      (cdr p2)
                      (unify-match (car p1)
                                   (car p2)
                                   frame)))
        (else 'failed)))
```

При унификации, как и при одностороннем сопоставлении с образцом, нам нужно принимать предлагаемое расширение кадра только в том случае, когда оно не противоречит имеющимся связываниям. Процедура `extend-if-possible`, используемая при унификации, подобна `extend-if-consistent` из сопоставителя, за исключением двух проверок, отмеченных в программе значком «***». В первом случае, если переменная, которую мы пытаемся сопоставить, не найдена, но значение, с которым мы ее сопоставляем, само является (другой) переменной, требуется проверить, нет ли у этой второй переменной значения, и если да, сопоставить его. Если же обе стороны сопоставления несвязаны, мы любую из них можем связать с другой.

Вторая проверка связана с попытками связать переменную с образцом, который ее саму содержит. Такая ситуация может возникнуть, когда в обоих образцах повторяются переменные. Рассмотрим, например, унификацию образцов `(?x ?x)` и `(?y` ⟨*выражение, содержащее* `?y`⟩`)` в кадре, где не связаны ни `?x`, ни `?y`. Сначала `?x` сопоставляется с `?y`, и возникает связывание переменной `?x` с `?y`. Затем та же переменная `?x` сопоставляется с данным выражением, которое включает `?y`. Поскольку `?x` уже связана со значением `?y`, это приводит к тому, что с выражением сопоставляется `?y`. Если мы считаем, что унификатор занят поиском набора значений для переменных, которые делают образцы одинаковыми, то значит, эти образцы содержат инструкции найти такое значение `?y`, чтобы `?y` был равен выражению, содержащему `?y`. Общего метода для решения таких задач не существует, так что мы такие связывания отвергаем; эти случаи распознаются предикатом `depends-on?`[80]. С другой стороны, нам не хочется отвергать попытки связать переменную саму с собой. Рассмотрим, например, унификацию `(?x ?x)` с `(?y ?y)`. Вторая попытка связать `?x` с `?y` вызывает сопоставление `?y` (старое значение `?x`) с `?y` (новым значением `?x`). Этот случай обрабатывается веткой `equal?` внутри `unify-match`.

---

[80] В общем случае унификация `?y` с выражением, содержащим `?y`, требует нахождения неподвижной точки уравнения `?y` = ⟨*выражение, содержащее* `?y`⟩. Иногда возможно синтаксическим образом создать выражение, которое кажется решением уравнения. Например, кажется, что `?y = (f y)` имеет неподвижную точку `(f (f (f ...)))`, которую мы можем получить, начав с выражения `(f ?y)` и систематически подставляя `(f ?y)` вместо `?y`. К сожалению, не у всякого такого уравнения имеется осмысленная неподвижная точка. Вопросы, возникающие здесь, подобны вопросам работы с бесконечными последовательностями в математике. Например, мы знаем, что решение уравнения $y = 1 + y/2$ равно 2. Если мы начнем с выражения $1 + y/2$ и будем подставлять $1 + y/2$ вместо y, то получим

$$2 = y = 1 + y/2 = 1 + (1 + y/2)/2 = 1 + 1/2 + y/4 = \cdots,$$

что ведет к

$$2 = 1 + 1/2 + 1/4 + 1/8 + \cdots.$$

Однако если мы попытаемся проделать те же преобразования, использовав тот факт, что решение уравнения $y = 1 + 2y$ равно -1, то получим

$$-1 = y = 1 + 2y = 1 = 2(1 + 2y) = 1 + 2 + 4y = \cdots,$$

что ведет к

$$-1 = 1 + 2 + 4 + 8 + \cdots.$$

Несмотря на то, что формальные преобразования, ведущие к этим двум уравнениям, одинаковы, первый результат является верным утверждением о бесконечных последовательностях, а второй нет. Подобным образом и при работе с унификациями работа с произвольными синтаксически правильными выражениями может привести к ошибкам.

```
(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
           (unify-match
            (binding-value binding) val frame))
          ((var? val)                      ; ***
           (let ((binding (binding-in-frame val frame)))
             (if binding
                 (unify-match
                  var (binding-value binding) frame)
                 (extend var val frame))))
          ((depends-on? val var frame)     ; ***
           'failed)
          (else (extend var val frame)))))
```

Процедура `depends-on?` — это предикат. Он проверяет, зависит ли выражение, которое предлагается сделать значением переменной образца, от этой переменной. Это нужно делать по отношению к текущему кадру, поскольку выражение может содержать вхождения переменной, уже обладающей значением, которое, в свою очередь, зависит от нашей переменной. По структуре `depends-on?` представляет собой простой рекурсивный обход дерева, во время которого мы по необходимости подставляем значения переменных.

```
(define (depends-on? exp var frame)
  (define (tree-walk e)
    (cond ((var? e)
           (if (equal? var e)
               true
               (let ((b (binding-in-frame e frame)))
                 (if b
                     (tree-walk (binding-value b))
                     false))))
          ((pair? e)
           (or (tree-walk (car e))
               (tree-walk (cdr e))))
          (else false)))
  (tree-walk exp))
```

4.4.4.5. Ведение базы данных

Одна из важных задач при разработке логических языков программирования — так организовать работу, чтобы при проверке каждого образца просматривалось как можно меньше ненужных записей из базы. В нашей системе, помимо того, что мы храним все утверждения в одном большом потоке, мы в отдельных потоках храним утверждения, `car`ы которых являются константными символами, в таблице, индексируемой по этим символам. Чтобы получить утверждения, которые могут сопоставляться с образцом, мы сначала смотрим, не является ли `car` образца константным символом. Если это так, то мы возвращаем (сопоставителю для проверки) все хранимые утверждения с тем же

car. Если car образца не является константным символом, мы возвращаем все хранимые утверждения. Более изысканные методы могли бы использовать еще информацию из кадра, либо пытаться оптимизировать и тот случай, когда car образца не является константным символом. Мы избегаем встраивания критериев для индексации (использование car, обработка только случая с константными символами) в программу: вместо этого мы вызываем предикаты и селекторы, реализующие эти критерии.

```
(define THE-ASSERTIONS the-empty-stream)

(define (fetch-assertions pattern frame)
  (if (use-index? pattern)
      (get-indexed-assertions pattern)
      (get-all-assertions)))

(define (get-all-assertions) THE-ASSERTIONS)

(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))
```

Процедура get-stream ищет поток в таблице и, если ничего там не находит, возвращает пустой поток.

```
(define (get-stream key1 key2)
  (let ((s (get key1 key2)))
    (if s s the-empty-stream)))
```

Правила хранятся подобным же образом, с использованием car заключения правила. Однако в заключениях правил могут стоять произвольные образцы, и таким образом, они отличаются от утверждений тем, что могут содержать переменные. Образец, в car которого стоит константный символ, может сопоставляться не только с правилами, у которых car заключения содержит тот же символ, но и с правилами, где в начале заключения стоит переменная. Таким образом, при поиске правил, которые могут сопоставляться с образцом, у которого в начале константный символ, мы возвращаем как правила с этим символом в car заключения, так и правила с переменной в начале заключения. Ради этого мы храним правила с переменными в начале заключения в отдельном потоке, который находится в таблице под индексом ?.

```
(define THE-RULES the-empty-stream)

(define (fetch-rules pattern frame)
  (if (use-index? pattern)
      (get-indexed-rules pattern)
      (get-all-rules)))

(define (get-all-rules) THE-RULES)

(define (get-indexed-rules pattern)
  (stream-append
   (get-stream (index-key-of pattern) 'rule-stream)
   (get-stream '? 'rule-stream)))
```

Процедура `add-rule-or-assertion!` вызывается из `query-driver-loop`, когда требуется добавить к базе данных правило или утверждение. Каждая запись сохраняется в индексе, если это требуется, а также в общем потоке правил либо утверждений базы данных.

```
(define (add-rule-or-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-assertion! assertion)))

(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS
          (cons-stream assertion old-assertions))
    'ok))

(define (add-rule! rule)
  (store-rule-in-index rule)
  (let ((old-rules THE-RULES))
    (set! THE-RULES (cons-stream rule old-rules))
    'ok))
```

Чтобы вставить в базу утверждение или правило, мы проверяем, можно ли его проиндексировать. Если да, то мы сохраняем его в соответствующем потоке.

```
(define (store-assertion-in-index assertion)
  (if (indexable? assertion)
      (let ((key (index-key-of assertion)))
        (let ((current-assertion-stream
               (get-stream key 'assertion-stream)))
          (put key
               'assertion-stream
               (cons-stream assertion
                            current-assertion-stream))))))

(define (store-rule-in-index rule)
  (let ((pattern (conclusion rule)))
    (if (indexable? pattern)
        (let ((key (index-key-of pattern)))
          (let ((current-rule-stream
                 (get-stream key 'rule-stream)))
            (put key
                 'rule-stream
                 (cons-stream rule
                              current-rule-stream)))))))
```

Следующие процедуры определяют, как используется индекс базы данных. Образец (утверждение или заключение правила) сохраняется в таблице, если он начинается с переменной или константного символа.

```
(define (indexable? pat)
  (or (constant-symbol? (car pat))
      (var? (car pat))))
```

Ключ, под которым образец сохраняется в таблице — это либо `?` (если он начинается с переменной), либо константный символ из его начала.

```
(define (index-key-of pat)
  (let ((key (car pat)))
    (if (var? key) '? key)))
```

Для поиска записей, которые могут соответствовать образцу, используется индекс в том случае, когда образец начинается с константного символа.

```
(define (use-index? pat)
  (constant-symbol? (car pat)))
```

Упражнение 4.70.
Какова цель выражений `let` в процедурах `add-assertion!` и `add-rule!`? Что неправильно в следующем варианте `add-assertion!`? Подсказка: вспомните определение бесконечного потока единиц из раздела 3.5.2: `(define ones (cons-stream 1 ones))`.

```
(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (set! THE-ASSERTIONS
        (cons-stream assertion THE-ASSERTIONS))
  'ok)
```

4.4.4.6. Операции над потоками

В запросной системе используется несколько операций над потоками, помимо представленных в главе 3.

Процедуры `stream-append-delayed` и `interleave-delayed` подобны `stream-append` и `interleave` (раздел 3.5.3), но только они принимают задержанный аргумент (как процедура `integral` из раздела 3.5.4). В некоторых случаях это откладывает зацикливание (см. упражнение 4.71).

```
(define (stream-append-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (stream-append-delayed (stream-cdr s1) delayed-s2))))

(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (interleave-delayed (force delayed-s2)
                           (delay (stream-cdr s1))))))
```

Процедура stream-flatmap, которая многократно используется в интерпретаторе, чтобы применить процедуру ко всем элементам потока кадров и соединить получающиеся потоки кадров, является потоковым аналогом процедуры flatmap для обычных списков, введенной в разделе 2.2.3. Однако, в отличие от обычного flatmap, потоки мы собираем с помощью чередующего процесса, а не просто сцепляем их (см. упражнения 4.72 и 4.73).

```
(define (stream-flatmap proc s)
  (flatten-stream (stream-map proc s)))

(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave-delayed
       (stream-car stream)
       (delay (flatten-stream (stream-cdr stream))))))
```

Кроме того, интерпретатор пользуется следующей простой процедурой для порождения потока, который состоит из одного элемента:

```
(define (singleton-stream x)
  (cons-stream x the-empty-stream))
```

4.4.4.7. Процедуры, определяющие синтаксис запросов

Процедуры type и contents, используемые в qeval (раздел 4.4.4.2), указывают, что особая форма определяется символом в ее car. Это те же процедуры, что type-tag и contents из раздела 2.4.2, с точностью до сообщения об ошибке.

```
(define (type exp)
  (if (pair? exp)
      (car exp)
      (error "Неизвестное выражение TYPE" exp)))

(define (contents exp)
  (if (pair? exp)
      (cdr exp)
      (error "Неизвестное выражение CONTENTS" exp)))
```

Следующие процедуры, вызываемые из query-driver-loop (раздел 4.4.4.1), указывают, что утверждения и правила добавляются в базу данных при помощи выражений вида (assert! ⟨*правило-или-утверждение*⟩):

```
(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))

(define (add-assertion-body exp)
  (car (contents exp)))
```

Вот синтаксические определения для особых форм and, or, not и lispvalue (раздел 4.4.4.2):

```
(define (empty-conjunction? exps) (null? exps))
(define (first-conjunct exps) (car exps))
(define (rest-conjuncts exps) (cdr exps))

(define (empty-disjunction? exps) (null? exps))
(define (first-disjunct exps) (car exps))
(define (rest-disjuncts exps) (cdr exps))

(define (negated-query exps) (car exps))

(define (predicate exps) (car exps))
(define (args exps) (cdr exps))
```

Следующие три процедуры определяют синтаксис правил:

```
(define (rule? statement)
  (tagged-list? statement 'rule))

(define (conclusion rule) (cadr rule))

(define (rule-body rule)
  (if (null? (cddr rule))
      '(always-true)
      (caddr rule)))
```

`Query-driver-loop` (раздел 4.4.4.1) вызывает `query-syntax-process`, чтобы преобразовать переменные образца в выражении, имеющие форму `?symbol`, к внутреннему формату `(? symbol)`. Это означает, что образец вроде `(должность ?x ?y)` на самом деле представляется внутри системы как `(должность (? x) (? y))`. Это повышает эффективность обработки запросов, потому что позволяет системе проверять, является ли выражение переменной, путем проверки `car` (не является ли `car` символом `?`), вместо того, чтобы извлекать из символа буквы. Преобразование синтаксиса осуществляется следующей процедурой[81].

```
(define (query-syntax-process exp)
  (map-over-symbols expand-question-mark exp))

(define (map-over-symbols proc exp)
  (cond ((pair? exp)
         (cons (map-over-symbols proc (car exp))
               (map-over-symbols proc (cdr exp))))
        ((symbol? exp) (proc exp))
```

[81] Большинство Лисп-систем позволяет пользователю изменять обыкновенную процедуру `read` и осуществлять такие преобразования путем определения *макросимволов ввода* (reader macro characters). Закавыченные выражения уже обрабатываются таким образом: процедура чтения автоматически переводит `'expression` в `(quote expression)`, прежде чем выражение видит интерпретатор. Можно было бы устроить преобразование `?expression` в `(? expression)` таким же образом; однако ради большей ясности мы здесь представили процедуру преобразования явно.

`Expand-question-mark` и `contract-question-mark` используют несколько процедур, имя которых содержит `string`. Это примитивы языка Scheme.

```
        (else exp)))

(define (expand-question-mark symbol)
  (let ((chars (symbol->string symbol)))
    (if (string=? (substring chars 0 1) "?")
        (list '?
              (string->symbol
               (substring chars 1 (string-length chars))))
        symbol)))
```

После того, как переменные таким образом преобразованы, переменные в образцах — это списки, начинающиеся с ?, а константные символы (которые требуется распознавать для индексирования базы данных, раздел 4.4.4.5) — это просто символы.

```
(define (var? exp)
  (tagged-list? exp '?))

(define (constant-symbol? exp) (symbol? exp))
```

Во время применения правил при помощи следующих процедур порождаются уникальные переменные (раздел 4.4.4.4). Уникальным идентификатором правила служит число, которое увеличивается при каждом применении правила:

```
(define rule-counter 0)

(define (new-rule-application-id)
  (set! rule-counter (+ 1 rule-counter))
  rule-counter)

(define (make-new-variable var rule-application-id)
  (cons '? (cons rule-application-id (cdr var))))
```

Когда `query-driver-loop` конкретизирует запрос для печати ответа, она преобразует все несвязанные переменные запроса обратно к печатной форме при помощи

```
(define (contract-question-mark variable)
  (string->symbol
   (string-append "?"
     (if (number? (cadr variable))
         (string-append (symbol->string (caddr variable))
                        "_"
                        (number->string (cadr variable)))
         (symbol->string (cadr variable))))))
```

4.4.4.8. Кадры и связывания

Кадры представляются как списки связываний, которые, в свою очередь, являются парами вида «переменная-значение»:

```
(define (make-binding variable value)
  (cons variable value))
```

```
(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (binding-in-frame variable frame)
  (assoc variable frame))

(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

Упражнение 4.71.
Хьюго Дум не понимает, почему процедуры `simple-query` и `disjoin` реализованы через явные операции `delay`, а не следующим образом:

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append (find-assertions query-pattern frame)
                    (apply-rules query-pattern frame)))
   frame-stream))

(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave
       (qeval (first-disjunct disjuncts) frame-stream)
       (disjoin (rest-disjuncts disjuncts) frame-stream))))
```

Можете ли Вы дать примеры запросов, с которыми эти простые определения приведут к нежелательному поведению?

Упражнение 4.72.
Почему `adjoin` и `stream-flatmap` чередуют потоки, а не просто их сцепляют? Приведите примеры, которые показывают, что чередование работает лучше. (Подсказка: зачем мы пользовались `interleave` в разделе 3.5.3?)

Упражнение 4.73.
Почему `flatten-stream` использует `delay` явно? Что было бы неправильно в таком определении:

```
(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave
       (stream-car stream)
       (flatten-stream (stream-cdr stream)))))
```

Упражнение 4.74.
Лиза П. Хакер предлагает использовать в `negate`, `lisp-value` и `find-assertions` упрощенную версию `stream-flatmap`. Она замечает, что в этих случаях процедура, которая отображает поток кадров, всегда порождает либо пустой поток, либо поток из одного элемента, и поэтому при слиянии этих потоков незачем использовать чередование.

а. Заполните пропущенные выражения в программе Лизы.

```
(define (simple-stream-flatmap proc s)
  (simple-flatten (stream-map proc s)))

(define (simple-flatten stream)
  (stream-map ⟨??⟩
              (stream-filter ⟨??⟩ stream)))
```

б. Если мы изменяем систему таким образом, меняется ли ее поведение?

Упражнение 4.75.
Реализуйте в языке запросов новую особую форму `unique`. Выражение `unique` должно быть успешно, если в базе данных ровно одна запись, удовлетворяющая указанному запросу. Например запрос

```
(unique (должность ?x (компьютеры гуру)))
```

должен печатать одноэлементный поток

```
(unique (должность (Битобор Бен) (компьютеры гуру)))
```

поскольку Бен — единственный компьютерный гуру, а

```
(unique (должность ?x (компьютеры программист)))
```

должно печатать пустой поток, поскольку программистов больше одного. Более того,

```
(and (должность ?x ?j) (unique (должность ?anyone ?j)))
```

должно перечислять все должности, на которых работает по одному человеку, а также самих этих людей.

Реализация `unique` состоит из двух частей. Первая заключается в том, чтобы написать процедуру, которая обрабатывает эту особую форму, а вторая в том, чтобы заставить `qeval` распознавать форму и вызывать ее процедуру. Вторая часть тривиальна, поскольку `qeval` написана в стиле программирования, управляемого данными. Если Ваша процедура называется `uniquely-asserted`, то нужно только написать

```
(put 'unique 'qeval uniquely-asserted)
```

и `qeval` будет передавать управление этой процедуре для всех запросов, у которых в `type` (`car`) стоит символ `unique`.

Собственно задача состоит в том, чтобы написать процедуру `uniquely-asserted`. В качестве входа она должна принимать `contents` (`cdr`) запроса `unique` и поток кадров. Для каждого кадра в потоке она должна с помощью `qeval` находить поток всех расширений, удовлетворяющих данному запросу. Потоки, в которых число элементов не равно одному, должны отбрасываться. Оставшиеся потоки нужно собирать в один большой поток. Он и становится результатом запроса `unique`. Эта процедура подобна реализации особой формы `not`.

Проверьте свою реализацию, сформировав запрос, который находит всех служащих, которые начальствуют ровно над одним человеком.

Упражнение 4.76.
Наша реализация `and` в виде последовательной комбинации запросов (рисунок 4.5) изящна, но неэффективна, поскольку при обработке второго запроса приходится просматривать базу данных для каждого кадра, порожденного первым запросом. Если в базе данных N записей, а типичный запрос порождает число записей, пропорциональное N (скажем, N/k), то проход базы данных для каждого кадра, порожденного первым запросом, потребует N^2/k вызовов сопоставителя. Другой подход может состоять в том, чтобы обрабатывать два подвыражения запроса `and` по отдельности а затем искать совместимые пары входных кадров. Если каждый запрос порождает N/k кадров, то нам придется проделать N^2/k^2 проверок на совместимость — в k раз меньше, чем число сопоставлений при нашем теперешнем методе.

Постройте реализацию `and` с использованием этой стратегии. Вам придется написать процедуру, которая принимает на входе два кадра, проверяет связывания в этих кадрах на совместимость и, если они совместимы, порождает кадр, в котором множества связываний слиты. Эта операция подобна унификации.

Упражнение 4.77.
В разделе 4.4.3 мы видели, что выражения `not` и `lisp-value` могут заставить язык запросов выдавать «неправильные» значения, если эти фильтрующие операции применяются к кадрам с несвязанными переменными. Придумайте способ избавиться от этого недостатка. Одна из возможностей состоит в том, чтобы проводить «задержанную» фильтрацию, цепляя к кадру «обещание» провести ее, которое выполняется только тогда, когда связано достаточно переменных, чтобы операция стала возможна. Можно ждать и проводить фильтрацию только тогда, когда выполнены все остальные операции. Однако из соображений эффективности хотелось бы фильтровать как можно раньше, чтобы уменьшить число порождаемых промежуточных кадров.

Упражнение 4.78.
Перестройте язык запросов в виде недетерминистской программы, реализуемой интерпретатором из раздела 4.3, а не в виде процесса обработки потоков. При таком подходе каждый запрос будет порождать один ответ (а не поток всех возможных ответов), а пользователь может ввести `try-again` и получить следующий ответ. Вы увидите, что существенная часть механизмов, которые мы построили в этом разделе, заменяется недетерминистским поиском и перебором с возвратами. Однако помимо этого, Вы обнаружите, что новый язык запросов отличается в тонких деталях поведения от реализованного нами в этом разделе. Можете ли Вы привести примеры, показывающие эти отличия?

Упражнение 4.79.
Когда мы реализовывали в разделе 4.1 интерпретатор, мы видели, как можно избежать конфликтов между именами параметров процедур при помощи локальных окружений. Например, при вычислении

```
(define (square x)
  (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
```

не возникает смешения между `x` из `square` и `x` из `sum-of-squares`, поскольку тело каждой процедуры мы вычисляем в окружении, которое специально построено для связывания локальных переменных. В запросной системе мы избегаем конфликтов имен при применении правил с помощью другой стратегии. Каждый раз при применении правила мы переименовываем переменные и даем им новые имена, которые обязаны быть уникальными. Аналогичная стратегия в интерпретаторе Лиспа заключалась бы в том, чтобы отменить внутренние окружения и просто переименовывать переменные в теле процедуры каждый раз, как мы ее вызываем.

Реализуйте для языка запросов метод применения правил, который использует не переименования, а окружения. Рассмотрите, можно ли использовать Вашу систему окружений для построения в языке запросов конструкций для работы с большими системами, например аналога блочной структуры процедур для правил. Можно ли связать это с проблемой ведения рассуждений в контексте (например: «Если бы я предположил, что истинно P, то я смог бы доказать A и B») в качестве метода решения задач? (Это упражнение не имеет однозначного решения. Хороший ответ, скорее всего, мог бы служить темой диссертации.)

Глава 5

Вычисления на регистровых машинах

> Моя цель — показать, что небесная машина не некое божественное живое существо, а скорее часовой механизм (а тот, кто верит, что у часов есть душа, приписывает славу творца творению), поскольку почти все из ее многочисленных движений вызываются простейшей материальной силой, так же, как все движения часов вызываются весом гири.
>
> Иоганн Кеплер
> (письмо к Герварту фон Гогенбургу, 1605)

Эта книга начинается с изучения процессов и с описания процессов в терминах процедур, написанных на Лиспе. Чтобы объяснить значение этих процедур, мы последовательно использовали несколько моделей вычисления: подстановочную модель из главы 1, модель с окружениями из главы 3 и метациклический интерпретатор из главы 4. Изучая последний, мы по большей части сняли покров тайны с деталей интерпретации лиспоподобных языков. Однако даже метациклический интерпретатор оставляет многие вопросы без ответа, поскольку он не проясняет механизмы управления Лисп-системы. Например, интерпретатор не показывает, как при вычислении подвыражения удается вернуть значение выражению, это значение использующему, или почему одни рекурсивные процедуры порождают итеративные процессы (то есть занимают неизменный объем памяти), в то время как другие процедуры порождают рекурсивные процессы. Эти вопросы остаются без ответа потому, что метациклический интерпретатор сам по себе является программой на Лиспе, а следовательно, наследует управляющую структуру нижележащей Лисп-системы. Чтобы предоставить более полное описание управляющей структуры вычислителя Лиспа, нам нужно работать на более элементарном уровне, чем сам Лисп.

В этой главе мы будем описывать процессы в терминах пошаговых операций традиционного компьютера. Такой компьютер, или *регистровая машина* (register machine), последовательно выполняет *команды* (instructions), которые работают с ограниченным числом элементов памяти, называемых *регистрами* (registers). Типичная команда регистровой машины применяет элементарную операцию к содержимому нескольких регистров и записывает результат еще в один регистр. Наши описания процессов, выполняемых регистровыми машинами, будут очень похожи на «машинный язык» обыкновенных компьютеров. Однако вместо того, чтобы сосредоточиться на машинном языке какого-то конкретного компьютера, мы рассмотрим несколько процедур на Лиспе и спроектируем специальную регистровую машину для выполнения каждой из этих процедур. Таким образом, мы будем решать задачу с точки зрения архитектора аппаратуры, а не с точки

зрения программиста на машинном языке компьютера. При проектировании регистровых машин мы разработаем механизмы для реализации важных программистских конструкций, таких, как рекурсия. Кроме того, мы представим язык для описания регистровых машин. В разделе 5.2 мы реализуем программу на Лиспе, которая с помощью этих описаний имитирует проектируемые нами машины.

Большинство элементарных операций наших регистровых машин очень просты. Например, такая операция может складывать числа, взятые из двух регистров, и сохранять результат в третьем. Несложно описать устройство, способное выполнять такие операции. Однако для работы со списковыми структурами мы будем использовать также операции `car`, `cdr` и `cons`, а они требуют сложного механизма выделения памяти. В разделе 5.3 мы изучаем их реализацию на основе более простых операций.

В разделе 5.4, накопив опыт выражения простых процессов в виде регистровых машин, мы спроектируем машину, которая реализует алгоритм, описываемый метациклическим интерпретатором из раздела 4.1. Таким образом, окажется заполненным пробел в нашем понимании того, как интерпретируются выражения языка Scheme, поскольку будет представлена явная модель механизмов управления вычислителя. В разделе 5.5 мы рассмотрим простой компилятор, переводящий программы на Scheme в последовательности команд, которые можно впрямую выполнить с помощью регистров и операций регистровой машины-вычислителя.

5.1. Проектирование регистровых машин

Чтобы спроектировать регистровую машину, требуется описать ее *пути данных* (data paths), то есть регистры и операции, а также *контроллер* (controller), который управляет последовательностью этих операций. Чтобы продемонстрировать строение простой регистровой машины, рассмотрим алгоритм Евклида для вычисления наибольшего общего делителя (НОД) двух натуральных чисел. Как мы видели в разделе 1.2.5, алгоритм Евклида можно реализовать в виде итеративного процесса, который описывается следующей процедурой:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Машина, реализующая алгоритм, должна отслеживать два числовых значения, a и b. Поэтому предположим, что эти числа хранятся в двух регистрах с такими же именами. Основные требуемые операции — это проверка, не является ли содержимое регистра `b` нулем, и вычисление остатка от деления содержимого регистра `a` на содержимое регистра `b`. Операция вычисления остатка — сложный процесс, однако пока что предположим, что у нас есть элементарное устройство, вычисляющее остатки. В каждом цикле алгоритма вычисления НОД содержимое регистра `a` требуется заменить содержимым регистра `b`, а содержимое регистра `b` следует заменить на остаток от деления старого содержимого `a` на старое содержимое `b`. Было бы удобно, если бы можно было производить эти замены одновременно, однако для нашей модели регистровых машин мы предположим, что на каждом шаге можно присвоить новое значение только одному регистру. Чтобы

произвести замены, наша машина использует третий «временный» регистр, который мы назовем `t`. (Сначала мы помещаем остаток в `t`, затем помещаем содержимое `b` в `a`, и наконец переносим остаток, хранимый в `t`, в `b`.)

Можно изобразить регистры и операции, требуемые нашей машине, при помощи диаграммы путей данных, показанной на рисунке 5.1. Регистры (`a`, `b` и `t`) на этой диаграмме изображаются в виде прямоугольников. Каждый способ присвоить регистру значение обозначается стрелкой, указывающей из источника данных на регистр, со значком X позади головки. Можно считать этот X кнопкой, которая при нажатии позволяет значению из источника «перетечь» в указанный регистр. Метка рядом — это имя для кнопки. Имена эти произвольны, и их можно подбирать с мнемоническим значением (например, `a<-b` обозначает нажатие кнопки, которая присваивает содержимое регистра `b` регистру `a`). Источником данных для регистра может служить другой регистр (как в случае присваивания `a<-b`), результат операции (как в случае присваивания `t<-r`) или константа (встроенное значение, которое нельзя изменять и которое представляется на диаграмме путей данных в виде треугольника со значением внутри).

Операция, которая вычисляет значение на основе констант и содержимого регистров, представляется на диаграмме путей данных в виде трапеции, содержащей имя операции. Например, фигура, обозначенная на рисунке 5.1 как `rem`, представляет операцию, вычисляющую остаток от деления содержимых регистров `a` и `b`, к которым она подсоединена. Стрелки (без кнопок) указывают из входных регистров и констант на фигуру, а другие стрелки связывают результат операции с регистрами. Сравнение изображается в виде круга, содержащего имя теста. К примеру, в нашей машине НОД имеется операция, которая проверяет, не равно ли содержимое регистра `b` нулю. У теста тоже есть входные стрелки, ведущие из входных регистров и констант, но у него нет исходящих стрелок; его значение используется контроллером, а не путями данных. В целом, диаграмма путей данных показывает регистры и операции, которые нужны машине, и как они должны быть связаны. Если мы рассматриваем стрелки как провода, а кнопки X как переключатели, то диаграмма путей данных оказывается очень похожа на схему машины, которую можно было бы построить из электронных деталей.

Для того, чтобы пути данных вычисляли НОД, нужно нажимать кнопки в правильной последовательности. Мы будем описывать эту последовательность с помощью диаграммы контроллера, показанной на рисунке 5.2. Элементы диаграммы контроллера показывают, как следует работать с компонентами путей данных. Прямоугольные блоки в такой диаграмме указывают, на какие кнопки следует нажимать, а стрелки описывают последовательный переход от одного шага к другому. Ромб на диаграмме обозначает выбор. Произойдет переход по одной из двух исходящих стрелок, в зависимости от значения того теста в потоке данных, имя которого указано в ромбе. Можно интерпретировать контроллер с помощью физической аналогии: представьте себе, что диаграмма — это лабиринт, в котором катается шарик. Когда шарик закатывается в прямоугольник, он нажимает на кнопку, имя которой в прямоугольнике написано. Когда шарик закатывается в узел выбора (например, тест `b` $= 0$), он покидает этот узел по стрелке, которую определяет результат указанного теста. Взятые вместе, пути данных и контроллер полностью определяют машину для вычисления НОД. Мы запускаем контроллер (катящийся шарик) в месте, обозначенном `start`, поместив предварительно числа в регистры `a` и `b`. Когда контроллер достигает точки, помеченной `done`, в регистре `a` оказывается значение НОД.

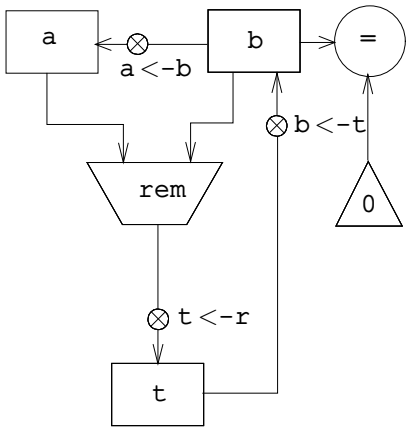


Рис. 5.1. Пути данных в машине НОД.

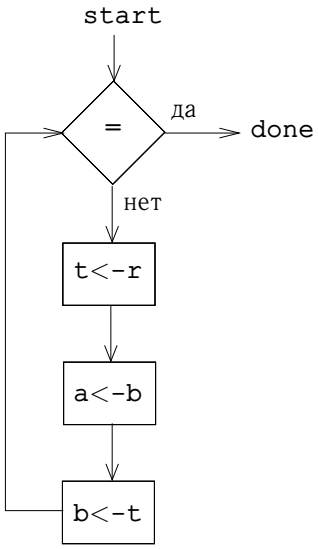


Рис. 5.2. Контроллер машины НОД.

Упражнение 5.1.
Спроектируйте регистровую машину для вычисления факториалов с помощью итеративного алгоритма, задаваемого следующей процедурой. Нарисуйте для этой машины диаграммы путей данных и контроллера.

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

5.1.1. Язык для описания регистровых машин

Диаграммы путей данных и контроллера адекватно представляют простые машины вроде машины НОД, но для описания сложных машин вроде интерпретатора Лиспа они непригодны. Чтобы можно было работать со сложными машинами, мы создадим язык, который представляет в текстовом виде всю информацию, содержащуюся в диаграммах путей данных и контроллера. Начнем с нотации, которая впрямую отражает диаграммы.

Мы определяем пути данных (`data-paths`) машины, описывая регистры (`registers`) и операции (`operations`). Чтобы описать регистр, мы даем ему имя (`name`) и указываем кнопки (`buttons`), которые присваивают ему значение. Каждой из этих кнопок мы даем имя (`name`) и указываем источник (`source`) для данных, которые попадают в регистр, управляемый кнопкой. Источником может служить регистр (`register`), константа (`constant`) или операция (`operation`). Для описания операции нам нужно дать ей имя и указать входы (`inputs`) — регистры или константы.

Контроллер машины мы определяем как последовательность *команд* (instructions) с *метками* (labels), которые определяют *точки входа* (entry points). Есть следующие виды команд:

- Имя кнопки на пути данных, которую следует нажать и присвоить регистру значение. (Это соответствует прямоугольнику на диаграмме контроллера.)
- Команда `test`, которая выполняет указанный тест.
- Условный переход (команда `branch`) в место, определяемое меткой контроллера, на основании предыдущего теста. (`Test` и `branch` вместе соответствуют ромбу на диаграмме контроллера.) Если тест дает результат «ложь», контроллер должен выполнять следующую команду в последовательности. В противном случае он должен выполнять команду, которая следует за меткой.
- Безусловный переход (команда `goto`), указывающий метку, с которой следует продолжить выполнение.

Машина начинает работу с начала последовательности оманд контроллера и заканчивает, когда выполнение достигает конца последовательности. Кроме тех случаев, когда переход изменяет поток управления, команды выполняются по порядку, так, как они перечислены.

На рисунке 5.3 изображена описанная на нашем языке машина НОД. Этот пример служит не более чем намеком на степень общности таких описаний, поскольку машина

```
(data-paths
 (registers
  ((name a)
   (buttons ((name a<-b) (source (register b)))))
  ((name b)
   (buttons ((name b<-t) (source (register t)))))
  ((name t)
   (buttons ((name t<-r) (source (operation rem))))))

 (operations
  ((name rem)
   (inputs (register a) (register b)))
  ((name =)
   (inputs (register b) (constant 0))))

 (controller
  test-b                        ; метка
    (test =)                    ; тест
    (branch (label gcd-done))   ; условный переход
    (t<-r)                      ; нажатие кнопки
    (a<-b)                      ; нажатие кнопки
    (b<-t)                      ; нажатие кнопки
    (goto (label test-b))       ; безусловный переход
  gcd-done))                    ; метка
```

Рис. 5.3. Описание машины НОД.

НОД — очень простой случай: у каждого регистра всего по одной кнопке, и каждая кнопка используется в контроллере только один раз.

К сожалению, такое описание неудобно читать. Чтобы понимать команды контроллера, нам все время приходится смотреть на определения имен кнопок и операций, а чтобы понять, что делают операции, приходится обращаться к определениям имен операций. Поэтому мы изменим свой способ записи и сольем информацию из описания контроллера и описания путей данных, так, чтобы видеть их одновременно.

В этой новой форме записи мы заменим произвольные имена кнопок и операций на описание их поведения. То есть, вместо того, чтобы говорить (в контроллере) «нажать кнопку `t<-r`» и отдельно (в путях данных) «кнопка `t<-r` присваивает регистру `t` значение операции `rem`», а также «входы операции `rem` — это содержимое регистров `a` и `b`», мы будем говорить (в контроллере) «нажать кнопку, которая присваивает регистру `t` результат операции `rem` от содержимого регистров `a` и `b`». Подобным образом, вместо «выполнить тест =» (в контроллере) и отдельно (в путях данных) «тест = применяется к содержимому регистра `b` и константе 0», будем говорить «выполнить тест = над содержимым регистра `b` и константой 0». Описание путей данных мы будем опускать, оставляя только последовательность команд контроллера. Таким образом, машину НОД можно описать так:

```
(controller
 test-b
   (test (op =) (reg b) (const 0))
   (branch (label gcd-done))
   (assign t (op rem) (reg a) (reg b))
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
 gcd-done)
```

Запись в такой форме проще читать, чем описания на разновидности языка, показанной на рисунке 5.3, но есть у нее и недостатки:

- Для больших машин такие описания длиннее, поскольку полные определения элементов путей данных повторяются каждый раз, как эти элементы упоминаются в последовательности команд контроллера. (В примере с НОД этой проблемы не возникает, поскольку каждая операция и каждая кнопка используются только по разу.) Более того, повторение описаний путей данных скрывает структуру этих путей в машине; для больших машин становится сложно определить, сколько в них регистров, операций и кнопок, и как они связаны.
- Поскольку команды контроллера в определении машины похожи на выражения Лиспа, легко забыть, что это не произвольные Лисп-выражения. Можно описывать только разрешенные операции машины. Например, операции впрямую могут работать только с константами и содержимым регистров, а не с результатами других операций.

Несмотря на указанные недостатки, мы будем использовать такой язык описания регистровых машин на всем протяжении этой главы, поскольку нас в большей мере будет занимать понимание работы контроллеров, чем понимание элементов и связей в путях данных. Следует, однако, помнить, что проектирование путей данных — ключевой элемент в разработке настоящих машин.

Упражнение 5.2.
С помощью языка регистровых машин опишите итеративную факториал-машину из упражнения 5.1.

Действия

Давайте теперь изменим машину НОД так, чтобы можно было вводить числа, НОД которых мы хотим получить, и видеть результаты, напечатанные на терминале. Мы не будем обсуждать, как построить машину для считывания и печати, а предположим (как и с процедурами `read` и `display` в Scheme), что эти действия доступны как элементарные операции[1].

`Read` подобна операциям, которые мы использовали ранее, поскольку она порождает значение, и его можно сохранить в регистре. Однако `read` не принимает входа ни из каких регистров; ее значение зависит от событий, происходящих за пределами тех компонентов машины, проектированием которых мы заняты. Мы позволим операциям нашей

[1] Такое предположение покрывает большую и сложную область. Обычно значительная часть реализации Лисп-систем посвящена работе ввода и вывода.

машины вести себя таким образом, и, следовательно, будем рисовать `read` и изображать ее в языке описания так же, как любую другую операцию, вычисляющую значение.

`Print`, с другой стороны, фундаментальным образом отличается от тех операций, которыми мы до сих пор пользовались: она не порождает результата, который можно было бы поместить в регистр. Хотя она и производит эффект, этот эффект не касается тех частей машины, которые мы проектируем. Этот тип операций мы будем называть *действиями* (actions). На диаграмме путей данных мы будем представлять действие так же, как и операции, вычисляющие значение — как трапецию с именем действия. В этот элемент входят стрелки из входов (регистров или констант). Кроме того, мы связываем с действием кнопку. Нажатие кнопки заставляет действие совершиться. Чтобы скомандовать контроллеру нажать кнопку действия, мы вводим новый тип команды `perform`. Таким образом, действие по распечатке содержимого регистра `a` представляется в последовательности контроллера командой

```
(perform (op print) (reg a))
```

На рисунке 5.4 показаны пути данных и контроллер для новой машины НОД. Вместо того, чтобы останавливать машину после печати ответа, мы приказываем ей начать сначала, так что она в цикле считывает пару чисел, вычисляет их НОД и печатает результат. Такая структура подобна управляющим циклам, которые мы использовали в интерпретаторах из главы 4.

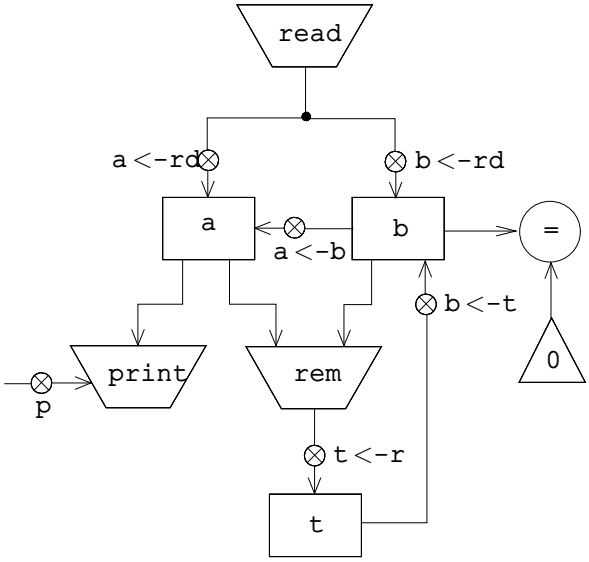
5.1.2. Абстракция в проектировании машин

Часто в определении машины мы будем использовать «элементарные» операции, которые на самом деле весьма сложны. Например, в разделах 5.4 и 5.5 мы будем рассматривать операции с окружениями Scheme как элементарные. Такая абстракция полезна, поскольку она позволяет нам игнорировать детали частей машины, так что мы можем сосредоточиться на других сторонах общего плана. Однако, хотя мы и скрываем существенную часть сложности, это не означает, что проект машины нереалистичен. Сложные «примитивы» всегда можно заменить более простыми операциями.

Рассмотрим машину НОД. В ней содержится команда, которая вычисляет остаток от деления содержимого регистров `a` и `b` и сохраняет результат в регистре `t`. Если мы хотим построить машину НОД без использования элементарной операции взятия остатка, нам нужно указать, как вычислять остатки с помощью более простых операций, например, вычитания. Действительно, можно написать на Scheme процедуру нахождения остатка таким образом:

```
(define (remainder n d)
  (if (< n d)
      n
      (remainder (- n d) d)))
```

Значит, мы можем заменить операцию взятия остатка в машине НОД операцией вычитания и тестом-сравнением. На рисунке 5.5 показаны пути данных и контроллер уточненной машины.




```
(controller
 gcd-loop
   (assign a (op read))
   (assign b (op read))
 test-b
   (test (op =) (reg b) (const 0)
   (branch (label gcd-done))
   (assign t (op rem) (reg a) (reg b))
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
gcd-done
  (perform (op print) (reg a))
  (goto (label gcd-loop)))
```

Рис. 5.4. Машина НОД, которая считывает входные числа и печатает результат.

Команда

```
(assign t (op rem) (reg a) (reg b))
```

в определении контроллера НОД заменяется на последовательность команд, содержащую цикл, как показано на рисунке 5.6.

Упражнение 5.3.
Спроектируйте машину для вычисления квадратных корней методом Ньютона, как описано в разделе 1.1.7:

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Для начала предположите, что операции `good-enough?` и `improve` имеются как примитивы. Затем покажите, как развернуть их с помощью арифметических операций. Опишите каждую из версий машины `sqrt`, нарисовав диаграмму путей данных, и написав определение контроллера на языке регистровых машин.

5.1.3. Подпрограммы

При проектировании машины для некоторого вычисления мы часто предпочтем устроить так, чтобы компоненты ее разделялись различными частями вычисления, а не дублировались. Рассмотрим машину, которая включает в себя два вычисления НОД — одно находит НОД содержимого регистров `a` и `b`, а другое НОД содержимого регистров `c` и `d`. Для начала можно предположить, что имеется элементарная операция `gcd`, а затем развернуть два экземпляра `gcd` в терминах более простых операций. На рисунке 5.7 показаны только части получившихся путей данных, относящиеся к НОД. Связи с остальными частями машины опущены. Кроме того, на рисунке показаны соответствующие сегменты последовательности команд контроллера машины.

В этой машине два блока вычисления остатка и два блока проверки на равенство. Если повторяющиеся компоненты сложны, как, например, блок вычисления остатка, такое построение машины окажется неэкономным. Можно избежать дублирования компонент путей данных, если использовать для обоих вычислений НОД одни и те же компоненты, при условии, что такое решение не повлияет на остальные вычисления большой машины. Если к тому времени, как контроллер добирается до `gcd-2`, значения в регистрах `a` и `b` не нужны (или если их можно временно сохранить в каких-то еще регистрах), то мы можем изменить машину так, чтобы она использовала регистры `a` и `b`, а не `c` и `d`, при вычислении второго НОД, так же как и при вычислении первого. Так у нас получится последовательность команд контроллера, показанная на рисунке 5.8.

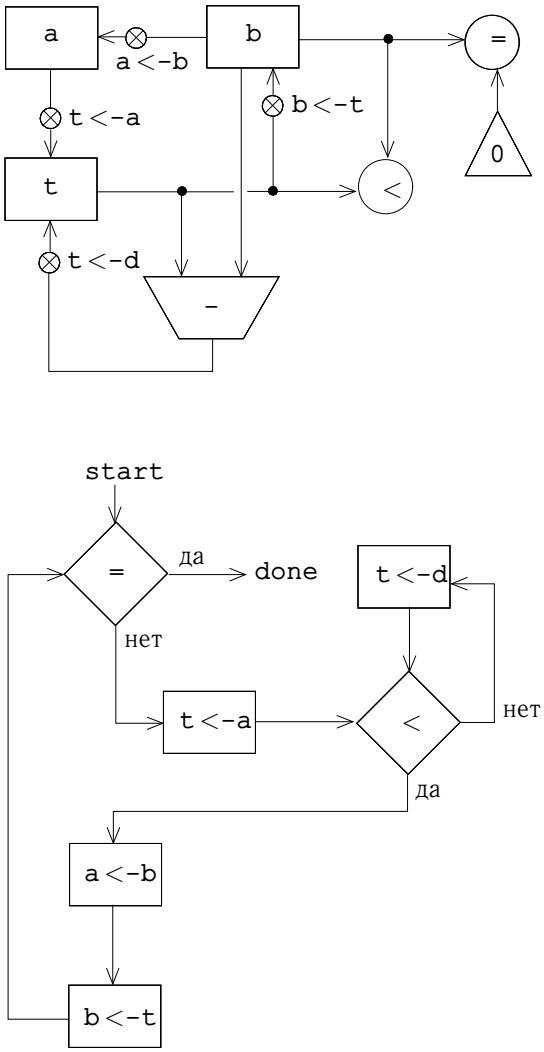


Рис. 5.5. Пути данных и контроллер уточненной машины НОД.

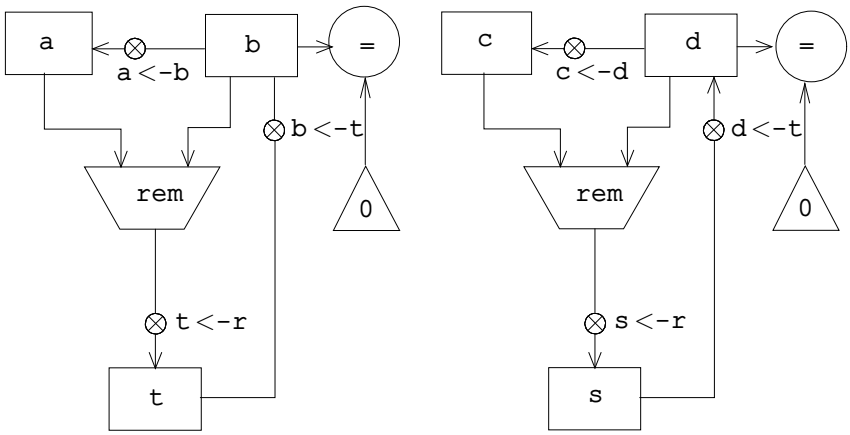
```
(controller
 test-b
 (test (op =) (reg b) (const 0))
 (branch (label gcd-done))
 (assign t (reg a))
rem-loop
 (test (op <) (reg t) (reg b))
 (branch (label rem-done))
 (assign t (op -) (reg t) (reg b))
 (goto (label rem-loop))
rem-done
 (assign a (reg b))
 (assign b (reg t))
(goto (label test-b))
gcd-done)
```

Рис. 5.6. Последовательность команд контроллера машины НОД с рисунка 5.5.

Мы удалили одинаковые компоненты путей данных (так что они снова стали такими, как на рисунке 5.1), но теперь в контроллере содержатся две последовательности команд вычисления НОД, которые различаются только метками. Было бы лучше заменить эти две последовательности переходами к единой последовательности — *подпрограмме* (subroutine), — в конце которой мы возвращаемся обратно к нужному месту в основной последовательности команд. Этого можно добиться так: прежде, чем перейти к `gcd`, мы помещаем определенное значение (0 или 1) в особый регистр, `continue`. В конце подпрограммы `gcd` мы переходим либо к `after-gcd-1`, либо к `after-gcd-2`, в зависимости от значения из регистра `continue`. На рисунке 5.9 показан соответствующий сегмент получающейся последовательности команд контроллера, который содержит только одну копию команд `gcd`.

Для маленьких задач это разумный подход, однако если бы в последовательности команд контроллера имелось много вызовов вычисления НОД, он стал бы неудобен. Чтобы решить, где продолжать вычисление после подпрограммы `gcd`, нам пришлось бы иметь в контроллере тесты и переходы для всех мест, где используется `gcd`. Более мощный метод реализации подпрограмм состоит в том, чтобы запоминать в регистре `continue` метку точки входа в последовательности контроллера, с которой выполнение должно продолжиться, когда подпрограмма закончится. Реализация этой стратегии требует нового вида связи между путями данных и контроллером регистровой машины: должно быть возможно присвоить регистру метку в последовательности команд контроллера таким образом, чтобы это значение можно было из регистра извлечь и с его помощью продолжить выполнение с указанной точки входа.

Чтобы отразить эту возможность, мы расширим команду `assign` языка регистровых машин и позволим присваивать регистру в качестве значения метку из последовательности команд контроллера (как особого рода константу). Кроме того, мы расширим команду `goto` и позволим вычислению продолжаться с точки входа, которая описывается содержимым регистра, а не только с точки входа, описываемой меткой-константой. С




```
gcd-1
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-1))
after gcd-2
  .
  .
  .
gcd-2
(test (op =) (reg d) (const 0))
(branch (label after-gcd-2))
(assign s (op rem) (reg c) (reg d))
(assign c (reg d))
(assign d (reg s))
(goto (label gcd-2))
after-gcd-2
```

Рис. 5.7. Части путей данных и последовательностей команд контроллера для машины с двумя вычислениями НОД.

```
gcd-1
  (test (op *) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
after-gcd-2
.
.
.
gcd-1
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-2))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-2))
after-gcd-2
```

Рис. 5.8. Сегменты последовательности команд контроллера для машины, которая использует одни и те же компоненты путей данных для двух различных вычислений НОД.

помощью этих двух команд мы можем завершить подпрограмму `gcd` переходом в место, хранимое в регистре `continue`. Это ведет к последовательности команд, показанной на рисунке 5.10.

Машина, в которой имеется более одной подпрограммы, могла бы использовать различные регистры продолжения (например, `gcd-continue`, `factorial-continue`), или же мы могли бы для всех подпрограмм использовать один регистр `continue`. Разделение регистра экономичнее, однако тогда требуется отслеживать случаи, когда из одной подпрограммы (`sub1`) зовется другая (`sub2`). Если `sub1` не сохранит значение `continue` в каком-то другом регистре, прежде чем использовать `continue` при вызове `sub2`, то `sub1` не будет знать, откуда продолжать выполнение после ее конца. Механизм, который разрабатывается в следующем разделе для работы с рекурсией, дает хорошее решение и для проблемы с вложенными вызовами подпрограмм.

5.1.4. Реализация рекурсии с помощью стека

При помощи описанных до сих пор механизмов мы можем реализовать любой итеративный процесс, задав регистровую машину, в которой имеется по регистру на каждую переменную состояния процесса. Машина выполняет цикл контроллера, изменяя при этом состояние регистров, до тех пор, пока не окажется выполнено некоторое условие окончания процесса. В каждой точке последовательности команд контроллера состояние машины (представляющее состояние итеративного процесса) полностью определяется состоянием регистров (значением переменных состояния).

Однако реализация рекурсивных процессов требует дополнительного механизма. Рас-

```
gcd
 (test (op =) (reg b) (const 0))
 (branch (label gcd-done))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd))
gcd-done
 (test (op =) (reg continue) (const 0))
 (branch (label after-gcd-1))
 (goto (label after-gcd-2))
   .
   .
   .
  ;; Прежде, чем перейти к  gcd из первого места, где
  ;; он нужен, заносим 0~в регистр  continue
 (assign continue (const 0))
 (goto (label gcd))
after-gcd-1
   .
   .
   .
  ;;  Перед вторым использованием  gcd помещаем 1~в регистр  continue.
 (assign continue (const 1))
 (goto (label gcd))
after-gcd-2
```

Рис. 5.9. Использование регистра `continue` ради избежания повторяющейся последовательности команд с рисунка 5.8.

```
gcd
 (test (op =) (reg b) (const 0))
 (branch (label gcd-done))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd))
gcd-done
 (goto (reg continue))
   .
   .
   .
  ;; Перед вызовом  gcd заносим~в  continue
  ;; метку, на которую  gcd должен вернуться.
 (assign continue (label after-gcd-1))
 (goto (label gcd))
after-gcd-1
   .
   .
   .
  ;;  Второй вызов gcd, с другим продолжением.
 (assign continue (label after-gcd-2))
 (goto (label gcd))
after-gcd-2
```

Рис. 5.10. Присваивание регистру `continue` меток упрощает и обобщает стратегию с рисунка 5.9.

смотрим следующий рекурсивный метод вычисления факториала, описанный нами в разделе 1.2.1:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

Как мы видим из этой процедуры, вычисление $n!$ требует вычисления $(n-1)!$. Машина НОД, которая моделирует процедуру

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

также должна была вычислять НОД других чисел, помимо начальных значений. Однако между машиной НОД, которая сводит исходное вычисление к вычислению другого НОД, и `factorial`, в котором нужно вычислить другой факториал как подзадачу, есть существенная разница. В машине НОД ответ, выдаваемый новым вычислением НОД — это и есть ответ на исходную задачу. Чтобы вычислить следующий НОД, мы просто помещаем новые аргументы во входные регистры машины и заново используем ее пути данных, прогоняя ту же самую последовательность команд контроллера. Когда машина заканчивает решение последней задачи НОД, исходное вычисление также заканчивается.

В случае с факториалом (и в любом другом рекурсивном процессе) ответ на подзадачу-факториал не является решением общей задачи. Значение, полученное для $(n-1)!$, требуется еще домножить на n, чтобы получить окончательный ответ. Если мы попытаемся сымитировать решение задачи НОД и решить подзадачу-факториал, уменьшив регистр `n` и запустив машину заново, у нас больше не будет старого значения `n`, на которое можно было бы домножить результат. Для решения подзадачи нам бы потребовалась еще одна факториальная машина. Во втором вычислении факториала также есть подзадача-факториал, для нее требуется третья факториальная машина, и так далее. Поскольку внутри каждой факториальной машины содержится другая факториальная машина, в общей машине должно содержаться бесконечное гнездо вложенных друг в друга машин, а следовательно, ее нельзя построить из заранее заданного конечного числа деталей.

Тем не менее реализовать факториальный процесс в виде регистровой машины можно, если использовать одни и те же компоненты для всех встроенных ее экземпляров. а именно, машина, которая вычисляет $n!$, должна использовать одни и те же детали для работы над подзадачей вычисления $(n-1)!$, $(n-2)!$ и так далее. Такое построение возможно, поскольку, несмотря на то, что факториальный процесс требует для своего вычисления неограниченное число одинаковых машин, в каждый момент времени только одна из этих машин активна. Когда машина встречает рекурсивную подзадачу, она может остановить работу над основной задачей, использовать свои физические детали для решения подзадачи, а затем продолжить остановленное вычисление.

Содержимое регистров внутри подзадачи будет отличаться от их значения в главной задаче. (В нашем случае регистр `n` уменьшается на единицу.) Чтобы суметь продолжить

остановленное вычисление, машина должна сохранить содержимое всех регистров, которые ей понадобятся после того, как подзадача будет решена, а затем восстановить их, прежде чем возобновить работу. В случае с факториалом мы сохраним старое значение `n` и восстановим его, когда закончим вычисление факториала от уменьшенного значения регистра `n`[2].

Поскольку нет никакого априорного ограничения на число вложенных рекурсивных вызовов, нам может понадобиться хранить произвольное число значений регистров. Значения эти требуется восстанавливать в порядке, обратном порядку их сохранения, поскольку в гнезде рекурсий последняя начатая подзадача должна завершаться первой. Поэтому требуется использовать для сохранения значений регистров *стек* (stack), или структуру данных вида «последним вошел, первым вышел». Можно расширить язык регистровых машин и добавить в него стек, если ввести два новых вида команд: значения заносятся на стек командой `save` и снимаются со стека при помощи команды `restore`. После того, как последовательность значений сохранена на стеке, последовательность команд `restore` восстановит их в обратном порядке[3].

С помощью стека можно использовать для всех подзадач-факториалов единую копию путей данных факториальной машины. Имеется подобная проблема и при использовании последовательности команд контроллера, который управляет путями данных. Чтобы запустить новое вычисление факториала, контроллер не может просто перейти в начало последовательности, как в итеративном процессе, поскольку после решения подзадачи поиска $(n-1)!$ машине требуется еще домножить результат на n. Контроллер должен остановить вычисление $n!$, решить подзадачу поиска $(n-1)!$ и затем продолжить вычисление $n!$. Такой взгляд на вычисление факториала приводит к использованию механизма подпрограмм из раздела 5.1.3, при котором контроллер с помощью регистра `continue` переходит к той части последовательности команд, которая решает подзадачу, а затем продолжает с того места, где он остановился в главной задаче. Мы можем таким образом написать факториальную подпрограмму, которая возвращается к точке входа, сохраненной в регистре `continue`. При каждом вызове подпрограммы мы сохраняем и восстанавливаем регистр `continue` подобно регистру `n`, поскольку все «уровни» вычисления факториала используют один и тот же регистр `continue`. Так что факториальная подпрограмма должна записать в `continue` новое значение, когда она вызывает сама себя для решения подзадачи, но для возврата в место, откуда она была вызвана для решения подзадачи, ей потребуется старое значение `continue`.

На рисунке 5.11 показаны пути данных и контроллер машины, реализующей рекурсивную процедуру `factorial`. В этой машине имеются стек и три регистра с именами `n`, `val` и `continue`. Чтобы упростить диаграмму путей данных, мы не стали давать имена кнопкам присваивания регистров, и поименовали только кнопки работы со стеком — `sc` и `sn` для сохранения регистров, `rc` и `rn` для их восстановления. В начале работы мы кладем в регистр `n` число, факториал которого желаем вычислить, и запускаем машину. Когда машина достигает состояния `fact-done`, вычисление закончено и результат находится в регистре `val`. В последовательности команд контроллера `n` и

[2]Казалось бы, незачем сохранять старое `n`; после того, как мы его уменьшим на единицу и решим подзадачу, можно эту единицу добавить и восстановить старое значение. Такая стратегия работает для факториала, но в общем случае она работать не может, поскольку старое значение регистра не всегда можно вычислить на основании нового.

[3]В разделе 5.3 мы увидим, как стек можно реализовать на основе более элементарных операций.

continue сохраняются перед каждым рекурсивным вызовом и восстанавливаются при возврате из этого вызова. Возврат из вызова происходит путем перехода к месту, хранящемуся в continue. В начале работы машины continue получает такое значение, что последний возврат переходит в fact-done. Регистр val, где хранится результат вычисления факториала, не сохраняется перед рекурсивным вызовом, поскольку после возврата из подпрограммы его старое содержимое не нужно. Используется только новое значение val, то есть результат подвычисления.

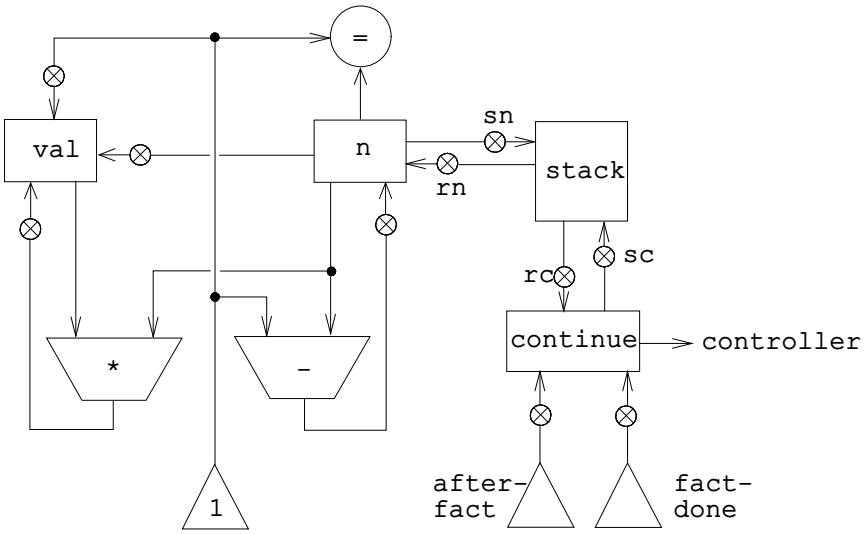
Несмотря на то, что в принципе вычисление факториала требует бесконечной машины, машина на рисунке 5.11 конечна, за исключением стека, который потенциально неограничен. Однако любая конкретная физическая реализация стека будет иметь конечный размер и таким образом будет ограничивать возможную глубину рекурсивных вызовов, которые машина сможет делать. Такая реализация факториала иллюстрирует общую стратегию реализации рекурсивных алгоритмов в виде обыкновенных регистровых машин, дополненных стеком. Когда нам требуется решить рекурсивную подзадачу, мы сохраняем на стеке регистры, текущее значение которых потребуется после решения этой подзадачи, решаем ее, затем восстанавливаем сохраненные регистры и продолжаем выполнение главной задачи. Регистр continue следует сохранять всегда. Нужно ли сохранять другие регистры, зависит от конкретной машины, поскольку не все рекурсивные вычисления нуждаются в исходных значениях регистров во время решения подзадачи (см. упражнение 5.4).

Двойная рекурсия

Рассмотрим более сложный рекурсивный процесс — древовидную рекурсию при вычислении чисел Фибоначчи, описанную в разделе 1.2.2:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Как и в случае с факториалом, рекурсивное вычисление чисел Фибоначчи можно реализовать в виде регистровой машины с регистрами n, val и continue. Машина более сложна, чем факториальная, поскольку в последовательности команд контроллера здесь два места, где нам нужно произвести рекурсивный вызов — один раз для вычисления $\mathrm{Fib}(n-1)$, а другой для вычисления $\mathrm{Fib}(n-2)$. При подготовке к этим вызовам мы сохраняем регистры, чье значение нам потребуется позже, устанавливаем в регистр n число, Fib от которого нам требуется вычислить ($n-1$ или $n-2$), и присваиваем регистру continue точку входа в главной последовательности, куда нужно вернуться (соответственно, afterfib-n-1 или afterfib-n-2). Затем мы переходим к метке fib-loop. При возврате из рекурсивного вызова ответ содержится в val. На рисунке 5.12 показана последовательность команд контроллера для этой машины.




```
(controller
   (assign continue (label fact-done))   ; установить адрес
    ; окончательного возврата
 fact-loop
   (test (op =) (reg n) (const 1))
   (branch (label base-case))
    ;; Подготовиться к рекурсивному вызову, сохраняя  n и  continue.
    ;; Установить   continue так, что вычисление продолжится
    ;; с  after-fact после возврата из подпрограммы.
   (save continue)
   (save n)
   (assign n (op -) (reg n) (const 1))
   (assign continue (label after-fact))
   (goto (label fact-loop))
 after-fact
   (restore n)
   (restore continue)
   (assign val (op *) (reg n) (reg val)) ; теперь  val содержит n(n − 1)!
   (goto (reg continue))              ; возврат в вызывающую программу
 base-case
 (assign val (const 1))               ; базовый случай: 1! = 1
   (goto (reg continue))              ; возврат в вызывающую программу
 fact-done)
```

Рис. 5.11. Рекурсивная факториальная машина.

```
(controller
   (assign continue (label fib-done))
 fib-loop
   (test (op <) (reg n) (const 2))
   (branch (label immediate-answer))
    ;; готовимся вычислить  Fib (n − 1)
   (save continue)
   (assign continue (label afterfib-n-1))
   (save n)                          ; сохранить старое значение n
   (assign n (op -) (reg n) (const 1)); записать в n n − 1
   (goto (label fib-loop))         ; произвести рекурсивный вызов
 afterfib-n-1                      ; при возврате val содержит Fib(n − 1)
   (restore n)
   (restore continue)
    ;; готовимся вычислить  Fib (n − 2)
   (assign n (op -) (reg n) (const 2))
   (save continue)
   (assign continue (label afterfib-n-2))
   (save val)                     ; сохранить  Fib(n − 1)
   (goto (label fib-loop))
 afterfib-n-2                     ; при возврате  val содержит  Fib(n − 2)
   (assign n (reg val))           ; теперь  n содержит  Fib(n − 2)
   (restore val)                  ; теперь  val содержит  Fib(n − 1)
   (restore continue)
   (assign val                    ;  Fib(n − 1)  +   Fib(n − 2)
           (op +) (reg val) (reg n))
   (goto (reg continue))          ; возврат, ответ~в  val
 immediate-answer
   (assign val (reg n))           ; базовый случай: Fib(n)  =  n
   (goto (reg continue))
 fib-done)
```

Рис. 5.12. Контроллер машины для вычисления чисел Фибоначчи.

Упражнение 5.4.
Опишите регистровые машины для реализации каждой из следующих процедур. Для каждой из этих машин напишите последовательность команд контроллера и нарисуйте диаграмму, показывающую пути данных.

а. Рекурсивное возведение в степень:

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

б. Итеративное возведение в степень:

```
(define (expt b n)
  (define (expt-iter counter product)
```

```
    (if (= counter 0)
        product
        (expt-iter (- counter 1) (* b product))))
  (expt-iter n 1))
```

Упражнение 5.5.
Смоделируйте вручную работу факториальной машины и машины Фибоначчи с каким-нибудь нетривиальным значением на входе (чтобы потребовался хотя бы один рекурсивный вызов). Покажите содержимое стека в каждый момент выполнения.

Упражнение 5.6.
Бен Битобор утверждает, что последовательность команд машины Фибоначчи содержит одну лишнюю команду `save` и одну лишнюю `restore`, которые можно убрать и получить более быструю машину. Что это за команды?

5.1.5. Обзор системы команд

Команда контроллера в нашей регистровой машине имеет одну из следующих форм, причем каждый ⟨*источник*$_i$⟩ — это либо `(reg` ⟨*имя-регистра*⟩`)`, либо `(const` ⟨*значение-константы*⟩`)`.

Команды, введенные в разделе 5.1.1:

- `(assign` ⟨*имя-регистра*⟩`(reg` ⟨*имя-регистра*⟩`))`
- `(assign` ⟨*имя-регистра*⟩ `(const` ⟨*значение-константы*⟩`))`
- `(assign` ⟨*имя-регистра*⟩ `(op` ⟨*имя-операции*⟩`)` ⟨*источник*$_1$⟩ `...` ⟨*источник*$_n$⟩`)`
- `(perform (op` ⟨*имя-операции*⟩`)` ⟨*источник*$_1$⟩ `...` ⟨*источник*$_n$⟩`)`
- `(test (op` ⟨*имя-операции*⟩`)` ⟨*источник*$_1$⟩ `...` ⟨*источник*$_n$⟩`)`
- `(branch (label` ⟨*имя-метки*⟩`))`
- `(goto (label` ⟨*имя-метки*⟩`))`

Использование регистров для хранения меток, введенное в разделе 5.1.3:

- `(assign` ⟨*имя-регистра*⟩ `(label` ⟨*имя-метки*⟩`))`
- `(goto (reg` ⟨*имя-регистра*⟩`))`

Команды для работы со стеком, введенные в разделе 5.1.4:

- `(save` ⟨*имя-регистра*⟩`)`
- `(restore` ⟨*имя-регистра*⟩`)`

До сих пор единственный вид ⟨*значений-константы*⟩, который нам встречался, — числа, но в дальнейшем мы будем использовать строки, символы и списки. Например, `(const "abc")` представляет строку `"abc"`, `(const abc)` представляет символ `abc`, `(const (a b c))` — список `(a b c)`, а `(const ())` — пустой список.

5.2. Программа моделирования регистровых машин

Чтобы как следует разобраться в работе регистровых машин, нам нужно уметь тестировать проектируемые нами машины и проверять, работают ли они в соответствии с ожиданиями. Один из способов проверки проекта состоит в ручном моделировании работы контроллера, как в упражнении 5.5. Однако этот способ подходит только для совсем простых машин. В этом разделе мы строим программу имитационного моделирования, *имитатор* (simulator), для машин, задаваемых на языке описания регистровых машин. Имитатор представляет собой программу на Scheme с четырьмя интерфейсными процедурами. Первая из них на основе описания регистровой машины строит ее модель (структуру данных, части которой соответствуют частям имитируемой машины), а остальные три позволяют имитировать машину, работая с этой моделью:

- `(make-machine` ⟨*имена-регистров*⟩ ⟨*операции*⟩ ⟨*контроллер*⟩`)` строит и возвращает модель машины с указанными регистрами, операциями и контроллером.
- `(set-register-contents!`⟨*модель-машины*⟩⟨*имя-регистра*⟩ ⟨*значение*⟩`)` записывает значение в имитируемый регистр указанной машины.
- `(get-register-contents` ⟨*модель-машины*⟩ ⟨*имя-регистра*⟩`)` возвращает содержимое имитируемого регистра указанной машины.
- `(start` ⟨*модель-машины*⟩`)` имитирует работу данной машины. Машина запускается с начала последовательности команд контроллера и останавливается, когда достигнут конец этой последовательности.

В качестве примера того, как используются эти процедуры, можно определить переменную `gcd-machine` как модель машины НОД из раздела 5.1.1 следующим образом:

```
(define gcd-machine
  (make-machine
   '(a b t)
   (list (list 'rem remainder) (list '= =))
   '(test-b
       (test (op =) (reg b) (const 0))
       (branch (label gcd-done))
       (assign t (op rem) (reg a) (reg b))
       (assign a (reg b))
       (assign b (reg t))
       (goto (label test-b))
     gcd-done)))
```

Первым аргументом `make-machine` является список имен регистров. Второй аргумент — таблица (список двухэлементных списков), связывающая каждое имя операции с процедурой Scheme, которая эту операцию реализует (то есть порождает тот же результат на тех же входных значениях). Последний аргумент описывает контроллер в виде списка из меток и машинных команд, как в разделе 5.1.

Чтобы вычислить НОД двух чисел с помощью этой машины, мы заносим значения во входные регистры, запускаем машину, а когда имитация ее работы завершается, считываем результат:

```
(set-register-contents! gcd-machine 'a 206)
done

(set-register-contents! gcd-machine 'b 40)
done

(start gcd-machine)
done

(get-register-contents gcd-machine 'a)
2
```

Эта модель будет работать значительно медленнее, чем процедура `gcd`, написанная на Scheme, поскольку она имитирует низкоуровневые команды машины, например, `assign`, с помощью значительно более сложных операций.

Упражнение 5.7.
Проверьте на имитаторе машины, построенные Вами в упражнении 5.4.

5.2.1. Модель машины

Модель машины, которую порождает `make-machine`, представляется в виде процедуры с внутренним состоянием при помощи методов передачи сообщений, разработанных в главе 3. При построении модели `make-machine` прежде всего вызывает процедуру `make-new-machine`, порождающую те части модели, которые у всех регистровых машин одинаковые. Эта базовая модель машины, создаваемая `make-new-machine`, является, в сущности, контейнером для нескольких регистров и стека, а кроме того, содержит механизм выполнения, который обрабатывает команды контроллера одну за другой.

Затем `make-machine` расширяет эту базовую модель (посылая ей сообщения) и добавляет в нее регистры, операции и контроллер для конкретной определяемой машины. Сначала она выделяет в новой машине по регистру на каждое из данных имен регистров и встраивает в нее указанные операции. Затем она с помощью *ассемблера* (assembler) (описанного в разделе 5.2.2) преобразует список контроллера в команды новой машины и устанавливает их ей в качестве последовательности команд. В качестве результата `make-machine` возвращает модифицированную модель машины.

```
(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each (lambda (register-name)
                ((machine 'allocate-register) register-name))
              register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

Регистры

Мы будем представлять регистры в виде процедур с внутренним состоянием, как в главе 3. Процедура `make-register` создает регистр. Регистр содержит значение, которое можно считать или изменить.

```
(define (make-register name)
  (let ((contents '*unassigned*))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value) (set! contents value)))
            (else
             (error "Неизвестная операция -- REGISTER" message))))
    dispatch))
```

Для доступа к регистрам используются следующие процедуры:

```
(define (get-contents register)
  (register 'get))

(define (set-contents! register value)
  ((register 'set) value))
```

Стек

Стек также можно представить в виде процедуры с внутренним состоянием. Процедура `make-stack` создает стек, внутреннее состояние которого состоит из списка элементов на стеке. Стек принимает сообщение `push`, кладущее элемент на стек, сообщение `pop`, снимающее со стека верхний элемент и возвращающее его, и сообщение `initialize`, которое дает стеку начальное пустое значение.

```
(define (make-stack)
  (let ((s '()))
    (define (push x)
      (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Пустой стек -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top)))
    (define (initialize)
      (set! s '())
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Неизвестная операция -- STACK"
                         message))))
    dispatch))
```

Для доступа к стеку используются следующие процедуры:

```
(define (pop stack)
  (stack 'pop))

(define (push stack value)
  ((stack 'push) value))
```

Базовая машина

Процедура `make-new-machine`, приведенная на рисунке 5.13, порождает объект, внутреннее состояние которого состоит из стека, изначально пустой последовательности команд, списка операций, где с самого начала присутствует операция инициализации стека, а также *таблицы регистров* (register table), в которой изначально содержатся два регистра, `flag` и `pc` (от *program counter*, «счетчик программы»). Внутренняя процедура `allocate-register` добавляет в таблицу новый элемент, а внутренняя процедура `lookup-register` ищет регистр в таблице.

Регистр `flag` используется для управления переходами в имитируемой машине. Команды `test` присваивают ему результат теста (истину или ложь). Команды `branch` определяют, нужно ли делать переход, в зависимости от значения регистра `flag`.

Регистр `pc` определяет порядок выполнения команд при работе машины. Этот порядок реализуется внутренней процедурой `execute`. В нашей имитационной модели каждая команда является структурой данных, в которой есть процедура без аргументов, называемая *исполнительная процедура команды* (instruction execution procedure). Вызов этой процедуры имитирует выполнение команды. Во время работы модели `pc` указывает на часть последовательности команд, начинающуюся со следующей подлежащей исполнению команды. Процедура `execute` считывает эту команду, выполняет ее при помощи вызова исполнительной процедуры, и повторяет этот процесс, пока имеется команды для выполнения (то есть пока `pc` не станет указывать на конец последовательности команд).

В процессе работы каждая исполнительная процедура изменяет `pc` и указывает, какую следующую команду надо выполнить. Команды `branch` и `goto` присваивают регистру `pc` значение, указывающее на новый адрес. Все остальные команды просто продвигают `pc` так, чтобы он указывал на следующую команду в последовательности. Заметим, что каждый вызов `execute` снова зовет `execute`, но это не приводит к бесконечному циклу, поскольку запуск исполнительной процедуры команды изменяет содержимое `pc`.

`Make-new-machine` возвращает процедуру `dispatch`, которая дает доступ к внутреннему состоянию на основе передачи сообщений. Запуск машины осуществляется путем установки `pc` в начало последовательности команд и вызова `execute`.

Ради удобства мы предоставляем альтернативный процедурный интерфейс для операции `start` регистровой машины, а также процедуры для доступа к содержимому регистров и их изменения, как указано в начале раздела 5.2:

```
(define (start machine)
  (machine 'start))

(define (get-register-contents machine register-name)
  (get-contents (get-register machine register-name)))
```

```
(define (set-register-contents! machine register-name value)
  (set-contents! (get-register machine register-name) value)
  'done)
```

Все эти процедуры (а также многие процедуры из разделов 5.2.2 и 5.2.3) следующим образом ищут регистр с данным именем в данной машине:

```
(define (get-register machine reg-name)
  ((machine 'get-register) reg-name))
```

5.2.2. Ассемблер

Ассемблер переводит последовательность выражений контроллера машины в соответствующий ей список машинных команд, каждая со своей исполнительной процедурой. По общему строению ассемблер подобен интерпретаторам, которые мы изучали в главе 4 — имеется входной язык (в этом случае язык регистровых машин), и нам нужно выполнить некоторое действие для каждого типа выражений этого языка.

Методика порождения исполнительной процедуры для каждой команды в точности та же, которой мы пользовались в разделе 4.1.7, чтобы ускорить интерпретацию путем отделения синтаксического анализа от выполнения. Как мы видели в главе 4, существенную часть полезного анализа выражений Scheme можно провести, не зная конкретных значений переменных. Подобным образом и здесь существенную часть анализа выражений машинного языка можно провести, не зная конкретного содержимого регистров машины. Например, можно заменить имена регистров указателями на объекты-регистры, а имена меток — указателями на те места в последовательности команд, которые метками обозначаются.

Прежде чем порождать исполнительные процедуры команд, ассемблер должен знать, куда указывают все метки, так что вначале он просматривает текст контроллера и отделяет метки от команд. При просмотре текста он строит список команд и таблицу, которая связывает каждую метку с указателем внутрь этого списка. Затем ассемблер дополняет список команд, вставляя в каждую команду исполнительную процедуру.

Процедура `assemble` — основной вход в ассемблер. Она принимает в качестве аргументов текст контроллера и модель машины, а возвращает последовательность команд, которую нужно сохранить в модели. `Assemble` вызывает `extract-labels`, чтобы построить из данного ей списка контроллера исходный список команд и таблицу меток. Вторым аргументом `extract-labels` служит процедура, которую следует позвать для обработки этих результатов: эта процедура зовет `update-insts!`, чтобы породить исполнительные процедуры для команд и вставить их в командный список, а затем возвращает модифицированный список команд.

```
(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      insts)))
```

`Extract-labels` принимает на входе список `text` (последовательность выражений, обозначающих команды контроллера) и процедуру `receive`. `Receive` будет вызвана с

```
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))))
          (register-table
           (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Многократно определенный регистр: " name)
            (set! register-table
                  (cons (list name (make-register name))
                        register-table)))
        'register-allocated)
      (define (lookup-register name)
        (let ((val (assoc name register-table)))
          (if val
              (cadr val)
              (error "Неизвестный регистр:" name))))
      (define (execute)
        (let ((insts (get-contents pc)))
          (if (null? insts)
              'done
              (begin
                ((instruction-execution-proc (car insts)))
                (execute)))))
      (define (dispatch message)
        (cond ((eq? message 'start)
               (set-contents! pc the-instruction-sequence)
               (execute))
              ((eq? message 'install-instruction-sequence)
               (lambda (seq) (set! the-instruction-sequence seq)))
              ((eq? message 'allocate-register) allocate-register)
              ((eq? message 'get-register) lookup-register)
              ((eq? message 'install-operations)
               (lambda (ops) (set! the-ops (append the-ops ops))))
              ((eq? message 'stack) stack)
              ((eq? message 'operations) the-ops)
              (else (error "Неизвестная операция -- MACHINE" message))))
      dispatch)))
```

Рис. 5.13. Процедура `make-new-machine`, реализующая базовую модель машины.

двумя аргументами: (1) списком insts структур данных, каждая из которых содержит команду из text; и (2) таблицей под названием labels, связывающей каждую метку из text с позицией в списке insts, которую эта метка обозначает.

```
(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels (cdr text)
       (lambda (insts labels)
         (let ((next-inst (car text)))
           (if (symbol? next-inst)
               (receive insts
                        (cons (make-label-entry next-inst
                                                insts)
                              labels))
               (receive (cons (make-instruction next-inst)
                              insts)
                        labels)))))))
```

Работа extract-labels заключается в последовательном просмотре элементов text и сборке insts и labels. Если очередной элемент является символом (то есть меткой), соответствующий вход добавляется в таблицу labels. В противном случае элемент добавляется к списку insts[4].

Update-insts! модифицирует командный список, который сначала содержит только текст команд, так, чтобы в нем имелись соответствующие исполнительные процедуры:

[4]Процедура receive используется здесь, в сущности, для того, чтобы заставить extract-labels вернуть два значения — labels и insts, — не создавая специально структуры данных для их хранения. Альтернативная реализация, которая явным образам возвращает пару значений, выглядит так:

```
(define (extract-labels text)
  (if (null? text)
      (cons '() '())
      (let ((result (extract-labels (cdr text))))
        (let ((insts (car result)) (labels (cdr result)))
          (let ((next-inst (car text)))
            (if (symbol? next-inst)
                (cons insts
                      (cons (make-label-entry next-inst insts) labels))
                (cons (cons (make-instruction next-inst) insts)
                      labels)))))))
```

Вызывать ее из assemble следовало бы таким образом:

```
(define (assemble controller-text machine)
  (let ((result (extract-labels controller-text)))
    (let ((insts (car result)) (labels (cdr result)))
      (update-insts! insts labels machine)
      insts)))
```

Можно считать, что использование receive показывает изящный способ вернуть несколько значений, а можно считать, что это просто оправдание для демонстрации программистского трюка. Аргумент, который, как receive, является процедурой, вызываемой в конце, называется «продолжением». Напомним, что мы уже использовали продолжения для того, чтобы реализовать управляющую структуру перебора с возвратом в разделе 4.3.3.

```
(define (update-insts! insts labels machine)
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag))
        (stack (machine 'stack))
        (ops (machine 'operations)))
    (for-each
     (lambda (inst)
       (set-instruction-execution-proc!
        inst
        (make-execution-procedure
         (instruction-text inst) labels machine
         pc flag stack ops)))
     insts)))
```

Структура данных для машинной команды просто сочетает текст команды с соответствующей исполнительной процедурой. Когда `extract-labels` создает команду, исполнительной процедуры еще нет, и она вставляется позже из процедуры `update-insts!`:

```
(define (make-instruction text)
  (cons text '()))

(define (instruction-text inst)
  (car inst))

(define (instruction-execution-proc inst)
  (cdr inst))

(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))
```

Текст команды не используется в имитаторе, но сохраняется в целях отладки (см. упражнение 5.16).

Элементы таблицы меток — это пары:

```
(define (make-label-entry label-name insts)
  (cons label-name insts))
```

Записи в таблице можно искать через

```
(define (lookup-label labels label-name)
  (let ((val (assoc label-name labels)))
    (if val
        (cdr val)
        (error "Неопределенная метка -- ASSEMBLE" label-name))))
```

Упражнение 5.8.

Следующий код для регистровой машины неоднозначен, поскольку метка `here` определена более одного раза:

```
start
  (goto (label here))
here
  (assign a (const 3))
  (goto (label there))
here
  (assign a (const 4))
  (goto (label there))
there
```

Каково будет содержимое регистра `a` в нашем имитаторе, когда управление достигнет `there`? Измените процедуру `extract-labels` так, чтобы ассемблер сообщал об ошибке в случае, когда одно и то же имя метки обозначает две различных точки в коде.

5.2.3. Порождение исполнительных процедур для команд

Чтобы породить для команды исполнительную процедуру, ассемблер зовет `make-execution-procedure`. Как и процедура `analyze` в интерпретаторе из раздела 4.1.7, она делает выбор на основе типа команды и порождает соответствующую исполнительную процедуру.

```
(define (make-execution-procedure inst labels machine
                                  pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else (error "Неизвестный тип команды -- ASSEMBLE"
                     inst))))
```

Для каждого типа команд языка регистровых машин имеется процедура-генератор, которая порождает исполнительные процедуры. Детали порождаемых процедур определяют как синтаксис, так и семантику отдельных команд машинного языка. Мы изолируем с помощью абстракции данных конкретный синтаксис выражений языка регистровых машин от общего механизма вычисления, подобно тому, как мы это делали для интерпретатора из раздела 4.1.2, и для считывания и классификации частей команды используем синтаксические процедуры.

Команды `assign`

Процедура `make-assign` обрабатывает команды `assign`:

```
(define (make-assign inst machine labels operations pc)
  (let ((target
         (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    (let ((value-proc
           (if (operation-exp? value-exp)
               (make-operation-exp
                value-exp machine labels operations)
               (make-primitive-exp
                (car value-exp) machine labels))))
      (lambda ()      ; исполнительная процедура для  assign
        (set-contents! target (value-proc))
        (advance-pc pc)))))
```

`Make-assign` извлекает имя целевого регистра (второй элемент команды) и выражение-значение (остаток списка, представляющего команду) из команды `assign` с помощью селекторов

```
(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))

(define (assign-value-exp assign-instruction)
  (cddr assign-instruction))
```

По имени регистра с помощью `get-register` находится объект-целевой регистр. Выражение-значение передается в `make-operation-exp`, если значение является результатом операции, и в `make-primitive-exp` в противном случае. Эти процедуры (приведенные ниже) рассматривают выражение-значение и порождают исполнительную процедуру для вычисления этого выражения. Это процедура без аргументов, называемая `value-proc`, которая будет вызвана во время работы имитатора и породит значение, которое требуется присвоить регистру. Заметим, что поиск регистра по имени и разбор выражения-значения происходят только один раз во время ассемблирования, а не каждый раз при выполнении команды. Именно ради такой экономии работы мы используем исполнительные процедуры, и этот выигрыш прямо соответствует экономии, полученной путем отделения синтаксического анализа от выполнения в интерпретаторе из раздела 4.1.7.

Результат, возвращаемый `make-assign` — это исполнительная процедура для команды `assign`. Когда эта процедура вызывается (из процедуры `execute` модели), она записывает в целевой регистр результат, полученный при выполнении `value-proc`. Затем она передвигает `pc` на следующую команду с помощью процедуры

```
(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))
```

`Advance-pc` вызывается в конце исполнения всех команд, кроме `branch` и `goto`.

Команды `test`, `branch` **и** `goto`

`Make-test` обрабатывает команду `test` похожим образом. Эта процедура извлекает выражение, которое определяет подлежащее проверке условие, и порождает для него исполнительную процедуру. Во время работы модели эта процедура для условия вызывается, результат ее сохраняется в регистре `flag`, и `pc` передвигается на шаг:

```
(define (make-test inst machine labels operations flag pc)
  (let ((condition (test-condition inst)))
    (if (operation-exp? condition)
        (let ((condition-proc
               (make-operation-exp
                condition machine labels operations)))
          (lambda ()
            (set-contents! flag (condition-proc))
            (advance-pc pc)))
        (error "Плохая команда TEST -- ASSEMBLE" inst))))

(define (test-condition test-instruction)
  (cdr test-instruction))
```

Исполнительная процедура для команды `branch` проверяет содержимое регистра `flag` и либо записывает в содержимое `pc` целевой адрес (если переход происходит), либо просто продвигает `pc` (если переход не происходит). Заметим, что указанная точка назначения для команды `branch` обязана быть меткой, и процедура `make-branch` это проверяет. Заметим также, что поиск метки происходит во время ассемблирования, а не при каждом исполнении команды `branch`.

```
(define (make-branch inst machine labels flag pc)
  (let ((dest (branch-dest inst)))
    (if (label-exp? dest)
        (let ((insts
               (lookup-label labels (label-exp-label dest))))
          (lambda ()
            (if (get-contents flag)
                (set-contents! pc insts)
                (advance-pc pc))))
        (error "Плохая команда BRANCH -- ASSEMBLE" inst))))

(define (branch-dest branch-instruction)
  (cadr branch-instruction))
```

Команда `goto` подобна `branch`, но только здесь в качестве целевого адреса может быть указана либо метка, либо регистр, и не надо проверять никакого условия — в `pc` всегда записывается новый адрес.

```
(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts
```

```
                  (lookup-label labels
                                (label-exp-label dest))))
            (lambda () (set-contents! pc insts))))
          ((register-exp? dest)
           (let ((reg
                  (get-register machine
                                (register-exp-reg dest))))
            (lambda ()
              (set-contents! pc (get-contents reg)))))
          (else (error "Плохая команда GOTO -- ASSEMBLE"
                       inst)))))

(define (goto-dest goto-instruction)
  (cadr goto-instruction))
```

Остальные команды

Команды работы со стеком `save` и `restore` просто используют стек и указанный регистр, а затем продвигают `pc`:

```
(define (make-save inst machine stack pc)

  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))

(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))

(define (stack-inst-reg-name stack-instruction)
  (cadr stack-instruction))
```

Последний тип команды, обрабатываемый процедурой `make-perform`, порождает исполнительную процедуру для требуемого действия. Во время работы имитатора действие выполняется и продвигается `pc`.

```
(define (make-perform inst machine labels operations pc)
  (let ((action (perform-action inst)))
    (if (operation-exp? action)
        (let ((action-proc
               (make-operation-exp
                action machine labels operations)))
          (lambda ()
            (action-proc)
```

```
            (advance-pc pc)))
        (error "Плохая команда PERFORM -- ASSEMBLE" inst))))

(define (perform-action inst) (cdr inst))
```

Исполнительные процедуры для подвыражений

Значение выражения `reg`, `label` или `const` может потребоваться для присваивания регистру (`make-assign`) или как аргумент операции (`make-operation-exp`, далее). Следующая процедура порождает исполнительные процедуры, которые вычисляют во время выполнения значения этих выражений:

```
(define (make-primitive-exp exp machine labels)
  (cond ((constant-exp? exp)
         (let ((c (constant-exp-value exp)))
           (lambda () c)))
        ((label-exp? exp)
         (let ((insts
                (lookup-label labels
                              (label-exp-label exp))))
           (lambda () insts)))
        ((register-exp? exp)
         (let ((r (get-register machine
                                (register-exp-reg exp))))
           (lambda () (get-contents r))))
        (else
         (error "Неизвестный тип выражения -- ASSEMBLE" exp))))
```

Синтаксис выражений `reg`, `label` и `const` определяется так:

```
(define (register-exp? exp) (tagged-list? exp 'reg))

(define (register-exp-reg exp) (cadr exp))

(define (constant-exp? exp) (tagged-list? exp 'const))

(define (constant-exp-value exp) (cadr exp))

(define (label-exp? exp) (tagged-list? exp 'label))

(define (label-exp-label exp) (cadr exp))
```

Команды `assign`, `test` и `perform` могут включать в себя применение машинной операции (определяемой выражением `op`) к нескольким операндам (определяемым выражениями `reg` или `const`). Следующая процедура порождает исполнительную процедуру для «выражения-операции» — списка, состоящего из выражений внутри команды, обозначающих операцию и операнды.

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
```

```
        (aprocs
         (map (lambda (e)
                (make-primitive-exp e machine labels))
              (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))
```

Синтаксис выражений-операций определяется через

```
(define (operation-exp? exp)
  (and (pair? exp) (tagged-list? (car exp) 'op)))

(define (operation-exp-op operation-exp)
  (cadr (car operation-exp)))

(define (operation-exp-operands operation-exp)
  (cdr operation-exp))
```

Заметим, что обработка выражений-операций очень напоминает обработку вызовов процедур процедурой `analyze-application` интерпретатора из раздела 4.1.7, поскольку там и тут мы порождаем исполнительные процедуры для каждого операнда. Во время работы мы вызываем эти процедуры для операндов и применяем процедуру Scheme, которая имитирует операцию, к полученным значениям. Имитирующая процедура получается путем поиска имени операции в таблице операций регистровой машины:

```
(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (error "Неизвестная операция -- ASSEMBLE" symbol))))
```

Упражнение 5.9.
Приведенная в тексте обработка позволяет машинным операциям принимать в качестве аргументов не только константы и содержимое регистров, но и метки. Измените процедуры, обрабатывающие выражения, и обеспечьте выполнение условия, что операции можно применять исключительно к регистрам и константам.

Упражнение 5.10.
Придумайте новый синтаксис для команд регистровой машины и измените имитатор так, чтобы он использовал Ваш новый синтаксис. Можете ли Вы реализовать свой синтаксис, ничего не трогая, кроме синтаксических процедур из этого раздела?

Упражнение 5.11.
Когда мы в разделе 5.1.4 определяли `save` и `restore`, мы не указали, что произойдет, если попытаться восстановить значение не в том регистре, который был сохранен последним, как в последовательности команд

```
(save y)
(save x)
(restore y)
```

Есть несколько разумных вариантов значения `restore`:

а. `(restore y)` переносит в `y` последнее значение, сохраненное на стеке, независимо от того, откуда это значение взялось. Так работает наш имитатор. Покажите, как с помощью такого поведения убрать одну команду из машины Фибоначчи (раздел 5.1.4, рисунок 5.12).

б. `(restore y)` переносит в `y` последнее значение, сохраненное на стеке, но только в том случае, когда это значение происходит из регистра `y`; иначе возникает сообщение об ошибке. Модифицируйте имитатор и заставьте его вести себя таким образом. Придется изменить `save` так, чтобы он сохранял имя регистра вместе со значением.

в. `(restore y)` переносит в `y` последнее значение, сохраненное из `y`, независимо от того, какие другие регистры были сохранены и не восстановлены после `y`. Модифицируйте имитатор так, чтобы он вел себя таким образом. С каждым регистром придется связать свой собственный стек. Операция `initialize-stack` должна инициализировать стеки всех регистров.

Упражнение 5.12.
При помощи имитатора можно определять пути данных, которые требуются для реализации машины с данным контроллером. Расширьте ассемблер и заставьте его хранить следующую информацию о модели машины:

- список всех команд, с удаленными дубликатами, отсортированный по типу команды (`assign`, `goto` и так далее).
- список (без дубликатов) регистров, в которых хранятся точки входа (это те регистры, которые упоминаются в командах `goto`).
- Список (без дубликатов) регистров, к которым применяются команды `save` или `restore`.
- Для каждого регистра, список (без дубликатов) источников, из которых ему присваивается значение (например, для регистра `val` в факториальной машине на рисунке 5.11 источниками являются `(const 1)` и `((op *) (reg n) (reg val))`).

Расширьте интерфейс сообщений машины и включите в него доступ к новой информации. Чтобы проверить свой анализатор, примените его к машине Фибоначчи с рисунка 5.12 и рассмотрите получившиеся списки.

Упражнение 5.13.
Модифицируйте имитатор так, чтобы он определял, какие регистры присутствуют в машине, из последовательности команд контроллера, а не принимал список регистров в качестве аргумента `make-machine`. Вместо того, чтобы выделять регистры в `make-machine` заранее, их можно создавать по одному, когда они встречаются в первый раз при ассемблировании команд.

5.2.4. Отслеживание производительности машины

Имитационное моделирование может служить не только для проверки правильности проекта машины, но и для измерения ее производительности. К примеру, можно установить в нашу машину «счетчик», измеряющий количество операций со стеком, задействованных при вычислении. Для этого мы изменяем моделируемый стек и следим, сколько раз регистры сохраняются на стеке, регистрируем максимальную глубину, которой он достигает, а также добавляем к интерфейсу стека сообщение, которое распечатывает статистику, как показано ниже. Кроме того, мы добавляем к базовой машине операцию для распечатки статистики, устанавливая значение `the-ops` в `make-new-machine` в

```
(list (list 'initialize-stack
            (lambda () (stack 'initialize)))
      (list 'print-stack-statistics
            (lambda () (stack 'print-statistics))))
```

Вот новая версия `make-stack`:

```
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop)
      (if (null? s)
          (error "Пустой стек -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            (set! current-depth (- current-depth 1))
            top)))
    (define (initialize)
      (set! s '())
      (set! number-pushes 0)
      (set! max-depth 0)
      (set! current-depth 0)
      'done)
    (define (print-statistics)
      (newline)
      (display (list 'total-pushes  '= number-pushes
                     'maximum-depth '= max-depth)))
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            ((eq? message 'print-statistics)
             (print-statistics))
            (else
             (error "Неизвестная операция -- STACK" message))))
    dispatch))
```

В упражнениях от 5.15 до 5.19 описываются другие полезные возможности для сбора информации и отладки, которые можно добавить к имитатору регистровых машин.

Упражнение 5.14.
Измерьте количество сохранений и максимальную глубину стека, требуемую для вычисления $n!$ при различных малых значениях n с помощью факториальной машины, показанной на рисун-

ке 5.11. По этим данным определите формулы в зависимости от n для числа сохранений и максимальной глубины стека, требуемых для вычисления $n!$ при любом $n > 1$. Обратите внимание, что это линейные функции от n, и они определяются двумя константами. Чтобы увидеть статистику, Вам придется добавить к факториальной машине команды для инициализации стека и распечатки статистики. Можно также заставить машину в цикле считывать `n`, вычислять факториал и печатать результат (как для машины НОД с рисунка 5.4), так, чтобы не нужно было все время вызывать `get-register-contents`, `set-register-contents!` и `start`.

Упражнение 5.15.
Добавьте к модели регистровой машины *подсчет команд* (instruction counting). Это значит, что машина должна подсчитывать число выполненных ею команд. Расширьте интерфейс модели и добавьте новое сообщение, которое печатает счетчик команд и переустанавливает его в ноль.

Упражнение 5.16.
Добавьте к имитатору *трассировку команд* (instruction tracing). а именно, перед тем, как выполнить каждую команду, имитатор должен распечатывать ее текст. Заставьте модель принимать сообщения `trace-on` и `trace-off`, которые включают и выключают трассировку.

Упражнение 5.17.
Расширьте трассировку команд из упражнения 5.16 так, чтобы перед тем, как печатать команду, имитатор распечатывал метки, которые стоят в последовательности контроллера непосредственно перед этой командой. Постарайтесь при этом не помешать подсчету команд (упражнение 5.15). Придется заставить имитатор хранить необходимую информацию о метках.

Упражнение 5.18.
Измените процедуру `make-register` из раздела 5.2.1, так, чтобы можно было трассировать регистры. Регистры должны принимать сообщения, которые включают и выключают трассировку. Когда регистр подвергается трассировке, присваивание ему значения должно вызывать распечатку имени регистра, старого его содержимого и нового, которое ему присваивается. Расширьте интерфейс модели и дайте пользователю возможность включать и выключать трассировку для указанных регистров машины.

Упражнение 5.19.
Лиза П. Хакер хочет добавить в имитатор *контрольные точки* (breakpoints) для облегчения отладки проектов машин. Вас наняли для реализации такой возможности. Лиза хочет, чтобы в последовательности команд контроллера можно было указать место, где имитатор остановится и позволит ей исследовать состояние машины. Вам нужно реализовать процедуру

```
(set-breakpoint ⟨машина⟩ ⟨метка⟩ ⟨n⟩)
```

которая устанавливает контрольную точку перед n-й командой, следующей за указанной меткой. Например,

```
(set-breakpoint gcd-machine 'test-b 4)
```

установит контрольную точку в `gcd-machine` непосредственно перед присваиванием регистру `a`. Когда моделирование достигает контрольной точки, имитатор должен распечатать метку и смещение точки, а затем прекратить выполнение команд. Тогда Лиза может с помощью `get-register-contents` и `set-register-contents!` исследовать и изменять состояние имитируемой машины. Затем она должна быть способна продолжить выполнение, сказав

`(proceed-machine ⟨машина⟩)`

Кроме того, необходимо иметь возможность удалить контрольную точку с помощью

`(cancel-breakpoint ⟨машина⟩ ⟨метка⟩ ⟨n⟩)`

и удалить все контрольные точки с помощью

`(cancel-all-breakpoints ⟨машина⟩)`

5.3. Выделение памяти и сборка мусора

В разделе 5.4 мы покажем, как реализовать вычислитель для Scheme в виде регистровой машины. Для того, чтобы упростить обсуждение, мы будем предполагать, что наши машины обладают *памятью со списковой структурой* (list-structured memory), в которой основные операции по работе с данными списковой структуры элементарны. Постулирование такой памяти — удобная абстракция, если мы хотим сконцентрировать внимание на механизмах управления в интерпретаторе Scheme, однако она не дает реалистической картины того, как на самом деле устроены элементарные операции с данными в современных компьютерах. Для того, чтобы получить более полное понимание работы Лисп-системы, требуется исследовать, как списковую структуру можно представить способом, совместимым с устройством памяти обыкновенных компьютеров.

При реализации списковой структуры возникает два вопроса. Первый относится только к способу представления: как изобразить структуру «ячеек и указателей», присущую лисповским парам, используя только механизмы хранения и адресации, которыми обладает обыкновенная компьютерная память. Второй вопрос связан с управлением памятью по мере того, как вычисление развивается. Работа Лисп-системы существенным образом зависит от ее способности постоянно создавать новые объекты данных. Сюда включаются как объекты, которые явным образом выделяются в интерпретируемых Лисп-процедурах, так и структуры, создаваемые самим интерпретатором, например окружения и списки аргументов. Несмотря на то, что постоянное создание новых объектов данных не вызвало бы проблемы на компьютере с бесконечным количеством быстродействующей памяти, в настоящих компьютерах объем доступной памяти ограничен (к сожалению). Поэтому Лисп-системы реализуют *автоматическое распределение памяти* (automatic storage allocation), которое поддерживает иллюзию бесконечной памяти. Когда объект данных перестает быть нужным, занятая под него память автоматически освобождается и используется для построения новых объектов данных. Имеются различные методы реализации такого автоматического распределителя памяти. Метод, обсуждаемый нами в этом разделе, называется *сборка мусора* (garbage collection).

5.3.1. Память как векторы

Обыкновенную память компьютера можно рассматривать как массив клеток, каждая из которых может содержать кусочек информации. У каждой клетки имеется собственное имя, которое называется ее *адресом* (address). Типичная система памяти предоставляет две элементарные операции: одна считывает данные, хранящиеся по указанному адресу, а вторая записывает по указанному адресу новые данные. Адреса памяти можно складывать с целыми числами и получать таким образом последовательный доступ к

некоторому множеству клеток. Если говорить более общо, многие важные операции с данными требуют, чтобы адреса памяти рассматривались как данные, которые можно записывать в ячейки памяти, и которыми можно манипулировать в регистрах машины. Представление списковой структуры — одно из применений такой *адресной арифметики* (address arithmetic).

Для моделирования памяти компьютера мы используем новый вид структуры данных, называемый *вектором* (vector). С абстрактной точки зрения, вектор представляет собой составной объект, к отдельным элементам которого можно обращаться при помощи целочисленного индекса за время, независимое от величины индекса[5]. Чтобы описать операции с памятью, мы пользуемся двумя элементарными процедурами Scheme для работы с векторами:

- `(vector-ref` ⟨*вектор*⟩ ⟨*n*⟩`)` возвращает n-ый элемент вектора.
- `(vector-set!` ⟨*вектор*⟩ ⟨*n*⟩ ⟨*значение*⟩`)` устанавливает n-ый элемент вектора в указанное значение.

Например, если `v` — вектор, то `(vector-ref v 5)` получает его пятый элемент, а `(vector-set! v 5 7)` устанавливает значение его пятого элемента равным 7[6]. В памяти компьютера такой доступ можно было бы организовать через адресную арифметику, сочетая *базовый адрес* (base address), который указывает на начальное положение вектора в памяти, с *индексом* (index), который указывает смещение определенного элемента вектора.

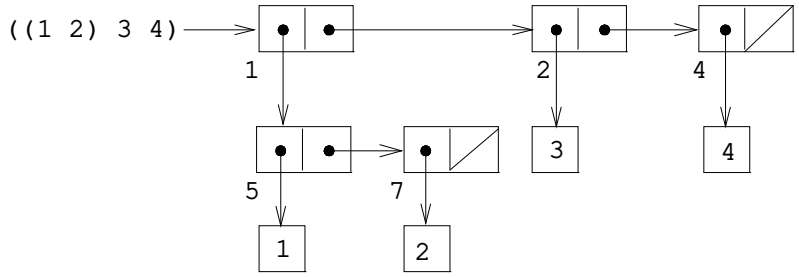
Представление лисповских данных

С помощью списков можно реализовать пары — основные объекты данных, нужные для памяти со списковой структурой. Представим, что память разделена на два вектора: `the-cars` и `the-cdrs`. Списковую структуру мы будем представлять следующим образом: указатель на пару есть индекс, указывающий внутрь этих двух векторов. Содержимое элемента `the-cars` с указанным индексом является `car` пары, а содержимое элемента `the-cdrs` с тем же индексом является `cdr` пары. Кроме того, нам нужно представление для объектов помимо пар (например, чисел и символов) и способ отличать один тип данных от другого. Есть много способов этого добиться, но все они сводятся к использованию *типизированных указателей* (typed pointers) — то есть понятие «указатель» расширяется и включает в себя тип данных[7]. Тип данных позволяет системе отличить указатель на пару (который состоит из метки типа данных «пара» и индекса в вектора памяти) от указателей на другие типы данных (которые состоят из метки какого-то другого типа и того, что используется для представления значений это-

[5] Можно было бы представить память в виде списка ячеек. Однако тогда время доступа не было бы независимым от индекса, поскольку доступ к n-му элементу списка требует $n-1$ операций `cdr`.

[6] Полноты ради, надо было бы указать еще операцию `make-vector`, которая создает вектора. Однако в текущем приложении мы используем вектора исключительно для моделирования заранее заданных участков компьютерной памяти.

[7] Это в точности понятие «помеченных данных», которое мы ввели в главе 2 для работы с обобщенными операциями. Однако здесь типы данных вводятся на элементарном машинном уровне, а не конструируются через списки.



Индекс	0	1	2	3	4	5	6	7	8	...
the-cars		p5	n3		n4	n1		n2		...
the-cdrs		p2	p4		e0	p7		e0		...

Рис. 5.14. Представления списка `((1 2) 3 4)` в виде стрелочной диаграммы и в виде вектора памяти.

го типа). Два объекта данных считаются равными (`eq?`), если равны указатели на них[8]. На рисунке 5.14 показано, как с помощью этого метода представляется список `((1 2) 3 4)`, а также дана его стрелочная диаграмма. Информацию о типах мы обозначаем через буквенные префиксы. Например, указатель на пару с индексом 5 обозначается `p5`, пустой список обозначается `e0`, а указатель на число 4 обозначается `n4`. На стрелочной диаграмме мы в левом нижнем углу каждой пары ставим индекс вектора, который показывает, где хранятся `car` и `cdr` пары. Пустые клетки в `the-cars` и `the-cdrs` могут содержать части других структур (которые нам сейчас неинтересны).

Указатель на число, например `n4`, может состоять из метки, указывающей, что это число, и собственно представления числа 4[9]. Для того, чтобы работать с числами, не умещающимися в ограниченном объеме, отводимом под указатель, может иметься особый тип данных *большое число* (bignum), для которого указатель обозначает список, где хранятся части числа[10].

Символ можно представить как типизированный указатель, обозначающий последо-

[8]Информация о типе может быть представлена различными способами, в зависимости от деталей машины, на которой реализована Лисп-система. Эффективность выполнения Лисп-программ будет сильно зависеть от того, насколько разумно сделан этот выбор, однако правила проектирования, определяющие, какой выбор хорош, сформулировать трудно. Самый простой способ реализации типизированных указателей состоит в том, чтобы в каждом указателе выделить несколько бит как *поле типа* (type field) (или *метку, тег типа* (type tag)) которое кодирует тип. При этом требуется решить следующие важные вопросы: сколько требуется битов для поля типа? Как велики должны быть индексы векторов? Насколько эффективно можно использовать элементарные команды машины для работы с полями типа в указателях? Про машины, в которых имеется специальная аппаратура для эффективной обработки полей типа, говорят, что они обладают *теговой архитектурой* (tagged architecture).

[9]Решение о представлении чисел определяет, можно ли сравнивать числа через `eq?`, который проверяет одинаковость указателей. Если указатель содержит само число, то равные числа будут иметь одинаковые указатели. Однако если в указателе содержится индекс ячейки, в которой хранится само число, то у равных чисел будут одинаковые указатели только в том случае, если нам удастся никогда не хранить одно и то же число в двух местах.

[10]Это представление очень похоже на запись числа в виде последовательности цифр, только каждая «цифра» является числом между 0 и максимальным значением, которое можно уместить в указателе.

вательность знаков, из которых состоит печатное представление символа. Эта последовательность создается процедурой чтения Лиспа, когда строка-представление в первый раз встречается при вводе. Поскольку мы хотим, чтобы два экземпляра символа всегда были «одинаковы» в смысле `eq?`, а `eq?` должно быть простым сравнением указателей, нам нужно обеспечить, чтобы процедура чтения, когда она видит ту же строку второй раз, использовала тот же самый указатель (к той же последовательности знаков) для представления обоих вхождений символа. Ради этого процедура чтения содержит таблицу, которая по традиции называется *обмассив* (obarray), и в ней хранит все когда-либо встреченные символы. Когда процедура видит строку и готова создать символ, она проверяет в обмассиве, не было ли уже ранее такой же строки. Если нет, она строит новый символ со встретившейся строкой в качестве имени (типизированный указатель на последовательность знаков) и включает его в обмассив. Если же процедура уже встречала указанную строку, она возвращает указатель на символ, хранимый в обмассиве. Процесс замены строк печатных знаков указателями без повторения называется *восприятием* (interning) символов.

Реализация элементарных списковых операций

Имея вышеописанную схему представления, можно заменить каждую «элементарную» списковую операцию регистровой машины одной или более элементарной векторной операцией. Мы будем обозначать векторы памяти двумя регистрами `the-cars` и `the-cdrs`, и предположим, что операции `vector-ref` и `vector-set!` даны как элементарные. Кроме того, предположим, что численные операции с указателями (добавление единицы, индексирование вектора с помощью указателя на пару и сложение чисел) используют только индексную часть типизированного указателя.

Например, можно заставить регистровую машину поддерживать команды

```
(assign ⟨рег1⟩ (op car) (reg ⟨рег2⟩))

(assign ⟨рег1⟩ (op cdr) (reg ⟨рег2⟩))
```

если мы реализуем их, соответственно, как

```
(assign ⟨рег1⟩ (op vector-ref) (reg the-cars) (reg ⟨рег2⟩))

(assign ⟨рег1⟩ (op vector-ref) (reg the-cdrs) (reg ⟨рег2⟩))
```

Команды

```
(perform (op set-car!) (reg ⟨рег1⟩) (reg ⟨рег⟨2⟩⟩))

(perform (op set-cdr!) (reg ⟨рег1⟩) (reg ⟨рег⟨2⟩⟩))
```

реализуются как

```
(perform
 (op vector-set!) (reg the-cars) (reg ⟨рег1⟩) (reg ⟨рег2⟩))

(perform
 (op vector-set!) (reg the-cdrs) (reg ⟨рег1⟩) (reg ⟨рег2⟩))
```

При выполнении `cons` выделяется неиспользуемый индекс, и аргументы `cons` записываются в `the-cars` и `the-cdrs` в ячейках с выделенным индексом. Мы предполагаем, что имеется особый регистр `free`, в котором всегда содержится указатель на следующую свободную пару, и что мы можем его увеличить и получить следующую свободную ячейку[11]. Например, команда

```
(assign ⟨рег1⟩ (op cons) (reg ⟨рег2⟩) (reg ⟨рег 3⟩))
```

реализуется как следующая последовательность векторных операций[12]:

```
(perform
 (op vector-set!) (reg the-cars) (reg free) (reg ⟨рег2⟩))
(perform
 (op vector-set!) (reg the-cdrs) (reg free) (reg ⟨рег3⟩))
(assign (reg ⟨рег1⟩) (reg free))
(assign free (op +) (reg free) (const 1))
```

Операция `eq?`

```
(op eq?) (reg ⟨рег1⟩) (reg ⟨рег2⟩)
```

попросту проверяет равенство всех полей регистров, а предикаты вроде `pair?`, `null?`, `symbol?` и `number?` смотрят только на поле типа.

Реализация стеков

Хотя наши регистровые машины используют стеки, нам ничего специально здесь делать не надо, поскольку стеки можно смоделировать на основе списков. Стек может быть списком сохраненных значений, на которые указывает особый регистр `the-stack`. Таким образом, `(save ⟨рег⟩)` может реализовываться как

```
(assign the-stack (op cons) (reg ⟨рег⟩) (reg the-stack))
```

Подобным образом, `(restore ⟨рег⟩)` можно реализовать как

```
(assign ⟨рег⟩ (op car) (reg the-stack))
(assign the-stack (op cdr) (reg the-stack))
```

а `(perform (op initialize-stack))` реализуется как

```
(assign the-stack (const ()))
```

Все эти операции можно далее расшифровать в терминах векторных операций, как показано выше. Однако в традиционных компьютерных архитектурах обычно удобно выделять стек в виде отдельного вектора. В таком случае сохранение на стеке и снятие с него реализуются через увеличение и уменьшение индекса, указывающего в этот вектор.

[11] Имеются и другие способы поиска свободной памяти. Например, можно было бы связать все неиспользуемые пары в *список свободных ячеек* (free list). Наши свободные ячейки идут подряд, поскольку мы пользуемся сжимающим сборщиком мусора, как будет описано в разделе 5.3.2.

[12] В сущности, это реализация `cons` через `set-car!` и `set-cdr!`, как описано в разделе 3.3.1. Операция `get-new-pair`, которая там используется, здесь реализуется через указатель `free`.

Упражнение 5.20.
Нарисуйте стрелочную диаграмму и представление в виде вектора (как на рисунке 5.14) списка, который порождается кодом

```
(define x (cons 1 2))
(define y (list x x))
```

если вначале указатель `free` равен `p1`. Чему равно значение `free` после исполнения кода? Какие указатели представляют значения `x` и `y`?

Упражнение 5.21.
Реализуйте регистровые машины для следующих процедур. Считайте, что операции с памятью, реализующие списковую структуру, имеются в машине как примитивы.

а. Рекурсивная `count-leaves`:

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                 (count-leaves (cdr tree))))))
```

б. Рекурсивная `count-leaves` с явным счетчиком:

```
(define (count-leaves tree)
  (define (count-iter tree n)
    (cond ((null? tree) n)
          ((not (pair? tree)) (+ n 1))
          (else (count-iter (cdr tree)
                            (count-iter (car tree) n)))))
  (count-iter tree 0))
```

Упражнение 5.22.
В упражнении 3.12 из раздела 3.3.1 были представлены процедура `append`, которая добавляет два списка друг к другу и получает третий, и процедура `append!`, которая склеивает два списка вместе. Спроектируйте регистровые машины, которые реализуют каждую из этих процедур. Предполагайте, что операции с памятью, реализующие списковую структуру, являются примитивами.

5.3.2. Иллюзия бесконечной памяти

Метод представления, намеченный в разделе 5.3.1, решает задачу реализации списковой структуры при условии, что у нас бесконечное количество памяти. В настоящем компьютере у нас в конце концов кончится свободное место, где можно строить новые пары[13]. Однако большинство пар, порождаемых во время типичного вычисления,

[13]На самом деле и это может быть неправдой, поскольку размеры памяти могут стать настолько большими, что свободная память просто не успеет исчерпаться за время жизни компьютера. Например, в году примерно 3×10^{13} секунд, так что если мы будем вызывать `cons` один раз в микросекунду, и у нас будет примерно 10^{15} ячеек памяти, то мы построим машину, которая сможет работать 30 лет, пока память не кончится. По теперешним понятиям такое количество памяти кажется абсурдно большим, однако физически оно вполне возможно. С другой стороны, процессоры становятся быстрее, и может быть, что в компьютерах будущего будет по многу процессоров, работающих параллельно в единой памяти, так что память можно будет расходовать намного быстрее, чем мы сейчас предполагаем.

используются только для хранения промежуточных результатов. После того, как эти результаты обработаны, пары больше не нужны — они становятся *мусором* (garbage). Например, при выполнении

```
(accumulate + 0 (filter odd? (enumerate-interval 0 n)))
```

создается два списка: перечисление и результат его фильтрации. После того, как проводится накопление, эти списки больше не нужны, и выделенную под них память можно освободить. Если нам удастся периодически собирать весь мусор, и память будет таким образом освобождаться приблизительно с той же скоростью, с которой мы строим новые пары, мы сможем поддерживать иллюзию, что у нас бесконечное количество памяти.

Для того, чтобы освобождать пары, нужен способ определять, какие из выделенных пар больше не нужны (в том смысле, что их содержимое не может уже повлиять на будущее вычисления). Метод, с помощью которого мы этого добиваемся, называется *сборка мусора* (garbage collection). Сборка мусора основана на наблюдении, что в каждый момент при интерпретации Лисп-программы на будущую судьбу вычисления могут повлиять только те объекты, до которых можно добраться с помощью какой-нибудь последовательности операций `car` и `cdr`, начиная с указателей, хранимых в этот момент в регистрах машины[14]. Любую ячейку памяти, до которой так добраться нельзя, можно освободить.

Есть множество способов сборки мусора. Метод, который мы опишем здесь, называется *остановка с копированием* (stop-and-copy). Основная идея состоит в том, чтобы разделить память на две половины: «рабочую память» и «свободную память». Когда `cons` строит пары, он это делает в рабочей памяти. Когда рабочая память заполняется, проводится сборка мусора: мы отыскиваем все используемые пары в рабочей памяти и копируем их в последовательные ячейки свободной памяти. (Используемые пары ищутся просмотром всех указателей `car` и `cdr`, начиная с машинных регистров.) Поскольку мусор мы не копируем, предполагается, что при этом появится дополнительная свободная память, где можно выделять новые пары. Кроме того, в рабочей памяти не осталось ничего нужного, поскольку все полезные пары из нее скопированы в свободную память. Таким образом, если мы поменяем роли рабочей и свободной памяти, мы можем продолжить работу; новые пары будут выделяться в новой рабочей памяти (бывшей свободной). Когда она заполнится, мы можем скопировать используемые пары из нее в новую свободную память (старую рабочую)[15].

---

[14]Здесь мы предполагаем, что стек представляется в виде списка, как описано в разделе 5.3.1, так что элементы стека доступны через указатель, хранящейся в стековом регистре.

[15]Эта идея была придумана и впервые реализована Минским, как часть реализации Лиспа для машины PDP-1 в Исследовательской лаборатории Электроники в MIT. Ее дополнили Феничель и Йохельсон (Fenichel and Yochelson 1969) для реализации Лиспа в системе разделения времени Multics. Позднее Бейкер (Baker 1978) разработал версию для «реального времени», в которой не требуется останавливать вычисления на время сборки. Идею Бейкера расширили Хьюитт, Либерман и Мун (Lieberman and Hewitt 1983), использовав то, что часть структуры более подвижна, а часть более постоянна.

Второй часто используемый метод сборки мусора — это *пометка с очисткой* (mark-sweep). Он состоит в том, что все структуры, доступные из машинных регистров, просматриваются и помечаются. Затем вся память просматривается, и всякий непомеченный участок «вычищается» как мусор и объявляется свободным. Полное обсуждение метода пометки с очисткой можно найти в Allen 1978.

В системах с большой памятью, как правило, используется алгоритм Минского-Феничеля-Йохельсона, поскольку он просматривает только используемую часть памяти. Напротив, при методе пометки с очисткой фаза

Реализация сборщика мусора методом остановки с копированием

Теперь мы можем с помощью языка регистровых машин описать алгоритм остановки с копированием более подробно. Предположим, что имеется регистр `root`, и в нем хранится указатель на корневой объект — структуру, которая (через перенаправления) указывает на все доступные данные. Такой конфигурации можно добиться, если переместить содержимое всех регистров машины в заранее выделенный список, на который и будет указывать `root`, непосредственно перед запуском сборщика мусора[16]. Кроме того, мы предполагаем, что помимо текущей рабочей памяти имеется свободная память, в которую мы можем копировать используемые данные. Текущая рабочая память состоит из векторов, базовые адреса которых хранятся в регистрах `the-cars` и `the-cdrs`, а свободная память доступна через регистры `new-cars` и `new-cdrs`.

Сборка мусора запускается, когда кончаются свободные ячейки в текущей рабочей памяти, то есть когда операция `cons` пытается сдвинуть указатель `free` за пределы вектора памяти. По завершении сборки мусора указатель `root` будет указывать на новую память, все объекты, доступные через `root`, будут перемещены в новую память, а указатель `free` будет указывать на место в новой памяти, начиная с которого можно выделять новые пары. Кроме того, поменяются местами роли рабочей памяти и свободной памяти — новые пары будут выделяться в новой памяти, начиная с места, на которое показывает `free`, а (предыдущая) рабочая память будет доступна в качестве новой памяти для следующей сборки мусора. На рисунке 5.15 показано устройство памяти непосредственно перед сборкой мусора и сразу после нее.

Состояние процесса сборки мусора управляется содержимым двух регистров: `free` и `scan`. Оба они сначала указывают на начало новой памяти. При запуске алгоритма пара, на которую указывает `root`, переносится в начало новой памяти. Пара копируется, регистр `root` переставляется в новое место, а указатель `free` увеличивается на единицу. В дополнение к этому в старом месте, где хранилась пара, делается пометка, которая говорит, что содержимое перенесено. Отметка делается так: в позицию `car` мы помещаем особое значение, показывающее, что объект перемещен. (По традиции, такой объект называется *разбитое сердце* (broken heart).)[17] В позицию `cdr` мы помещаем *перенаправляющий адрес* (forwarding address), который указывает на место, куда перемещен объект.

После того, как перемещен корневой объект, сборщик мусора входит в основной цикл. На каждом шаге алгоритма регистр `scan` (вначале он указывает на перемещенный корневой объект) содержит адрес пары, которая сама перемещена в новую память, но `car` и `cdr` которой по-прежнему указывают на объекты в старой памяти. Каждый из этих объектов перемещается, а затем регистр `scan` увеличивается на единицу. При перемещении объекта (например, того, на который указывает `car` сканируемой пары) мы прежде всего проверяем, не перемещен ли он уже (об этом нам говорит разбитое

очистки должна проверять всю память. Второе преимущество остановки с копированием состоит в том, что это *сжимающий* (compacting) сборщик мусора. Это означает, что в конце фазы сборки мусора все используемые данные оказываются в последовательной области памяти, а весь мусор «выжат». На машинах с виртуальной памятью это может давать весьма большой выигрыш в производительности, поскольку доступ к сильно различающимся адресам в памяти может приводить к дополнительной подкачке страниц.

[16] Этот список регистров не включает в себя регистры, которые используются подсистемой выделения памяти — `root`, `the-cars`, `the-cdrs` и еще несколько регистров, которые будут введены в этом разделе.

[17] Термин *разбитое сердце* придумал Дэвид Кресси, когда он писал сборщик мусора для MDL, диалекта Лиспа, разработанного в MIT в начале 70-х годов.

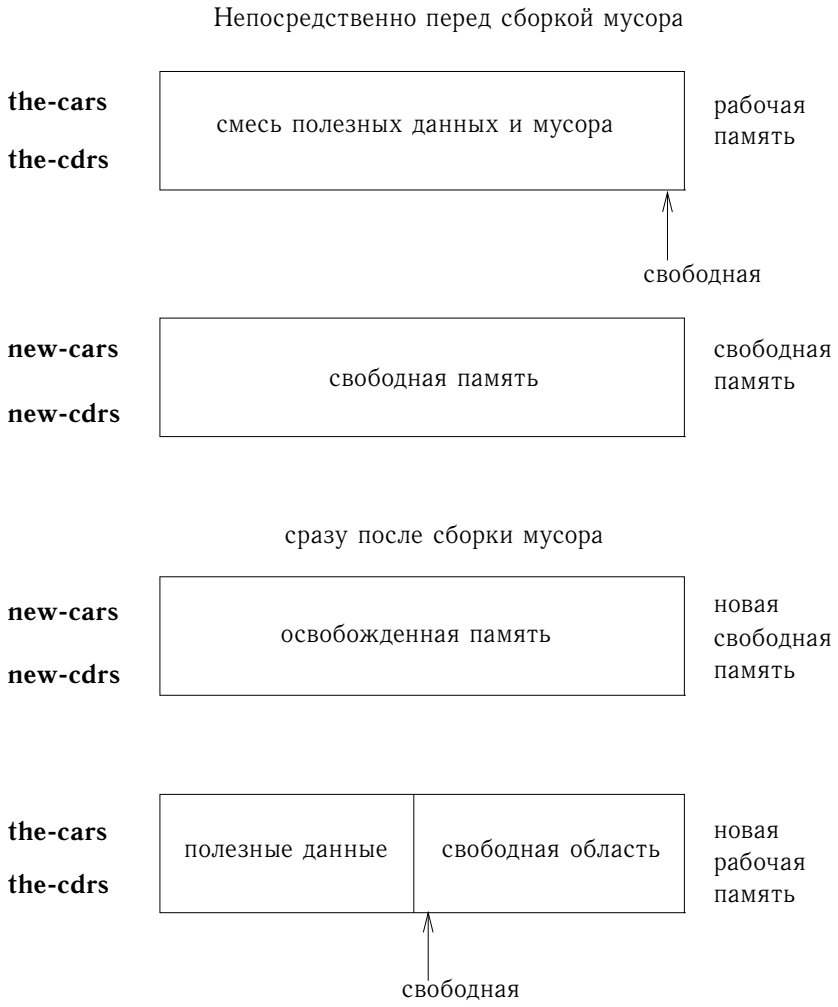


Рис. 5.15. Перестройка памяти в процессе сборки мусора.

сердце в позиции car объекта). Если объект еще не перемещен, мы переносим его в место, на которое указывает free, увеличиваем free, записываем разбитое сердце по старому местоположению объекта, и обновляем указатель на объект (в нашем случае, car пары, которую мы сканируем) так, чтобы он показывал на новое место. Если же объект уже был перемещен, то его перенаправляющий адрес (он находится в позиции cdr разбитого сердца) подставляется на место указателя в сканируемой паре. В конце концов все доступные объекты окажутся перемещены и просканированы. В этот момент указатель scan догонит указатель free, и процесс завершится.

Алгоритм остановки с копированием можно описать как последовательность команд регистровой машины. Базовый шаг, состоящий в перемещении объекта, проделывается подпрограммой relocate-old-result-in-new. Эта подпрограмма принимает свой аргумент, указатель на объект, подлежащий перемещению, в регистре old. Она перемещает данный объект (по пути обновляя free), помещает указатель на перемещенный объект в регистр new, и возвращается, переходя на точку входа, хранимую в регистре relocate-continue. В начале процесса сборки мы с помощью этой подпрограммы перемещаем указатель root, после инициализации free и scan. Когда root перемещен, мы записываем новый указатель в регистр root и входим в основной цикл сборщика мусора.

```
begin-garbage-collection
  (assign free (const 0))
  (assign scan (const 0))
  (assign old (reg root))
  (assign relocate-continue (label reassign-root))
  (goto (label relocate-old-result-in-new))
reassign-root
  (assign root (reg new))
  (goto (label gc-loop))
```

В основном цикле сборщика мусора нам нужно определить, есть ли еще подлежащие сканированию объекты. Для этого мы проверяем, совпадает ли указатель scan с указателем free. Если указатели равны, значит, все доступные объекты перемещены, и мы переходим на gc-flip. По этому адресу расположены восстановительные действия, после которых можно продолжить прерванные вычисления. Если же еще есть пары, которые требуется просканировать, мы зовем подпрограмму перемещения для car следующей пары (при вызове мы помещаем указатель car в регистр old). Регистр relocate-continue устанавливается таким образом, чтобы при возврате мы могли обновить указатель car.

```
gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip))
  (assign old (op vector-ref) (reg new-cars) (reg scan))
  (assign relocate-continue (label update-car))
  (goto (label relocate-old-result-in-new))
```

На метке update-car мы изменяем указатель car сканируемой пары. После этого мы перемещаем ее cdr, а затем возвращаемся на метку update-cdr. Наконец, когда

cdr перемещен и обновлен, сканирование пары закончено, и мы продолжаем главный цикл.

```
update-car
  (perform
   (op vector-set!) (reg new-cars) (reg scan) (reg new))
  (assign old (op vector-ref) (reg new-cdrs) (reg scan))
  (assign relocate-continue (label update-cdr))
  (goto (label relocate-old-result-in-new))

update-cdr
  (perform
   (op vector-set!) (reg new-cdrs) (reg scan) (reg new))
  (assign scan (op +) (reg scan) (const 1))
  (goto (label gc-loop))
```

Подпрограмма relocate-old-result-in-new перемещает объекты следующим образом: если перемещаемый объект (на него указывает old) не является парой, мы возвращаем указатель на него неизменным (в регистре new). (К примеру, мы можем сканировать пару, в car которой находится число 4. Если значение в car представляется как n4, как описано в разделе 5.3.1, то нам нужно, чтобы «перемещенный» car по-прежнему равнялся n4.) В противном случае требуется произвести перемещение. Если позиция car перемещаемой пары содержит метку разбитого сердца, значит, сама пара уже перенесена, и нам остается только взять перенаправляющий адрес (из позиции cdr разбитого сердца) и вернуть его в регистре new. Если же регистр old указывает на еще пе перенесенную пару, то мы ее переносим в первую пустую ячейку новой памяти (на нее указывает free), а затем строим разбитое сердце, записывая в старой ячейке метку разбитого сердца и перенаправляющий адрес. Процедура relocate-old-result-in-new хранит car или cdr объекта, на который указывает old, в регистре oldcr[18].

```
relocate-old-result-in-new
  (test (op pointer-to-pair?) (reg old))
  (branch (label pair))
  (assign new (reg old))
  (goto (reg relocate-continue))
pair
  (assign oldcr (op vector-ref) (reg the-cars) (reg old))
  (test (op broken-heart?) (reg oldcr))
  (branch (label already-moved))
  (assign new (reg free))  ; новое место для пары
   ;; Обновить указатель  free.
  (assign free (op +) (reg free) (const 1))
   ;; Скопировать  car и  cdr~в новую память.
  (perform (op vector-set!)
```

[18]Сборщик мусора использует низкоуровневый предикат pointer-to-pair? вместо обыкновенного pair?, поскольку в настоящей системе могут иметься различные объекты, которые с точки зрения сборщика будут являться парами. Например, в системе, которая соответствует стандарту Scheme IEEE, объект-процедура может реализовываться особого рода «парой», которая не удовлетворяет предикату pair?. В нашем имитаторе можно реализовать pointer-to-pair? как pair?.

```
          (reg new-cars) (reg new) (reg oldcr))
  (assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
  (perform (op vector-set!)
           (reg new-cdrs) (reg new) (reg oldcr))
   ;; Построить разбитое сердце.
  (perform (op vector-set!)
           (reg the-cars) (reg old) (const broken-heart))
  (perform
   (op vector-set!) (reg the-cdrs) (reg old) (reg new))
  (goto (reg relocate-continue))
already-moved
  (assign new (op vector-ref) (reg the-cdrs) (reg old))
  (goto (reg relocate-continue))
```

В самом конце процесса сборки мусора мы меняем ролями старую и новую память, обменивая указатели: `cars` меняется с `new-cars`, а `cdrs` с `new-cdrs`. Теперь мы готовы запустить следующую сборку, когда память опять кончится.

```
gc-flip
  (assign temp (reg the-cdrs))
  (assign the-cdrs (reg new-cdrs))
  (assign new-cdrs (reg temp))
  (assign temp (reg the-cars))
  (assign the-cars (reg new-cars))
  (assign new-cars (reg temp))
```

5.4. Вычислитель с явным управлением

В разделе 5.1 мы видели, как простые программы на Scheme можно преобразовывать в описания регистровых машин. Теперь мы проделаем такое преобразование с более сложной программой — метациклическим интерпретатором из разделов 4.1.1–4.1.4, который показал нам, как поведение процедур Scheme можно описать через процедуры `eval` и `apply`. *Вычислитель с явным управлением* (explicit-control evaluator), который мы разработаем в этом разделе, демонстрирует, как механизмы вызова процедур и передачи параметров, используемые в процессе вычисления, могут быть описаны в терминах действий, производимых над регистрами и стеками. В дополнение к этому вычислитель с явным управлением может служить реализацией интерпретатора Scheme, написанной на языке, весьма близком к машинному языку обыкновенных компьютеров. Этот вычислитель можно выполнить на имитаторе регистровых машин из раздела 5.2. С другой стороны, его можно использовать как основу для построения вычислителя Scheme, написанного на машинном языке, или даже специализированной машины для вычисления выражений на Scheme. На рисунке 5.16 показана такая аппаратная реализация: кремниевый чип, который работает как вычислитель Scheme. Проектировщики чипа начали с описания путей данных и контроллера, подобного вычислителю из этого раздела, а затем с помощью программ автоматизированного проектирования построили разводку интегральной микросхемы[19].

[19] Информацию о микросхеме и методе ее проектирования можно найти в Batali et al. 1982.

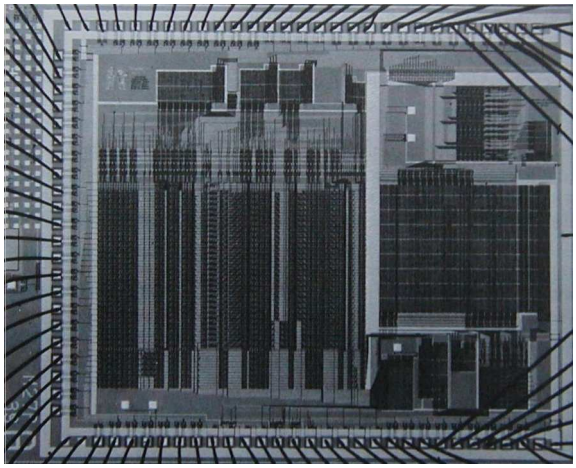


Рис. 5.16. Реализация вычислителя для Scheme в виде кремниевой микросхемы.

Регистры и операции

При проектировании вычислителя с явным управлением требуется указать операции, которые будут использоваться в нашей регистровой машине. Метациклический интерпретатор мы описали, используя абстрактный синтаксис, с процедурами вроде `quoted?` и `make-procedure`. При реализации регистровой машины мы могли бы развернуть эти процедуры в последовательности элементарных операций с памятью, работающих со списковой структурой, и уже эти операции реализовать в нашей регистровой машине. Однако при этом вычислитель получился бы слишком длинным, а его структура была бы загромождена мелкими деталями. Для большей ясности представления мы будем считать элементарными операциями регистровой машины синтаксические процедуры из раздела 4.1.2, а также процедуры для представления окружений и прочих данных интерпретатора из разделов 4.1.3 и 4.1.4. Для того, чтобы полностью описать вычислитель, который можно было бы запрограммировать на машинном языке низкого уровня или реализовать аппаратно, пришлось бы заменить эти операции более примитивными на основе реализации списковой структуры, которую мы описали в разделе 5.3.

Наша регистровая машина-вычислитель Scheme имеет стек и семь регистров: `exp`, `env`, `val`, `continue`, `proc`, `argl` и `unev`. В регистре `exp` хранится выражение, подлежащее вычислению, а в регистре `env` окружение, в котором нужно это вычисление произвести. В конце вычисления `val` содержит значение, полученное при вычислении выражения в указанном окружении. Регистр `continue` используется для реализации рекурсии, как описано в разделе 5.1.4. (Вычислитель вызывает сам себя рекурсивно, поскольку вычисление выражения требует вычисления его подвыражений.) Регистры `proc`, `argl` и `unev` используются при работе с комбинациями.

Мы не будем ни рисовать диаграмму путей данных, показывающую, как связаны между собой регистры и операции, ни давать полный список машинных операций. Эти данные можно получить из текста контроллера, который будет представлен полностью.

5.4.1. Ядро вычислителя с явным управлением

Центральным элементом вычислителя является последовательность команд, расположенная по метке `eval-dispatch`. Она соответствует процедуре `eval` метациклического интерпретатора из раздела 4.1.1. Начиная с `eval-dispatch`, контроллер вычисляет выражение, хранящееся в `exp`, в окружении, которое содержится в `env`. Когда вычисление заканчивается, контроллер переходит на точку входа, которая хранится в `continue`, а значение выражения при этом находится в `val`. Как и в метациклическом `eval`, структура `eval-dispatch` представляет собой анализ случаев по синтаксическому типу выполняемого выражения[20].

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

Вычисление простых выражений

В числах и строках (значением которых являются они сами), переменных, закавыченных выражениях и выражениях `lambda` не содержится подлежащих вычислению подвыражений. В этих случаях вычислитель попросту помещает нужное значение в регистр `val` и продолжает работу с точки, указанной в регистре `continue`. Вычисление простых выражений производится следующим кодом контроллера:

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
```

[20] В нашем контроллере анализ случаев написан как последовательность команд `test` и `branch`. Можно было бы написать его и в стиле программирования, управляемого данными (в реальной системе так, скорее всего, и было бы сделано). При этом исчезла бы необходимость проводить последовательные проверки, и легче было бы определять новые типы выражений. Машина, рассчитанная на выполнение Лисп-программ, скорее всего, имела бы команду `dispatch-on-type`, которая бы эффективно выполняла выбор, управляемый данными.

```
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
              (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

Обратите внимание, как `ev-lambda` пользуется регистрами `unev` и `exp` для параметров и тела лямбда-выражения, которые наряду с окружением в регистре `env` требуется передать операции `make-procedure`.

Вычисление вызовов процедур

Вызов процедуры описывается комбинацией, состоящей из оператора и операндов. Оператор — подвыражение, значением которого является процедура, а операнды — подвыражения, значения которых являются аргументами, к которым процедуру следует применить. Метациклический `eval` при обработке вызовов зовет себя рекурсивно и вычисляет таким образом все элементы комбинации, а затем передает результаты в `apply`, которая и осуществляет собственно применение процедуры. Вычислитель с явным управлением ведет себя точно так же; рекурсивные вызовы осуществляются командами `goto`, и при этом на стеке сохраняются регистры, значения которых нужно восстановить после возврата из рекурсивного вызова. Перед каждым вызовом мы тщательно смотрим, какие именно регистры требуется сохранить (поскольку их значения потребуются позже)[21].

В начале обработки процедурного вызова мы вычисляем оператор и получаем процедуру, которую позже надо будет применить к вычисленным операндам. Для того, чтобы вычислить оператор, мы переносим его в регистр `exp` и переходим на `eval-dispatch`. В регистре `env` уже находится то окружение, в котором оператор требуется вычислить. Однако мы сохраняем `env`, поскольку его значение нам еще потребуется для вычисления операндов. Кроме того, мы переносим операнды в регистр `unev` и сохраняем его на стеке. Регистр `continue` мы устанавливаем так, чтобы после вычисления оператора работа продолжилась с `ev-appl-did-operator`. Однако перед этим мы сохраняем старое значение `continue`, поскольку оно говорит нам, куда требуется перейти после вычисления вызова.

[21] Это важная, но сложная деталь при переводе алгоритмов из процедурного языка типа Лиспа на язык регистровой машины. В качестве альтернативы сохранению только нужных регистров мы могли бы сохранять их все (кроме `val`) перед каждым рекурсивным вызовом. Такая тактика называется *дисциплиной кадрированного стека* (framed-stack discipline). Она работает, но при этом сохраняется больше регистров, чем нужно; в системе, где стековые операции дороги, это может оказаться важным. Кроме того, сохранение регистров, с ненужными значениями может привести к удлиннению жизни бесполезных данных, которые в противном случае освободились бы при сборке мусора.

```
ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

После того, как вычислено значение подвыражения-оператора, мы вычисляем операнды комбинации и собираем их значения в список, хранимый в регистре `argl`. Прежде всего мы восстанавливаем невычисленные операнды и окружение. Мы заносим пустой список в `argl` как начальное значение. Затем заносим в регистр `proc` процедуру, порожденную при вычислении оператора. Если операндов нет, мы напрямую идем в `apply-dispatch`. В противном случае сохраняем `proc` на стеке и запускаем цикл вычисления операндов[22]:

```
ev-appl-did-operator
  (restore unev)                  ; операнды
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))         ; оператор
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

При каждом проходе цикла вычисления аргументов вычисляется один аргумент, и его значение добавляется к `argl`. Для того, чтобы вычислить операнд, мы помещаем его в регистр `exp` и переходим к `eval-dispatch`, установив предварительно `continue` так, чтобы вычисление продолжилось на фазе сбора аргументов. Но еще до этого мы сохраняем уже собранные аргументы (из `argl`), окружение (из `env`) и оставшиеся операнды, подлежащие вычислению (из `unev`). Вычисление последнего операнда считается особым случаем и обрабатывается кодом по метке `ev-appl-last-arg`.

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
```

[22] К процедурам работы со структурой данных вычислителя из раздела 4.1.3 мы добавляем следующие процедуры для работы со списками аргументов:

```
(define (empty-arglist) '())
(define (adjoin-arg arg arglist)
  (append arglist (list arg)))
```

Кроме того, мы проверяем, является ли аргумент в комбинации последним, при помощи дополнительной синтаксической процедуры:

```
(define (last-operand? ops)
  (null? (cdr ops)))
```

```
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

После того, как аргумент вычислен, его значение подсоединяется в конец списка, который хранится в `argl`. Затем операнд убирается из списка невычисленных операндов в `unev`, и продолжается цикл вычисления аргументов.

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))
```

Последний аргумент обрабатывается отдельно. Здесь нет необходимости сохранять окружение и список невычисленных операндов перед переходом в `eval-dispatch`, поскольку после вычисления последнего операнда они не понадобятся. Поэтому после вычисления мы возвращаемся к метке `ev-appl-accum-last-arg`, где восстанавливается список аргументов, на него наращивается новый (последний) аргумент, восстанавливается сохраненная процедура, и, наконец, происходит переход к применению процедуры[23]:

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
```

Детали цикла вычисления аргументов определяют порядок, в котором интерпретатор вычисляет операнды комбинации (то есть справа налево или слева направо — см. упражнение 3.8). Этот порядок оставался неопределенным в метациклическом интерпретаторе, который наследует свою управляющую структуру из нижележащей Scheme, на которой он написан[24]. Поскольку селектор `first-operand` (который используется в `ev-appl-operand-loop` для последовательного извлечения операндов из `unev`) реализован как `car`, а селектор `rest-operands` реализуется как `cdr`, вычислитель с явным управлением будет вычислять операнды комбинации в порядке слева направо.

[23]Оптимизация при обработке последнего аргумента известна как *хвостовая рекурсия в списке аргументов* (evlis tail recursion) (см. Wand 1980). Можно было бы особым образом обрабатывать еще и первый аргумент и получить некоторую дополнительную выгоду. Это позволило бы отложить инициализацию `argl` до того времени, когда будет вычислен первый операнд, и избежать в этом случае сохранения `argl`. Компилятор в разделе 5.5 проделывает эту оптимизацию. (Сравните с процедурой `construct-arglist` из раздела 5.5.3.)

[24]Порядок вычисления операндов в метациклическом интерпретаторе определяется порядком вычисления аргументов `cons` в процедуре `list-of-values` из раздела 4.1.1 (см. упражнение 4.1).

Применение процедур

Точка входа apply-dispatch соответствует процедуре apply метациклического интерпретатора. К тому времени, когда мы попадаем в apply-dispatch, в регистре proc содержится подлежащая применению процедура, а в регистре argl список вычисленных аргументов, к которым ее требуется применить. Сохраненное значение continue (исходно оно передается подпрограмме eval-dispatch, а затем сохраняется внутри ev-application), которое определяет, куда нам надо вернуться с результатом применения процедуры, находится на стеке. Когда вызов вычислен, контроллер передает управление в точку входа, указанную в сохраненном continue, а результат при этом хранится в val. Подобно метациклическому apply, нужно рассмотреть два случая. Либо применяемая процедура является примитивом, либо это составная процедура.

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

Мы предполагаем, что все примитивы реализованы так, что они ожидают аргументы в регистре argl, а результат возвращают в val. Чтобы описать, как машина обрабатывает примитивы, нам пришлось бы дать последовательность команд, которая реализует каждый примитив, и заставить primitive-apply переходить к этим командам в зависимости от содержимого proc. Поскольку нас интересуют не детали примитивов, а структура процесса вычисления, мы вместо этого будем просто использовать операцию apply-primitive-procedure, которая применяет процедуру, содержащуюся в proc, к аргументам, содержащимся в argl. Чтобы смоделировать вычислитель имитатором из раздела 5.2, мы используем процедуру apply-primitiveprocedure, которая исполняет процедуру с помощью нижележащей Scheme-системы, как мы это делали и в метациклическом интерпретаторе из раздела 4.1.4. После того, как элементарная процедура вычислена, мы восстанавливаем регистр continue и переходим на указанную точку входа.

```
primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
  (restore continue)
  (goto (reg continue))
```

Чтобы применить составную процедуру, мы действуем так же, как и в метациклическом интерпретаторе. Мы строим кадр, который связывает параметры процедуры с ее аргументами, расширяем этим кадром окружение, хранимое в процедуре, и в этом расширенном окружении вычисляем последовательность выражений, которая представляет собой тело процедуры. Подпрограмма ev-sequence, описанная ниже в разделе 5.4.2, проводит вычисление последовательности.

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
```

```
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

Подпрограмма compound-apply — единственное место в интерпретаторе, где регистру env присваивается новое значение. Подобно метациклическому интерпретатору, новое окружение строится из окружения, хранимого в процедуре, а также списка аргументов и соответствующего ему списка связываемых переменных.

5.4.2. Вычисление последовательностей и хвостовая рекурсия

Сегмент кода вычислителя с явным управлением, начинающийся с метки ev-sequence, соответствует процедуре eval-sequence метациклического интерпретатора. Он обрабатывает последовательности выражений в телах процедур, а также явные выражения begin.

Явные выражения begin обрабатываются так: последовательность подлежащих выполнению выражений помещается в unev, регистр continue сохраняется на стеке, а затем происходит переход на ev-sequence.

```
ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))
```

Неявные последовательности в телах процедур обрабатываются переходом на ev-sequence из compound-apply, причем continue в этот момент уже находится на стеке, поскольку он был сохранен в ev-application.

Метки ev-sequence и ev-sequence-continue образуют цикл, который по очереди вычисляет все выражения в последовательности. Список невычисленных пока выражений хранится в unev. Прежде, чем вычислять каждое выражение, мы смотрим, есть ли в последовательности после него еще выражения, подлежащие вычислению. Если да, то мы сохраняем список невычисленных выражений (из регистра unev) и окружение, в котором их надо вычислить (из env), а затем вызываем eval-dispatch, чтобы вычислить выражение. После того, как вычисление закончено, два сохраненных регистра восстанавливаются на метке ev-sequence-continue.

Последнее выражение в последовательности обрабатывается особым образом, на метке ev-sequence-last-exp. Поскольку после этого выражения никаких других вычислять не требуется, не нужно и сохранять unev и env перед переходом на eval-dispatch. Значение всей последовательности равно значению последнего выражения, так что после вычисления последнего выражения требуется только продолжить вычисление по адресу, который хранится на стеке (помещенный туда из ev-application или ev-begin). Мы не устанавливаем continue так, чтобы вернуться в текущее место, восстановить continue со стека и продолжить с хранящейся в нем точки входа, а восстанавливаем continue со стека перед переходом на eval-dispatch, так что после вычисления выражения eval-dispatch передаст управление по этому адресу.

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

Хвостовая рекурсия

В главе 1 мы говорили, что процесс, который описывается процедурой вроде

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

итеративен. Несмотря на то, что синтаксически процедура рекурсивна (определена через саму себя), вычислителю нет логической необходимости сохранять информацию при переходе от одного вызова `sqrt-iter` к другому[25]. Про вычислитель, который способен вычислить процедуру типа `sqrt-iter`, не требуя дополнительной памяти по мере ее рекурсивных вызовов, говорят, что он обладает свойством *хвостовой рекурсии* (tail recursion). Метациклическая реализация вычислителя из главы 4 не указывает, обладает ли вычислитель хвостовой рекурсией, поскольку он наследует механизм сохранения состояния из нижележащей Scheme. Однако в случае вычислителя с явным управлением мы можем проследить процесс вычисления и увидеть, когда вызовы процедур приводят к росту информации на стеке.

Наш вычислитель обладает хвостовой рекурсией, поскольку при вычислении последнего выражения последовательности мы напрямую переходим к `eval-dispatch`, никакую информацию не сохраняя на стеке. Следовательно, при вычислении последнего выражения последовательности — даже если это рекурсивный вызов (как в `sqrt-iter`, где выражение `if`, последнего выражения в теле процедуры, сводится к вызову `sqrt-iter`) — не происходит никакого роста глубины стека[26].

[25] В разделе 5.1 мы видели, как такие процессы можно реализовывать с помощью регистровой машины, не имеющей стека; состояние машины хранилось в ограниченном количестве регистров.

[26] Наша реализация хвостовой рекурсии в `ev-sequence` — вариант хорошо известного метода оптимизации, используемого во многих компиляторах. При компиляции процедуры, которая заканчивается вызовом процедуры, можно заменить вызов переходом на начало вызываемой процедуры. Встраивание такой стратегии в интерпретатор (как сделано в этом разделе) единым образом распространяет эту оптимизацию на весь язык.

Мы использовали тот факт, что сохранять информацию необязательно. Если бы мы до этого не додумались, можно было бы реализовать `eval-sequence` так, что все выражения в последовательности обрабатываются одинаково — сохранение регистров, вычисление выражения, возврат с сохранением регистров, и повторение до тех пор, пока не вычислятся все выражения[27].

```
ev-sequence
  (test (op no-more-exps?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))
```

Может показаться, что это мелкое изменение в предыдущем коде для выполнения последовательности: единственная разница состоит в том, что мы проходим через цикл сохранения-восстановления для последнего выражения последовательности так же, как и для остальных. Интерпретатор по-прежнему будет возвращать для всех выражений то же самое значение. Однако такое изменение теряет свойство хвостовой рекурсии, поскольку теперь после вычисления последнего выражения в последовательности нам придется возвращаться и отменять (бесполезные) сохранения регистров. Эти дополнительные сохранения будут накапливаться в гнезде рекурсивных вызовов. Следовательно, процессы вроде `sqrt-iter` потребуют памяти пропорционально количеству итераций, а не фиксированного объема. Такая разница может быть существенна. Например, при наличии хвостовой рекурсии можно выразить бесконечный цикл с помощью одного только механизма вызова процедур:

```
(define (count n)
  (newline)
  (display n)
  (count (+ n 1)))
```

Без хвостовой рекурсии такая процедура рано или поздно исчерпает место в стеке, а итерацию придется выражать с помощью какого-то другого механизма помимо вызовов процедур.

[27] Можно определить `no-more-exps?` как

```
(define (no-more-exps? seq) (null? seq))
```

5.4.3. Условные выражения, присваивания и определения

Как и в метациклическом интерпретаторе, особые формы обрабатываются путем частичного выполнения частей выражения. В выражении `if` нам нужно вычислить предикат, а затем на основании его значения решить, требуется нам выполнять следствие или альтернативу.

Прежде чем вычислять предикат, мы сохраняем само выражение `if`, поскольку позже из него потребуется извлекать следствие либо альтернативу. Кроме того, мы сохраняем окружение, которое потребуется при вычислении следствия или альтернативы, и `continue`, который потребуется нам при возврате значения выражению, ждущему результата `if`.

```
ev-if
  (save exp)                      ; сохраняем выражение
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))    ; вычисляем предикат
```

Вернувшись после вычисления предиката, мы смотрим, является ли его значение истиной или ложью, в зависимости от этого переносим в регистр `exp` следствие либо альтернативу, и идем на `eval-dispatch`. Заметим, что после восстановления `env` и `continue` `eval-dispatch` будет выполняться в правильном окружении и вернется после вычисления выражения в правильное место.

```
ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))
```

Присваивания и определения

Присваивания обрабатываются по метке `ev-assignment`, на которую переход происходит из `eval-dispatch` с выражением-присваиванием в регистре `exp`. Код `ev-assignment` сначала вычисляет значение присваиваемого выражения, а затем заносит это значение в окружение. Предполагается, что `set-variable-value!` дана как машинная операция.

```
ev-assignment
  (assign unev (op assignment-variable) (reg exp))
```

```
  (save unev)                    ; сохранить переменную
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))  ; вычислить присваиваемое значение
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Подобным образом обрабатываются и определения:

```
ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev)                     ; сохранить переменную
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))   ; вычислить значение переменной
ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Упражнение 5.23.
Расширьте вычислитель так, чтобы обрабатывались производные выражения `cond`, `let` и тому подобные (раздел 4.1.2). Можно «сжульничать» и считать, что синтаксические трансформации вроде `cond->if` имеются как машинные операции[28].

Упражнение 5.24.
Реализуйте `cond` как новую особую форму, не сводя его к `if`. Придется организовать цикл, проверяющий предикаты последовательных ветвей `cond`, пока один не окажется истинным, а затем с помощью `ev-sequence` выполнять действия этой ветви.

Упражнение 5.25.
Измените вычислитель так, чтобы он использовал нормальный порядок вычислений, на основе ленивого интерпретатора из раздела 4.2.

[28] На самом деле это не жульничество. В настоящей реализации, построенной с нуля, мы бы на синтаксическом этапе, происходящем раньше собственно выполнения, интерпретировали при помощи вычислителя с явным управлением Scheme-программу, которая производит трансформации исходного кода вроде `cond->if`.

5.4.4. Запуск вычислителя

Реализовав вычислитель с явным управлением, мы подходим к концу сюжета, начатого в главе 1 — построения все более точных моделей для процесса вычисления. Мы начали с относительно неформальной подстановочной модели, затем в главе 3 расширили ее до модели с окружениями, позволившей работать с состоянием и его изменением. В метациклическом интерпретаторе из главы 4 мы, используя как язык саму Scheme, сделали более явной структуру окружений, которые строятся при вычислении выражения. Теперь, рассмотрев регистровые машины, мы внимательнее исследовали механизмы вычислителя для работы с памятью, передачи аргументов и управления. На каждом новом уровне нам приходилось сталкиваться с вопросами и разрешать неясности, которые не были заметны при предыдущем, менее строгом описании вычислений. Для того, чтобы понять поведение вычислителя с явным управлением, мы можем построить его имитационную модель и рассмотреть ее работу.

Введем в нашу машину-вычислитель управляющий цикл. Он играет роль процедуры `driver-loop` из раздела 4.1.4. Вычислитель будет в цикле печатать подсказку, считывать выражение, выполнять его с помощью перехода на `eval-dispatch`, и печатать результат. Следующая группа команд стоит в начале последовательности команд контроллера в вычислителе с явным управлением[29].

```
read-eval-print-loop
  (perform (op initialize-stack))
  (perform
   (op prompt-for-input) (const ";;; Ввод EC-Eval:"))
  (assign exp (op read))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (label eval-dispatch))
print-result
  (perform
   (op announce-output) (const ";;; Значение EC-Eval:"))
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
```

Когда мы сталкиваемся с ошибкой (например, с ошибкой «неизвестный тип процедуры» из `apply-dispatch`), мы печатаем сообщение об ошибке и возвращаемся в управляющий цикл[30].

---

[29] Мы предполагаем, что `read` и различные операции печати имеются как элементарные машинные операции. Такое предположение разумно в целях имитации, но на практике совершенно нереалистично. Эти операции чрезвычайно сложны. На практике они реализовывались бы с помощью низкоуровневых операций ввода-вывода, например, посимвольного ввода и вывода печатных знаков на устройство.

Для поддержки операции `get-global-environment` мы определяем

```
(define the-global-environment (setup-environment))

(define (get-global-environment)
  the-global-environment)
```

[30] Хотелось бы обрабатывать и другие типы ошибок, но этого не так легко добиться. См. упражнение 5.30.

```
unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))
unknown-procedure-type
  (restore continue)     ; очистить стек (после  apply-dispatch)
  (assign val (const unknown-procedure-type-error))
  (goto (label signal-error))

signal-error
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
```

Для целей имитации мы каждый раз в начале прохождения управляющего цикла инициализируем стек, поскольку после того, как ошибка (например, неопределенная переменная) прерывает вычисление, он может не быть пуст[31].

Если мы соберем все фрагменты кода, представленные в разделах 5.4.1–5.4.4, то можно создать модель машины-вычислителя, которая запускается имитатором регистровых машин из раздела 5.2.

```
(define eceval
  (make-machine
   '(exp env val proc argl continue unev)
   eceval-operations
  '(
    read-eval-print-loop
      ⟨контроллер машины, как описано выше⟩
   )))
```

Требуется определить процедуры Scheme, имитирующие операции, которые считаются элементарными в вычислителе. Это те же операции, которые использовались в метациклическом интерпретаторе из раздела 4.1, а также несколько дополнительных, определенных в примечаниях к разделу 5.4.

```
(define eceval-operations
  (list (list 'self-evaluating? self-evaluating)
        ⟨полный список операций машины-вычислителя⟩))
```

Наконец, мы можем проинициализировать глобальное окружение и запустить вычислитель:

```
(define the-global-environment (setup-environment))

(start eceval)

;;; Ввод EC-Eval:
(define (append x y)
  (if (null? x)
```

[31] Можно было бы инициализировать стек только после ошибок, однако если мы это делаем в управляющем цикле, оказывается удобнее следить за производительностью вычислителя, как это описано ниже.

```
      y
      (cons (car x)
            (append (cdr x) y))))
;;; Значение EC-Eval:
ok

;;; Ввод EC-Eval:
(append '(a b c) '(d e f))
;;; Значение EC-Eval:
(a b c d e f)
```

Разумеется, вычисление выражений таким образом занимает намного больше времени, чем если их вводить напрямую в Scheme, по причине многослойной имитации. Наши выражения выполняются регистровой машиной-вычислителем с явным управлением, которая имитируется программой на Scheme, которая, в свою очередь, выполняется интерпретатором Scheme.

Отслеживание производительности вычислителя

Имитационное моделирование может служить мощным инструментом, помогающим в реализации вычислителей. С помощью имитации легко можно не только исследовать варианты построения регистровых машин, но и отслеживать производительность имитируемого вычислителя. Например, один из важных параметров производительности состоит в том, насколько эффективно вычислитель использует стек. Можно отследить количество стековых операций, которые требуются для вычисления различных выражений, если взять версию имитатора, которая собирает статистику по использованию стека (раздел 5.2.4) и добавить на точку входа `print-result` дополнительную команду, распечатывающую эту статистику:

```
print-result
  (perform (op print-stack-statistics))  ; добавленная команда
  (perform
   (op announce-output) (const ";;; Значение EC-Eval:"))
...  ; как и раньше
```

Теперь работа с вычислителем выглядит так:

```
;;; Ввод EC-Eval:
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
(total-pushes = 3 maximum-depth = 3)
;;; Значение EC-Eval:
ok

;;; Ввод EC-Eval:
(factorial 5)
(total-pushes = 144 maximum-depth = 28)
```

```
;;; Значение EC-Eval:
120
```

Заметим, что управляющий цикл вычислителя на каждом проходе заново инициализирует стек, так что печатаемая статистика будет относиться только к стековым операциям, произведенным при выполнении последнего выражения.

Упражнение 5.26.
С помощью отслеживания стека исследуйте хвостовую рекурсию в нашем вычислителе (раздел 5.4.2). Запустите вычислитель и определите итеративную процедуру `factorial` из раздела 1.2.1:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Запустите эту процедуру с несколькими небольшими значениями n. Для каждого из этих значений запишите максимальную глубину стека и количество операций сохранения, потребных для вычисления $n!$.

а. Вы увидите, что максимальная глубина стека, нужная для вычисления $n!$, от n не зависит. Какова эта глубина?

б. Составьте на основе своих данных формулу в зависимости от n числа операций сохранения, необходимых для вычисления $n!$ для любого $n \geq 1$. Обратите внимание, что число операций — линейная функция от n и, следовательно, определяется двумя константами.

Упражнение 5.27.
Для сравнения с упражнением 5.26, изучите поведение следующей процедуры для рекурсивного вычисления факториала:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

Запуская эту процедуру и отслеживая поведение стека, определите как функции от n максимальную глубину стека и общее число сохранений, требуемых для вычисления $n!$, при $n \geq 1$. (Эти функции также будут линейны.) Опишите общие результаты экспериментов, записав в следующую таблицу соответствующие выражения как формулы, зависящие от n:

	Максимальная глубина	Количество сохранений
Рекурсивный факториал		
Итеративный факториал		

Максимальная глубина служит мерой объема памяти, используемой вычислителем при обработке выражения, а количество сохранений хорошо коррелирует со временем вычисления.

Упражнение 5.28.
Измените в определении вычислителя `eval-sequence` так, как описано в разделе 5.4.2, чтобы вычислитель перестал обладать хвостовой рекурсией. Заново проведите эксперименты из упражнений 5.26 и 5.27 и покажите, что теперь обе версии процедуры `factorial` требуют количества памяти, которое линейно зависит от значения аргумента.

Упражнение 5.29.
Проследите за использованием стека в вычислении чисел Фибоначчи с помощью древовидной рекурсии:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

а. Дайте формулу, зависящую от n, для максимальной глубины стека, требуемой при вычислении $\mathrm{Fib}(n)$ при $n \geq 2$. Подсказка: в разделе 1.2.2 мы утверждали, что требуемый объем памяти линейно зависит от n.

б. Постройте формулу для количества сохранений, требуемых при вычислении $\mathrm{Fib}(n)$, если $n \geq 2$. Вы увидите, что количество сохранений (которое хорошо коррелирует со временем исполнения) экспоненциально растет с ростом n. Подсказка: пусть при вычислении $\mathrm{Fib}(n)$ требуется $S(n)$ сохранений. Нужно показать, что имеется формула, которая выражает $S(n)$ в зависимости от $S(n-1)$, $S(n-2)$ и некоторой фиксированной «дополнительной» константы k, независимой от n. Приведите эту формулу и найдите, чему равно `k`. Покажите теперь, что $S(n)$ выражается как $a\,\mathrm{Fib}(n+1) + b$ и укажите значения a и b.

Упражнение 5.30.
Наш вычислитель отлавливает только два вида ошибок (и сообщает о них) — неизвестные типы выражений и неизвестные типы процедур. При других ошибках он будет выпадать из управляющего цикла. Когда мы запускаем вычислитель с помощью имитатора регистровых машин, эти ошибки будут пойманы нижележащей Scheme-системой. Это похоже на «падение» компьютера в случае ошибки пользовательской программы[32]. Построить настоящую систему обработки ошибок — большой проект, но понимание, что за вопросы здесь возникают, стоит затраченных усилий.

а. При ошибках, возникающих в процессе вычисления, например, при попытке получить значение неопределенной переменной, можно заставить операцию просмотра окружения возвращать особый код ошибки, который не может служить значением пользовательской переменной. Тогда вычислитель может проверять этот код и организовывать переход на `signal-error`. Найдите в вычислителе все места, где нужно провести подобные изменения, и исправьте их. (Потребуется много работы.)

б. Значительно тяжелее проблема, которую представляют ошибки, возникающие в элементарных процедурах, например, попытки поделить на ноль или взять `car` символа. В профессионально написанной системе высокого качества всякий вызов примитива проверяется на безопасность внутри процедуры-примитива. Например, при всяком вызове `car` требуется проверить, что аргумент — пара. Если аргумент не является парой, вызов вернет особый код ошибки, который вычислитель может проверить и сообщить об ошибке. В нашем имитаторе регистровых машин этого можно было бы добиться, если бы мы проверяли в каждой элементарной процедуре правильность аргументов и

[32] К сожалению, в обычных компиляторных языковых системах, например, C, это обычное дело. В UNIX™ система «кидает дамп», в DOS/Windows™ впадает в кататонию. Macintosh™, если повезет, рисует на экране взрывающуюся бомбу и предлагает перегрузиться.

при необходимости возвращали соответствующий код. В таком случае код `primitive-apply` мог бы проверять этот код и, если надо, переходить на `signal-error`. Постройте такую структуру и заставьте ее работать. (Это большой проект.)

5.5. Компиляция

Вычислитель с явным управлением из раздела 5.4 — регистровая машина, контроллер которой исполняет Scheme-программы. В этом разделе мы увидим, как выполнять программы на Scheme с помощью регистровой машины, контроллер которой не является интерпретатором Scheme.

Машина-вычислитель с явным управлением универсальна — она способна выполнить любой вычислительный процесс, который можно описать на Scheme. Контроллер вычислителя выстраивает использование своих путей данных так, чтобы исполнялось нужное вычисление. Таким образом, пути данных вычислителя универсальны: они достаточны для того, чтобы выполнить любое необходимое нам вычисление, если снабдить их подходящим контроллером[33].

Коммерческие компьютеры общего назначения представляют собой регистровые машины, построенные на множестве регистров и операций, составляющем эффективный и удобный набор путей данных. Контроллер машины общего назначения — это интерпретатор языка регистровых машин, подобный нашему. Язык называется *внутренним языком* (native language) машины, или попросту *машинным языком* (machine language). Программы, написанные на машинном языке — это последовательности команд, использующих пути данных машины. Например, последовательность команд вычислителя с явным управлением можно рассматривать как программу на машинном языке компьютера общего назначения, а не как контроллер специализированной машины-интерпретатора.

Есть две стратегии борьбы с разрывом между языками высокого и низкого уровня. Вычислитель с явным управлением иллюстрирует стратегию интерпретации. Интерпретатор, написанный на внутреннем языке машины, конфигурирует машину так, что она начинает исполнять программы на языке (называемом *исходный язык* (source language)), который может отличаться от внутреннего языка машины, производящей вычисление. Элементарные процедуры исходного языка реализуются в виде библиотеки подпрограмм, написанных на внутреннем языке данной машины. Интерпретируемая программа (называемая *исходная программа* (source program)) представляется как структура данных. Интерпретатор просматривает эту структуру и анализирует исходную программу. В процессе анализа он имитирует требуемое поведение исходной программы, вызывая соответствующие элементарные подпрограммы из библиотеки.

В этом разделе мы исследуем альтернативную стратегию — *компиляцию* (compilation). Компилятор для данного исходного языка и данной машины переводит исходную программу в эквивалентную ей программу (называемую *объектной* (object program)), написанную на внутреннем языке машины. Компилятор, который мы реализуем в этом разделе, переводит программы, написанные на Scheme, в последовательности

[33]Это теоретическое утверждение. Мы не говорим, что пути данных вычислителя как-то особенно удобны или эффективны для компьютера общего назначения. Например, они не слишком хороши для реализации высокоскоростных вычислений с плавающей точкой или для вычислений, интенсивно работающих с битовыми векторами.

команд, которые подлежат исполнению с помощью путей данных машины-вычислителя с явным управлением[34].

По сравнению с интерпретацией, компиляция может дать большой выигрыш в эффективности исполнения программы. Это будет объяснено ниже, при обзоре компилятора. С другой стороны, интерпретатор предоставляет более мощную среду для интерактивной разработки программы и отладки, поскольку исполняемая исходная программа присутствует во время выполнения, и ее можно исследовать и изменять. В дополнение к этому, поскольку библиотека примитивов присутствует целиком, во время отладки можно конструировать и добавлять в систему новые программы.

Исходя из взаимно дополнительных преимуществ компиляции и интерпретации, современные среды разработки программ следуют смешанной стратегии. Как правило, интерпретаторы Лиспа устроены таким образом, что интерпретируемые и скомпилированные процедуры могут вызывать друг друга. Это позволяет программисту компилировать те части программы, которые он считает отлаженными, пользуясь при этом преимуществом в эффективности, предоставляемом компиляцией, но при этом сохранять интерпретационный режим выполнения для тех частей программы, которые находятся в гуще интерактивной разработки и отладки. В разделе 5.5.7, после того, как компилятор будет разработан, мы покажем, как построить его взаимодействие с нашим интерпретатором и получить интегрированную систему разработки, состоящую из компилятора и интерпретатора.

Обзор компилятора

Наш компилятор во многом похож на наш интерпретатор, как по структуре, так и по функции, которую он осуществляет. Соответственно, механизмы анализа выражений, используемые компилятором, будут подобны тем же механизмам для интерпретатора. Более того, чтобы упростить взаимодействие компилируемого и интерпретируемого кода, мы построим компилятор так, чтобы порождаемый им код следовал тем же соглашениям, что и интерпретатор: окружение будет храниться в регистре `env`, списки аргументов будут собираться в `argl`, применяемая процедура — в `proc`, процедуры будут возвращать свое значение в `val`, а место, куда им следует вернуться, будет храниться в регистре `continue`. В общем, компилятор переводит исходную программу в объектную программу, которая проделывает, в сущности, те же самые операции с регистрами, которые провел бы интерпретатор при выполнении той же самой исходной программы.

Это описание подсказывает стратегию для реализации примитивного компилятора: разбирать выражение таким же образом, как это делает интерпретатор. Когда мы встречаем команду работы с регистром, которую интерпретатор выполнил бы при работе с выражением, мы эту команду не выполняем, а добавляем к порождаемой нами последовательности. Полученная последовательность команд и будет объектным кодом. Отсюда видно преимущество в эффективности, которое компиляция имеет перед интерпретаци-

[34]На самом деле, машина, исполняющая скомпилированный код, может быть проще, чем машина-интерпретатор, поскольку регистры `exp` и `unev` мы использовать не будем. В интерпретаторе они использовались для хранения невычисленных выражений. Однако при использовании компилятора эти выражения встраиваются в компилируемый код, который будет выполняться на регистровой машине. По той же причине нам не нужны машинные операции, работающие с синтаксисом выражений. Однако скомпилированный код будет использовать некоторые дополнительные машинные операции (представляющие скомпилированные объекты-процедуры), которых не было в машине-вычислителе с явным управлением.

ей. Каждый раз, когда интерпретатор выполняет выражение — например, `(f 48 96)`, — он проделывает работу по распознаванию выражения (определение того, что это вызов процедуры) и проверке, не кончился ли список операндов (определение того, что операндов два). В случае с компилятором выражение анализируется только один раз, когда во время компиляции порождается последовательность команд. Объектный код, порожденный компилятором, содержит только команды, которые вычисляют оператор и два операнда, собирают список аргументов и применяют процедуру (из `proc`) к аргументам (из `argl`).

Это тот же самый вид оптимизации, который мы применяли в анализирующем интерпретаторе из раздела 4.1.7. Однако в случае компиляции имеются дополнительные возможности повысить эффективность. Интерпретатор при работе следует процессу, который обязан быть приложимым к любому выражению языка. В противоположность этому, всякий данный сегмент скомпилированного кода должен вычислять только одно выражение. Это может приводить к большой разнице, например, при использовании стека для сохранения регистров. Интерпретатор, выполняя выражение, должен быть готов к любым неожиданностям. При вычислении подвыражения он сохраняет все регистры, которые понадобятся в дальнейшем, поскольку в подвыражении могут содержаться произвольные действия. Напротив, компилятор может пользоваться структурой конкретного выражения и порождать код, который избегает лишних операций со стеком.

Рассмотрим в качестве примера выражение `(f 84 96)`. Интерпретатор, прежде чем вычислять оператор комбинации, подготавливается к этому вычислению и сохраняет регистры с операндами и окружением, чьи значения ему потребуются позже. Затем интерпретатор вычисляет оператор, получая значение в `val`, восстанавливает сохраненные регистры, и, наконец, переносит `val` в `proc`. Однако в данном конкретном вычислении оператором служит символ `f`, и его вычисление осуществляется командой `lookup-variable-value`, которая никаких регистров не изменяет. Компилятор, который мы разработаем в этом разделе, пользуется этим фактом и порождает код для вычисления оператора командой

```
(assign proc (op lookup-variable-value) (const f) (reg env))
```

Этот код не только избегает ненужных сохранений и восстановлений, но и записывает значение переменной напрямую в регистр `proc`, в то время как интерпретатор сначала получает его в `val`, а уж затем переносит в `proc`.

Кроме того, компилятор может оптимизировать доступ к среде. Во многих случаях при анализе кода компилятор может определять, в каком кадре будет находиться конкретная переменная, и обращаться к этому кадру напрямую, а не через поиск `lookup-variable-value`. Мы рассмотрим, как реализуется такой доступ к переменным, в разделе 5.5.6. До тех пор, впрочем, мы сосредоточим внимание на оптимизациях доступа к регистрам и стеку, описанным выше. Имеются и другие виды оптимизаций, которые может производить компилятор: например, «вставка» кода элементарных операций вместо общего механизма `apply` (см. упражнение 5.38); однако эти оптимизации мы здесь рассматривать не будем. В этом разделе наша цель — проиллюстрировать процесс компиляции в упрощенном (но все же интересном) контексте.

5.5.1. Структура компилятора

В разделе 4.1.7 мы модифицировали исходный метациклический интерпретатор и отделили анализ от выполнения. При анализе каждого выражения порождалась исполнительная процедура, которая в качестве аргумента принимала окружение и проделывала требуемые операции. В компиляторе мы будем проводить, в сущности, такой же анализ. Однако вместо исполнительных процедур мы будем порождать последовательности команд, предназначенных для исполнения на нашей регистровой машине.

Процедура `compile` проводит в компиляторе анализ верхнего уровня. Она соответствует процедуре `eval` из раздела 4.1.1, процедуре `analyze` из раздела 4.1.7 и точке входа `eval-dispatch` вычислителя с явным управлением из раздела 5.4.1. Подобно интерпретаторам, компилятор использует процедуры разбора синтаксиса выражений из раздела 4.1.2[35]. Процедура `compile` проводит разбор по случаям на основе синтаксического типа выражения, подлежащего компиляции. Для каждого типа выражения она вызывает специальный *генератор кода* (code generator).

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp target linkage))
        ((assignment? exp)
         (compile-assignment exp target linkage))
        ((definition? exp)
         (compile-definition exp target linkage))
        ((if? exp) (compile-if exp target linkage))
        ((lambda? exp) (compile-lambda exp target linkage))
        ((begin? exp)
         (compile-sequence (begin-actions exp)
                           target
                           linkage))
        ((cond? exp) (compile (cond->if exp) target linkage))
        ((application? exp)
         (compile-application exp target linkage))
        (else
         (error "Неизвестный тип выражения -- COMPILE" exp))))
```

Целевые регистры и типы связи

`Compile` и вызываемые оттуда генераторы кода принимают, помимо подлежащего компиляции выражения, еще два аргумента. Во-первых, *целевой регистр* (target), в который компилируемый код должен поместить значение выражения. Во-вторых, *тип связи* (linkage descriptor), который описывает, что код, который получается при компи-

[35] Заметим, однако, что наш компилятор является программой на Scheme, и для анализа синтаксиса он использует те же процедуры на Scheme, которые использовал метациклический интерпретатор. Для вычислителя с явным управлением мы, наоборот, предполагали, что эквивалентные синтаксические операции присутствуют как примитивы в регистровой машине. (Разумеется, когда мы имитировали эту машину на Scheme, в модели регистровой машины мы использовали эти же процедуры Scheme.)

ляции, должен делать после того, как он закончит выполняться. Описатель типа связи может потребовать одного из трех следующих действий:

- продолжить со следующей команды в последовательности (на это указывает описатель типа связи `next`),
- вернуться из компилируемой процедуры (на это указывает описатель типа связи `return`), или
- перейти на указанную метку (на это указывает использование метки в качестве описателя связи).

Например, компиляция выражения 5 (значение которого равно ему самому) с целевым регистром `val` и типом связи `next` должна породить команду

```
(aasign val (const 5))
```

Компиляция того же самого выражения с типом связи `return` должна породить команды

```
(assign val (const 5))
(goto (reg continue))
```

В первом случае выполнение продолжится на следующей команде последовательности. Во втором мы вернемся из процедуры. В обоих случаях значение выражения будет помещено в целевой регистр `val`.

Последовательности команд и использование стека

Каждый генератор кода возвращает *последовательность команд* (instruction sequence), содержащую порожденный для выражения объектный код. Порождение кода для составных выражений достигается путем сочетания более простых сегментов, порожденных генераторами кода для подвыражений, так же, как вычисление составного выражения проходит через вычисление подвыражений.

Простейший способ сочетания последовательностей команд — процедура под названием `append-instruction-sequences`. Она принимает в качестве аргументов произвольное число последовательностей команд, которые надо выполнить одну за другой. Процедура склеивает их и возвращает полученную последовательность. а именно, если $\langle посл_1 \rangle$ и $\langle посл_2 \rangle$ — последовательности команд, то вычисление

`(append-instruction-sequences` $\langle посл_1 \rangle$ $\langle посл_2 \rangle$`)`

вернет последовательность

$\langle посл_1 \rangle$
$\langle посл_2 \rangle$

Когда требуется сохранять регистры, генераторы кода используют `preserving`, более сложный метод сочетания последовательностей команд. `Preserving` принимает три аргумента: множество регистров и две последовательности, которые требуется выполнить одна за другой. Эта процедура склеивает последовательности таким образом, что

содержимое всех регистров из множества сохраняется во время выполнения первой последовательности, если оно нужно при выполнении второй. Таким образом, если первая последовательность изменяет регистр, а второй последовательности нужно его исходное содержимое, `preserving` оборачивает вокруг первой последовательности команды `save` и `restore` для этого регистра, прежде чем склеить последовательности. В противном случае она просто возвращает склеенные последовательности команд. Так, например,

(preserving (list ⟨*рег*$_1$⟩ ⟨*рег*$_2$⟩) ⟨*посл*$_1$⟩ ⟨*посл*$_2$⟩)

порождает одну из следующих четырех последовательностей команд, в зависимости от того, как ⟨*посл*$_1$⟩ и ⟨*посл*$_2$⟩ используют ⟨*рег*$_1$⟩ и ⟨*рег*$_2$⟩:

⟨*посл*$_1$⟩
⟨*посл*$_2$⟩

или

(save ⟨*рег*$_1$⟩)
⟨*посл*$_1$⟩
(restore ⟨*рег*$_1$⟩)
⟨*посл*$_2$⟩

или

(save ⟨*рег*$_2$⟩)
⟨*посл*$_1$⟩
(restore ⟨*рег*$_2$⟩)
⟨*посл*$_2$⟩

или

(save ⟨*рег*$_2$⟩)
(save ⟨*рег*$_1$⟩)
⟨*посл*$_1$⟩
(restore ⟨*рег*$_1$⟩)
(restore ⟨*рег*$_2$⟩)
⟨*посл*$_2$⟩

Сочетая последовательности команд с помощью `preserving`, компилятор избегает лишних операций со стеком. Кроме того, при этом забота о том, стоит ли порождать `save` и `restore`, целиком оказывается заключенной в процедуре `preserving` и отделяется от забот, которые будут нас волновать при написании отдельных генераторов кода. В сущности, ни одна команда `save` или `restore` не порождается генераторами кода явно.

В принципе мы могли бы представлять последовательность команд просто как список отдельных команд. В таком случае `append-instruction-sequences` могла бы склеивать последовательности с помощью обычного `append` для списков. Однако тогда `preserving` оказалась бы более сложной операцией, поскольку ей пришлось бы исследовать каждую последовательность команд и выяснять, как там используются регистры.

`Preserving` была бы при этом сложной и неэффективной, поскольку она анализировала бы каждый из своих аргументов-последовательностей, при том, что сами эти последовательности могли быть созданы вызовами `preserving`, и в этом случае их части были бы уже проанализированы. Чтобы избежать такого многократного анализа, мы с каждой последовательностью команд будем связывать некоторую информацию о том, как она использует регистры. При порождении элементарной последовательности мы будем указывать эту информацию явно, а процедуры, сочетающие последовательности, будут выводить информацию об использовании регистров для составной последовательности из информации, связанной с ее последовательностями-компонентами.

Последовательность команд будет содержать три вида информации:

- множество регистров, которые должны быть инициализированы, прежде чем выполняются команды из последовательности (говорится, что последовательность *нуждается* (needs) в этих регистрах),
- множество регистров, значения которых последовательность изменяет, и
- сами команды (называемые также *предложениями* (statements)) в последовательности.

Мы будем представлять последовательность команд в виде списка из трех частей. Таким образом, конструктор для последовательностей команд таков:

```
(define (make-instruction-sequence needs modifies statements)
  (list needs modifies statements))
```

Например, последовательность из двух команд, которая ищет значение переменной `x` в текущем окружении, присваивает его `val`, а затем возвращается, требует, чтобы были проинициализированы регистры `env` и `continue`, и изменяет регистр `val`. Следовательно, эту последовательность можно построить так:

```
(make-instruction-sequence '(env continue) '(val)
 '((assign val
           (op lookup-variable-value) (const x) (reg env))
   (goto (reg continue))))
```

Иногда нам нужно будет строить последовательность без команд:

```
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
```

Процедуры для сочетания последовательностей команд приведены в разделе 5.5.4.

Упражнение 5.31.
Во время вычисления вызова процедуры вычислитель с явным управлением всегда сохраняет и восстанавливает регистр `env` при вычислении оператора, сохраняет и восстанавливает `env` при вычислении каждого операнда (кроме последнего), сохраняет и восстанавливает `argl` при вычислении каждого операнда, а также сохраняет и восстанавливает `proc` при вычислении последовательности операндов. Для каждой из следующих комбинаций скажите, какие из этих операций `save` и `restore` излишни и могут быть исключены с помощью механизма `preserving`:

```
(f 'x 'y)

((f) 'x 'y)

(f (g 'x) y)

(f (g 'x) 'y)
```

Упражнение 5.32.
С помощью механизма `preserving` компилятор сможет избежать сохранения и восстановления `env` при вычислении оператора комбинации в случае, если это символ. Такие оптимизации можно было бы встроить и в интерпретатор. В сущности, вычислитель с явным управлением из раздела 5.4 уже проводит одну подобную оптимизацию, поскольку рассматривает комбинацию без операндов как особый случай.

а. Расширьте вычислитель с явным управлением так, чтобы он как особый случай рассматривал комбинации, в которых оператором является символ, и при вычислении таких выражений использовал это свойство оператора.

б. Лиза П. Хакер говорит, что если заставить интерпретатор рассматривать все больше особых случаев, то можно включить в него все оптимизации компилятора, и при этом все преимущество компиляции пропадет. Каково Ваше мнение?

5.5.2. Компиляция выражений

В этом и следующем разделе мы реализуем генераторы кода, на которые ссылается процедура `compile`.

Компиляция связующего кода

В общем случае результат работы каждого генератора кода будет заканчиваться командами — порожденными процедурой `compile-linkage`, — которые реализуют требуемый тип связи. Если это тип `return`, то нам надо породить команду `(goto (reg continue))`. Она нуждается в регистре `continue` и никаких регистров не меняет. Если тип связи `next`, то никаких дополнительных команд порождать не надо. В остальных случаях тип связи — переход по метке, и мы порождаем команду `goto` на эту метку, команду, которая ни в чем не нуждается и не изменяет никакие регистры[36].

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
```

[36] В этой процедуре используется конструкция Лиспа, называемая *обратная кавычка* (backquote) или *квазикавычка* (quasiquote), с помощью которой удобно строить списки. Обратная кавычка перед списком работает почти так же, как обычная, но при этом все выражения внутри списка, перед которыми стоит запятая, вычисляются.

Например, если значение `linkage` равно символу `branch25`, то результатом выражения `` `((goto (label ,linkage))) `` будет список `((goto (label branch25)))`. Подобным образом, если значением `x` является список `(a b c)`, то `` `(1 2 ,(car x)) `` дает при вычислении список `(1 2 a)`.

```
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          `((goto (label ,linkage)))))))
```

Связующий код добавляется к последовательности команд с сохранением через `preserving` регистра `continue`, поскольку связь `return` нуждается в этом регистре: если данная последовательность команд изменяет `continue`, а связующий код в нем нуждается, `continue` будет сохранен и восстановлен.

```
(define (end-with-linkage linkage instruction-sequence)
  (preserving '(continue)
   instruction-sequence
   (compile-linkage linkage)))
```

Компиляция простых выражений

Генераторы кода для самовычисляющихся выражений, кавычек и переменных строят последовательности команд, которые присваивают нужное значение целевому регистру, а затем ведут себя в соответствии с описателем связи.

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,exp))))))

(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,(text-of-quotation exp)))))))

(define (compile-variable exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '(env) (list target)
    `((assign ,target
              (op lookup-variable-value)
              (const ,exp)
              (reg env))))))
```

Все эти последовательности команд изменяют целевой регистр, а для поиска значения переменной требуется регистр `env`.

Присваивания и определения обрабатываются во многом так же, как в интерпретаторе. Мы рекурсивно порождаем код, вычисляющий значение, которое следует присвоить переменной, и присоединяем его к последовательности из двух команд, которая собственно присваивает значение переменной или определяет ее, а затем заносит в целевой регистр значение всего выражения (символ `ok`). Рекурсивная компиляция вызывается с целевым регистром `val` и типом связи `next`, так что порождаемый код положит результат в регистр `val`, а затем продолжит выполнение с той последовательности, которая идет за ним. При объединении кода сохраняется `env`, поскольку для определения и

присваивания переменной требуется окружение, а код, вычисляющий значение переменной, может оказаться сложным выражением, которое изменяет регистры произвольным образом.

```
(define (compile-assignment exp target linkage)
  (let ((var (assignment-variable exp))
        (get-value-code
         (compile (assignment-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       `((perform (op set-variable-value!)
                  (const ,var)
                  (reg val)
                  (reg env))
         (assign ,target (const ok))))))))

(define (compile-definition exp target linkage)
  (let ((var (definition-variable exp))
        (get-value-code
         (compile (definition-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       `((perform (op define-variable!)
                  (const ,var)
                  (reg val)
                  (reg env))
         (assign ,target (const ok))))))))
```

Двухкомандная последовательность в конце нуждается в `env` и `val` и изменяет свой целевой регистр. Заметим, что мы сохраняем в последовательности `env`, но не сохраняем `val`, поскольку `get-value-code` для того и нужна, чтобы поместить в `val` результат, которым затем воспользуется эта последовательность. (На самом деле сохранение `val` было бы ошибкой, поскольку тогда сразу после выполнения `get-value-code` восстановилось бы старое значение `val`.)

Компиляция условных выражений

Код для выражения `if` с указанными целевым регистром и типом связи имеет форму

```
 ⟨скомпилированный код для предиката с целевым регистром val
 и типом связи next⟩
 (test (op false?) (reg val))
 (branch (label false-branch))
true-branch
 ⟨скомпилированный код для следствия с указанным целевым регистром и
 указанным типом связи либо after-if⟩
```

```
false-branch
 ⟨скомпилированный код для альтернативы с указанными целевым регистром
 и типом связи⟩
after-if
```

Для того, чтобы породить этот код, мы компилируем предикат, следствие и альтернативу, а затем сочетаем то, что получилось, с командами, проверяющими значение предиката и со свежепорожденными метками, которые отмечают истинную ветвь, ложную ветвь и конец условного выражения[37]. В этом блоке кода нам требуется обойти истинную ветвь, если предикат ложен. Единственная небольшая сложность состоит в том, какой тип связи нужно указывать для истинной ветви. Если тип связи условного выражения `return` или метка, то и истинная, и ложная ветка будут этот тип и использовать. Если же тип связи `next`, то истинная ветвь заканчивается переходом, обходящим код для ложной ветви, на метку, которая стоит в конце условного выражения.

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
      (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
             (compile
              (if-consequent exp) target consequent-linkage))
            (a-code
             (compile (if-alternative exp) target linkage)))
        (preserving '(env continue)
         p-code
         (append-instruction-sequences
          (make-instruction-sequence '(val) '()
           `((test (op false?) (reg val))
             (branch (label ,f-branch))))
          (parallel-instruction-sequences
```

[37] Просто использовать метки `true-branch`, `false-branch` и `after-if` нельзя, потому что в программе может быть больше одного `if`. Компьютер порождает метки при помощи процедуры `make-label`. Она принимает символ в качестве аргумента и возвращает новый символ, имя которого начинается с данного. Например, последовательные вызовы `(make-label 'a)` будут возвращать `a1`, `a2` и так далее. Процедуру `make-label` можно написать аналогично тому, как порождаются новые имена переменных в языке запросов, а именно:

```
(define label-counter 0)

(define (new-label-number)
  (set! label-counter (+ 1 label-counter))
  label-counter)

(define (make-label name)
  (string->symbol
    (string-append (symbol->string name)
                   (number->string (new-label-number)))))
```

```
          (append-instruction-sequences t-branch c-code)
          (append-instruction-sequences f-branch a-code))
         after-if))))))
```

При вычислении предиката сохраняется `env`, поскольку он может потребоваться в истинной и ложной ветке, и `continue`, поскольку он может потребоваться связующему коду в этих ветвях. Код для истинной и ложной ветви (которые не выполняются последовательно) склеивается с помощью особого комбинатора `parallel-instruction-sequences`, описанного в разделе 5.5.4.

Заметим, что поскольку `cond` является производным выражением, для его обработки компилятор должен только запустить преобразование `cond->if`, а затем скомпилировать получившееся выражение `if`.

Компиляция последовательностей

Компиляция последовательностей (тел процедур и явных выражений `begin`) происходит так же, как их выполнение. Компилируется каждое из выражений последовательности — последнее с типом связи, который указан для всей последовательности, а остальные с типом связи `next` (для того, чтобы потом выполнялись остальные выражения последовательности). Последовательности команд для отдельных выражений склеиваются и образуют единую последовательность, при этом сохраняются `env` (необходимый для остатка последовательности) и `continue` (возможно, требуемый для связи в конце последовательности).

```
(define (compile-sequence seq target linkage)
  (if (last-exp? seq)
      (compile (first-exp seq) target linkage)
      (preserving '(env continue)
       (compile (first-exp seq) target 'next)
       (compile-sequence (rest-exps seq) target linkage))))
```

Компиляция выражений `lambda`

Выражения `lambda` строят процедуры. Объектный код для выражения `lambda` должен иметь вид

⟨*построить процедурный объект и присвоить его целевому регистру*⟩
⟨*связь*⟩

Компилируя выражения `lambda`, мы одновременно генерируем код для тела процедуры. Несмотря на то, что во время построения процедурного объекта тело исполняться не будет, удобно вставить его в код сразу после кода для `lambda`. Если связь для выражения `lambda` — метка или `return`, никаких сложностей при этом не возникает. Если же у нас тип связи `next`, то нужно обойти код для тела процедуры, использовав связь, которая переходит на метку, вставляемую сразу вслед за телом. Таким образом, объектный код принимает вид

⟨*построить процедурный объект и присвоить его целевому регистру*⟩
⟨*код для указанной связи*⟩ *либо* `(goto (label after-lambda))`

⟨*скомпилированное тело процедуры*⟩
after-lambda

Процедура compile-lambda порождает код, строящий процедурный объект, вслед за которым идет код тела процедуры. Процедурный объект порождается во время выполнения путем сочетания текущего окружения (окружения, в котором исполняется определение) и точки входа для скомпилированного тела процедуры (свежесгенерированной метки)[38].

```
(define (compile-lambda exp target linkage)
  (let ((proc-entry (make-label 'entry))
        (after-lambda (make-label 'after-lambda)))
    (let ((lambda-linkage
           (if (eq? linkage 'next) after-lambda linkage)))
      (append-instruction-sequences
       (tack-on-instruction-sequence
        (end-with-linkage lambda-linkage
         (make-instruction-sequence '(env) (list target)
          `((assign ,target
                    (op make-compiled-procedure)
                    (label ,proc-entry)
                    (reg env)))))
        (compile-lambda-body exp proc-entry))
       after-lambda))))
```

В compile-lambda для того, чтобы добавить тело процедуры к коду lambda-выражения, используется специальный комбинатор tack-on-instructionsequence (раздел 5.5.4), а не обыкновенный append-instructionsequences, поскольку тело процедуры не является частью последовательности команд, выполняемой при входе в общую последовательность; оно стоит в последовательности только потому, что его удобно было сюда поместить.

Процедура compile-lambda-body строит код для тела процедуры. Этот код начинается с метки для точки входа. Затем идут команды, которые заставят машину во время выполнения войти в правильное окружение для вычисления тела — то есть окружение, где определена процедура, расширенное связываниями формальных параметров с аргументами, с которыми она вызвана. Затем следует код для последовательности выражений, составляющих тело процедуры. Последовательность эта компилируется с типом

[38] Нам потребуются машинные операции, которые реализуют структуру данных, представляющую скомпилированные процедуры, аналогичную структуре для составных процедур, описанной в разделе 4.1.3:

```
(define (make-compiled-procedure entry env)
  (list 'compiled-procedure entry env))

(define (compiled-procedure? proc)
  (tagged-list? proc 'compiled-procedure))

(define (compiled-procedure-entry c-proc) (cadr c-proc))

(define (compiled-procedure-env c-proc) (caddr c-proc))
```

связи `return` и целевым регистром `val`, так что она закончится возвратом из процедуры с результатом в регистре `val`.

```
(define (compile-lambda-body exp proc-entry)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
     (make-instruction-sequence '(env proc argl) '(env)
      `(,proc-entry
        (assign env (op compiled-procedure-env) (reg proc))
        (assign env
                (op extend-environment)
                (const ,formals)
                (reg argl)
                (reg env))))
     (compile-sequence (lambda-body exp) 'val 'return))))
```

5.5.3. Компиляция комбинаций

Соль процесса компиляции заключается в компилировании вызовов процедур. Код для комбинации, скомпилированный с данными целевым регистром и типом связи, имеет вид

⟨*скомпилированный код оператора с целевым регистром* `proc` *и типом связи* `next`⟩
⟨*вычисление операндов и построение списка аргументов в* `argl`⟩
⟨*скомпилированный код вызова процедуры с указанными целевым регистром и типом связи*⟩

Во время вычисления оператора и операндов может потребоваться сохранить и восстановить регистры `env`, `proc` и `argl`. Заметим, что это единственное место в компиляторе, где указывается целевой регистр, отличный от `val`.

Требуемый код порождается процедурой `compile-application`. Она рекурсивно компилирует оператор, порождая код, который помещает подлежащую вызову процедуру в `proc`, и операнды, порождая код, который по одному вычисляет операнды процедурного вызова. Последовательности команд для операндов собираются (в процедуре `construct-arglist`) вместе с кодом, который строит список аргументов в регистре `argl`, а полученный код для порождения списка аргументов склеивается с кодом вычисления процедуры и кодом, который производит собственно вызов (он порождается с помощью `compile-procedure-call`). При склеивании последовательностей команд требуется сохранить регистр `env` на время вычисления оператора (поскольку в это время `env` может измениться, а он еще потребуется во время вычисления операндов), а регистр `proc` требуется сохранить на время построения списка аргументов (при вычислении операндов `proc` может измениться, а он потребуется во время собственно вызова процедуры). Наконец, все время следует сохранять `continue`, поскольку этот регистр нужен для связующего кода.

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
        (operand-codes
```

```
        (map (lambda (operand) (compile operand 'val 'next))
             (operands exp))))
    (preserving '(env continue)
     proc-code
     (preserving '(proc continue)
      (construct-arglist operand-codes)
      (compile-procedure-call target linkage)))))
```

Код для построения списка аргументов вычисляет каждый операнд, помещая результат в `val`, а затем с помощью `cons` прицепляет его к списку аргументов, собираемому в `argl`. Поскольку мы по очереди нацепляем аргументы на `argl` через `cons`, нам нужно начать с последнего аргумента и закончить первым, чтобы в получившемся списке аргументы стояли в порядке от первого к последнему. Чтобы не тратить команду на инициализацию `argl` пустым списком, прежде чем начать последовательность вычислений, мы строим исходное значение `argl` в первом участке кода. Таким образом, общая форма построения списка аргументов такова:

```
⟨компиляция последнего операнда с целью val⟩
(assign argl (op list) (reg val))
⟨компиляция следующего аргумента с целью val⟩
(assign argl (op cons) (reg val) (reg argl))
...
⟨компиляция первого аргумента с целью val⟩
(assign argl (op cons) (reg val) (reg argl))
```

Нужно сохранять `argl` при вычислении всех операндов, кроме как в самом начале (чтобы уже набранные аргументы не потерялись), а при вычислении всех операндов, кроме как в самом конце, нужно сохранять `env` (его могут использовать последующие вычисления операндов).

Компилировать код для аргументов довольно сложно, поскольку особым образом обрабатывается первый вычисляемый операнд, и в различных местах требуется сохранять `argl` и `env`. Процедура `construct-arglist` принимает в качестве аргументов участки кода, которые вычисляют отдельные операнды. Если никаких операндов нет вообще, она попросту порождает команду

```
(assign argl (const ()))
```

В остальных случаях она порождает код, инициализирующий `argl` последним аргументом, и добавляет к нему код, который по очереди вычисляет остальные аргументы и добавляет их к `argl`. Для того, чтобы аргументы обрабатывались от конца к началу, нам следует обратить список последовательностей кода для операндов, подаваемый из `compile-application`.

```
(define (construct-arglist operand-codes)
  (let ((operand-codes (reverse operand-codes)))
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl)
         '((assign argl (const ()))))
        (let ((code-to-get-last-arg
```

```
                (append-instruction-sequences
                 (car operand-codes)
                 (make-instruction-sequence '(val) '(argl)
                  '((assign argl (op list) (reg val)))))))
          (if (null? (cdr operand-codes))
              code-to-get-last-arg
              (preserving '(env)
               code-to-get-last-arg
               (code-to-get-rest-args
                (cdr operand-codes))))))))

(define (code-to-get-rest-args operand-codes)
  (let ((code-for-next-arg
         (preserving '(argl)
          (car operand-codes)
          (make-instruction-sequence '(val argl) '(argl)
           '((assign argl
               (op cons) (reg val) (reg argl)))))))
    (if (null? (cdr operand-codes))
        code-for-next-arg
        (preserving '(env)
         code-for-next-arg
         (code-to-get-rest-args (cdr operand-codes))))))
```

Применение процедур

После того, как элементы комбинации вычислены, скомпилированный код должен применить процедуру из регистра `proc` к аргументам из регистра `argl`. Этот код рассматривает, в сущности, те же самые случаи, что и процедура `apply` из метациклического интерпретатора в разделе 4.1.1 или точка входа `apply-dispatch` из вычислителя с явным управлением в разделе 5.4.1. Нужно проверить какая процедура применяется — элементарная или составная. В случае элементарной процедуры используется `apply-primitive-procedure`; как ведется работа с составными процедурами, мы скоро увидим. Код применения процедуры имеет такую форму:

```
 (test (op primitive-procedure?) (reg proc))
 (branch (label primitive-branch))
compiled-branch
 ⟨код для применения скомпилированной процедуры с указанной целью
  и подходящим типом связи⟩
primitive-branch
 (assign ⟨целевой регистр⟩
         (op apply-primitive-procedure)
         (reg proc)
         (reg argl))
 ⟨связующий код⟩
after-call
```

Заметим, что если выбрана ветвь для скомпилированной процедуры, машина должна обойти ветвь для элементарной процедуры. Следовательно, если тип связи для исходного

вызова процедуры был `next`, ветвь для составной процедуры должна использовать связь с переходом на метку, стоящую после ветви для элементарной процедуры. (Подобным образом работает связь для истинной ветви в `compile-if`.)

```
(define (compile-procedure-call target linkage)
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (after-call (make-label 'after-call)))
    (let ((compiled-linkage
           (if (eq? linkage 'next) after-call linkage)))
      (append-instruction-sequences
       (make-instruction-sequence '(proc) '()
        `((test (op primitive-procedure?) (reg proc))
          (branch (label ,primitive-branch))))
       (parallel-instruction-sequences
        (append-instruction-sequences
         compiled-branch
         (compile-proc-appl target compiled-linkage))
        (append-instruction-sequences
         primitive-branch
         (end-with-linkage linkage
          (make-instruction-sequence '(proc argl)
                                     (list target)
           `((assign ,target
                     (op apply-primitive-procedure)
                     (reg proc)
                     (reg argl)))))))
       after-call))))
```

Ветви для элементарных и составных процедур, подобно истинной и ложной ветвям в `compile-if`, склеиваются через `parallel-instruction-sequences`, а не обыкновенной `append-instruction-sequences`, поскольку они не выполняются последовательно.

Применение скомпилированных процедур

Код, обрабатывающий применение процедур, — наиболее тонко устроенная часть компилятора, при том, что он порождает очень короткие последовательности команд. У скомпилированной процедуры (порожденной с помощью `compile-lambda`) имеется точка входа, то есть метка, указывающая, где начинается тело процедуры. Код, расположенный по этой метке, вычисляет результат, помещая его в `val`, а затем возвращается, исполняя команду `(goto (reg continue))`. Таким образом, если в качестве связи выступает метка, мы можем ожидать, что код для вызова скомпилированной процедуры (порождаемый с помощью `compile-proc-appl`) с указанным целевым регистром будет выглядеть так:

```
(assign continue (label proc-return))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

```
proc-return
 (assign ⟨целевой регистр⟩ (reg val))  ; включается, если целевой
                                       ; регистр не val
 (goto (label ⟨связь⟩))                ; связующий код
```

либо, если тип связи `return`, так:

```
 (save continue)
 (assign continue (label proc-return))
 (assign val (op compiled-procedure-entry) (reg proc))
 (goto (reg val))
proc-return
 (assign ⟨целевой регистр⟩ (reg val))  ; включается, если целевой
                                       ; регистр не val
 (restore continue)
 (goto (label ⟨связь⟩))                ; связующий код
```

Этот код устанавливает `continue` так, чтобы процедура вернулась на метку `proc-return`, а затем переходит на входную точку процедуры. Код по метке `proc-return` переносит результат процедуры из `val` в целевой регистр (если нужно), а затем переходит в место, определяемое типом связи. (Связь всегда `return` или метка, поскольку процедура `compile-procedure-call` заменяет связь `next` для ветви составной процедуры на переход к метке `after-call`.)

На самом деле, если целевой регистр не равен `val`, то именно такой код наш компилятор и породит[39]. Однако чаще всего целевым регистром является `val` (единственное место, в котором компилятор заказывает другой целевой регистр — это когда вычисление оператора имеет целью `proc`), так что результат процедуры помещается прямо в целевой регистр, и возвращаться в особое место, где он копируется, незачем. Вместо этого мы упрощаем код, так устанавливая `continue`, что процедура «возвращается» прямо на то место, которое указано типом связи вызывающего кода:

```
⟨установить continue в соответствии с типом вызова⟩
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

Если в качестве связи указана метка, мы устанавливаем `continue` так, что возврат происходит на эту метку. (Таким образом, в приведенной выше `proc-return`, команда `(goto (reg continue))`, которой кончается процедура, оказывается равносильной `(goto (label ⟨связь⟩))`.)

```
(assign continue (label ⟨связь⟩))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

Если тип связи у нас `return`, нам вообще ничего не надо делать с `continue`: там уже хранится нужное место возврата. (То есть команда `(goto (reg continue))`, которой заканчивается процедура, переходит прямо туда, куда перешла бы `(goto (reg-continue))`, расположенная по метке `proc-return`.)

[39] Мы сообщаем об ошибке, если целевой регистр не `val`, а тип связи `return`, поскольку единственное место, где мы требуем связи `return` — это компиляция процедур, а по нашему соглашению процедуры возвращают значение в регистре `val`.

```
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

При такой реализации типа связи `return` компилятор порождает код, обладающий свойством хвостовой рекурсии. Вызов процедуры, если это последнее действие в теле процедуры, приводит к простой передаче управления, когда на стек ничего не кладется.

Предположим, однако, что мы реализовали случай вызова процедуры с типом связи `return` и целевым регистром `val` так, как показано выше для случая с целью не-`val`. Хвостовая рекурсия оказалась бы уничтожена. Наша система по-прежнему вычисляла бы то же значение для всех выражений. Однако при каждом вызове процедур мы сохраняли бы `continue`, а после вызова возвращались бы для (ненужного) восстановления. В гнезде рекурсивных вызовов эти дополнительные сохранения накапливались бы[40].

При порождении вышеописанного кода для применения процедуры `compile-proc-appl` рассматривает четыре случая, в зависимости от того, является ли `val` целевым регистром, и от того, дан ли нам тип связи `return`. Обратите внимание: указано, что эти последовательности команд изменяют все регистры, поскольку при выполнении тела процедуры регистрам разрешено меняться как угодно[41]. Заметим, кроме того, что в случае с целевым регистром `val` и типом связи `return` говорится, что участок кода нуждается в `continue`: хотя в этой двухкомандной последовательности `continue` явно не упоминается, нам нужно знать, что при входе в скомпилированную процедуру `continue` будет содержать правильное значение.

```
(define (compile-proc-appl target linkage)
  (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
         (make-instruction-sequence '(proc) all-regs
           `((assign continue (label ,linkage))
             (assign val (op compiled-procedure-entry)
                         (reg proc))
             (goto (reg val)))))
        ((and (not (eq? target 'val))
              (not (eq? linkage 'return)))
         (let ((proc-return (make-label 'proc-return)))
           (make-instruction-sequence '(proc) all-regs
            `((assign continue (label ,proc-return))
```

[40]Казалось бы, заставить компилятор порождать код с хвостовой рекурсией — естественная идея. Однако большинство компиляторов для распространенных языков, включая C и Паскаль, так не делают, и, следовательно, в этих языках итеративный процесс нельзя представить только через вызовы процедур. Сложность с хвостовой рекурсией в этих языках состоит в том, что их реализации сохраняют на стеке не только адрес возврата, но еще и аргументы процедур и локальные переменные. Реализации Scheme, описанные в этой книге, хранят аргументы и переменные в памяти и подвергают их сборке мусора. Причина использования стека для переменных и аргументов — в том, что при этом можно избежать сборки мусора в языках, которые не требуют ее по другим причинам, и вообще считается, что так эффективнее. На самом деле изощренные реализации Лиспа могут хранить аргументы на стеке, не уничтожая хвостовую рекурсию. (Описание можно найти в Hanson 1990.) Кроме того, ведутся споры о том, правда ли, что выделение памяти на стеке эффективнее, чем сборка мусора, но тут результат, кажется, зависит от тонких деталей архитектуры компьютера. (См. Appel 1987 и Miller and Rozas 1994, где по этому вопросу высказываются противоположные мнения.)

[41]Значением переменной `all-regs` является список имен всех регистров:

```
(define all-regs '(env proc val argl continue))
```

```
            (assign val (op compiled-procedure-entry)
                        (reg proc))
            (goto (reg val))
            ,proc-return
            (assign ,target (reg val))
            (goto (label ,linkage))))))
      ((and (eq? target 'val) (eq? linkage 'return))
       (make-instruction-sequence '(proc continue) all-regs
        '((assign val (op compiled-procedure-entry)
                      (reg proc))
          (goto (reg val)))))
      ((and (not (eq? target 'val)) (eq? linkage 'return))
       (error "Тип связи return, цель не val -- COMPILE"
              target))))
```

5.5.4. Сочетание последовательностей команд

В этом разделе в деталях описывается представление последовательностей команд и их сочетание друг с другом. Напомним, что в разделе 5.5.1 мы решили, что последовательность представляется в виде списка, состоящего из множества требуемых регистров, множества изменяемых регистров, и собственно кода. Кроме того, мы будем считать метку (символ) особым случаем последовательности, которая не требует и не изменяет никаких регистров. Таким образом, для определения регистров, в которых нуждается и которые изменяет данная последовательность, мы пользуемся селекторами

```
(define (registers-needed s)
  (if (symbol? s) '() (car s)))

(define (registers-modified s)
  (if (symbol? s) '() (cadr s)))

(define (statements s)
  (if (symbol? s) (list s) (caddr s)))
```

а для того, чтобы выяснить, нуждается ли последовательность в регистре и изменяет ли она его, используются предикаты

```
(define (needs-register? seq reg)
  (memq reg (registers-needed seq)))

(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))
```

С помощью этих селекторов и предикатов мы можем реализовать все многочисленные комбинаторы последовательностей команд, которые используются в тексте компилятора.

Основным комбинатором является `append-instruction-sequences`. Он принимает как аргументы произвольное число последовательностей команд, которые следует выполнить последовательно, а возвращает последовательность команд, предложениями которой служат предложения всех последовательностей, склеенные вместе. Сложность

состоит в том, чтобы определить регистры, которые требуются, и регистры, которые изменяются в получаемой последовательности. Изменяются те регистры, которые изменяются в какой-либо из подпоследовательностей; требуются те регистры, которые должны быть проинициализированы прежде, чем можно запустить первую подпоследовательность (регистры, требуемые первой подпоследовательностью), а также регистры, которые требует любая из оставшихся подпоследовательностей, не измененные (проинициализированные) одной из подпоследовательностей, идущих перед ней.

Последовательности сливаются по две процедурой `append-2-sequences`. Она берет две последовательности команд `seq1` и `seq2`, и возвращает последовательность команд, в которой предложениями служат предложения `seq1`, а затем в конце добавлены предложения `seq2`. Ее изменяемые регистры — те, которые изменяет либо `seq1`, либо `seq2`, а требуемые регистры — те, что требует `seq1` плюс те, что требует `seq2` и не изменяет `seq1`. (В терминах операций над множествами, новое множество требуемых регистров является объединением множества требуемых регистров `seq1` с множественной разностью требуемых регистров `seq2` и изменяемых регистров `seq1`.) Таким образом, `append-instruction-sequences` реализуется так:

```
(define (append-instruction-sequences . seqs)
  (define (append-2-sequences seq1 seq2)
    (make-instruction-sequence
     (list-union (registers-needed seq1)
                 (list-difference (registers-needed seq2)
                                  (registers-modified seq1)))
     (list-union (registers-modified seq1)
                 (registers-modified seq2))
     (append (statements seq1) (statements seq2))))
  (define (append-seq-list seqs)
    (if (null? seqs)
        (empty-instruction-sequence)
        (append-2-sequences (car seqs)
                            (append-seq-list (cdr seqs)))))
  (append-seq-list seqs))
```

В этой процедуре используются некоторые операции для работы с множествами, представленными в виде списков, подобные (неотсортированному) представлению множеств, описанному в разделе 2.3.3:

```
(define (list-union s1 s2)
  (cond ((null? s1) s2)
        ((memq (car s1) s2) (list-union (cdr s1) s2))
        (else (cons (car s1) (list-union (cdr s1) s2)))))

(define (list-difference s1 s2)
  (cond ((null? s1) '())
        ((memq (car s1) s2) (list-difference (cdr s1) s2))
        (else (cons (car s1)
                    (list-difference (cdr s1) s2)))))
```

Второй основной комбинатор последовательностей команд, `preserving`, принимает список регистров `regs` и две последовательности команд `seq1` и `seq2`, которые следует

выполнить последовательно. Он возвращает последовательность команд, чьи предложения — это предложения seq1, за которыми идут предложения seq2, с командами save и restore вокруг seq1, для того, чтобы защитить регистры из множества regs, изменяемые в seq1, но требуемые в seq2. Для того, чтобы построить требуемую последовательность, сначала preserving создает последовательность, содержащую требуемые команды save, команды из seq1 и команды restore. Эта последовательность нуждается в регистрах, которые подвергаются сохранению/восстановлению, а также регистрах, требуемых seq1. Она изменяет регистры, которые меняет seq1, за исключением тех, которые сохраняются и восстанавливаются. Затем эта дополненная последовательность и seq2 сочетаются обычным образом. Следующая процедура реализует эту стратегию рекурсивно, двигаясь по списку сохраняемых регистров[42]:

```
(define (preserving regs seq1 seq2)
  (if (null? regs)
      (append-instruction-sequences seq1 seq2)
      (let ((first-reg (car regs)))
        (if (and (needs-register? seq2 first-reg)
                 (modifies-register? seq1 first-reg))
            (preserving (cdr regs)
             (make-instruction-sequence
              (list-union (list first-reg)
                          (registers-needed seq1))
              (list-difference (registers-modified seq1)
                               (list first-reg))
              (append `((save ,first-reg))
                      (statements seq1)
                      `((restore ,first-reg))))
             seq2)
            (preserving (cdr regs) seq1 seq2)))))
```

Еще один комбинатор последовательностей, tack-on-instruction-sequence, используется в compile-lambda для того, чтобы добавить тело процедуры к другой последовательности. Поскольку тело процедуры не находится «в потоке управления» и не должно выполняться как часть общей последовательности, используемые им регистры никак не влияют на регистры, используемые последовательностью, в которую оно включается. Таким образом, когда мы добавляем тело процедуры к другой последовательности, мы игнорируем его множества требуемых и изменяемых регистров.

```
(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
   (registers-needed seq)
   (registers-modified seq)
   (append (statements seq) (statements body-seq))))
```

В процедурах compile-if и compile-procedure-call используется специальный комбинатор parallel-instruction-sequences, который склеивает две альтернативные ветви, следующие за тестом. Эти две ветви никогда не исполняются одна за

[42] Заметим, что preserving зовет append с тремя аргументами. Хотя определение append, приводимое в этой книге, принимает только два аргумента, в стандарте Scheme имеется процедура append, принимающая любое их количество.

другой; при каждом исполнении теста будет запущена либо одна, либо другая ветвь. Поэтому регистры, требуемые во второй ветви, по-прежнему требуются составной последовательности, даже если первая ветвь их изменяет.

```
(define (parallel-instruction-sequences seq1 seq2)
  (make-instruction-sequence
   (list-union (registers-needed seq1)
               (registers-needed seq2))
   (list-union (registers-modified seq1)
               (registers-modified seq2))
   (append (statements seq1) (statements seq2))))
```

5.5.5. Пример скомпилированного кода

Теперь, когда мы рассмотрели все элементы компилятора, можно разобрать пример скомпилированного кода и увидеть, как сочетаются его элементы. Мы скомпилируем определение рекурсивной процедуры `factorial` с помощью вызова `compile`:

```
(compile
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n)))
 'val
 'next)
```

Мы указали, что значение выражения `define` требуется поместить в регистр `val`. Нам неважно, что будет делать скомпилированный код после того, как будет выполнено `define`, так что выбор `next` в качестве типа связи произволен.

Процедура `compile` распознает выражение как определение, так что она зовет `compile-definition`, чтобы породить код для вычисления присваиваемого значения (с целью `val`), затем код для внесения определения в среду, затем код, который помещает значение `define` (символ `ok`) в целевой регистр, и, наконец, связующий код. При вычислении значения сохраняется `env`, поскольку этот регистр требуется, чтобы внести определение в среду. Поскольку тип связи у нас `next`, никакого связующего кода не порождается. Таким образом, скелет скомпилированного кода таков:

```
⟨сохранить env, если его изменяет код для вычисления значения⟩
⟨скомпилированный код для значения определения, цель val, связь
next⟩
⟨восстановить env, если он сохранялся⟩
(perform (op define-variable!)
         (const factorial)
         (reg val)
         (reg env))
(assign val (const ok))
```

Выражение, которое нужно скомпилировать, чтобы получить значение переменной `factorial` — это выражение `lambda`, и значением его является процедура, вычисляющая факториалы. `Compile` обрабатывает его путем вызова `compile-lambda`.

`Compile-lambda` компилирует тело процедуры, снабжает его меткой как новую точку входа и порождает команду, которая склеит тело процедуры по новой метке с окружением времени выполнения и присвоит значение регистру `val`. Затем порожденная последовательность перепрыгивает через скомпилированный код, который вставляется в этом месте. Сам код процедуры начинается с того, что окружение, где процедура определена, расширяется кадром, в котором формальный параметр `n` связывается с аргументом процедуры. Затем идет собственно тело процедуры. Поскольку код для определения значения переменной не изменяет регистр `env`, команды `save` и `restore`, которые показаны выше как возможные, не порождаются. (В этот момент не выполняется код процедуры по метке `entry2`, так что детали его работы с `env` значения не имеют.) Следовательно, наш скелет скомпилированного кода становится таким:

```
  (assign val (op make-compiled-procedure)
              (label entry2)
              (reg env))
  (goto (label after-lambda1))
entry2
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment)
              (const (n))
              (reg argl)
              (reg env))
  ⟨скомпилированный код тела процедуры⟩
after-lambda1
  (perform (op define-variable!)
           (const factorial)
           (reg val)
           (reg env))
  (assign val (const ok))
```

Тело процедуры всегда компилируется (в `compile-lambda-body`) как последовательность команд с целевым регистром `val` и типом связи `return`. В данном случае в последовательности одно выражение `if`:

```
(if (= n 1)
    1
    (* (factorial (- n 1)) n))
```

`Compile-if` порождает код, который сначала вычисляет предикат (с целью `val`), затем проверяет его значение и, если предикат ложен, обходит истинную ветвь. При вычислении предиката сохраняются `env` и `continue`, поскольку они могут потребоваться в оставшейся части выражения `if`. Поскольку выражение `if` последнее (и единственное) в последовательности, составляющей тело процедуры, оно имеет цель `val` и тип связи `return`, так что и истинная, и ложная ветви компилируются с целью `val` и типом связи `return`. (Это значит, что значение условного выражения, которое вычисляется одной из его ветвей, является значением процедуры.)

```
  ⟨сохранить continue, env, если они изменяются предикатом и
  требуются в ветвях⟩
```

```
  ⟨скомпилированный код предиката, цель val, связь next⟩
  ⟨восстановить continue, env, если они сохранялись⟩
  (test (op false?) (reg val))
  (branch (label false-branch4)
true-branch5
  ⟨скомпилированный код истинной ветви, цель val, связь return⟩
false-branch4
  ⟨скомпилированный код ложной ветви, цель val, связь return⟩
after-if3
```

Предикат `(= n 1)` является вызовом процедуры. Он ищет в окружении оператор (символ `=`) и помещает его значение в `proc`. Затем он собирает аргументы — `1` и значение `n`, — в `argl`. Затем он проверяет, лежит ли в `proc` примитив или составная процедура, и соответствующим образом переходит на ветвь элементарной или составной процедуры. Обе ветви приводят к метке `after-call`. Требование сохранять регистры при вычислении оператора и операндов не приводит ни к каким операциям сохранения, поскольку в этом случае вычисления не трогают нужные регистры.

```
  (assign proc
          (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto reg val)
primitive-branch17
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
after-call15
```

Истинная ветвь, константа 1, компилируется (с целевым регистром `val` и типом связи `return`) в

```
  (assign val (const 1))
  (goto (reg continue))
```

Код для ложной ветви является еще одним вызовом процедуры, где процедурой служит значение символа `*`, а аргументами — `n` и значение еще одного вызова (вызова `factorial`). Каждый из этих вызовов устанавливает значения `proc` и `argl`, а также свои собственные ветви для элементарных и составных процедур. На рисунке 5.17 показан полный скомпилированный код для определения процедуры `factorial`. Заметим, что возможные команды `save` и `restore` для `continue` и `env` при вычислении предиката, указанные выше, на самом деле порождаются, поскольку эти регистры изменяются во время вызова процедуры в предикате и нужны для вызова процедуры и связи `return` в ветвях.

```
 ;; построить процедуру и обойти ее тело
  (assign val (op make-compiled-procedure) (label entry2) (reg env))
  (goto (label after-lambda1))

entry2      ; вызовы factorial будут попадать сюда
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
          (op extend-environment) (const (n)) (reg argl) (reg env))

 ;; начинается собственно тело процедуры
  (save continue)
  (save env)

 ;; вычислить (= n 1)
  (assign proc (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto reg val)
primitive-branch17
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call15    ; здесь val содержит результат (= n 1)
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch4))
true-branch5   ; вернуть 1
  (assign val (const 1))
  (goto (reg continue))

false-branch4
 ;; вычислить и вернуть (* (factorial (- n 1) n))
  (assign proc (op lookup-variable-value) (const *) (reg env))
  (save continue)
  (save proc)    ; сохранить процедуру *
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op list) (reg val))
  (save argl)    ; сохранить частичный список аргументов для *
```

Рис. 5.17. Скомпилированный код определения процедуры `factorial`. (Продолжение на следующей странице.)

```
 ;; вычислить (factorial (- n 1)), еще один аргумент *
  (assign proc
          (op lookup-variable-value) (const factorial) (reg env))
  (save proc)   ; сохранить процедуру factorial

 ;; вычислить (- n 1), аргумент factorial
  (assign proc (op lookup-variable-value) (const -) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch8))
compiled-branch7
  (assign continue (label after-call6))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch8
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call6    ; теперь в val содержится результат (- n 1)
  (assign argl (op list) (reg val))
  (restore proc)   ; восстановить factorial
 ;; применить factorial
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch11))
compiled-branch10
  (assign continue (label after-call9))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch11
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call9        ; теперь val содержит результат (factorial (- n 1))
  (restore argl)   ; восстановить частичный список аргументов для *
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)   ; восстановить *
  (restore continue)
 ;; применить * и вернуть
 ;; его результат
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch14))
compiled-branch13
 ;; обратите внимание:
 ;; скомпилированная процедура здесь зовется
 ;; с хвостовой рекурсией
```

Рис. 5.17. Скомпилированный код определения процедуры `factorial`. Продолжение. (Окончание на следующей странице.)

```
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch14
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call12
after-if3
after-lambda1
 ;; присвоить процедуру переменной factorial
  (perform
   (op define-variable!) (const factorial) (reg val) (reg env))
  (assign val (const ok))
```

Рис. 5.17. Скомпилированный код определения процедуры `factorial`. Окончание.

Упражнение 5.33.
Рассмотрим следующее определение процедуры для вычисления факториала, которое незначительно отличается от рассмотренного в тексте:

```
(define (factorial-alt n)
  (if (= n 1)
      1
      (* n (factorial-alt (- n 1)))))
```

Скомпилируйте эту процедуру и сравните получившийся код с кодом для `factorial`. Объясните обнаруженные различия. Есть ли разница в эффективности программ?

Упражнение 5.34.
Скомпилируйте итеративную процедуру вычисления факториала:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Прокомментируйте полученный код и покажите существенное различие между кодом для итеративной и рекурсивной версий `factorial`, благодаря которому один процесс наращивает глубину стека, а второй выполняется при фиксированной глубине.

Упражнение 5.35.
При компиляции какого выражения был получен код на рисунке 5.18?

```
  (assign val (op make-compiled-procedure) (label entry16)
                                           (reg env))
  (goto (label after-lambda15))
entry16
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
          (op extend-environment) (const (x)) (reg argl) (reg env))
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (save continue)
  (save proc)
  (save env)
  (assign proc (op lookup-variable-value) (const g) (reg env))
  (save proc)
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch19))
compiled-branch18
  (assign continue (label after-call17))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch19
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call17
  (assign argl (op list) (reg val))
  (restore proc)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch22))
compiled-branch21
  (assign continue (label after-call20))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch22
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
```

Рис. 5.18. Пример вывода компилятора. См. упражнение 5.35. (Продолжение на следующей странице.)

```
after-call20
  (assign argl (op list) (reg val))
  (restore env)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch25))
compiled-branch24
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch25
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call23
after-lambda15
  (perform (op define-variable!) (const f) (reg val) (reg env))
  (assign val (const ok))
```

Рис. 5.18. Пример вывода компилятора (продолжение).

.

Упражнение 5.36.
Какой порядок вычисления задает наш компилятор для операндов комбинации — слева направо, справа налево, или какой-либо иной? Где в компиляторе задается этот порядок? Измените компилятор так, чтобы он порождал какой-нибудь другой порядок вычисления. (См. обсуждение порядка вычисления для вычислителя с явным управлением из раздела 5.4.1.) Как смена порядка вычисления операндов влияет на эффективность кода, который строит список аргументов?

Упражнение 5.37.
Вот один из способов понять, как механизм `preserving` оптимизирует использование стека: рассмотреть, какие дополнительные операции порождались бы, если бы мы этот механизм не использовали. Измените `preserving` так, чтобы операции `save` и `restore` порождались всегда. Скомпилируйте несколько простых выражений и отметьте ненужные операции со стеком, которые станут порождаться. Сравните этот код с тем, который порождается, если механизм `preserving` присутствует.

Упражнение 5.38.
Наш компилятор тщательно избегает ненужных операций со стеком, но с точки зрения перевода вызовов элементарных процедур языка в операции машины он очень слаб. Рассмотрим, например, сколько кода порождается для вычисления `(+ a 1)`: код порождает список аргументов в `argl`, помещает элементарную процедуру сложения (которую он находит через поиск символа `+` в окружении) в `proc`, затем проверяет, является ли эта процедура элементарной или составной. Компилятор всегда порождает код этой проверки, а также код для ветви элементарной процедуры и ветви составной процедуры (из которых только одна будет выполняться). Мы не показали ту часть контроллера, которая реализует примитивы, но мы предполагаем, что эти команды используют элементарные арифметические операции в путях данных машины. Рассмотрим, насколько

меньше кода будет порождаться, если компилятор сможет вставлять примитивы в виде *явного кода* (open coding) — то есть порождать код, который прямо использует эти машинные операции. Выражение `(+ a 1)` можно было бы скомпилировать в простую последовательность вроде[43]

```
(assign val (op lookup-variable-value) (const a) (reg env))
(assign val (op +) (reg val) (const 1))
```

В этом упражнении мы расширим компилятор так, чтобы он поддерживал явное кодирование отдельных примитивов. При обращениях к этим примитивам будет порождаться специально написанный код, а не общий код для вызова процедуры. Для того, чтобы поддержать такую работу, мы дополним машину специальными регистрами для аргументов `arg1` и `arg2`. Элементарные арифметические операции машины будут принимать свои аргументы в `arg1` и `arg2`. Они могут помещать результаты в `val`, `arg1` или `arg2`.

Компилятор должен уметь распознавать вызов явно кодируемого примитива в исходной программе. Мы дополним распознаватель в процедуре `compile`, так, чтобы он узнавал имена этих примитивов в дополнение к зарезервированным словам (особым формам), которые он узнаёт сейчас[44]. Для каждой особой формы в компиляторе есть свой генератор кода. В этом упражнении мы построим семью генераторов кода для явно кодируемых примитивов.

а. В отличие от особых форм, явно кодируемые примитивы требуют, чтобы их аргументы вычислялись. Напишите генератор кода `spread-arguments`, который будут использовать генераторы явного кода. `Spread-arguments` должен принимать список операндов и компилировать данные ему операнды, направляя их в последовательные аргументные регистры. Заметим, что операнд может содержать вызов явно кодируемого примитива, так что во время вычисления операндов придется сохранять аргументные регистры.

б. Для каждой из элементарных процедур `=`, `*`, `-` и `+` напишите по генератору кода, который принимает комбинацию, содержащую этот оператор вместе с целевым регистром и описателем связи, и порождает код, который раскидывает аргументы по регистрам, а затем проводит операцию с данным целевым регистром и указанным типом связи. Достаточно обрабатывать только выражения с двумя операндами. Заставьте `compile` передавать управление этим генераторам кода.

в. Опробуйте обновленный компилятор на примере с процедурой `factorial`. Сравните полученный результат с результатом, который получается без открытого кодирования.

г. Расширьте свои генераторы кода для `+` и `*` так, чтобы они могли обрабатывать выражения с произвольным числом операндов. Выражение, в котором операндов больше двух, придется компилировать в последовательность операций, каждая из которых работает с двумя входами.

5.5.6. Лексическая адресация

Одна из наиболее часто встречающихся в компиляторах оптимизаций связана с поиском переменных. В нынешнем виде наш компилятор использует операцию `lookup-variable-value` машины-вычислителя. Эта операция ищет переменную, сравнивая ее со всеми переменными, связанными в данный момент, и проходя кадр за кадром по окружению, имеющемуся во время выполнения. Если кадр глубоко вложен или если имеется

[43] Здесь мы одним символом `+` обозначаем и процедуру исходного языка, и машинную операцию. В общем случае может не быть однозначного соответствия примитивов исходного языка примитивам машины.

[44] Вообще говоря, превращение примитивов в зарезервированные слова — плохая идея, потому что тогда пользователь не может связать эти имена с другими процедурами. Более того, если мы добавим зарезервированные слова в компилятор, который уже используется, перестанут работать существующие программы, которые определяют процедуры с такими именами. Идеи, как можно избежать этой проблемы, можно найти в упражнении 5.44.

много переменных, этот поиск может оказаться дорогим. Рассмотрим, например, задачу поиска значения `x` при вычислении выражения `(* x y z)` внутри процедуры, возвращаемой при вычислении

```
(let ((x 3) (y 4))
  (lambda (a b c d e)
    (let ((y (* a b x))
          (z (+ c d x)))
      (* x y z))))
```

Поскольку выражение `let` — всего лишь синтаксический сахар для комбинации `lambda`, это выражение равносильно

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) (* x y z))
      (* a b x)
      (+ c d x))))
 3
 4)
```

Каждый раз, когда `lookup-variable-value` ищет `x`, она должна убедиться, что символ `x` не равен (через `eq?`) ни `y`, ни `z` (в первом кадре), ни `a`, `b`, `c`, `d`, ни `e` (во втором). Предположим, временно, что в наших программах не используется `define` — что переменные связываются только через `lambda`. Поскольку в нашем языке лексическая сфера действия, во время выполнения окружение любого выражения будет иметь структуру, параллельную лексической структуре программы, в которой это выражение встречается[45]. Таким образом, компилятор при анализе вышеуказанного выражения может узнать, что каждый раз, когда процедура применяется, переменная с именем `x` будет найдена на два кадра выше текущего, и в этом кадре будет первая.

Мы можем это использовать и ввести новый вид операции поиска переменной, `lexical-address-lookup`, который в качестве аргументов берет окружение и *лексический адрес* (lexical address), состоящий из двух чисел: *номера кадра* (frame number), который показывает, сколько кадров надо пропустить, и *смещения* (displacement number), которое показывает, сколько переменных нужно пропустить в этом кадре. `Lexical-address-lookup` будет возвращать значение переменной, которая имеет указанный лексический адрес по отношению к текущему окружению. Добавив в свою машину `lexical-address-lookup`, мы можем научить компилятор порождать код, который обращается к переменным через эту операцию, а не через `lookup-variable-value`. Подобным образом, скомпилированный код может использовать новую операцию `lexical-address-set!` вместо `set-variable-value!`.

Для того, чтобы порождать такой код, компилятор должен уметь определять лексический адрес переменной, ссылку на которую он намерен скомпилировать. Лексический адрес переменной зависит от того, где она находится в коде. Например, в следующей программе адрес `x` в выражении ⟨*e1*⟩ есть (2,0) — на два кадра назад и первая переменная в кадре. В этом же месте `y` имеет адрес (0,0), а `c` — адрес (1,2). В выражении ⟨*e2*⟩ `x` имеет адрес (1,0), `y` адрес (1,1), а `c` адрес (0,2).

[45]Это не так, если мы разрешаем внутренние определения и если мы от них не избавляемся. См. упражнение 5.43.

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) ⟨e1⟩)
      ⟨e2⟩
      (+ c d x))))
 3
 4)
```

Один из способов породить в компиляторе код, который использует лексическую адресацию, состоит в поддержании структуры данных, называемой *окружение времени компиляции* (compile-time environment). Она следит за тем, какие переменные в каких позициях и в каких кадрах будут находиться в окружении времени выполнения, когда будет выполняться определенная операция доступа к переменной. Окружение времени компиляции представляет собой список кадров, каждый из которых содержит список переменных. (Разумеется, с переменными не будет связано никаких значений, поскольку во время компиляции значения не вычисляются.) Окружение времени компиляции становится дополнительным аргументом процедуры `compile` и передается всем генераторам кода. Вызов `compile` верхнего уровня использует пустое окружение времени компиляции. Когда компилируется тело `lambda`, `compile-lambda-body` расширяет это окружение кадром, содержащим параметры процедуры, так что последовательность, которая является телом, компилируется в этом расширенном окружении. В каждой точке компиляции `compile-variable` и `compile-assignment` используют окружение времени компиляции для порождения соответствующих лексических адресов.

Упражнения с 5.39 по 5.43 описывают, как завершить этот набросок лексической адресации и включить в компилятор лексический поиск. В упражнении 5.44 описывается еще один способ использовать окружение времени компиляции.

Упражнение 5.39.
Напишите процедуру `lexical-address-lookup`, которая реализует новую операцию поиска. Она должна брать два аргумента — лексический адрес и окружение времени компиляции, — и возвращать значение переменной, находящейся по указанному лексическому адресу. `Lexical-address-lookup` должна сообщать об ошибке, если значением переменной является символ `*unassigned*`[46]. Кроме того, напишите процедуру `lexical-address-set!`, реализующую операцию, которая изменяет значение переменной по указанному лексическому адресу.

Упражнение 5.40.
Модифицируйте компилятор так, чтобы он поддерживал окружение времени компиляции, как описано выше. а именно, добавьте аргумент-окружение к `compile` и всем генераторам кода, и расширяйте его в `compile-lambda-body`.

Упражнение 5.41.
Напишите процедуру `find-variable`, которая в качестве аргументов принимает переменную и окружение времени компиляции, а возвращает лексический адрес переменной по отношению к

[46]Эта модификация в поиске переменной требуется в том случае, если мы реализуем просмотр текста и уничтожение внутренних определений (упражнение 5.43). Чтобы лексическая адресация работала, их следует уничтожить.

этому окружению. Например, во фрагменте программы, который приведен выше, окружение времени компиляции при обработке выражения ⟨*e1*⟩ равно `((y z) (a b c d e) (x y))`. `Find-variable` должна давать

```
(find-variable 'c '((y z) (a b c d e) (x y)))
(1 2)

(find-variable 'x '((y z) (a b c d e) (x y)))
(2 0)

(find-variable 'w '((y z) (a b c d e) (x y)))
not-found
```

Упражнение 5.42.
С помощью `find-variable` из упражнения 5.41 перепишите `compile-variable` и `compile-assignment` так, чтобы они порождали команды лексической адресации. В случаях, когда `find-variable` возвращает `not-found` (то есть, когда переменной нет в окружении времени компиляции), нужно заставлять генераторы кода использовать, как и раньше, операции вычислителя для поиска связывания. (Единственное место, где может оказаться переменная, не найденная во время компиляции — это глобальное окружение, которое является частью окружения времени выполнения, но не окружения времени компиляции[47]. Поэтому, если хотите, можете заставить операции вычислителя искать сразу в глобальном окружении, которое можно получить с помощью операции `(op get-global-environment)`, а не в полном локальном окружении, которое хранится в `env`.) Проверьте измененный компилятор на нескольких простых примерах, например, на вложенной комбинации `lambda` из начала этого раздела.

Упражнение 5.43.
В разделе 4.1.6 мы показали, что определения внутри блочной структуры не следует рассматривать как «настоящие» `define`. Вместо этого тело процедуры следует интерпретировать так, как будто внутренние переменные, определяемые через `define`, были введены как обыкновенные переменные `lambda`, а их настоящее значение было им присвоено через `set!`. В разделе 4.1.6 и упражнении 4.16 показывалось, как можно изменить метациклический интерпретатор и добиться этого просмотром внутренних определений. Измените компилятор так, чтобы он проводил такое же преобразование, прежде чем компилировать тело процедуры.

Упражнение 5.44.
В этом разделе мы в основном говорили о том, как с помощью окружения времени компиляции порождать лексические адреса. Однако такие окружения можно использовать и другими способами. Например, в упражнении 5.38 мы повысили эффективность скомпилированного кода путем явного кодирования элементарных процедур. Наша реализация обрабатывала имена явно кодируемых процедур как зарезервированные слова. Если бы какая-либо программа переопределяла такое имя, механизм, описанный в упражнении 5.38, продолжал бы явно кодировать его как примитив и игнорировал бы новое связывание. Рассмотрим, например, процедуру

[47] Для доступа к переменным в глобальном окружении нельзя использовать лексические адреса, поскольку эти имена можно определять и переопределять интерактивно когда угодно. Если внутренние определения вычищены, как в упражнении 5.43, то компилятор видит только определения верхнего уровня, которые действуют на глобальное окружение. Компиляция определения не приводит к тому, что определяемое имя вводится в окружение времени компиляции.

```
(lambda (+ * a b x y)
  (+ (* a x) (* b y)))
```

которая вычисляет линейную комбинацию `x` и `y`. Мы могли бы вызвать такую процедуру с аргументами `+matrix`, `*matrix` и четырьмя матрицами, но явно кодирующий компилятор по-прежнему вставлял бы код для `+` и `*` в `(+ (* a x) (* b y))` как для примитивов `+` и `*`. Измените компилятор с явным кодированием так, чтобы он проверял окружение времени компиляции и на его основе порождал правильный код для выражений, в которых встречаются имена элементарных процедур. (Код будет работать правильно, пока программа не применяет к этим именам `define` или `set!`.)

5.5.7. Связь скомпилированного кода с вычислителем

Пока что мы не объяснили, как загружать скомпилированный код в машину-вычислитель и как его запускать. Предположим, что машина-вычислитель с явным управлением определена как в разделе 5.4.4 с дополнительными операциями из примечания 38. Мы реализуем процедуру `compile-and-go`, которая компилирует выражение на Scheme, загружает получившийся код в машину-вычислитель, а затем заставляет машину исполнить код в глобальном окружении вычислителя, напечатать результат и войти в управляющий цикл. Вычислитель мы изменим так, чтобы интерпретируемые выражения могли вызывать не только интерпретируемые, но и скомпилированные процедуры. После этого мы можем поместить скомпилированную процедуру в машину и вызвать ее с помощью интерпретатора:

```
(compile-and-go
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
;;; Значение EC-Eval:
ok

;;; Ввод EC-Eval:
(factorial 5)
;;; Значение EC-Eval:
120
```

Для того, чтобы вычислитель мог обрабатывать скомпилированные процедуры (например, выполнить вызов `factorial`, как показано выше), нужно изменить код `apply-dispatch` (раздел 5.4.1), чтобы он распознавал их (в отличие от составных и элементарных процедур) и передавал управление прямо во входную точку скомпилированного кода[48]:

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
```

[48]Разумеется, скомпилированные процедуры являются составными (неэлементарными) точно так же, как и интерпретируемые. Однако ради совместимости с терминологией, которая используется при обсуждении вычислителя с явным управлением, мы в этом разделе будем считать, что слово «составная» означает «интерпретируемая» (а не скомпилированная).

```
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (test (op compiled-procedure?) (reg proc))
  (branch (label compiled-apply))
  (goto (label unknown-procedure-type))

compiled-apply
  (restore continue)
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
```

Обратите внимание на восстановление `continue` в `compiled-apply`. Вспомним, что вычислитель был так устроен, что при выполнении `apply-dispatch` продолжение находилось на вершине стека. С другой стороны, входная точка скомпилированного кода ожидает, что продолжение будет находиться в `continue`, так что этот регистр надо восстановить, прежде чем передать управление скомпилированному коду.

Для того, чтобы позволить нам запускать скомпилированный код при запуске вычислителя, мы в начало машины добавляем команду `branch`, которая переводит машину на новую точку входа, если установлен регистр `flag`[49].

```
  (branch (label external-entry)) ; переход, если установлен flag
read-eval-print-loop
  (perform (op initialize-stack))
  ...
```

`External-entry` предполагает, что при запуске машины регистр `val` содержит местоположение последовательности команд, которая помещает результат в `val` и заканчивается командой `(goto (reg continue))`. Запуск машины с этой точки входа приводит к переходу в место, куда показывает `val`, но сначала `continue` устанавливается в `print-result`, которая распечатает значение `val`, а затем направится в начало управляющего цикла вычислителя[50].

---

[49] Теперь, когда код вычислителя начинается с `branch`, нам перед запуском машины всегда нужно устанавливать значение `flag`. Для того, чтобы запустить машину в обычном управляющем цикле, можно использовать

```
(define (start-eceval)
  (set! the-global-environment (setup-environment))
  (set-register-contents! eceval 'flag false)
  (start eceval))
```

[50] Поскольку скомпилированная процедура является объектом, который система может попытаться напечатать, нужно еще изменить системную операцию печати `user-print` (из раздела 4.1.4), чтобы она не пыталась печатать компоненты скомпилированной процедуры:

```
(define (user-print object)
  (cond ((compound-procedure? object)
         (display (list 'compound-procedure
                        (procedure-parameters object)
                        (procedure-body object)
                        '<procedure-env>)))
        ((compiled-procedure? object)
```

```
external-entry
  (perform (op initialize-stack))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (reg val))
```

Теперь с помощью следующей процедуры можно скомпилировать определение, выполнить скомпилированный код и войти в управляющий цикл, откуда мы можем процедуру оттестировать. Поскольку мы хотим, чтобы скомпилированная процедура возвращалась в место, указанное `continue`, со значением в `val`, мы компилируем выражение с целевым регистром `val` и типом связи `return`. Чтобы преобразовать объектный код, порождаемый компилятором, в исполняемые команды регистровой машины-вычислителя, мы пользуемся процедурой `assemble` из имитатора регистровых машин (раздел 5.2.2). Затем мы устанавливаем `val`, чтобы он указывал на список команд, устанавливаем `flag`, чтобы вычислитель пошел на `external-entry`, и запускаем вычислитель.

```
(define (compile-and-go expression)
  (let ((instructions
         (assemble (statements
                    (compile expression 'val 'return))
                   eceval)))
    (set! the-global-environment (setup-environment))
    (set-register-contents! eceval 'val instructions)
    (set-register-contents! eceval 'flag true)
    (start eceval)))
```

Если мы установим отслеживание стека, как в конце раздела 5.4.4, то сможем исследовать использование стека скомпилированным кодом:

```
(compile-and-go
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))

(total-pushes = 0 maximum-depth = 0)
;;; Значение EC-Eval:
ok

;;; Ввод EC-Eval:
(factorial 5)
(total-pushes = 31 maximum-depth = 14)
;;; Значение EC-Eval:
120
```

```
         (display '<compiled-procedure>))
        (else (display object))))
```

Сравните этот пример с вычислением `(factorial 5)` с помощью интерпретируемой версии той же самой процедуры, приведенным в конце раздела 5.4.4. В интерпретируемой версии потребовалось 144 сохранения и максимальная глубина стека 28. Это показывает, какую оптимизацию удалось получить с помощью нашей стратегии компиляции.

Интерпретация и компиляция

При помощи программ из этого раздела мы можем проводить эксперименты с различными стратегиями выполнения: интерпретацией и компиляцией[51]. Интерпретатор поднимает машину до уровня пользовательской программы; компилятор опускает пользовательскую программу до уровня машинного языка. Можно рассматривать язык Scheme (или любой язык программирования) как согласованную систему абстракций, построенных поверх машинного языка. Интерпретаторы хороши для интерактивной разработки программ и их отладки, поскольку шаги выполнения программы организованы в терминах этих абстракций, и, следовательно, лучше понятны программисту. Скомпилированный код может работать быстрее, поскольку шаги выполнения программы организованы в терминах машинного языка, и компилятор может проводить оптимизации, которые нарушают абстракции верхнего уровня[52].

Компиляция и интерпретация также ведут к различным стратегиям при переносе языков на новые компьютеры. Предположим, что нам надо реализовать Лисп для новой машины. Одна стратегия будет состоять в том, чтобы взять вычислитель с явным управлением из раздела 5.4 и перевести его команды в команды новой машины. Вторая — в том, чтобы начать с компилятора и изменить генераторы кода так, чтобы они порождали код новой машины. Вторая стратегия позволяет запускать на новой машине любую программу на Лиспе, если сначала скомпилировать ее компилятором, который работает на исходной Лисп-системе, а затем связать со скомпилированной версией рабочей библиотеки[53]. Более того, мы можем скомпилировать сам компилятор, и, запустив его на

[51] Можно добиться даже большего, если расширить компилятор так, чтобы скомпилированный код мог вызывать интерпретируемые процедуры. См. упражнение 5.47.

[52] Независимо от стратегии выполнения, мы сталкиваемся с существенным замедлением, если требуем, чтобы ошибки, возникающие при выполнении пользовательской программы, были обнаружены и отмечены, а не приводили к смерти системы или к неверным результатам работы. Например, индексирование за границы массива можно обнаружить, если перед обращением к массиву проверить правильность индекса. Однако затраты на проверку могут быть во много раз больше, чем стоимость самого обращения, и программисту приходится взвешивать преимущества скорости и безопасности, когда он решает, нужна ли такая проверка. Хороший компилятор должен уметь порождать код с проверками, избегать лишних проверок и позволять программистам управлять количеством и видами проверок на ошибки в скомпилированном коде.

Компиляторы для популярных языков программирования, вроде C или C++, почти никаких проверок в работающий код не помещают, чтобы программы выполнялись как можно быстрее. В результате программистам приходится самим проводить проверку на ошибки. К сожалению, многие этим пренебрегают, даже в критических приложениях, где скорость не является существенным ограничением. Их программы ведут бурную и опасную жизнь. Например, знаменитый «Червь», который парализовал Интернет в 1988 году, использовал то, что операционная система UNIX™ не проверяла переполнение буфера в демоне `finger`. (См. Spafford 1989.)

[53] Разумеется, как при интерпретации, так и при компиляции придется еще реализовать для новой машины управление памятью, ввод и вывод, а также все операции, которые мы считали «элементарными» при обсуждении интерпретатора и компилятора. Один из способов минимизировать эту работу заключается в том, чтобы как можно большее число этих операций написать на Лиспе, а затем скомпилировать для новой машины. В конце концов все сводится к простому ядру (например, сборка мусора и механизм для применения настоящих машинных примитивов), которое кодируется для новой машины вручную.

новой машине, компилировать другие программы на Лиспе[54]. Либо же мы можем скомпилировать один из интерпретаторов из раздела 4.1 и получить интерпретатор, который работает на новой машине.

Упражнение 5.45.
Сравнивая статистику операций со стеком, порождаемую скомпилированным кодом, с такой же статистикой для интерпретатора, ведущего то же самое вычисление, можно определить, насколько компилятор оптимизирует работу со стеком, как по скорости (уменьшая общее число стековых операций), так и по памяти (уменьшая максимальную глубину стека). Сравнение той же статистики с результатами работы специализированной машины, предназначенной для того же вычисления, дает некоторое представление о качестве компилятора.

а. В упражнении 5.27 от Вас требовалось определить как функцию от n число сохранений и максимальную глубину стека, которые требуются вычислителю для того, чтобы получить $n!$ с помощью указанной факториальной процедуры. В упражнении 5.14 Вас просили провести те же измерения для специализированной факториальной машины, показанной на рисунке 5.11. Проведите теперь тот же анализ для скомпилированной процедуры `factorial`.

Возьмем отношение числа сохранений в скомпилированной версии к числу сохранений в интерпретируемой версии и проделаем то же для максимальной глубины стека. Поскольку число операций и глубина стека при вычислении $n!$ линейно зависят от n, эти отношения должны приближаться к константам при росте n. Чему равны эти константы? Найдите также отношения показателей использования стека в специализированной машине к показателям интерпретируемой версии.

Сравните отношения специализированной версии к интерпретируемой и отношения скомпилированной версии к интерпретируемой. Вы должны увидеть, что специализированная машина работает намного лучше скомпилированного кода, поскольку настроенный вручную код контроллера должен быть намного лучше, чем результаты работы нашего рудиментарного компилятора общего назначения.

б. Можете ли Вы предложить изменения в компиляторе, помогающие ему порождать код, который приблизится к показателям версии, построенной вручную?

Упражнение 5.46.
Проведите анализ, подобный анализу из упражнения 5.45, и определите эффективность компиляции для процедуры Фибоначчи с древовидной рекурсией

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

по сравнению с эффективностью работы специализированной машины Фибоначчи с рисунка 5.12. (Измерения интерпретируемой версии см. в упражнении 5.29.) Для процедуры Фибоначчи время растет в нелинейной зависимости от n; следовательно, отношение числа стековых операций не будет приближаться к независимому от n пределу.

Упражнение 5.47.
В этом разделе рассказывалось, как модифицировать вычислитель с явным управлением и позволить интерпретируемому коду вызывать скомпилированные процедуры. Покажите, как можно из-

[54]Такая стратегия приводит к забавным проверкам корректности компилятора. Можно, например, сравнить, совпадает ли результат компиляции программы на новой машине, с помощью скомпилированного компилятора, с результатом компиляции той же программы на исходной Лисп-системе. Поиск причин расхождения — занятие интересное, но зачастую довольно сложное, поскольку результат может зависеть от микроскопических деталей.

менить компилятор и позволить скомпилированным процедурам вызывать не только элементарные и скомпилированные процедуры, но и интерпретируемые. При этом придется изменить `compile-procedure-call` так, чтобы обрабатывался случай составных (интерпретируемых) процедур. Проследите, чтобы обрабатывались все те же сочетания `target` и `linkage`, что и в `compile-proc-appl`. Для того, чтобы произвести собственно применение процедуры, код должен переходить на точку входа `compound-apply` вычислителя. На эту метку нельзя напрямую ссылаться из объектного кода (потому что ассемблер требует, чтобы все метки, на которые явно ссылается объектный код, в нем и определялись), так что придется добавить в машину-вычислитель специальный регистр `compapp`, в котором будет храниться эта точка входа, и команду для его инициализации:

```
  (assign compapp (label compound-apply))
  (branch (label external-entry)) ; переход, если установлен flag
read-eval-print-loop
  ...
```

Чтобы проверить свой код, для начала определите процедуру `f`, которая вызывает процедуру `g`. С помощью `compile-and-go` скомпилируйте определение `f` и запустите вычислитель. Теперь, вводя код для интерпретации, определите `g` и попробуйте вызвать `f`.

Упражнение 5.48.
Интерфейс `compile-and-go`, реализованный в этом разделе, неудобен, поскольку компилятор можно вызвать только один раз (при запуске машины-вычислителя). Дополните интерфейс между компилятором и интерпретатором, введя примитив `compile-and-run`, который можно будет вызывать из вычислителя с явным управлением так:

```
;;; Ввод EC-Eval:
(compile-and-run
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
;;; Значение EC-Eval:
ok

;;; Ввод EC-Eval:
(factorial 5)
;;; Значение EC-Eval:
120
```

Упражнение 5.49.
В качестве альтернативы циклу ввод-выполнение-печать вычислителя с явным управлением, спроектируйте регистровую машину, которая работала бы в цикле ввод-компиляция-выполнение-печать. А именно, машина должна работать в цикле, при каждой итерации которого она считывает выражение, компилирует его, ассемблирует и исполняет получившийся код, и печатает результат. В нашей имитируемой среде это легко устроить, поскольку мы можем заставить `compile` и `assemble` работать как «операции регистровой машины».

Упражнение 5.50.
С помощью компилятора оттранслируйте метациклический интерпретатор из раздела 4.1 и запустите эту программу через имитатор регистровых машин. (Чтобы скомпилировать за раз более

одного определения, можно упаковать определения в `begin`.) Получившийся интерпретатор будет работать очень медленно из-за многочисленных уровней интерпретации, однако заставить все детали работать — полезное упражнение.

Упражнение 5.51.
Разработайте рудиментарную реализацию Scheme на C (или на другом низкоуровневом языке по Вашему выбору), переведя на C вычислитель с явными управлением из раздела 5.4. Для того, чтобы запустить этот код, Вам потребуется предоставить также функции для выделения памяти и прочую поддержку времени выполнения.

Упражнение 5.52.
В качестве обратной задачи к упражнению 5.51, измените компилятор так, чтобы он компилировал процедуры Scheme в последовательности команд C. Скомпилируйте метациклический интерпретатор из раздела 4.1 и получите интерпретатор Scheme, написанный на C.

Литература

Abelson, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman, and Jack Wisdom. 1992. The Supercomputer Toolkit: A general framework for special-purpose computing. *International Journal of High-Speed Electronics* 3(3):337-361.

Allen, John. 1978. *Anatomy of Lisp.* New York: McGraw-Hill.

ANSI X3.226-1994. *American National Standard for Information Systems—Programming Language—Common Lisp.*

Appel, Andrew W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25(4):275-279.

Backus, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8):613–641.

Baker, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4):280-293.

Batali, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman, and Daniel Weise. 1982. The Scheme-81 architecture—System and chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI,* edited by Paul Penfield, Jr. Dedham, MA: Artech House.

Borning, Alan. 1977. ThingLab—An object-oriented system for building simula-tions using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence.*

Borodin, Alan, and Ian Munro. 1975. *The Computational Complexity of Algebraic and Numeric Problems.* New York: American Elsevier.

Chaitin, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5):47–52.

Church, Alonzo. 1941. *The Calculi of Lambda-Conversion.* Princeton, N.J.: Princeton University Press.

Clark, Keith L. 1978. Negation as failure. In *Logic and Data Bases.* New York: Plenum

Press, pp. 293–322.

Clinger, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Symposium on Lisp and Functional Programming,* pp. 226–234.

Clinger, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages,* pp. 155–162.

Colmerauer A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy.

Cormen, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms.* Cambridge, MA: MIT Press. (Русский перевод: Кормен Т., Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. – М.: МЦНМО, 2001.)

Darlington, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications.* New York: Cambridge University Press.

Dijkstra, Edsger W. 1968a. The structure of the "THE" multiprogramming system. *Communications of the ACM* 11(5):341–346.

Dijkstra, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43–112. (Русский перевод: Дийкстра Э. Взаимодействие последовательных процессов. // в кн.: Языки программирования, под ред. Ф. Женюи. – М.: Мир, 1972, с. 9–86.)

Dinesman, Howard P. 1968. *Superior Mathematical Puzzles.* New York: Simon and Schuster.

deKleer, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages,* pp. 116–125.

Doyle, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12:231–272.

Feigenbaum, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems,* vol. 9, pp. 105–117.

Feeley, Marc. 1986. Deux approches à l'implantation du language Scheme. Masters thesis, Université de Montréal.

Feeley, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1):47–66.

Feller, William. 1957. *An Introduction to Probability Theory and Its Applications,* volume 1. New York: John Wiley & Sons. (Русский перевод: Феллер В. Введение в теорию вероятностей и её приложения, в 2-х томах. – М.: Мир, 1984)

Fenichel, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11):611–612.

Floyd, Robert. 1967. Nondeterministic algorithms. *JACM,* 14(4):636–644.

Forbus, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press.

Friedman, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium,* edited by S. Michaelson and R. Milner, pp. 257–284.

Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages*. Cambridge, MA: MIT Press/McGraw-Hill.

Gabriel, Richard P. 1988. The Why of *Y. Lisp Pointers* 2(2):15–25.

Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley.

Gordon, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag.

Gray, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models*. San Mateo, CA: Morgan-Kaufman.

Green, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 219–240.

Green, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the ACM National Conference,* pp. 169–181.

Griss, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah.

Guttag, John V. 1977. Abstract data types and the development of data structures. *Communications of the ACM* 20(6):397–404.

Hamming, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, N.J.: Prentice-Hall.

Hanson, Christopher P. 1990. Efficient stack allocation for tail-recursive languages. In *Proceedings of ACM Conference on Lisp and Functional Programming,* pp. 106–118.

Hanson, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers,* 4(3).

Hardy, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2).

Hardy, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers*. 4th edition. New York: Oxford University Press.

Havender, J. 1968. Avoiding deadlocks in multi-tasking systems. *IBM Systems Journal* 7(2):74–84.

Hearn, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University.

Henderson, Peter. 1980. *Functional Programming: Application and Implementation.* Englewood Cliffs, N.J.: Prentice-Hall. (Русский перевод: П. Хендерсон. Функциональное программирование. – М.: Мир, 1983.)

Henderson. Peter. 1982. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pp. 179-187.

Hewitt, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 295–301.

Hewitt, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3):323–364.

Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1).

Hodges, Andrew. 1983. *Alan Turing: The Enigma.* New York: Simon and Schuster.

Hofstadter, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid.* New York: Basic Books. (Русский перевод: Хофштадтер Д. Гёдель, Эшер, Бах: эта бесконечная гирлянда. – Самара: Бахрах, 2001.)

Hughes, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17–42.

IEEE Std 1178-1990. 1990. *IEEE Standard for the Scheme Programming Language.*

Ingerman, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Неопубликованная рукопись. (А также частное сообщение от Уоллеса Ферцейга.)

Kaldewaij, Anne. 1990. *Programming: The Derivation of Algorithms.* New York: Prentice-Hall.

Kohlbecker, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University.

Konopasek, Milos, and Sundaresan Jayaraman. 1984. *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education.* Berkeley, CA: Osborne/McGraw-Hill.

Knuth, Donald E. 1973. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming.* 2nd edition. Reading, MA: Addison-Wesley. (Русский перевод: Кнут Д. Ис-

кусство программирования для ЭВМ. Т. 1.: Основные алгоритмы. – СПб.: Вильямс, 2000.)

Knuth, Donald E. 1981. *Seminumerical Algorithms*. Volume 2 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley. (Русский перевод: Кнут Д. Искусство программирования для ЭВМ. Т. 2. Получисленные алгоритмы. – СПб.: Вильямс, 2000.)

Kowalski, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh.

Kowalski, Robert. 1979. *Logic for Problem Solving*. New York: North-Holland. (Русский перевод: Ковальски Р. Логика в решении проблем. – М.: Наука, 1990.)

Lamport, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558–565.

Lampson, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek. 1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto.

Landin, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101.

Lieberman, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6):419–429.

Liskov, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* 1(1):7–19.

McAllester, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory.

McAllester, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory.

McCarthy, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4):184–195.

McCarthy, John. 1967. A basis for a mathematical theory of computation. In *Computer Programing and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland.

McCarthy, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages*.

McCarthy, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual*. 2nd edition. Cambridge, MA: MIT Press.

McDermott, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259,

MIT Artificial Intelligence Laboratory.

Miller, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3):300–317.

Miller, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory.

Moon, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science.

Moon, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory.

Morris, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages.*

Phillips, Hubert. 1934. *The Sphinx Problem Book*. London: Faber and Faber.

Pitman, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science.

Rabin, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12:128–138.

Raymond, Eric. 1993. *The New Hacker's Dictionary.* 2nd edition. Cambridge, MA: MIT Press. (Русский перевод: Рэймонд Э. Новый словарь хакера. – М.: ЦентрКом, 1996.)

Raynal, Michel. 1986. *Algorithms for Mutual Exclusion.* Cambridge, MA: MIT Press.

Rees, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pp. 114–122.

Rees, Jonathan, and William Clinger (eds). 1991. The revised^4 report on the algorithmic language Scheme. *Lisp Pointers,* 4(3).

Rivest, Ronald, Adi Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LCS/TM82, MIT Laboratory for Computer Science.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1):23.

Robinson, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1:107–124. (Русский перевод: Робинсон Дж. Логическое программирование – прошлое, настоящее и будущее. // В кн. Логическое программирование. – М.: Мир, 1988, с. 7–26.)

Spafford, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the ACM* 32(6):678–688.

Steele, Guy Lewis, Jr. 1977. Debunking the “expensive procedure call” myth. In *Proceedings of the National Conference of the ACM,* pp. 153–62.

Steele, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming,* pp. 98–107.

Steele, Guy Lewis, Jr. 1990. *Common Lisp: The Language*. 2nd edition. Digital Press.

Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory.

Steele, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker’s Dictionary*. New York: Harper & Row.

Stoy, Joseph E. 1977. *Denotational Semantics*. Cambridge, MA: MIT Press.

Sussman, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided circuit analysis. *IEEE Transactions on Circuits and Systems* CAS-22(11):857–865.

Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints – A language for expressing almost-hierachical descriptions. *AI Journal* 14:1–39.

Sussman, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257:256-262.

Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory.

Sutherland, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory.

Teitelman, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center.

Thatcher, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 119–132.

Turner, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics,* Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334–348.

Wand, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):164–180.

Waters, Richard C. 1979. A method for analyzing loop programs. *IEEE Transactions on*

Software Engineering 5(3):237–247.

Winograd, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory.

Winston, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley.

Zabih, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. *AAAI-87*, pp. 59–64.

Zippel, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT.

Zippel, Richard. 1993. *Effective Polynomial Computation*. Boston, MA: Kluwer Academic Publishers.

Предметный указатель

Все неточности в этом указателе объясняются тем, что его готовили при помощи вычислительной машины.

Дональд Э. Кнут, *Основные алгоритмы* (*Исскусство программирования для ЭВМ*, том 1)

Номера страниц для определений процедур даны курсивом.
Буква *n* после номера страницы отсылает к примечанию.
Буквы *нс* после названия элементарной функции либо особой формы означают, что она не входит в стандарт Scheme IEEE.

" (двойная кавычка) 147*n*
` (обратная кавычка) 524*n*
' (одинарная кавычка) 147*n*
  и read 356*n*, 444*n*
< (элементарный предикат сравнения чисел) *36*
!, в именах процедур 214*n*
* (элементарная процедура умножения) *26*
+ (элементарная процедура сложения) *26*
, (запятая, внутри обратной кавычки) 524*n*
- (элементарная процедура вычитания) *26*
  как смена знака *36n*
/ (элементарная процедура деления) *26*
; *см.* точка с запятой
= (элементарный предикат сравнения чисел) *36*
=number? *153*
=zero? (обобщенная) 191 (упр. 2.80)
  для многочленов 204 (упр. 2.87)
> (элементарный предикат сравнения чисел) *36*
>= (элементарный предикат сравнения чисел) *38*
?, в именах предикатов 41*n*
$\mapsto$, математическая запись для функций 81*n*
λ-исчисление *см.* лямбда-исчисление
π *см.* пи
$\sum$ *см.* сигма-запись
$\Theta(f(n))$ *см.* тета от $f(n)$

abs *36*, *37*
accelerated-sequence *316*
accumulate 74 (упр. 1.32), *123*
  то же, что fold-right 127 (упр. 2.38)
accumulate-n 126 (упр. 2.36)
actual-value *373*
Ada (Ада)
  рекурсивные процедуры 51
add (обобщенная) *187*
  примененная к коэффициентам многочленов 202

add-action! 264, *266*
add-binding-to-frame! *352*
add-complex *173*
add-complex-to-schemenum *191*
add-interval *103*
add-lists *380*
add-poly *200*
add-rat *94*
add-rule-or-assertion! *441*
add-streams *309*
add-terms *201*
add-to-agenda! 267, *270*
add-vect 141 (упр. 2.46)
addend 150, *152*
adder (элементарное ограничение) *275*
adjoin-arg *504n*
adjoin-set 154
  для множеств взвешенных элементов *168*
  представление в виде бинарных деревьев *159*
  представление в виде неупорядоченных списков *155*
  представление в виде упорядоченных списков 157 (упр. 2.61)
adjoin-term 201, *204*
advance-pc *481*
after-delay 264, *267*
Algol (Алгол)
  бедность средств работы с составными объектами 281*n*
  блочная структура 47
  передача аргументов по имени [call by name] 306*n*, 372*n*
  санки 306*n*, 372*n*
all-regs (компилятор) *535n*
always-true *434*
amb *383*
ambeval *396*
an-element-of *383*
an-integer-starting-from *383*
analyze
  метациклическая *366*
  недетерминистская 396
analyze-amb *401*
analyze-...
  метациклические 366, 369 (упр. 4.23)
  недетерминистские 397
and (особая форма) 37
  без подвыражений 348 (упр. 4.4)
  вычисление 37
  почему особая форма 38
and (язык запросов) 411
  обработка 419, 432, 448 (упр. 4.76)
and-gate *264*
angle
  декартово представление *174*
  полярное представление *175*
  с помеченными данными *178*
  управляемая данными *182*
angle-polar *177*
angle-rectangular *176*
announce-output *356*
APL 125*n*
append 110, *110*, *244* (упр. 3.12)
  vs. append! 244 (упр. 3.12)
  как накопление 125 (упр. 2.33)
  как регистровая машина 494 (упр. 5.22)
  с произвольным числом аргументов 538*n*
  «что такое» (правила) или «как сделать» (процедура) 405
append! *245* (упр. 3.12)
  как регистровая машина 494 (упр. 5.22)
append-instruction-sequences 521, *537*
append-to-form (правила) *415*
application? *346*
apply (ленивая) *373*
apply (метациклическая) *341*
  vs. элементарная apply 355*n*
apply (элементарная процедура) *182n*
apply-dispatch *506*
  с учетом скомпилированных процедур *551*
apply-generic *182*
  с башней типов 194
  с приведением *193*, 197 (упр. 2.81)
  с приведением нескольких аргументов 197 (упр. 2.82)
  с приведением через последовательный подъем 198 (упр. 2.84)
  с упрощением типа 198 (упр. 2.85)

через передачу сообщений *185*
`apply-primitive-procedure` 341, 351, *356*
`apply-rules` *436*
`argl`, регистр 501
`articles` *389*
`assemble` *476*, *478n*
`assert!` (интерпретатор запросов) 424
`assign` (в регистровой машине) 455
имитация 481
сохранение метки в регистре 461
`assign-reg-name` *481*
`assign-value-exp` *481*
`assignment-value` *344*
`assignment-variable` *344*
`assignment?` *344*
`assoc` *256*
`atan` (элементарная процедура) *174n*
`attach-tag` *176*
использование типов Scheme 191 (упр. 2.78)
`augend` 150, *152*
`average` *41*
`average-damp` *83*
`averager` (ограничение) 279 (упр. 3.33)

B-деревья [B-trees] 160*n*
Basic (Бейсик)
бедность средств работы с составными объектами 281*n*
ограничения на составные данные 107*n*
`begin-actions` *346*
`begin?` *346*
`below` 134, 145 (упр. 2.51)
`beside` 133, *144*
`branch` (в регистровой машине) 454
имитация 482
`branch-dest` *482*

C (Си)
интерпретатор Scheme, написанный на Си 557 (упр. 5.51), 557 (упр. 5.52)
компиляция процедур Scheme в команды Си 557 (упр. 5.52)
обработка ошибок 516*n*, 554*n*
ограничения на составные данные 107*n*
рекурсивные процедуры 51
`cadr` 108*n*
`ca...r` 108*n*
`call-each` *266*
`car` (элементарная процедура) 95
как операция над списком 108
описывающая аксиома 100
происхождение имени 95*n*
процедурная реализация *101*, *102* (упр. 2.4), *249*, *379*
реализация с мутаторами *249*
реализация через векторы 492
свойство замыкания *106*
`cd...r` 108*n*
`cdr` (элементарная процедура) 95
как операция над списком 108
описывающая аксиома 100
происхождение имени 95*n*
процедурная реализация *101*, 102 (упр. 2.4), *249*, *379*
реализация с мутаторами *249*
реализация через векторы 492
`celsius-fahrenheit-converter` *273*
в формате выражения *280* (упр. 3.37)
`center` *104*
`cesaro-stream` *330*
`cesaro-test` *219*
`coeff` 201, *204*
Common Lisp 24*n*
трактовка `nil` 109*n*
`compile` *520*
`compile-and-go` 551, *553*
`compile-and-run` 556 (упр. 5.48)
`compile-application` *530*
`compile-assignment` *526*
`compile-definition` *526*
`compile-if` *527*
`compile-lambda` *529*
`compile-linkage` *524*
`compile-proc-appl` *535*
`compile-procedure-call` *533*
`compile-quoted` *525*
`compile-self-evaluating` *525*
`compile-sequence` *528*
`compile-variable` *525*
`compiled-apply` *552*

`compiled-procedure` *529n*
`compiled-procedure-entry` *529n*
`compiled-procedure-env` *529n*
`complex`, пакет *189*
`complex->complex` *197* (упр. 2.81)
`compound-apply` *506*
`compound-procedure?` *351*
`cond` (особая форма) *36*
  vs. `if` 37*n*
  вариант синтаксиса ветвей *349* (упр. 4.5)
  ветвь 36
  вычисление 36
  неявный `begin` в следствиях 214*n*
`cond->if` *347*
`cond-actions` *347*
`cond-clauses` *347*
`cond-else-clause?` *347*
`cond-predicate` *347*
`cond?` *347*
`conjoin` *432*
`connect` 275, *279*
Conniver (Коннивер) 385*n*
`cons` (элементарная процедура) 95
  как операция над списком 109
  описывающие аксиомы 100
  происхождение имени 95*n*
  процедурная реализация *101*, *101* (упр. 2.4), *244*, *249*, *379*
  реализация с мутаторами *249*
  реализация через векторы 493
`cons-stream` (особая форма) 301, *302*
  и ленивые вычисления 379
  почему особая форма 303*n*
`const` (в регистровой машине) 455
  имитация 484
  синтаксис 471
`constant` (элементарное ограничение) *277*
`constant-exp-value` *484*
`constant-exp?` *484*
`construct-arglist` *531*
`contents` *176*
  использование типов Scheme 191 (упр. 2.78)
`continue`, регистр *461*
  в вычислителе с явным управлением *501*
  и рекурсия *467*
`corner-split` *137*
`cos` (элементарная процедура) *81*
`count-change` *56*
`count-leaves` 115, *117*
  как накопление 126 (упр. 2.35)
  как регистровая машина 494 (упр. 5.21)
`count-pairs` *248* (упр. 3.16)
`cube` *59* (упр. 1.15), *69*, *85*
`cube-root` *84*
`current-time` 267, *270*

`decode` *167*
`deep-reverse` 117 (упр. 2.27)
`define` (особая форма) *28*
  vs. `lambda` 75
  внутренняя *см.* внутренние определения
  для процедур *32*, 75
  значение выражения 28*n*
  модель с окружениями 231
  почему особая форма 30
  синтаксический сахар 344
  точечная запись 112 (упр. 2.20)
`define-variable!` 351, *353*
`definition-value` *345*
`definition-variable` *344*
`definition?` *344*
`delay` (особая форма) *302*
  и ленивые вычисления 379
  мемоизированная 305, 312 (упр. 3.57)
  почему особая форма 303*n*
  реализация с помощью `lambda` 305
  явная 325
  явная vs. автоматическая 380
`delay-it` *375*
`delete-queue!` 251, *253*
`denom` 94, *96*
  описывающая аксиома *100*
  с приведением к наименьшему знаменателю *99*
`deposit` (сообщение для банковского счета) *216*
`deposit`, с внешним сериализатором *292*
`deriv` (символическая) *151*
  управляемая данными *183* (упр. 2.73)
`deriv` (численная) *85*

`disjoin` *432*
`display` (элементарная процедура) 67 (упр. 1.22), *96n*
`display-line` *302*
`display-stream` *302*
`distinct?` *387n*
`div` (обобщенная) *187*
`div-complex` *174*
`div-interval` *103*
  деление на ноль 104 (упр. 2.10)
`div-poly` 205 (упр. 2.91)
`div-rat` *94*
`div-series` 314 (упр. 3.62)
`div-terms` *205* (упр. 2.91)
`divides?` *64*
`divisible?` *307*
DOS/Windows 516*n*
`dot-product` *127* (упр. 2.37)
`draw-line` 141
`driver-loop`
  для ленивого интерпретатора *374*
  для метациклического интерпретатора *356*
  для недетерминистского интерпретатора *402*

e
  как решение дифференциального уравнения 326
  как цепная дробь 83 (упр. 1.38)
e^x, степенной ряд 312 (упр. 3.59)
`edge1-frame` 140
`edge2-frame` 140
EIEIO 298*n*
`element-of-set?` 154
  представление в виде бинарных деревьев *159*
  представление в виде неупорядоченных списков *155*
  представление в виде упорядоченных списков *156*
`else` (особый символ в `cond`) *37*
`empty-agenda?` 267, *270*
`empty-arglist` *504n*
`empty-instruction-sequence` *523*
`empty-queue?` 251, *252*
`empty-termlist?` 201, *204*
`enclosing-environment` *352*
`encode` *169* (упр. 2.68)
`end-segment` 99 (упр. 2.2), 142 (упр. 2.48)
`end-with-linkage` *525*
`entry` *159*
`enumerate-interval` *123*
`enumerate-tree` *123*
`env`, регистр 501
`eq?` (элементарная процедура) *148*
  для произвольных объектов 247
  и равенство чисел 491*n*
  как равенство указателей 247, 492
  реализация для символов 492
`equ?` (обобщенный предикат) 191 (упр. 2.79)
`equal-rat?` *95*
`equal?` 149 (упр. 2.54)
`error` (элементарная процедура) *80n*
`estimate-integral` 221 (упр. 3.5)
`estimate-pi` *219*
`euler-transform` *316*
`ev-application` *504*
`ev-assignment` *510*
`ev-begin` *507*
`ev-definition` *511*
`ev-if` *510*
`ev-lambda` *503*
`ev-quoted` *503*
`ev-self-eval` *502*
`ev-sequence`
  без хвостовой рекурсии *509*
  с хвостовой рекурсией *508*
`ev-variable` *502*
`eval` (ленивая) 373
`eval` (метациклическая) 339, *340*
  vs. элементарная `eval` 359*n*
  анализирующий вариант *366*
  управляемая данными 348 (упр. 4.3)
`eval` (элементарная процедура) *359*
  MIT Scheme *360n*
  в интерпретаторе запросов *433*
`eval-assignment` *342*
`eval-definition` *343*
`eval-dispatch` *502*

`eval-if` (ленивая) *374*
`eval-if` (метациклическая) *342*
`eval-sequence` *342*
`even-fibs` *120*, *123*
`even?` *60*
`exchange` *292*
`execute` 475
`execute-application`
  метациклическая *368*
  недетерминистская *400*
`exp`, регистр 501
`expand-clauses` *347*
`expmod` *65*, *68* (упр. 1.25), *68* (упр. 1.26)
`expt`
  линейно итеративный вариант *60*
  линейно рекурсивный вариант *59*
  регистровая машина 470 (упр. 5.4)
`extend-environment` 351, *352*
`extend-if-consistent` *435*
`extend-if-possible` *439*
`external-entry` *553*
`extract-labels` *478*, *478n*

`#f` 36*n*
`factorial`
  использование стека, интерпретируемый вариант 515 (упр. 5.26), 515 (упр. 5.27)
  использование стека, регистровая машина 488 (упр. 5.14)
  использование стека, скомпилированный вариант 555 (упр. 5.45)
  как абстрактная машина 357
  компиляция 539, 544
  линейно итеративный вариант *50*
  линейно рекурсивный вариант *49*
  регистровая машина (итеративная) 454 (упр. 5.1), 456 (упр. 5.2)
  регистровая машина (рекурсивная) 466, 469
  с присваиванием *226*
  структура окружений при вычислении 233 (упр. 3.9)
  через процедуры высших порядков 73 (упр. 1.31)
`false` *36n*
`false?` *350*
`fast-expt` *60*
`fast-prime` *66*
`fermat-test` *66*
`fetch-assertions` *440*
`fetch-rules` *440*
`fib`
  древовидно-рекурсивный вариант *53*, *516* (упр. 5.29)
  использование стека, интерпретируемый вариант 516 (упр. 5.29)
  использование стека, скомпилированный вариант 555 (упр. 5.46)
  линейно-итеративный вариант 55
  логарифмический вариант 62 (упр. 1.19)
  регистровая машина (с древовидной рекурсией) 468, 470
  с именованным `let` *350* (упр. 4.8)
  с мемоизацией *260* (упр. 3.27)
`fibs` (бесконечный поток) *308*
  неявное определение *310*
FIFO 250
`filter` *122*
`filtered-accumulate` 74 (упр. 1.33)
`find-assertions` *434*
`find-divisor` *64*
`first-agenda-item` 267, *271*
`first-exp` *346*
`first-frame` *352*
`first-operand` *346*
`first-segment` *270*
`first-term` 201, *204*
`fixed-point` *80*
  как пошаговое улучшение 89 (упр. 1.46)
`fixed-point-of-transform` *86*
`flag`, регистр 475
`flatmap` *129*
`flatten-stream` *443*
`flip-horiz` 134, 144 (упр. 2.50)
`flip-vert` 134, *143*
`flipped-pairs` *136*, *139*, *139n*
`fold-left` *127* (упр. 2.38)
`fold-right` 127 (упр. 2.38)
`for-each` 114 (упр. 2.23), *377* (упр. 4.30)
`for-each-except` *279*
`force` 302, *305*

vs. вынуждение санка 372*n*
`force-it` *375*
вариант с мемоизацией *376*
`forget-value!` 275, *279*
Fortran (Фортран) 24, 125*n*
изобретатель 333*n*
ограничения на составные данные 107*n*
`frame-coord-map` *140*
`frame-values` *352*
`frame-variables` *352*
Franz Lisp (Франц Лисп) 24*n*
`free`, регистр 493, 496
`fringe` 118 (упр. 2.28)
как перечисление листьев дерева 123*n*
`front-ptr` *251*
`front-queue` 251, *252*
`full-adder` *263*

`gcd` *63*
регистровая машина 451, 472
`gcd-terms` *208*
`generate-huffman-tree` *169* (упр. 2.69)
`get` 180, *259*
`get-contents` *474*
`get-global-environment` *512n*
`get-register` *476*
`get-register-contents` 472, *475*
`get-signal` 264, *266*
`get-value` 275, *279*
`goto` (в регистровой машине) 454
имитация 482
переход на содержимое регистра 461
`goto-dest` *483*

`half-adder` *262*
`half-interval-method` *80*
`has-value?` 275, *279*
Hassle (Закавыка) 371*n*

IBM 704 95*n*
`identity` *72*
`if` (особая форма) *37*
vs. `cond` 37*n*
вычисление 37
нормальный порядок вычисления формы 39 (упр. 1.5)
одностороннее предложение (без альтернативы) 271*n*
почему особая форма 42 (упр. 1.6)
предикат, следствие и альтернатива 37
`if-alternative` *345*
`if-consequent` *345*
`if-predicate` *345*
`if?` *345*
`imag-part`
декартово представление *174*
полярное представление *175*
с помеченными данными *177*
управляемая данными *182*
`imag-part-polar` *177*
`imag-part-rectangular` *176*
`inc` *71*
`inform-about-no-value` *276*
`inform-about-value` *276*
`initialize-stack`, операция регистровой машины 474, 493
`insert!`
для двумерной таблицы *258*
для одномерной таблицы *256*
`insert-queue!` 251, *252*
`install-complex-package` *189*
`install-polar-package` *181*
`install-polynomial-package` *200*
`install-rational-package` *188*
`install-rectangular-package` *181*
`install-scheme-number-package` *187*
`instantiate` *431*
`instruction-execution-proc` *479*
`instruction-text` *479*
`integers` (бесконечный поток) *307*
вариант с ленивыми списками *380*
неявное определение *309*
`integers-starting-from` *307*
`integral` *72*, *322*, *326* (упр. 3.77)
вариант с ленивыми списками *380*
необходимость задержанного вычисления 325
с задержанным аргументом *325*
с использованием `lambda` *75*
`integrate-series` 313 (упр. 3.59)

`interleave` *319*
`interleave-delayed` *442*
InterLisp (ИнтерЛисп) 24*n*
`intersection-set` 154
  представление в виде бинарных деревьев 162 (упр. 2.65)
  представление в виде неупорядоченных списков *155*
  представление в виде упорядоченных списков *157*
`inverter` *264*

KRC 128*n*, 319*n*

`label` (в регистровой машине) 454
  имитация 484
`label-exp-label` *484*
`label-exp?` *484*
`lambda` (особая форма) *74*
  vs. `define` 75
  точечная запись 112*n*
`lambda-body` *345*
`lambda-parameters` *345*
`lambda-выражение`
  значение 231
`lambda?` *345*
`last-operand?` *504n*
`last-pair` 111 (упр. 2.17), *245* (упр. 3.12)
  правила 416 (упр. 4.62)
`leaf?` *166*
`left-branch` *159*, *167*
`length` 110
  итеративный вариант *110*
  как накопление 125 (упр. 2.33)
  рекурсивный вариант *110*
`let` (особая форма) *76*
  vs. внутреннее определение 78
  именованный *349* (упр. 4.8)
  как синтаксический сахар 77, 238 (упр. 3.10)
  модель вычисления 238 (упр. 3.10)
  область действия переменных 77
`let*` (особая форма) *349* (упр. 4.7)
`letrec` (особая форма) *363* (упр. 4.20)
`lexical-address-lookup` 548, 549 (упр. 5.39)
Lisp (Лисп)
  vs. Паскаль 31*n*
  vs. Фортран 24
  аппликативный порядок вычислений 35
  диалекты *см.* диалекты Лиспа
  история 23
  исходная реализация на IBM 704 95*n*
  на DEC PDP-1 495*n*
  процедуры как объекты первого класса 87
  система внутренних типов 191 (упр. 2.78)
  сокращение от LISt Processing 23
  удобство для написания вычислителей 337
  уникальные свойства 24
  эффективность 24, 27*n*
`lisp-value` (интерпретатор запросов) *433*
`lisp-value` (язык запросов) 412, 427
  вычисление 421, 433, 448 (упр. 4.77)
`list` (элементарная процедура) *108*
`list->tree` *161* (упр. 2.64)
`list-difference` *537*
`list-exp?` *346*
`list-of-arg-values` *374*
`list-of-delayed-args` *374*
`list-of-values` *341*
`list-ref` *109*, *379*
`list-union` *537*
`log` (элементарная процедура) *82* (упр. 1.36)
`logical-not` *264*
`lookup`
  в двумерной таблице *258*
  в множестве записей *162*
  в одномерной таблице *255*
`lookup-label` *479*
`lookup-prim` *485*
`lookup-variable-value` 351, *352*
  при исключенных внутренних определениях 362 (упр. 4.16)
`lower-bound` 103 (упр. 2.7)

machine *484*
Macintosh 516*n*
MacLisp (МакЛисп) 24*n*
magnitude
  декартово представление *174*
  полярное представление *175*
  с помеченными данными *177*
  управляемая данными *182*
magnitude-polar *177*
magnitude-rectangular *176*
make-account *216*
  в модели с окружениями 240 (упр. 3.11)
  с сериализацией *289*, *290* (упр. 3.41), *291* (упр. 3.42)
make-account-and-serializer *292*
make-accumulator 217 (упр. 3.1)
make-agenda 267, *270*
make-assign *481*
make-begin *346*
make-branch *118* (упр. 2.29), *482*
make-center-percent *104* (упр. 2.12)
make-center-width *104*
make-code-tree *166*
make-compiled-procedure *529n*
make-complex-from-mag-ang *190*
make-complex-from-real-imag *189*
make-connector *278*
make-cycle *245* (упр. 3.13)
make-decrementer *222*
make-execution-procedure *480*
make-frame 140, *141* (упр. 2.47), *352*
make-from-mag-ang *178*, *183*
  в виде передачи сообщений 185 (упр. 2.75)
  декартово представление *174*
  полярное представление *175*
make-from-mag-ang-polar *177*
make-from-mag-ang-rectangular *177*
make-from-real-imag *178*, *183*
  в виде передачи сообщений 185
  декартово представление *174*
  полярное представление *175*
make-from-real-imag-polar *177*
make-from-real-imag-rectangular *176*
make-goto *482*
make-if *345*
make-instruction *479*
make-instruction-sequence *523*
make-interval 103, *103* (упр. 2.7)
make-joint 226 (упр. 3.7)
make-label *527n*
make-label-entry *479*
make-lambda *345*
make-leaf *166*
make-leaf-set *168*
make-machine 472, *473*
make-monitored 217 (упр. 3.2)
make-mutex *295*
make-new-machine *477*
make-operation-exp *484*
make-perform *483*
make-point 99 (упр. 2.2)
make-poly *200*
make-polynomial *204*
make-procedure *351*
make-product 150, *151*
make-queue 251, *252*
make-rat 94, *96*, *98*
  описывающая аксиома *100*
  приведение к наименьшему знаменателю *97*
make-rational *188*
make-register *474*
make-restore *483*
make-save *483*
make-scheme-number *188*
make-segment 99 (упр. 2.2), 142 (упр. 2.48)
make-serializer *295*
make-simplified-withdraw *222*, *332*
make-stack *474*
  с отслеживанием стека *487*
make-sum 150, *151*
make-table
  одномерная таблица *256*
  реализация через передачу сообщений *258*
make-tableau *316*
make-term 201, *204*
make-test *482*
make-time-segment *269*

`make-tree` *159*
`make-vect` 141 (упр. 2.46)
`make-wire` 262, *265*, 269 (упр. 3.31)
`make-withdraw` *215*
  в модели с окружениями 234
  с использованием `let` *237* (упр. 3.10)
`map` *113*, *379*
  как накопление 125 (упр. 2.33)
  с несколькими аргументами 113*n*
`map-over-symbols` *444*
`map-successive-pairs` *330*
`matrix-*-matrix` 127 (упр. 2.37)
`matrix-*-vector` 127 (упр. 2.37)
`max` (элементарная процедура) *103*
MDL 496*n*
`member` 387*n*
`memo-fib` *260* (упр. 3.27)
`memo-proc` *305*
`memoize` *260* (упр. 3.27)
`memq` *148*
`merge` *311* (упр. 3.56)
`merge-weighted` 320 (упр. 3.70)
MicroPlanner (МикроПлэнер) 385*n*
`min` (элементарная процедура) *103*
Miranda (Миранда) 128*n*
MIT 405*n*
  Исследовательская лаборатория по Электронике 23, 495*n*
  лаборатория Искусственного Интеллекта 24*n*
  проект MAC 24*n*
  ранняя история 133*n*
MIT Scheme
  `eval` 360*n*
  `random` 221*n*
  `user-initial-environment` 360*n*
  `without-interrupts` 296*n*
  внутренние определения 362*n*
  пустой поток 301*n*
  числа 42*n*
ML 329*n*
`modifies-register?` *536*
modus ponens 425*n*
`monte-carlo` *219*
  бесконечный поток *331*
`mul` (обобщенная) *187*
  примененная к коэффициентам многочленов 202
`mul-complex` *174*
`mul-interval` *103*
  более эффективная версия *104* (упр. 2.11)
`mul-poly` *200*
`mul-rat` *94*
`mul-series` 313 (упр. 3.60)
`mul-streams` 311 (упр. 3.54)
`mul-terms` *202*
Multics, система разделения времени [Multics time-sharing system] 495*n*
`multiple-dwelling` *387*
`multiplicand` *152*
`multiplier`
  селектор *152*
  элементарное ограничение *276*
`mystery` *245* (упр. 3.14)

n-кратно сглаженная функция [n-fold smoothed function] 88 (упр. 1.44)
`needs-register?` *536*
`negate` *433*
`new`, регистр 498
`new-cars`, регистр 496
`new-cdrs`, регистр 496
`new-withdraw` *214*
`newline` (элементарная процедура) 67 (упр. 1.22), *96n*
`newton-transform` *85*
`newtons-method` *85*
`next` (описатель связи) 521
`next-to` (правила) *416* (упр. 4.61)
`nil`
  избавление от 148
  как обыкновенная переменная в Scheme 109*n*
  как показатель конца списка 107
  как пустой список 109
`no-more-exps?` *509n*
`no-operands?` *346*
`not` (элементарная процедура) *37*
`not` (язык запросов) 412, 427
  вычисление 421, 433, 448 (упр. 4.77)
`null?` (элементарная процедура) *110*

реализация через типизированные указатели 493
number? (элементарная процедура) *150*
и тип данных 191 (упр. 2.78)
реализация через типизированные указатели 493
numer 94, *96*
описывающая аксиома *100*
с приведением к наименьшему знаменателю *98*

old, регистр 498
oldcr, регистр 499
ones (бесконечный поток)
вариант с ленивыми списками *380*
op (в регистровой машине) 455
имитация 484
operands *183* (упр. 2.73), *346*
operation-exp-op *485*
operation-exp-operands *485*
operation-exp? *485*
operation-table *259*
operator *183* (упр. 2.73), *346*
or (особая форма) 37
без подвыражений 348 (упр. 4.4)
вычисление 37
почему особая форма 38
or (язык запросов) 411
обработка 420, 432
or-gate 265 (упр. 3.28), 265 (упр. 3.29)
order 201, *204*
origin-frame 140

P-операция на семафоре 294*n*
pair? (элементарная процедура) *117*
реализация через типизированные указатели 493
pairs *319*
parallel-execute 288
parallel-instruction-sequences *539*
parse *390*
parse-... 389
partial-sums 311 (упр. 3.55)
Pascal (Паскаль) 31*n*
бедность средств работы с составными объектами 281*n*
ограничения на составные данные 107*n*
рекурсивные процедуры 51
сложности с процедурами высших порядков 328*n*
pattern-match *435*
pc, регистр 475
perform (в регистровой машине) 457
имитация 483
perform-action *484*
permutations *129*
pi-stream *315*
pi-sum *71*
через lambda *75*
через процедуры высших порядков *72*
Planner (Плэнер) 385*n*
polar, пакет 181
polar? *176*
poly 200
polynomial, пакет 200
pop *475*
Portable Standard Lisp (Переносимый Стандартный Лисп) 24*n*
PowerPC 298*n*
prepositions *391*
preserving 521, 523 (упр. 5.31), *538*, 546 (упр. 5.37)
prime *310*
prime-sum-pair *382*
prime-sum-pairs *129*
бесконечный поток 318
prime? *64*
primes (бесконечный поток) *308*
неявное определение *310*
primitive-apply *506*
primitive-implementation *355*
primitive-procedure-names *355*
primitive-procedure-objects *355*
primitive-procedure? 351, *355*
print, операция в регистровой машине 457
print-point *99* (упр. 2.2)
print-queue 254 (упр. 3.21)
print-rat *96*
print-result *512*

с отслеживаемым количеством стековых операций 514
`probe`
в имитаторе цифровых схем *268*
в системе ограничений *277*
`proc`, регистр 501
`procedure-body` *351*
`procedure-environment` *351*
`procedure-parameters` *351*
`product` 73 (упр. 1.31)
как накопление 74 (упр. 1.32)
`product?` 150, *152*
Prolog (Пролог) 385*n*, 405*n*
`prompt-for-input` *356*
`propagate` *267*
`push` *475*
`put` 180, *259*

`qeval` 424, *431*
`queens` *130* (упр. 2.42)
`query-driver-loop` *430*
`quote` (особая форма) 147*n*
и `read` 356*n*, 444*n*
`quoted?` *344*
`quotient` (элементарная процедура) 312 (упр. 3.58)

`rand` *219*
со сбрасыванием 221 (упр. 3.6)
`random` (элементарная процедура) 66
MIT Scheme 221*n*
необходимость присваивания 213*n*
`random-in-range` *221* (упр. 3.5)
`rational-package` (пакет) *188*
RC-цепь [RC circuit] 322 (упр. 3.73)
`read` (элементарная процедура) *356n*
макросимволы ввода 444*n*
обработка точечной записи 436
`read`, операция регистровой машины 456
`read-eval-print` loop *512*
`real-part`
декартово представление *174*
полярное представление *175*
с помеченными данными *177*
управляемая данными *182*
`real-part-polar` *177*
`real-part-rectangular` *176*
`rear-ptr` *251*
`receive`, процедура 476
`rectangular`, пакет 181
`rectangular?` *176*
`reg` (в регистровой машине) 455
имитация 484
`register-exp-reg` *484*
`register-exp?` *484*
`registers-modified` *536*
`registers-needed` *536*
`remainder` (элементарная процедура) *60*
`remainder-terms` 208 (упр. 2.94)
`remove` *130*
`remove-first-agenda-item!` 267
`require` *383*
как особая форма 403 (упр. 4.54)
`rest-exps` *346*
`rest-operands` *346*
`rest-segments` *270*
`rest-terms` 201, *204*
`restore` (в регистровой машине) 467
моделирование 483
реализация 493
`return` (описатель связи) 521
`reverse` 111 (упр. 2.18)
как свертка 128 (упр. 2.39)
правила 429 (упр. 4.68)
`right-branch` *159*, *167*
`right-split` *136*
RLC-цепь последовательная [series RLC circuit] 328 (упр. 3.80)
`root`, регистр 496
`rotate90` *144*
`round` (элементарная процедура) *198n*
RSA алгоритм [RSA algorithm] 67*n*
`runtime` (элементарная процедура) *67* (упр. 1.22)

`same` (правило) *413*
`same-variable?` 150, *151*, 200
`save` (в регистровой машине) 467
моделирование 483
реализация 493
`scale-list` *112*, *113*, *380*
`scale-stream` *310*

`scale-tree` *119*
`scale-vect` 141 (упр. 2.46)
`scan`, регистр 496
`scan-out-defines` 362 (упр. 4.16)
Scheme (Схема) 24
  история 24*n*
`search` *79*
`segment-queue` *269*
`segment-time` *269*
`segments` *270*
`segments->painter` *142*
`self-evaluating?` *343*
`sequence->exp` *346*
`serialized-exchange` *293*
  с избежанием тупиков 297 (упр. 3.48)
`set!` (особая форма) 214, *см. также* присваивание
  значение 214*n*
  модель с окружениями 231*n*
`set-car!` *242*
`set-cdr!` *242*
`set-contents!` *474*
`set-current-time!` *270*
`set-front-ptr!` *251*
`set-instruction-execution-proc!` *479*
`set-rear-ptr!` *251*
`set-register-contents!` 472, *476*
`set-segments!` *270*
`set-signal!` 264, *266*
`set-value!` 275, *279*
`set-variable-value!` 351, *353*
`setup-environment` *354*
`shrink-to-upper-right` *143*
`signal-error` *513*
`simple-query` *431*
`sin` (элементарная процедура) *81*
`singleton-stream` *443*
SKETCHPAD 272*n*
`smallest-divisor` *64*
  более эффективный вариант 68 (упр. 1.23)
Smalltalk 272*n*
`solve` *325*, *326*
  вариант с ленивыми списками *380*
  с прочесанными внутренними определениями *363* (упр. 4.18)
`split` 139 (упр. 2.45)
`sqare` *31*
`sqare-limit` *137*, *139*
`sqrt` *42*
  в модели с окружениями *238*
  как неподвижная точка *81*, *84*, *86*
  как пошаговое улучшение 89 (упр. 1.46)
  как предел потока 317 (упр. 3.64)
  регистровая машина 459 (упр. 5.3)
  с блочной структурой *47*
  с методом Ньютона *86*
`sqrt-stream` *314*
`square`
  в модели с окружениями *229*
`square-of-four` *139*
`squarer` (ограничение) *280* (упр. 3.34), *280* (упр. 3.35)
`squash-inwards` *144*
`stack-inst-reg-name` *483*
`start` 472, *475*
`start-eceval` *552n*
`start-segment` 99 (упр. 2.2), 142 (упр. 2.48)
`statements` *536*
`stream-append` *319*
`stream-append-delayed` *442*
`stream-car` 301, *302*
`stream-cdr` 301, *302*
`stream-enumerate-interval` *303*
`stream-filter` *303*
`stream-flatmap` *443*, 447 (упр. 4.74)
`stream-for-each` *301*
`stream-limit` 317 (упр. 3.64)
`stream-map` *301*
  с несколькими аргументами *306* (упр. 3.50)
`stream-null?` 301
  в MIT Scheme 301*n*
`stream-ref` *301*
`stream-withdraw` *332*
`sub` (обобщенная) *187*
`sub-complex` *173*
`sub-interval` 103 (упр. 2.8)
`sub-rat` *94*

sub-vect 141 (упр. 2.46)
subsets *120* (упр. 2.32)
sum *71*
  итеративный вариант 73 (упр. 1.30)
  как накопление 74 (упр. 1.32)
sum-cubes *70*
  через процедуры высших порядков *71*
sum-integers *70*
  через процедуры высших порядков *72*
sum-odd-squares *120*, *123*
sum-of-squares *32*
  в модели с окружениями 231
sum-primes *300*
sum? 150, *152*
symbol-leaf *166*
symbol? (элементарная процедура) *151*
  и тип данных 191 (упр. 2.78)
  реализация через типизированные указатели 493
symbols *167*
SYNC 298*n*

#t 36*n*
tack-on-instruction-sequence *538*
tagged-list? *344*
term-list *201*
test (в регистровой машине) 454
  имитация 482
test-and-set! *295*, *296n*
test-condition *482*
text-of-quotation *344*
lambda-выражение
  как оператор в комбинации 75
THE, Система Мультипрограммирования 294*n*
the-cars
  вектор 490
  регистр 492, 496
the-cdrs
  вектор 490
  регистр 492, 496
the-empty-environment *352*
the-empty-stream 301
  в MIT Scheme 301*n*
the-empty-termlist 201, *204*
the-global-environment *355*, *512n*
thunk-env *375*
thunk-exp *375*
thunk? *375*
timed-prime-test *67* (упр. 1.22)
TK!Solver 272*n*
transform-painter *143*
transpose 127 (упр. 2.37)
tree->list... *161* (упр. 2.63)
tree-map 119 (упр. 2.31)
true *36n*
true? *350*
try-again *385*
type-tag *176*
  использование типов Scheme 191 (упр. 2.78)

unev, регистр 501
unify-match *437*
union-set 154
  представление в виде бинарных деревьев 162 (упр. 2.65)
  представление в виде неупорядоченных списков 156 (упр. 2.59)
  представление в виде упорядоченных списков 158 (упр. 2.62)
unique (язык запросов) 447 (упр. 4.75)
unique-pairs 130 (упр. 2.40)
UNIX (Юникс) 516*n*, 554*n*
unknown-expression-type *513*
unknown-procedure-type *513*
up-split 137 (упр. 2.44)
update-insts! *479*
upper-bound 103 (упр. 2.7)
user-initial-environment *360n*
user-print *356*
  измененная для скомпилированного кода *552n*

V-операция на семафоре 294*n*
val, регистр 501
value-proc 481
variable *201*
variable? 150, *343*
vector-ref (элементарная процедура) *490*

`vector-set!` (элементарная процедура) *490*
`verbs` *389*

`weight` *167*
`weight-leaf` *166*
`width` *104*
`withdraw` *213*
  сложности в параллельных системах *284*
`without-interrupts` 296*n*

`xcor-vect` 141 (упр. 2.46)
Xerox, исследовательский центр в Пало Альто [Xerox Palo Alto Research Center] 24*n*

Y-оператор [Y operator] 364*n*
`ycor-vect` 141 (упр. 2.46)

Zetalisp (Зеталисп) 24*n*

Абельсон, Харольд [Harold Abelson] 24*n*
абстрактные данные [abstract data] 93, *см. также* абстракция данных
абстрактные модели данных [abstract models for data] 100*n*
абстрактный синтаксис [abstract syntax]
  в метациклическом интерпретаторе 339
  в языке запросов 430
абстракция [abstraction] *см. также* средства абстракции; абстракция данных; процедуры высших порядков
  выделение общей схемы 71
  метаязыковая 336
  поиска в недетерминистском программировании 387
  при проектировании регистровых машин 457
  процедурная 44
абстракция данных [data abstraction] 91, 93, 170, 173, 343, *см. также* метациклический интерпретатор
  для очереди 250
автомагически [automagically] 384
автоматический поиск [automatic search] 381, *см. также* поиск
  история 384*n*
автоматическое распределение памяти [automatic storage allocation] 489
Ада, сыновья 417 (упр. 4.63)
Адамс, Норман И., IV [Norman I. Adams IV] 366*n*
аддитивность [additivity] 93, 171, 179
Адельман, Леонард [Leonard Adleman] 67*n*
адрес [address] 489
адресная арифметика [address arithmetic] 490
азбука Морзе [Morse code] 163
Аккермана функция [Ackermann's function] 52 (упр. 1.10)
алгебра символьная *см.* символьная алгебра
алгебраическая спецификация [algebraic specification] 100*n*
алгебраическое выражение [algebraic expression] 198
  дифференцирование 149
  представление 151
  упрощение 152
алгоритм [algorithm]
  RSA [RSA] 67*n*
  вероятностный 66
  Евклида [Euclid's] 63, 451
  оптимальный 125*n*
  унификации [unification algorithm] 405*n*
Аллен, Джон [John Allen] 495*n*
альтернатива `if` [alternative of `if`] 37
анализирующий интерпретатор [analyzing evaluator] 365
  `let` 369 (упр. 4.22)
  как основа для недетерминистского интерпретатора 394
Аппель, Эндрю У. [Andrew W. Appel] 535*n*
аппликативный порядок вычислений [applicative-order evaluation] 35
  vs. нормальный порядок 39 (упр. 1.5), 64 (упр. 1.20), 370
  в Лиспе 35
арбитр [arbiter] 296*n*
аргумент(ы) [argument(s)] 26
  задержанный 325

произвольное количество 26, 112 (упр. 2.20)
Аристотель, *De caelo* (комментарий Ж. Буридана) 296*n*
арифметика [arithmetic]
интервальная 102
комплексных чисел 171, *см.* арифметика комплексных чисел
многочленов *см.* арифметика многочленов
обобщенные операции 186
рациональная 93
элементарные процедуры 26
арифметика комплексных чисел [complex-number arithmetic] 171
взаимодействие с общими арифметическими системами 189
структура системы 178
арифметика многочленов [polynomial arithmetic] 199
алгоритм Евклида 207*n*
вероятностный алгоритм для НОД 210*n*
включение в систему обобщенной арифметики 200
вычитание 205 (упр. 2.88)
деление 205 (упр. 2.91)
наибольший общий делитель 207, 210*n*
рациональные функции 207
сложение 200
умножение 200
арктангенс [arctangent] 174*n*
ассемблер [assembler] 473, 476
атомарность [atomicity] 295
атомарные операции, поддерживаемые на уровне аппаратуры [atomic operations supported in hardware] 296
A'x-мосе [A'h-mose] 61*n*
Ачарья, Бхаскара [Bháscara Áchárya] 57*n*

база данных [data base]
и логическое программирование 407
и программирование, управляемое данными 184 (упр. 2.74)
индексирование 418*n*, 439
как множество записей 162
персонал компании Insatiable Enterprises, Inc. 184 (упр. 2.74)
персонал «Микрошафт» 407
базовый адрес [base address] 490
банковский счет [bank account] 213, 240 (упр. 3.11)
защищенный паролем 217 (упр. 3.3)
мена балансов местами 291
перенос денег 293 (упр. 3.44)
потоковая модель 332
сериализованный 289
совместный 225, 226 (упр. 3.7), 284
совместный, смоделированный с помощью потоков 334
Барт, Джон [John Barth] 335
барьерная синхронизация [barrier synchronization] 298*n*
барьеры абстракции [abstraction barriers] 92, 97, 170
в обобщенной арифметической системе 186
в системе работы с комплексными числами 171
Батали, Джон Дин [John Dean Batali] 500*n*
башня типов [tower of types] 194
Бейкер, Генри Дж., мл. [Henry J. Baker Jr.] 495*n*
Бертрана гипотеза [Bertrand's hypothesis] 311*n*
бесконечная последовательность [infinite series] 438*n*
бесконечные потоки [infinite streams] 307
простых чисел *см.* `primes`
целых чисел *см.* `integers`
чисел Фибоначчи *см.* `fibs`
бесконечный поток
для моделирования сигналов 321
для суммирования ряда 315
пар 318
представление степенных рядов 312 (упр. 3.59)
слияние 311 (упр. 3.56), 319, 320 (упр. 3.70), 334
слияние как отношение 334*n*
случайных чисел 330

факториалов 311 (упр. 3.54)
бинарное дерево [binary tree] 158
для кодирования по Хаффману 164
представленное при помощи списков 159
преобразование в список 161 (упр. 2.63)
преобразование из списка 161 (упр. 2.64)
списки, представленные как 158
таблицы 260 (упр. 3.26)
бинарный поиск [binary search] 158
биномиальный коэффициент [binomial coefficients] 57*n*
блоха [bug] 23
блочная структура [block structure] 47, 360
в модели окружения 238
в языке запросов 449 (упр. 4.79)
«Болт, Беранек и Ньюман» [Bolt, Beranek and Newman, inc.] 24*n*
большое число [bignum] 491
Борнинг, Алан [Alan Borning] 272*n*
Бородин, Алан [Alan Borodin] 126*n*
Буридан, Жан [Jean Buridan] 296*n*
буфер [buffer]
FIFO 250
LIFO *см.* стек
Бытие 417 (упр. 4.63)
Бэкус, Джон [John Backus] 333*n*
бюрократия [bureaucracy] 425

Вагнер, Эрик Дж. [Eric G. Wagner] 100*n*
Ванд, Митчелл [Mitchell Wand] 337*n*, 505*n*
Вейль, Герман [Hermann Weyl] 90
вектор (математический) [vector (mathematical)]
в кадре из языка описания изображений 140
операции 126 (упр. 2.37), 141 (упр. 2.46)
представленный в виде пары 141 (упр. 2.46)
представленный в виде последовательности 126 (упр. 2.37)
вектор (структура данных) [vector (data structure)] 490
Венера [Venus] 147*n*
вероятностный алгоритм [probabilistic algorithm] 66, 67, 210*n*, 308*n*
вершина дерева [node of a tree] 29
ветвь [clause] 36
ветвь `cond` [clause, of a `cond`] 36
дополнительный синтаксис 349 (упр. 4.5)
ветвь дерева [branch of a tree] 29
вечерняя звезда [evening star] *см.* Венера
взаимное исключение [mutual exclusion] 294*n*
взгляд на вычисления как на обработку сигналов [signal-processing view of computation] 121
Виноград, Терри [Terry Winograd] 385*n*
вложение комбинаций [nested combinations] 27
вложенные определения [nested definitions] *см.* внутренние определения
вложенные отображения [nested mappings] *см.* отображение
вложенные применения `car` и `cdr` [nested applications of `car` and `cdr`] 108*n*
внутреннее состояние [local state] 212
поддерживаемое в кадрах 234
внутренние определения [internal definitions] 46
vs. `let` 78
в модели с окружениями 238
в недетерминистском интерпретаторе 399*n*
ограничения 361*n*
позиция 47*n*
прочесывание и уничтожение 361
свободная переменная внутри 47
сфера действия имени 360
внутренний язык машины [native language of machine] 517
внутренняя переменная состояния [local state variable] 212, 213
возведение в степень [exponentiation] 59
по модулю n 65
возврат [backtracking] 384, *см. также* недетерминистские вычисления
возврат нескольких значений [returning multiple values] 478*n*

волшебник [wizard] *см.* специалист по численному анализу
вопросительный знак, в именах предикатов [question mark] 41*n*
восклицательный знак, в именах процедур [exclamation point] 214*n*
восприятие [interning] 492
временна́я диаграмма [timing diargam] 283
временной отрезок [time segment] 269
время [time] 281
  в недетерминистских вычислениях 382
  в недетерминистском вычислении 384
  в параллельных системах 284
  и взаимодействие процессов 298
  и присваивание 281
  и функциональное программирование 331
  предназначение 284*n*
встроенный язык, использование в разработке языков [embedded language] 369
выборки ключа, процедура [key selector] 162
вывод типов [type inference] 329*n*
выделение стека и хвостовая рекурсия [stack allocation and tail recursion] 535*n*
вызов по имени [call by name] 372*n*
вызов по необходимости [call by need] 306*n*, 372*n*
  связь с мемоизацией 312*n*
вынуждение санка [forcing a thunk] 372
выражение [expression] 26, *см. также* составные выражения; элементарные выражения
  алгебраическое *см.* алгебраическое выражение
  самовычисляющееся 340
  символьное 92, *см.также* символ(ы)
выражение-следствие [consequent expression]
  `cond` 36
  `if` 37
высокоуровневый язык vs. машинный язык 336
выход из тупика [deadlock recovery] 297*n*
вычисление [evaluation] 26
  `and` 37
  `cond` 36
  `if` 37
  `or` 37
  аппликативный порядок *см.* аппликативный порядок вычислений
  задержанное *см.* задержанные вычисления
  комбинации 29
  модели 354
  модель с окружениями *см.* модель вычисления с окружениями
  нормальный порядок *см.* нормальный порядок вычислений
  особых форм 31
  подстановочная модель *см.* подстановочная модель применения процедуры
  порядок вычисления подвыражений *см.* порядок вычисления
  элементарных выражений 30
вычислимость [computability] 359*n*, 360*n*
вычислители *см.* метациклический интерпретатор; анализирующий интерпретатор; недетерминистский интерпретатор; ленивый интерпретатор; интерпретатор запросов; вычислитель с явным управлением
вычислитель [evaluator] 336, *см. также* интерпретатор
  как универсальная машина 359
  метациклический 338
вычислитель с хвостовой рекурсией [tail-recursive evaluator] 508
вычислитель с нормальным порядком *см.* ленивый интерпретатор
вычислитель с явным управлением для Scheme [explicit-control evaluator for Scheme] 500
  выражения без подвыражений, подлежащих вычислению 502
  вычисление операндов 504
  запуск 512
  использование стека 503

как программа на машинном языке 517
как универсальная машина 517
комбинации 503
контроллер 502
модель машины 513
модифицированный для скомпилированного кода 551
нормальный порядок вычислений 511 (упр. 5.25)
обработка ошибок 512, 516 (упр. 5.30)
операции 501
определения 511
оптимизация 524 (упр. 5.32)
отслеживание производительности (использование стека) 514
последовательности выражений 507
применение процедур 503
присваивания 510
производные выражения 511 (упр. 5.23), 511 (упр. 5.24)
пути данных 501
регистры 501
составная процедура 506
управляющий цикл 512
условные выражения 510
хвостовая рекурсия 508, 515 (упр. 5.26), 516 (упр. 5.28)
элементарные процедуры 506
вычислительный процесс [computational process] 22, *см. также* процесс

Гаттэг, Джон Фогель [John Vogel Guttag] 100*n*
генератор кода [code generator] 520
аргументы 520
возвращаемое значение 521
генератор случайных чисел [random-number generator] 213*n*, 218
в моделировании методом Монте-Карло 219
в тесте на простоту 65
со сбрасыванием 221 (упр. 3.6)
со сбрасыванием, потоковая версия 331 (упр. 3.81)
Гераклит 211
Герон Александрийский 40*n*
гипотеза о замкнутости мира [closed world assumption] 428
глобальное окружение [global environment] 28, 228
в метациклическом интерпретаторе 354
глобальное поведение процесса [global behavior of a process] 48
глобальный кадр [global frame] 227
глюк [glitch] 23
Гоген, Джозеф [Joseph Goguen] 100*n*
Гордон, Майкл [Michael Gordon] 329*n*
Горнер, У.Дж. [W. J. Horner] 125 (упр. 2.34)
грамматика [grammar] 389
графика [graphics] *см.* язык описания изображений
Грей, Джим [Jim Gray] 297*n*
Грин, Корделл [Cordell Green] 405*n*
Грисс, Мартин Льюис [Martin Lewis Griss] 24*n*
Гэбриел, Ричард П. [Richard P. Gabriel] 364*n*

Дайнсман, Говард П. [Howard P. Dinesman] 387
данные [data] 22, 25, 100
абстрактные 93, *см. также* абстракция данных
абстрактные модели 100*n*
алгебраическая спецификация 100*n*
значение 100
иерархические 106, 115
изменяемые *см.* изменяемые объекты данных
как программы 357
«конкретное представление» 93
помеченные 175, 490*n*
процедурное представление 100
разделенные 246
символьные 146
со списковой структурой [list-structured] 96
составные 90
численные 25
Дарлингтон, Джон [John Darlington] 333*n*

двоичные числа, сложение [binary numbers, addition of] *см.* сумматор
де Клеер, Йохан [Johan deKleer] 385*n*, 427*n*
Дейкстра, Эдсгер Вибе [Edsger Wybe Dijkstra] 294*n*
действия, в регистровой машине [actions, in register machine] 457
дек [deque] 254 (упр. 3.23)
декларативное vs. императивное знание [declarative vs. imperative knowledge] 40, 404
  и логическое программирование 405, 425
  и недетерминистское вычисление 382*n*
декомпозиция программы [decomposition of program into parts] 44
деление целых чисел [division of integers] 42*n*
деньги, размен *см.* размен денег
дерево [tree] 115
  B-дерево 160*n*
  бинарное 158, *см. также* бинарное дерево
  красно-черное дерево 160*n*
  ленивое 380*n*
  листва 118 (упр. 2.28)
  обращение на всех уровнях 117 (упр. 2.27)
  отображение 119
  перечисление листьев 123
  подсчет числа листьев 115
  представление комбинации 29
  представленное в виде пар 115
  Хаффмана 164
десятичная точка в числах [decimal point in numbers] 42*n*
Джаяраман, Сундаресан [Sundaresan Jayaraman] 272*n*
диаграмма потока сигналов [signal-flow diagram] 121, 322 (упр. 3.73)
диалекты Лиспа [Lisp dialects]
  Common Lisp 24*n*
  Franz Lisp (Франц Лисп) 24*n*
  InterLisp (ИнтерЛисп) 24*n*
  MacLisp (МакЛисп) 24*n*
  MDL 496*n*
  Portable Standard Lisp (Переносимый Стандартный Лисп) 24*n*
  Scheme (Схема) 24
  Zetalisp (Зеталисп) 24*n*
Диофант, *Арифметика*; экземпляр Ферма 65*n*
диспетчеризация [dispatching]
  по типу [on type] 179
  сравнение различных стилей 186 (упр. 2.76)
диспетчирование
  по типу *см. также* программирование, управляемое данными
дисциплина кадрированного стека [framed-stack discipline] 503*n*
дифференциальное уравнение [differential equation] 324, *см. также* `solve`
  второго порядка 327 (упр. 3.78), 328 (упр. 3.79)
дифференцирование [differentiation]
  правила 150, 153 (упр. 2.56)
  символьное 149, 183 (упр. 2.73)
  численное 85
диффузия, имитация [imitation of diffusion] 287
Дойл, Джон [Jon Doyle] 385*n*
доказательство корректности программы [proving programs correct] 40*n*
доказательство теорем (автоматическое) [automatic theorem proving] 405*n*
древовидная рекурсия [tree recursion] 53
  порядок роста 58
древовидно-рекурсивное вычисление чисел Фибоначчи [tree-recursive Fibonacci-number computation] 53
дробь [fraction] *см.* рациональные числа

Евклид, *Начала* 63*n*
Евклида алгоритм [Euclid's Algorithm] 63, 451, *см. также* наибольший общий делитель
  для многочленов 207*n*
  порядок роста 63
единичный квадрат [unit square] 140
естественный язык [natural language]
  кавычки 146

синтаксический анализ *см.* синтаксический анализ естественного языка

`живет-около` (правило) *413*, 415 (упр. 4.60)

Заби, Рамин [Ramin Zabih] 385*n*
заблокированный процесс [blocked process] 295*n*
зависимость от реализации *см. также* неопределенные значения
порядок вычисления подвыражений 229*n*
числа 42*n*
загадки
задача о восьми ферзях 130 (упр. 2.42), 389 (упр. 4.44)
логические 387
задача о восьми ферзях 130 (упр. 2.42), 389 (упр. 4.44)
задержанные вычисления [delayed evaluation] 212, 299
в ленивом интерпретаторе 369
и нормальный порядок вычислений 328
и печать 306 (упр. 3.51)
и потоки 324
и присваивание 306 (упр. 3.52)
явные и автоматические 380
задержанный аргумент [delayed argument] 325
задержанный объект [delayed object] 302
задержка, в цифровой схеме [delay, in digital circuit] 261
замкнутости мира гипотеза *см.* гипотеза о замкнутости мира
замыкание [closure] 92
в абстрактной алгебре 106*n*
отсутствие во многих языках 107*n*
свойство замыкания `cons` 106
свойство замыкания языка описания картинок 132, 134
замыкания свойство [closure property] 106
занятое ожидание [busy waiting] 295*n*
запись, в базе данных [record, in a data base] 162
запрос [query] 406, *см. также* простой запрос; составной запрос
правило *см. также* правило (в языке запросов)
простой *см.* простой запрос
составной *см.* составной запрос
запятая внутри обратной кавычки [comma, used with backquote] 524*n*
зарезервированные слова [reserved words] 547 (упр. 5.38), 550 (упр. 5.44)
захват мьютекса [acquiring a mutex] 294
захват свободной переменной [free variable capture] 46
защищенный паролем банковский счет [password-protected bank account] 217 (упр. 3.3)
Земля, измерение длины окружности [measuring circumference of Earth] 308*n*
Зиллес, Стивен Н. [Stephen N. Zilles] 100*n*
Зиппель, Ричард Э. [Richard E. Zippel] 210*n*
значение [value]
выражения 27*n*, *см. также* неопределенные значения
комбинации 26
переменной 28
золотое сечение [golden ratio] 53
как неподвижная точка 82 (упр. 1.35)
как цепная дробь 82 (упр. 1.37)

И-элемент [and-gate] 261
иерархические структуры данных 106, 115
иерархия типов [hierarchy of types] 194
в символьной алгебре 206
неадекватность 195
извлечение информации [information retrieval] *см.* база данных
изменение и тождественность [change and sameness]
значение 223
и разделяемые данные 246
изменяемые объекты данных 241, *см. также* очередь; таблица
пары 242
процедурное представление 249

разделяемые данные 248
реализованные с помощью присваиваний 249
списковая структура 242
изменяемые регистры [modified registers] *см.* последовательность команд
ИЛИ-элемент [or-gate] 261
именование [naming]
вычислительных объектов 27
процедур 32
именованный `let` (особая форма) [named `let`] *349* (упр. 4.8)
имитатор регистровых машин [register-machine simulator] 472
имитация [simulation]
для отслеживания производительности регистровой машины 486
как инструмент для проектирования машин 514
регистровых машин *см.* имитатор регистровых машин
управляемая событиями [event-driven] 261
цифровых схем *см.* имитация цифровых схем
имитация цифровых схем [digital-circuit simulation] 260
план действий 267
представление проводов 265
пример работы модели 268
реализация плана действий 269
элементарные функциональные элементы 263
императивное vs. декларативное знание [imperative vs. declarative knowledge] 40, 404
и логическое программирование 405, 425
и недетерминистское вычисление 382*n*
императивное программирование [imperative programming] 225
императивный стиль vs. стиль, ориентированный на выражения [imperative vs. expression-oriented programming style] 281*n*
имя [name] *см. также* локальная переменная; локальное имя; переменная
инкапсулированное 215*n*
процедуры 32
формального параметра 45
инвариант итеративного процесса [invariant quantity of an iterative process] 61 (упр. 1.16)
инвертор [inverter] 261
Ингерман, Питер [Peter Ingerman] 372*n*
индекс [index] 490
индексирование базы данных [database indexing] 418*n*, 439
инкапсулированное имя [encapsulated name] 215*n*
инкапсуляция [encapsulation] 215*n*
интеграл [integral] *см. также* опредеденный интеграл; Монте-Карло, интегрирование методом
степенного ряда 313 (упр. 3.59)
интегратор [integrator] 322
интерпретатор [interpreter] 23, 336, *см. также* вычислитель
vs. компилятор 517, 554
цикл чтение-вычисление-печать 27
интерпретатор `amb` [`amb` evaluator] *см.* недетерминисткий интерпретатор
интерпретатор языка запросов [query interpreter] 406
vs. интерпретатор для Лиспа 423, 424, 448 (упр. 4.79)
база данных 439
бесконечные циклы 426, 429 (упр. 4.67)
вычислитель запросов 424, 431
добавление утверждения или правила 424
кадр 227
кадры 445
конкретизация 430
обзор 417
операции над потоками 442
потоки кадров 418, 424*n*
представление переменной образца 430
преобразование переменной образца 444
проблемы с `not` и `lisp-value` 427, 448 (упр. 4.77)
синтаксис языка запросов 443

сопоставление с образцом 417, 434
структура окружений 449 (упр. 4.79)
улучшения 429 (упр. 4.67), 448 (упр. 4.76), 448 (упр. 4.77)
унификация 421, 437
управляющий цикл 424, 429
инфиксная нотация vs. префиксная нотация [infix notation vs. prefix notation] 154 (упр. 2.58)
информатика [computer science] 336, 359*n*
vs. математика 40, 404
исполнительная процедура [execution procedure] 366
в анализирующем вычислителе 366
в имитаторе регистровых машин 475, 480
в недетерминистском вычислителе 394, 396
исполнительная процедура команды [instruction execution procedure] 475
истина [true] 36*n*
исходная программа [source program] 517
исходный язык [source language] 517
итеративные конструкции [iteration costructs] *см.* циклические конструкции
итеративный процесс [iterative process] 50
vs. рекурсивный процесс 48, 233 (упр. 3.9), 463, 544 (упр. 5.34)
как потоковый процесс 314
линейный 51, 58
построение алгоритмов 61 (упр. 1.16)
реализованный с помощью вызова процедуры 509, *см. также* хвостовая рекурсия
реализованный с помощью вызова процедуры 42, 51
регистровая машина 463

Йохельсон, Джером К. [Jerome C. Yochelson] 495*n*

кавычка [quote] 146
кавычки 146
в естественном языке 146
одинарная vs. двойные 147*n*
с объектами данных Лиспа 147
со строкой символов 147*n*
кадр (в интерпретаторе запросов) [frame] 418, *см. также* сопоставление с образцом; унификация
представление 445
кадр (в модели с окружениями) [frame] 227
глобальный 227
как хранилище внутреннего состояния 234
кадрированного стека дисциплина *см.* дисциплина кадрированного стека
“как сделать” vs. “что такое” [“what is” vs. “how to”] *см.* императивное vs. декларативное знание
Калдевай, Анне [Anne Kaldewaij] 62*n*
Калифорнийский университет в Беркли [University of California at Berkeley] 24*n*
калькулятор; поиск неподвижных точек [fixed points with calculator] 81*n*
каноническая форма многочленов [canonical form for polynomials] 206
Кармайкла числа [Carmichael numbers] 66*n*, 69 (упр. 1.27)
Карр, Альфонс [Alphonse Karr] 211
каскадный сумматор [ripple-carry adder] 265 (упр. 3.30)
квадратный корень 40, *см. также* `sqrt`
поток приближений 314
квазикавычка [quasiquote] 524*n*
квантовая механика [quantum mechanics] 334*n*
Кеплер, Иоганн [Johannes Kepler] 450
Кларк, Кит Л. [Keith L. Clark] 428*n*
Клингер, Уильям [William Clinger] 348*n*, 372*n*
ключ записи [key of a record]
в базе данных 162
в таблице 254
проверка на равенство 259 (упр. 3.24)
Кнут, Дональд Э. [Donald E. Knuth] 57*n*, 61*n*, 63*n*, 125*n*, 218*n*, 219*n*, 566
Ковальски, Роберт [Robert Kowalski] 405*n*
код [code]

ASCII 163
префиксный 164
с переменной длиной [variable-length code] 163
с фиксированной длиной [fixed-length code] 163
Хаффмана *см.* код Хаффмана
код-разделитель [separator code] 164
Колмогоров, А.Н. 218*n*
Кольбеккер, Юджин Эдмунд, мл. [Eugene Edmund Kohlbecker Jr.] 348*n*
Кольмероэ, Ален [Alain Colmerauer] 405*n*
кольцо [ring]
Евклидово [Euclidean] 207*n*
команда [instruction] 450, 454
комбинация 26
`lambda`-выражение как оператор 75
в виде дерева 29
вычисление 29
как оператор комбинации 84*n*
с оператором-комбинацией 84*n*
составное выражение как оператор 39 (упр. 1.4)
комментарии в программах [comments in programs] 129*n*
компилятор [compiler] 517
vs. интерпретатор 517, 554
хвостовая рекурсия, выделение памяти на стеке и сборка мусора 535*n*
компилятор для Scheme 518, *см. также* генератор кода; окружение времени компиляции; последовательность команд; тип связи; целевой регистр
`lambda`-выражения 528
vs. анализирующий интерпретатор 519, 520
vs. вычислитель с явным управлением 518
генераторы кода *см.* `compile...`
запуск скомпилированного кода 551
использование машинных операций 518*n*
использование регистров 518, 518*n*, 535*n*
использование стека 521, 523 (упр. 5.31), 546 (упр. 5.37)
кавычки 525
комбинации 530
лексическая адресация 547
определения 525
отслеживание производительности (использования стека) скомпилированного кода 553, 555 (упр. 5.45), 555 (упр. 5.46)
переменные 525
порождение меток 527*n*
порожденный код, обладающий свойством хвостовой рекурсии 535
порядок вычисления операндов 546 (упр. 5.36)
последовательности выражений 528
применение процедур 530
пример скомпилированного кода 539
присваивания 525
процедуры разбора синтаксиса 520
самовычисляющиеся выражения 525
связующий код 524
связь с вычислителем 551
структура 520
уничтожение внутренних определений 549*n*, 550 (упр. 5.43)
условные выражения 526
эффективность 518
компиляция [compilation] 517, *см.* компилятор
комплексные числа [complex numbers]
декартово vs. полярное представление 172
декартово представление 174
полярное представление 174
помеченные данные 175
композиция функций [composition of functions] 88 (упр. 1.42)
компьютер общего назначения, как универсальная машина [general-purpose computer as universal machine] 517
конвейеризация [pipelining] 284*n*
конечная цепная дробь [finite continued fraction] 82 (упр. 1.37)
конкретизация образца [instantiation of a pattern] 410
конкретное представление данных

[concrete data representation] 93
Конопасек, Милош [Milos Konopasek] 272*n*
конструктор [constructor] 93
как барьер абстракции 98
контроллер для регистровой машины [controller for register machine] 451
диаграмма 452
контрольная точка [breakpoint] 488 (упр. 5.19)
концевая вершина дерева [terminal node of a tree] 29
корень n-й степени как неподвижная точка [nth root as fixed point] 88 (упр. 1.45)
корень четвертой степени как неподвижная точка [fourth root as fixed point] 88 (упр. 1.45)
Кормен, Томас Г. [Thomas H. Cormen] 160*n*
корни уравнения [roots of equation] *см.* Ньютона метод; половинного деления метод
корректность программы [correctness of a program] 40*n*
косинус [cosine]
неподвижная точка функции 81
степенной ряд 312 (упр. 3.59)
космическое излучение [cosmic radiation] 66*n*
красивая печать [pretty printing] 27
красно-черные деревья [red-black trees] 160*n*
кредитные счета, международные [international credit-card accounts] 298*n*
Кресси, Дэвид [David Cressey] 496*n*
криптография [cryptography] 67*n*
кубический корень [cubic root]
как неподвижная точка 84
метод Ньютона 43 (упр. 1.8)

Лагранжа формула интерполяции [Lagrange interpolation formula] 200*n*
Ламэ, Габриэль [Gabriel Lamé] 63*n*
Ландин, Питер [Peter Landin] 31*n*, 306*n*
Лапальм, Ги [Guy Lapalme] 366*n*
Лапицкий, Виктор 335
Лейбниц, барон Готфрид Вильгельм фон [Baron Gottfried Wilhelm von Leibniz]
доказательство Малой теоремы Ферма 65*n*
ряд для вычисления π 70*n*, 315
Лейзерсон, Чарльз Э. [Charles E. Leiserson] 160*n*
лексическая адресация [lexical addressing] 354*n*, 547
лексический адрес 548
лексическая сфера действия [lexical scoping] 47
и структура окружений 548
лексический адрес [lexical address] 548
лекция, что на ней делать [something to do during a lecture] 81*n*
ленивая пара [lazy pair] 379
ленивое вычисление [lazy evaluation] 370
ленивое дерево [lazy tree] 380*n*
ленивый интерпретатор 369
ленивый список [lazy list] 379
Либерман, Генри [Henry Lieberman] 495*n*
линейно итеративный процесс [linear iterative process] 51
порядок роста 58
линейно рекурсивный процесс [linear recursive process] 50
порядок роста 58
линейный рост [linear growth] 50, 58
Лисков, Барбара Хьюберман [Barbara Huberman Liskov] 100*n*
логарифм, аппроксимация $\ln 2$ [logarithm, approximating $\ln 2$] 318 (упр. 3.65)
логарифмический рост [logarithmic growth] 58, 60, 158*n*
логические загадки [logic puzzles] 387
логический вывод [logical deduction] 415, 425
логическое И [logical and] 261
логическое ИЛИ [logical or] 261
логическое программирование [logic programming] 337, 404, *см. также* интерпретатор запросов; язык запросов

vs. математическая логика 425
в Японии 406*n*
история 405*n*, 406*n*
компьютеры 406*n*
язык логического программирования 406
ложь [false] 36*n*
локальная переменная [local variable] 76
локальная эволюция процесса [local evolution of a process] 48
локальное имя 45, 76
Локк, Джон [John Locke] 22
Лэмпорт, Лесли [Leslie Lamport] 298*n*
Лэмпсон, Батлер [Butler Lampson] 225*n*
лямбда-исчисление (λ-исчисление) [λ calculus] 75*n*

Макаллестер, Дэвид Аллен [David Allen McAllester] 385*n*
Макдермот, Дрю [Drew McDermott] 385*n*
Маккарти, Джон [John McCarthy] 23, 23*n*, 383*n*
макрос [macro] 348*n*, *см. также* макросимволы ввода
макросимволы ввода [reader macro characters] 444*n*
Марсельский университет [University of Marseille] 405*n*
математика [mathematics]
vs. информатика 40, 404
vs. техника 66*n*
математическая функция *см.* функция (математическая)
матрица, представленная как последовательность [matrix, represented as sequence] 126 (упр. 2.37)
Маус, Минни и Микки [Mickie and Minnie Mouse] 426
машина Тьюринга [Turing machine] 359*n*
машинный язык [machine language] 517
vs. высокоуровневый язык 336
мемоизация [memoization] 57*n*, 260 (упр. 3.27), 372
и вызов по необходимости 312*n*
и сборка мусора 375*n*
санков 372
через `delay` 305
мера, в Евклидовом кольце [measure in a Euclidean ring] 207*n*
метациклический интерпретатор для Scheme 338
`eval` и `apply` 339
`eval`, управляемая данными 348 (упр. 4.3)
`true` и `false` 354
абстракция данных 339, 350 (упр. 4.10), 353
анализирующая версия 365
глобальное окружение 354
действия над окружениями 351
задача 338*n*
запуск 354
и символьное дифференцирование 343
комбинации (применение процедур) 348 (упр. 4.2)
компиляция 556 (упр. 5.50), 557 (упр. 5.52)
модель вычислений с окружениями 338
наличие хвостовой рекурсии (неизвестно) 508
особые формы 348 (упр. 4.4), 349 (упр. 4.5), 349 (упр. 4.6), 349 (упр. 4.7), 349 (упр. 4.8), 350 (упр. 4.9)
особые формы как производные выражения 346
порядок вычисления операндов 343 (упр. 4.1)
представление выражений 339, 343
представление истины и лжи 350
представление окружений 352
представление процедур 351
производные выражения 346
процедура высшего порядка 341*n*
реализуемый язык vs. язык реализации 342
синтаксис интерпретируемого языка 343, 348 (упр. 4.2), 350 (упр. 4.10)
управляющий цикл 356
цикл `eval-apply` 339
элементарные процедуры 354
эффективность 365

метаязыковая абстракция [metalinguistic abstraction] 336
метка [label] 454
метка типа [type tag] 171, 175, 491*n*
двухуровневая 190
мечтать не вредно [wishful thinking] 94, 150
микросхема для Scheme [Scheme chip] 500, 501
«Микрошафт» (Microshaft) 407
Миллер, Гэри Л. [Gary L. Miller] 69 (упр. 1.28)
Миллер, Джеймс С. [James S. Miller] 535*n*
Миллера-Рабина тест [Miller-Rabin test] 69 (упр. 1.28)
Милнер, Робин [Robin Milner] 329*n*
Минский, Марвин [Marvin Minsky] 15, 495*n*
мировая линия частицы [world line of a particle] 299*n*, 333*n*
многочлен(ы) [polynomial(s)] 199
вычисление по схеме Горнера 125 (упр. 2.34)
иерархия типов 206
каноническая форма 206
от одной переменной 199
плотный [dense] 203
разреженный [sparse] 203
множество [set] 154
база данных как множество 162
операции 154
перестановки 129
подмножество 120 (упр. 2.32)
представленное в виде бинарного дерева 158
представленное в виде неупорядоченного списка 155
представленное в виде упорядоченного списка 156
множитель целости [integerizing factor] 209
мобиль [mobile] 118 (упр. 2.29)
модели вычисления [models of evaluation] 512
моделирование [modeling]
в науке и технике 34
как стратегия разработки 211
моделирование методом Монте-Карло [Monte Carlo simulation] 219
формулировка в терминах потоков 330
модель вычисления с окружениями [environment model of evaluation] 212, 227
внутреннее состояние 234
внутренние определения 238
и метациклический интерпретатор 338
и хвостовая рекурсия 234*n*
передача сообщений 240 (упр. 3.11)
правила вычисления 228
пример применения процедуры 231
структура окружений 227
модуль [absolute value] 36
модульность 124, 211
границы модулей, проведенные по границам объектов 334*n*
и потоки 314
принцип сокрытия 215*n*
функциональных программ vs. объектов 330
через бесконечные потоки 331
через диспетчеризацию по типу 179
через моделирование объектов 218
момент времени [moment in time] 282
Монте-Карло, интегрирование методом [Monte Carlo integration] 221 (упр. 3.5)
формулировка в терминах потоков 331 (упр. 3.82)
Морзе азбука *см.* азбука Морзе
Моррис, Дж. Г. [J. H. Morris] 225*n*
Мун, Дэвид А. [David A. Moon] 24*n*, 495*n*
Мунро, Иан [Ian Munro] 126*n*
мусор [garbage] 495
мутатор [mutator] 241
мьютекс [mutex] 294

наименьших обязательств принцип [principle of least commitment] 175
надтип [supertype] 194
несколько надтипов 195
наибольший общий делитель [greatest common divisor] 62, *см. также* `gcd`

для многочленов 207
используемый в арифметике рациональных чисел 97
используемый для оценки π 219
обобщенный 208 (упр. 2.94)
накопитель [accumulator] 121, 217 (упр. 3.1)
накопление [accumulation] 74 (упр. 1.32)
по дереву 29
невычислимость [noncomputability] 360*n*
недетерминизм в поведении параллельных программ 287*n*, 334*n*
недетерминистские вычисления [nondeterministic computing] 337
недетерминистские программы [nondeterministic programs]
логические загадки 387
пары чисел с простой суммой 381
Пифагоровы тройки 386 (упр. 4.35), 386 (упр. 4.36), 386 (упр. 4.37)
синтаксический анализ естественного языка 389
недетерминистский интерпретатор [nondeterministic evaluator] 394
порядок вычисления операндов 393 (упр. 4.46)
недетерминистское вычисление [nondeterministic computing] 381
недетерминистское программирование vs. программирование на Scheme [nondeterministic programming vs. Scheme programming] 381, 388 (упр. 4.41), 389 (упр. 4.44), 448 (упр. 4.78)
неопределенные значения [unspecified values]
`define` 28*n*
`display` 96*n*
`if` без альтернативы 271*n*
`newline` 96*n*
`set!` 214*n*
`set-car!` 242*n*
`set-cdr!` 242*n*
неподвижная точка [fixed point, of a function] 80
в методе Ньютона 84
вычисление с помощью калькулятора 81*n*
золотое сечение 82 (упр. 1.35)
и унификация 438*n*
как пошаговое улучшение 89 (упр. 1.46)
квадратный корень 81, 84, 86
корень *n*-ной степени 88 (упр. 1.45)
корень четвертой степени 88 (упр. 1.45)
кубический корень 84
трансформации некоторой функции 86
функции косинус 81
несвязанная переменная [unbound variable] 227
нестрогая процедура [non-strict procedure] 371
неудача, в недетерминистских вычислениях
vs. ошибка 397
и поиск 384
неудача, в недетерминистском вычислении [failure in nondeterministic computation] 383
НОД *см.* наибольший общий делитель
номер кадра [frame number] 548
нормальный порядок вычислений [normal-order evaluation] 35, 337
`if` 39 (упр. 1.5)
vs. аппликативный порядок 39 (упр. 1.5), 64 (упр. 1.20), 370
в вычислителе с явным управлением 511 (упр. 5.25)
и задержанные вычисления 328
нотация в настоящей книге [notation in this book]
наклонный шрифт для ответов интерпретатора 26*n*
нотация в настоящей книге
курсив в синтаксисе выражений 32*n*
нужные регистры [needed registers] 523
Ньютона метод [Newton's method] 84
vs. метод половинного деления 85*n*
для дифференцируемых функций 84
для квадратных корней 40, 86
для кубических корней 43 (упр. 1.8)

область действия переменной [scope of a

variable] 45, *см. также* лексическая сфера действия
в `let` 77
внутренняя `define` 360
формальные параметры процедуры 45
обмассив [obarray] 492
обобщенные арифметические операции [generic arithmetic operations] 187
структура системы 187
обобщенные операции [generic operations] 93
обобщенные процедуры [generic procedures] 167, 171
обобщенный селектор 177, 179
обработка ошибок [error handling]
в вычислителе с явным управлением 512, 516 (упр. 5.30)
в скомпилированном коде 554*n*
обработка потоков [stream processing] 35*n*
обработка сигналов [signal processing]
переход сигнала через ноль 323 (упр. 3.74), 323 (упр. 3.75), 324 (упр. 3.76)
потоковая модель 321
сглаживание сигнала 323 (упр. 3.75), 324 (упр. 3.76)
сглаживание функции 88 (упр. 1.44)
образец [pattern] 409
обратная кавычка [backquote] 524*n*
объект(ы) [object(s)] 211
преимущества введения в моделирование 218
с состоянием, меняющимся во времени 213
объектная программа [object program] 517
объектно-ориентированные языки программирования [object-oriented programming languages] 195*n*
обычные числа [ordinary numbers]
в обобщенной арифметической системе 187
ограничения [constraints]
распространение 271
элементарные 272
окружение [environment] 28, 227
в интерпретаторе запросов 449 (упр. 4.79)
времени компиляции *см.* окружение времени компиляции
глобальное [global] 227, *см.* глобальное окружение
и лексическая сфера действия 47
и переименование 448 (упр. 4.79)
как контекст для вычисления 30
объемлющее [enclosing] 227
окружение времени компиляции [compile-time environment] 549, 549 (упр. 5.40), 549 (упр. 5.41)
и явное кодирование 550 (упр. 5.44)
операнды комбинации [operands of a combination] 26
оператор комбинации 26
в виде `lambda`-выражения 75
в виде составного выражения 39 (упр. 1.4)
как комбинация 84*n*
оператор присваивания [assignment operator] 212, *см. также* `set!`
операция [operation]
в регистровой машине 451
обобщенные 93
со смешанными типами 191
определение процедуры [procedure definition] 31
определения *см.* `define`; внутренние определения
определенный интеграл [definite integral] 72
приближенное вычисление методом Монте-Карло 221 (упр. 3.5), 331 (упр. 3.82)
оптимальность [optimality]
кода Хаффмана 165
схемы Горнера 125*n*
освобождение мьютекса 294
особая форма [special form] 31
vs. процедура 371 (упр. 4.26), 379
как производное выражение в интерпретаторе 346
необходимость 42 (упр. 1.6)
особые формы
`and` 37

`begin` 214
`cond` 36
`cons-stream` *(нс)* 302
`define` 28, 32
`delay` *(нс)* 302
`if` 37
`lambda` 75
`let` 76
`let*` 349 (упр. 4.7)
`letrec` 363 (упр. 4.20)
`or` 37
`quote` 147*n*
`set!` 214
именованный `let` 349 (упр. 4.8)
остановки проблема [halting problem] 360 (упр. 4.15)
остаток по модулю [remainder modulo n] 65
Островский, А. М. 125*n*
отладка [debugging] 23
отложенная операция [deferred operation] 50
относительности теория [theory of relativity] 298
отношения, вычисления в терминах отношений 272, 405
отображение [map] 121
вложенное 128, 318
деревьев 119
как преобразователь 121
списков 112
отображение координат рамки [frame coordinate map] 140
отрезок [line segment]
представление в виде пары векторов 142 (упр. 2.48)
представление в виде пары точек 99 (упр. 2.2)
отслеживаемая процедура [monitored procedure] 217 (упр. 3.2)
очередь [queue] 250
в плане действий 269
голова [front] 250
двусторонняя 254 (упр. 3.23)
операции 250
реализация в виде процедуры 254 (упр. 3.22)
хвост [rear] 250
ошибка [bug]
захват свободной переменной 46
побочного эффекта 225*n*
порядок присваиваний 226
ошибка округления [roundoff error] 25*n*
ошибки округления 172*n*

пакет [package] 180
Scheme-number 187
декартово представление 181
комплексные числа 189
многочлены 200
полярное представление 181
рациональные числа 188
память [memory]
в 1964 г. 384*n*
со списковой структурой [list-structured] 489
Пан, В. Я. 126*n*
пара (пары) [pair(s)] 95
аксиоматическое определение 100
бесконечные потоки 318
изменяемые 242
использование для представления дерева 115
использование для представления последовательности 107
ленивые 379
представленные с помощью векторов 490
процедурное представление 100, 249, 379
стрелочная нотация 106
параллелизм 281, 282
и функциональное программирование 333
механизмы для управления 287
правильность параллельных программ 285
тупик 296
параметр [parameter] *см.* формальные параметры
Паскаль, Блез [Blaise Pascal] 57*n*

передача сообщений [message passing] 101, 184, 185, 216
в банковском счете 216
в имитаторе цифровых схем 265
и модель с окружениями 240 (упр. 3.11)
и хвостовая рекурсия 51*n*
переменная [variable] 28, *см. также* локальная переменная
значение [value] 28, 227
несвязанная [unbound] 227
область действия 45, *см. также* область действия переменной
свободная 45
связанная 45
переменная образца [pattern variable] 409
представление 430, 444
переменная состояния [state variable] 51, 212
внутренняя 213
перенаправляющий адрес [forwarding address] 496
перенос языка (на новый компьютер) [porting a language] 554
перестановки множества [permutations of a set] 129
переход сигнала через ноль 323 (упр. 3.74), 323 (упр. 3.75), 324 (упр. 3.76)
перечислитель [enumerator] 121
Перлис, Алан Дж. [Alan J. Perlis] 13, 107*n*
афоризмы 27*n*, 31*n*
печать входных выражений [typing input expressions] 27*n*
печать, элементарные процедуры [primitives for printing] 96*n*
пи (π)
аппроксимация через интегрирование методом Монте-Карло 221 (упр. 3.5)
оценка Чезаро 219, 330
поток приближений 315
приближенное вычисление методом половинного деления 80, 331 (упр. 3.82)
приближенное вычисление через интегрирование методом Монте-Карло 221 (упр. 3.5), 331 (упр. 3.82)
ряд Лейбница 70*n*, 315
формула Уоллиса 73 (упр. 1.31)
Пингала, Ачарья [Ácharya Pingala] 61*n*
Питман, Кент [Kent Pitman] 24*n*
Пифагоровы тройки [Pythagorean triples]
в недетерминистских программах 386 (упр. 4.35), 386 (упр. 4.36), 386 (упр. 4.37)
в потоках 320 (упр. 3.69)
план действий [agenda] 267, *см. также* имитация цифровых схем
плотный многочлен [dense polynomial] 203
поддержание истины [truth maintenance] 385*n*
подкоренное число [radicand] 41
подпрограмма, в регистровой машине [subroutine] 461
подсказка [prompt] 356
вычислитель с явным управлением 512
интерпретатор запросов 430
ленивый интерпретатор 374
метациклический интерпретатор 356
недетерминистский вычислитель 401
подсказка вывода [output prompt] 356
подстановка [substitution] 34*n*
подстановочная модель применения процедуры 33, 34, 227
неадекватность 222
форма процесса 49
подсчет команд [instruction counting] 488 (упр. 5.15)
подтип [subtype] 194
несколько подтипов 195
`подчиняется` (правило) *414*, 428 (упр. 4.64)
поиск [search]
в глубину [depth-first] 384
по бинарным деревьям 158
систематический [systematic] 384
поиск с возвратом [backtracking]
управляемый зависимостями [dependency-directed] 385*n*
хронологический [chronological] 384
показатель конца списка [end-of-list marker] 107

поле типа [type field] 491*n*
полноправные элементы вычисления [first-class elements of computation] 87, *см. также* элементы вычисления первого класса
половинного деления метод [half-interval method] 78
  `half-interval-method` 80
  vs. метод Ньютона 86*n*
полусумматор [half-adder] 261
  `half-adder` 262
  имитация 268
пометка с очисткой [mark-sweep] 495*n*
помеченные данные [tagged data] 175, 490*n*
порождение предложений [generating sentences] 393 (упр. 4.49)
порядок вычисления [order of evaluation]
  в Scheme 227 (упр. 3.8)
  в вычислителе с явным управлением 505
  в компиляторе 546 (упр. 5.36)
  в метациклическом интерпретаторе 343 (упр. 4.1)
  зависимость от реализации 229*n*
  и присваивания 227 (упр. 3.8)
порядок вычисления подвыражений [order of subexpression evaluation] *см.* порядок вычисления
порядок роста [order of growth] 58
  древовидно-рекурсивный процесс 58
  линейно итеративный процесс 58
  линейно рекурсивный процесс 58
  логарифмический 60
  способ записи 58
порядок событий [order of events]
  неопределенность в параллельных системах 284
  отделение от внешней структуры процедур 304
последовательное возведение в квадрат [successive squaring] 60
последовательности [sequences] 73*n*, 107
  как источник модульности 124
  как стандартные интерфейсы 120
  операции 122
  представленные в виде пар 107
последовательность выражений [sequence of expressions]
  в следствии `cond` 37*n*
  в теле процедуры 32*n*
последовательность команд [instruction sequence] 521, 536
поток(и) [stream(s)] 211, 298, 299
  бесконечные *см.* бесконечные потоки
  и задержанное вычисление 324
  используемые в интерпретаторе языка запросов 418, 424*n*
  неявное определение 309
  пустые 301
  реализованные в виде задержанных списков 299
  реализованные в виде ленивых списков 379
пошаговое написание [incremental development] 28
пошаговое улучшение [iterative improvement] 89 (упр. 1.46)
правило (в языке запросов) [rule] 407, 412
  без тела 413*n*, 415
  применение 422, 434, 436, 449 (упр. 4.79)
предикат
  `if` 37
предикат [predicate] 36
  ветви `cond` 36
  соглашение об именах 41*n*
предложения [statements] 523, *см.* последовательность команд
представление множеств в виде неупорядоченных списков [unordered-list representation of sets] 155
представление множеств в виде упорядоченных списков [ordered-list representation of sets] 156
префикс [prefix] 164
префиксная нотация [prefix notation] 26
  vs. инфиксная нотация 154 (упр. 2.58)
префиксный код [prefix code] 164
приведение к наименьшему знаменателю [reducing to lowest terms] 97, 98, 209
приведение типов [coercion] 192

в алгебраических манипуляциях 206
в арифметике многочленов 202
процедура 192
таблица 192
применение процедур [procedure application]
модель вычисления с окружениями 231
обозначение комбинаций 26
подстановочная модель *см.* подстановочная модель применения процедуры
присваивание [assignment] 212, *см. также* `set!`
вызванные ошибки 225*n*, 226
преимущества 218
расплата за 221
провод, в цифровой схеме [wire] 261
программа [program] 22
как абстрактная машина 357
как данные 357
комментарии 129*n*
пошаговое написание 28
структура 28, 44, 46, *см. также* барьеры абстракции
структурирование с помощью подпрограмм 359*n*
программирование [programming]
императивное 225
управляемое данными *см.* программирование, управляемое данными
управляемое потребностями 304
функциональное *см.* функциональное программирование
чрезвычайно плохой стиль 306*n*
элементы 25
программирование, управляемое данными [data-directed programming] 93, 171, 179, 180
vs. разбор случаев 341
в интерпретаторе запросов 431
в метациклическом интерпретаторе 348 (упр. 4.3)
продолжение [continuation]
в имитаторе регистровых машин 478*n*
в недетерминистском интерпретаторе 394, 396, *см. также* продолжение неудачи, продолжение успеха
продолжение неудачи [failure continuation] 394, 396
`amb` 401
присваивания 399
управляющий цикл 401
продолжение успеха [success continuation] 394, 396
прозрачность референциальная 224
производная функции 85
производные выражения в интерпретаторе [derived expressions] 348
производные выражения в интерпретаторе в вычислителе с явным управлением 511 (упр. 5.23)
простой запрос 409
обработка 418, 423, 431
простые числа 64
бесконечный поток *см.* `primes`
и криптография 67*n*
проверка на простоту 64
решето Эратосфена 308
тест Миллера-Рабина 69 (упр. 1.28)
тест Ферма 65
процедура [procedure] 24, 25
vs. математическая функция 40
vs. особая форма 371 (упр. 4.26), 379
безымянная 75
в качестве аргумента 70
возврат нескольких значений 478*n*
высшего порядка *см.* процедура высшего порядка
именование (с помощью `define`) 32
имя 32
как возвращаемое значение 83
как данные 24
как обобщенный метод 78
как черный ящик 44
как шаблон локальной эволюции процесса 48
мемоизированная 260 (упр. 3.27)
нестрогая [non-strict] 371
неявный `begin` в теле 214*n*
область действия формальных параметров 45

обобщенная 167, 171
определение 31
отслеживаемая 217 (упр. 3.2)
полноправный статус в Лиспе 87
произвольное количество аргументов 26, 112 (упр. 2.20)
создание с помощью `define` 32
создание с помощью `lambda` 229, 230
создание с помощью `lambda` 75
составная [compound] 31
тело 32
формальные параметры 32
процедура высшего порядка [higher-order procedure] 70
в метациклическом интерпретаторе 341*n*
и сильная типизация 328*n*
процедура как возвращаемое значение 83
процедура как обобщенный метод 78
процедура в качестве аргумента 70
процедурная абстракция [procedural abstraction] 44
процедурное представление данных [procedural representation of data] 100
изменяемые данные 249
процесс [process] 22
древовидно-рекурсивный 53
итеративный 50
линейно итеративный 51
линейно рекурсивный 50
локальная эволюция 48
необходимые ресурсы 58
порядок роста 58
рекурсивный 50
форма 50
прочесывание внутренних определений [scanning out internal definitions] 361
в компиляторе 549*n*, 550 (упр. 5.43)
прямоугольники, их представление [representing rectangles] 99 (упр. 2.3)
псевдоделение многочленов [pseudodivision] 209
псевдоним [aliasing] 225*n*
псевдоостаток многочленов [pseudoremainder] 209
псевдослучайная последовательность чисел [pseudo-random sequence] 218*n*
пустой поток [empty stream] 301
пустой список [empty list] 109
обозначенный `'()` 109*n*
распознавание с помощью `null?` 110
пути данных регистровой машины [data paths for register machine] 451
диаграмма путей данных 452

Рабин, Майкл О. [Michael O. Rabin] 69 (упр. 1.28)
равенство [equality]
в обобщенной арифметической системе 191 (упр. 2.79)
и референциальная прозрачность 224
символов 148*n*
списков 149 (упр. 2.54)
чисел 36, 149*n*, 491*n*
равенство по модулю n [congruency modulo n] 65
радиус интервала [width, of an interval] 104 (упр. 2.9)
разбитое сердце [broken heart] 496
разбор случаев [case analysis] 36
vs. стиль, управляемый данными 341
обший *см. также* `cond`
с двумя случаями 37
разделение времени [time slicing] 296
разделяемое состояние [shared state] 285
разделяемые данные 246
разделяемые ресурсы [shared resources] 291
размен денег [counting change] 55, 111 (упр. 2.19)
разреженный многочлен [sparse polynomial] 203
Райт, Джесси Б. [Jesse B. Wright] 100*n*
Райт, Э. М. [E. M. Wright] 311*n*
рак точки с запятой [cancer of the semicolon] 31*n*
Рамануджан, Шриниваса [Srinivasa Ramanujan] 321*n*
Рамануджана числа [Ramanujan numbers] 321 (упр. 3.71)
рамка [frame] 133, 140

отображение координат 140
Ранкл, Джон Дэниел [John Daniel Runkle] 133*n*
распространение ограничений [propagation of constraints] 271
Рафаэль, Бертрам [Bertram Raphael] 405*n*
рациональная арифметика [rational-number arithmetic] 93
взаимодействие с обобщенной арифметической системой 188
необходимость составных данных 91
рациональная функция [rational function] 207
приведение к наименьшему знаменателю 209
рациональные числа [rational numbers]
арифметические операции 93
в MIT Scheme 42*n*
печать 96
представление в виде пар 96
приведение к наименьшему знаменателю 97, 98
регистр [register] 450
представление 474
регистровая машина [register machine] 450
действия 456
диаграмма контроллера 452
диаграмма путей данных 452
контроллер 451
отслеживание производительности 486
подпрограмма 459
программа моделирования 472
проектирование 451
пути данных 451
стек 463
тест 452
трассировка 488 (упр. 5.18)
язык для описания 454
резолюции принцип [resolution principle] 405*n*
Хорновские формы 405*n*
Рейтер, Андреас [Andreas Reuter] 297*n*
рекурсивная процедура [recursive procedure]
vs. рекурсивный процесс 51
определение рекурсивной процедуры 43
построение без `define` 364 (упр. 4.21)
рекурсивный процесс [recursive process] 50
vs. итеративный процесс 48, 233 (упр. 3.9), 463, 544 (упр. 5.34)
vs. рекурсивная процедура 51
древовидный 53, 58
линейный 50, 58
регистровая машина 463
рекурсии теория [recursion theory] 359*n*
рекурсия [recursion] 29, 43
в правилах 414
выражение сложного процесса 29
для работы с деревьями 115
управляемая данными 203
референциальная прозрачность 224
решение уравнений [solving equations] *см.* Ньютона метод; половинного деления метод; `solve`
решето Эратосфена [sieve of Eratosthenes] 308
`sieve` 308
Ривест, Рональд Л. [Ronald L. Rivest] 67*n*, 160*n*
Рис, Джонатан А. [Jonathan A. Rees] 348*n*, 366*n*
рисовалка [painter] 133
операции 133
операции высших порядков 137
представление в виде процедуры 141
преобразование и комбинирование 143
Робинсон, Дж. А. [J. A. Robinson] 405*n*
Роджерс, Уильям Бартон [William Barton Rogers] 133*n*
рок-песни 1950-х гг [1950s rock songs] 169 (упр. 2.70)
Росас, Гильермо Хуан [Guillermo Juan Rozas] 535*n*
русского крестьянина метод умножения [Russian peasant method of multiplication] 61*n*
Рэймонд, Эрик [Eric Raymond] 370*n*, 384*n*

`cconsить` [`cons` up] 110
Савин, А.Н. 22
Сазерленд, Айвен [Ivan Sutherland] 272*n*

самовычисляющееся выражение [self-evaluating expression] 340
самосознание, повышение уровня [expansion of consciousness] 342*n*
санк [thunk] 372
  вызова по имени [call-by-name] 306*n*
  вызова по необходимости [call-by-need] 306*n*
  вынуждение 372
  происхождение названия 372*n*
  реализация 375
Сассман, Джеральд Джей [Gerald Jay Sussman] 24*n*, 272*n*, 385*n*
Сассман, Джули Эстер Мазель, племянницы [nieces of Julie Esther Mazel Sussman] 146
сбалансированное бинарное дерево [balanced binary tree] 160, *см. также* бинарное дерево
сбалансированный мобиль [balanced mobile] 118 (упр. 2.29)
сборка мусора [garbage collection] 375*n*, 489, 494, 495
  и изменения данных 242*n*
  и мемоизация 375*n*
  и хвостовая рекурсия 535*n*
  сжимающая [compacting] 496*n*
  через остановку с копированием [stop-and-copy] 495
  через пометку с очисткой [mark-sweep] 495*n*
сборщик мусора [garbage collector] 242*n*
  остановка с копированием 495
  сжимающий 496*n*
  через пометку с очисткой 495*n*
свободная переменная [free variable] 45
  захват 46
  лексическая сфера действия 47
связанная переменная [bound variable] 45
связывание [binding] 45, 227
  глубокое [deep] 354*n*
сглаживание сигнала [smoothing a signal] 323 (упр. 3.75), 324 (упр. 3.76)
сглаживание функции 88 (упр. 1.44)
секретарь, его важное значение [importance of a secretary] 408
селектор [selector] 93
  как барьер абстракции 98
  обобщенный 177, 179
семафор [semaphore] 294*n*
  размера n 296 (упр. 3.47)
сериализатор [serializer] 288, 289
  реализация 294
  с множественными разделяемыми ресурсами 291
сеть ограничений [constraint network] 272
сжимающий сборщик мусора [compacting garbage collector] 496*n*
сигма-запись ($\sum$) 71
сильно типизированный язык [strongly typed language] 328*n*
символ в коде ASCII [character in ASCII encoding] 163
символ(ы) [symbol(s)] 146
  восприятие 492
  кавычки 147
  представление 491
  равенство 148
  уникальность 247*n*
символьная алгебра [symbolic algebra] 198
символьное выражение [symbolic expression] 92, *см. также* символ(ы)
символьное дифференцирование [symbolic differentiation] 149, 183 (упр. 2.73)
Симпсона правило для численного интегрирования [Simpson's Rule for numerical integration] 73 (упр. 1.29)
синтаксис [syntax] 339, *см. также* особые формы
  абстрактный *см.* абстрактный синтаксис
  выражений, описание 32*n*
  языка программирования 31
синтаксический анализ естественного языка [parsing] 389
  настоящая обработка языка vs. игрушечный анализатор 393*n*
синтаксический анализ, отделение от выполнения [separating syntactic analysis from execution]
  в имитаторе регистровых машин 476, 481
  в метациклическом интерпретаторе 365

синтаксический интерфейс [syntax interface] 266*n*
синтаксический сахар [syntactic sugar] 31*n*
  `define` 344
  `let` 77
  процедура vs. данные 266*n*
  циклические конструкции 51
синус [sine]
  приближение при малых углах 59 (упр. 1.15)
  степенной ряд 312 (упр. 3.59)
синхронизация *см.* параллелизм
систематический поиск [systematic search] 384
скобки [parentheses]
  в определении процедуры 32
  обозначающие клаузу `cond` 36
  обозначение применения функции к аргументам 26
слизывание [snarfing] 370*n*
слияние [merge] 334
слияние бесконечных потоков [merging infinite streams] *см.* бесконечные потоки
смещение [displacement] 548
собака идеально разумная, поведение [behavior of a perfectly rational dog] 296*n*
совместимое расширение [upward-compatible extension] 378 (упр. 4.31)
согласованность кэша [cache coherence] 285*n*
соглашение об именах [naming conventions]
  `!` для присваиваний и изменений 214*n*
  `?` для предикатов 41*n*
соединитель (соединители) в системе ограничений [connector(s), in constraint system] 272
  операции 275
  представление 278
сокрытие связывания [shadowing a binding] 228
сокрытия принцип [hiding principle] 215*n*
Соломонофф, Рэй [Ray Solomonoff] 218*n*
сопоставитель [pattern matcher] 417
сопоставление с образцом [pattern matching] 417
  vs. унификация 422, 424*n*
  реализация 434
сопротивление [resistance]
  погрешность 102
  формула для параллельных резисторов 102, 105
составная процедура [compound procedure] 31, *см. также* процедура
  использование в качестве элементарных 33
составное выражение [compound expression] 26, *см. также* комбинация; особая форма
  как оператор комбинации 39 (упр. 1.4)
составной запрос [compound query] 411
  обработка 419, 432, 447 (упр. 4.75), 448 (упр. 4.76), 448 (упр. 4.77)
составной объект данных [compound data object] 91
составные данные, их необходимость [compound data, need for] 90
состояние [state]
  внутреннее *см.* внутреннее состояние
  исчезновение в потоковой формулировке 332
  разделяемое 285
Спаффорд, Юджин Г. [Eugene H. Spafford] 554*n*
специалист по численному анализу [numerical analyst] 79*n*
списковая структура 96, 108*n*
  vs. список 108*n*
  изменяемая 242
  представленная с помощью векторов 490
список [list] 108, 108*n*
  n-й элемент 109
  vs. списковая структура 108*n*
  длина 110
  кавычки 147
  ленивый 379
  манипуляции с `car`, `cdr` и `cons` 108
  методы работы 109
  обратная кавычка 524*n*

обращение 111 (упр. 2.18)
операции 109
отображение 112
последняя пара 111 (упр. 2.17)
представление на печати 108
преобразование в бинарное дерево 161 (упр. 2.64)
преобразование из бинарного дерева 161 (упр. 2.63)
пустой *см.* пустой список
равенство 149 (упр. 2.54)
с заголовком [headed] 255, 269*n*
«`cons`ивание» 110
соединение через `append` 110
«`ycdr`ивание» 109
список свободных ячеек [free list] 493*n*
список термов многочлена [term list of polynomial] 199, 200
представление 203
средства абстракции [means of abstraction] 25
`define` 28
средства комбинирования [means of cvombination] 25, *см. также* замыкание
стандартный интерфейс [conventional interface] 92, 120
последовательность 120
стек [stack] 51*n*, 467
для рекурсии в регистровой машине 463
кадрированный 503*n*
представление 474, 493
степенной ряд, представленный как поток [power series, as a stream] 312 (упр. 3.59)
деление 314 (упр. 3.62)
перемножение 313 (упр. 3.60)
сложение 313 (упр. 3.60)
степенные ряды как потоки
интегрирование 313 (упр. 3.59)
Стил, Гай Льюис мл. [Guy Lewis Steele Jr.] 24*n*, 52*n*, 226*n*, 272*n*, 370*n*, 384*n*
стиль, ориентированный на выражения, vs. императивный стиль [expression-oriented style vs. imperative style] 281*n*
Стой, Джозеф Э. [Joseph E. Stoy] 34*n*, 62*n*, 364*n*
Столлман, Ричард М. [Richard M. Stallman] 272*n*, 385*n*
Стрейчи, Кристофер [Christopher Strachey] 87*n*
стрелка [pointer] 106
в стрелочной диаграмме 106
стрелочная диаграмма [box-and-pointer notation] 106
строгая процедура [procedure] 371
строка [string]
печатание 147*n*
элементарные процедуры 444*n*, 527*n*
структура управления [control structure] 425
сумматор [full-adder] 263
`full-adder` 263
каскадный [ripple-carry] 265 (упр. 3.30)
полу- 261
сумматор для сигналов [summer] 322
суммирование последовательности [summation of a series] 71
с потоками 315
ускорение последовательности приближений 315
схема [circuit]
моделируемая с помощью потоков 322 (упр. 3.73), 328 (упр. 3.80)
цифровая *см.* имитация цифровых схем
схема Горнера [Horner's rule] 125 (упр. 2.34)
счет банковский *см.* банковский счет
счетчик программы [program counter] 475

таблица [table] 254
n-мерная 259 (упр. 3.25)
двумерная 256
для программирования, управляемого данными 180
используемая для моделирования плана действий 269
используемая для хранения вычисленных значений 260 (упр. 3.27)
локальная 258

одномерная 255
операций и типов *см.* таблица операций и типов
представленная в виде бинарного дерева vs. неупорядоченного списка 260 (упр. 3.26)
приведения 192
проверка равенства ключей 259 (упр. 3.24)
хребет [backbone] 255
таблица операций и типов [operation-and-type table] 180
необходимость присваивания 213*n*
реализация 259
таблица регистров, в программе моделирования [register table] 475
табло [tableau] 316
табуляризация [tabulation] 57*n*, 260 (упр. 3.27)
тангенс [tangent]
как цепная дробь 83 (упр. 1.39)
степенной ряд 314 (упр. 3.62)
тег типа [type tag] 491*n*
теговая архитектура [tagged architecture] 491*n*
Тейтельман, Уоррен [Warren Teitelman] 24*n*
текущее время, для имитации плана [current time] 269
тело процедуры [body of a procedure] 32
теорема об остановке [Halting Theorem] 360*n*
Тёрнер, Дэвид [David Turner] 128*n*, 319*n*, 333*n*
тест на равенство нулю (обобщенный) [zero test] 191 (упр. 2.80)
для многочленов 204 (упр. 2.87)
тест, операция в регистровой машине [test] 452
тета от $f(n)$ ($\Theta(f(n))$) 58
техника vs. математика [engineering vs. mathematics] 66*n*
тип данных
башня [tower of types] 194
в Лиспе 191 (упр. 2.78)
в сильно типизированных языках 328*n*
диспетчеризация 179
иерархия в символьной алгебре 206
надтип 194
несколько подтипов и надтипов 195
операции со смешанными типами 191
подтип 194
подъем 194, 197 (упр. 2.83)
спуск 195, 198 (упр. 2.85)
тип связи [linkage descriptor] 520
типизированный указатель [typed pointer] 490
тождественность и изменение [sameness and change]
значение 223
и разделяемые данные 246
торможение усреднением [average damping] 82
точечная запись [dotted-tail notation]
в образцах запросов 410, 435
в правилах языка запросов 415
для процедурных параметров 112 (упр. 2.20), 182*n*
и `read` 436
точка входа [entry point] 454
точка недетерминистского выбора [nondeterministic choice point] 384
точка с запятой [semicolon] 31*n*
как символ комментария 129*n*
точка, представленная в виде пары чисел [point, represented as a pair] 99 (упр. 2.2)
точные целые числа [exact integers] 42*n*
трассировка [tracing]
команд [instruction] 488 (упр. 5.16)
регистров [register] 488 (упр. 5.18)
требуемые регистры [needed registers] *см.* последовательность команд
треугольник Паскаля [Pascal's triangle] 57 (упр. 1.12)
тригонометрические тождества [trigonometric relations] 175
тупик 296, 297
выход 297*n*
способ избежания 297
Тьюринг, Алан М. [Alan M. Turing] 359*n*, 360*n*

Тэтчер, Джеймс У. [James W. Thatcher] 100*n*

Уайз, Дэвид С. [David S. Wise] 306*n*
Уайлс, Эндрю [Andrew Wiles] 65*n*
Уайльд, Оскар, парафраза Перлиса [Oscar Wilde] 27*n*
уголь, битумный [bituminous coal] 133*n*
удовлетворение запросу [satisfying a pattern] 410
  составному 411
Уиздом, Джек [Jack Wisdom] 24*n*
Уинстон, Патрик Генри [Patrick Henry Winston] 385*n*, 393*n*
указатель [pointer]
  типизированный [typed] 490
умножение методом русского крестьянина [Russian peasant method of multiplication] 61*n*
универсальная машина [universal machine] 359
  вычислитель с явным управлением 517
  компьютер общего назначения 517
унификация [unification] 405, 417, 421
  vs. сопоставление с образцом 422, 424*n*
  открытие алгоритма 405*n*
  реализация 437
Уодлер, Филип [Philip Wadler] 225*n*
Уодсворт, Кристофер [Christopher Wadsworth] 329*n*
Уокер, Фрэнсис Амаса [Francis Amasa Walker] 133*n*
Уоллис, Джон [John Wallis] 73*n*
Уотерс, Ричард К. [Richard C. Waters] 125*n*
управляющий цикл [driver loop] 356
  в вычислителе с явным управлением 512
  в интерпретаторе запросов 424, 429
  в ленивом интерпретаторе 374
  в метациклическом интерпретаторе 356
  в недетерминистском интерпретаторе 385, 401
упрощение алгебраических выражений [simplification of algebraic expressions] 152
уравнения рекурсии [recursion equations] 23
уравнения, решения *см.* метод половинного деления; Ньютона метод; `solve`
уровневое проектирование [stratified design] 145
ускоритель последовательности [sequence accelerator] 315
условное выражение [conditional expression]
  `cond` 36
  `if` 37
устойчивость программы [robustness] 145
утверждение [assertion] 407
  неявное 413
утренняя звезда [morning star] *см.* Венера

факториал [factorial] 48, *см. также* `factorial`
  без `letrec` и `define` 364 (упр. 4.21)
  бесконечный поток 311 (упр. 3.54)
  с `letrec` 364 (упр. 4.20)
Фейгенбаум, Эдвард [Edward Feigenbaum] 406*n*
Феничель, Роберт [Robert Fenichel] 495*n*
Ферма Малая теорема [Fermat's Little Theorem] 65
  альтернативная формулировка 69 (упр. 1.28)
  доказательство 65*n*
Ферма тест на простоту [Fermat test for primality] 65
  вариант 69 (упр. 1.28)
Ферма, Пьер де [Pierre de Fermat] 65*n*
Фибоначчи числа [Fibonacci numbers] 53, *см. также* `fib`
  бесконечный поток *см.* `fibs`
  и алгоритм Евклида для НОД 63
Фили, Марк [Marc Feeley] 366*n*
Филипс, Хьюберт [Hubert Phillips] 388 (упр. 4.42)
фильтр [filter] 74 (упр. 1.33), 121
Флойд, Роберт [Robert Floyd] 385*n*
Форбус, Кеннет Д. [Kenneth D. Forbus] 385*n*
форма процесса [shape of a process] 50

формальные параметры процедуры [formal parameters] 32
имена 45
область действия 45
форматирование входных выражений [formatting input expressions] 27*n*
Фридман, Дэниел П. [Daniel P. Friedman] 306*n*, 337*n*
функциональное программирование [functional programming] 222, 330
и время 331
и параллельность 333
функциональный язык программирования 333
функциональный элемент в цифровых схемах [function box] 261
функциональный язык программирования [functional programming language] 333
функция (математическая) [function]
$\mapsto$-нотация 81*n*
vs. процедура 40
Аккермана 52 (упр. 1.10)
композиция 88 (упр. 1.42)
многократное применение 88 (упр. 1.43)
неподвижная точка 80
производная 85
рациональная [rational] 207
сглаживание 88 (упр. 1.44)

Хайтин, Грегори [Gregory Chaitin] 218*n*
Хайям, Омар 57*n*
Хансон, Кристофер П. [Cristopher P. Hanson] 348*n*, 535*n*
хаос в динамике Солнечной системы [chaos in Solar system] 24*n*
Харди, Годфри Харольд [Godfrey Harold Hardy] 311*n*, 321*n*
Хаффман, Дэвид [David Huffman] 164
Хаффмана код [Huffman code] 163, *см.* код Хаффмана
оптимальность 165
порядок роста для кодирования 170 (упр. 2.72)
хвостовая рекурсия [tail recursion] 51, 508
в Scheme 52*n*
и вычислитель с явным управлением 508, 515 (упр. 5.26), 516 (упр. 5.28)
и компилятор 535
и метациклический интерпретатор 508
и модель с окружениями 234*n*
и сборка мусора 535*n*
хвостовая рекурсия в списке аргументов [evlis tail recursion] 505*n*
Хейвендер, Дж. [J. Havender] 297*n*
Хейнс, Кристофер Т. [Christopher T. Haynes] 337*n*
Хендерсон, Питер [Peter Henderson] 132*n*, 308*n*, 333*n*
хендерсоновская диаграмма 308
Херн, Энтони К. [Anthony C. Hearn] 24*n*
Хилфингер, Пол [Paul Hilfinger] 162*n*
Хоар, Чарльз Энтони Ричард [Charles Anthony Richard Hoare] 100*n*
Ходжес, Эндрю [Andrew Hodges] 359*n*
Хофштадтер, Дуглас Р. [Douglas R. Hofstadter] 359*n*
Хьюз, Р. Дж. М. [R. J. M. Hughes] 380*n*
Хьюитт, Карл Эдди [Carl Eddie Hewitt] 51*n*, 385*n*, 405*n*, 495*n*
Хэмминг, Ричард Уэсли [Richard Wesley Hamming] 165*n*, 311 (упр. 3.56)

целевой регистр [target] 520
целые числа [integers] 25*n*
деление 42*n*
точные 42*n*
цепная дробь [continued fraction] 82 (упр. 1.37)
e 83 (упр. 1.38)
золотое сечение 82 (упр. 1.37)
тангенс 83 (упр. 1.39)
цепь электрическая [circuit]
моделируемая с помощью потоков 322 (упр. 3.73), 328 (упр. 3.80)
Цзю Ши-Цзе [Chu Shih-Chieh] 57*n*
цикл в списке [cycle in list] 245 (упр. 3.13)
обнаружение 248 (упр. 3.18)
цикл обратной связи, моделируемый с помощью потоков [feedback loop] 324
цикл чтение-вычисление-печать [read-eval-print loop] 27, *см. также*

управляющий цикл
циклические конструкции [looping constructs] 42, 51
реализация в метациклическом интерпретаторе 350 (упр. 4.9)
цифровой сигнал [digital signal] 261

Чанда-сутра 61*n*
Чарняк, Юджин [Eugene Charniak] 385*n*
Чебышев, Пафнутий Львович 311*n*
Чезаро, Эрнесто [Ernesto Cesàro] 219*n*
Чепман, Дэвид [David Chapman] 385*n*
«Червь» в Интернете [Internet “worm”] 554*n*
черный ящик [black box] 44
Чёрч, Алонсо [Alonzo Church] 75*n*, 102 (упр. 2.6)
Чёрча числа [Church numerals] 102 (упр. 2.6)
Чёрча-Тьюринга тезис [Church-Turing thesis] 359*n*
чисел теория [number theory] 65*n*
числа [numbers]
в Лиспе 26
в обобщенной арифметической системе 187
вещественные [real] 25*n*
десятичная точка 42*n*
зависимость от реализации 42*n*
простые по отношению к другому числу 74 (упр. 1.33)
равенство 36, 149*n*, 491*n*
рациональные 42*n*
сравнение 36
точные целые 42*n*
целые [integer] 25*n*
целые vs. вещественные 25*n*
численные данные [numerical data] 25
численный анализ [numerical analysis] 25*n*
“что такое” vs. “как сделать” [“how to” vs. “what is”] *см.* декларативное vs. императивное знание
Чёрча–Тьюринга тезис *см.* тезис Черча–Тьюринга

Шамир, Ади [Adi Shamir] 67*n*
шахматы, задача о восьми ферзях [chess, eight-queens puzzle] 130 (упр. 2.42), 389 (упр. 4.44)
шишка (правило) *413*, 428 (упр. 4.65)
Шмидт, Эрик [Eric Schmidt] 225*n*
Шроуб, Ховард Э. [Howard E. Shrobe] 406*n*

Эдинбургский университет [University of Edinburgh] 405*n*
Эйлер, Леонард [Leonhard Euler] 83 (упр. 1.38)
доказательство Малой теоремы Ферма 65*n*
ускоритель рядов 315
Эйндховенский технический Университет 294*n*
экспоненциальный рост [exponential growth] 58
электрические цепи, моделируемые с помощью потоков 322 (упр. 3.73), 328 (упр. 3.80)
элементарное ограничение [primitive constraint] 272
элементарные выражения [primitive expressions] 25
вычисление 30
имя переменной 28
имя элементарной процедуры 26
числа 26
элементарные процедуры [primitives]
< 36
> 36
* 26
+ 26
- 26, 36*n*
/ 26
= 36
`apply` 182*n*
`atan` 174*n*
`car` 95
`cdr` 95
`cons` 95
`cos` 81
`display` 96*n*
`eq?` 148

`error` *(нс)* 80*n*
`eval` *(нс)* 360*n*
`list` 108
`log` 82 (упр. 1.36)
`max` 103
`min` 103
`newline` 96*n*
`not` 37
`null?` 110
`number` 150
`pair?` 117
`quotient` 312 (упр. 3.58)
`random` *(нс)* 66, 221*n*
`read` 356*n*
`remainder` 60
`round` 198*n*
`runtime` *(нс)* 67 (упр. 1.22)
`set-car!` 242
`set-cdr!` 242
`sin` 81
`symbol?` 151
`vector-ref` 490
`vector-set!` 490
элементарный запрос [primitive query] *см.* простой запрос
элементы вычисления первого класса [first-class elements of computation] 87, *см. также* полноправные элементы вычисления
Эратосфен 308*n*
эффективность [efficiency] *см. также* порядок роста
вычисления 365
доступа к базе данных 418*n*
древовидно-рекурсивного процесса 57
компиляции 518
Лиспа 24
обработки запросов 420
Эшер, Мориц Корнелис [Maurits Cornelis Escher] 132*n*

явное кодирование элементарных процедур [open coding of primitives] 547 (упр. 5.38), 550 (упр. 5.44)
язык [language] *см.* естественный язык; язык программирования
язык запросов [query language] 406, 407
vs. математическая логика 425
абстракция 412
база данных 407
логический вывод 415
правило *см.* правило (в языке запросов)
проверка на равенство 412*n*
простой запрос *см.* простой запрос
расширения 428 (упр. 4.66), 447 (упр. 4.75)
составной запрос *см.* составной запрос
язык описания изображений [picture language] 133
язык программирования [programming language] 22
логического 406
объектно-ориентированный 195*n*
с аппликативным порядком вычислений [applicative-order] 370
с нормальным порядком вычислений [normal-order] 370
сверхвысокого уровня 40*n*
сильно типизированный 328*n*
строение 369
функциональный 333
язык регистровых машин [register machine language]
`assign` 455, 471
`branch` 454, 471
`const` 455, 471
`goto` 454, 471
`label` 454
`op` 455, 471
`perform` 457, 471
`reg` 455, 471
`restore` 467, 471
`save` 467, 471
`test` 454, 471
`метка` 454
команды 454, 471
точки входа 454
язык сверхвысокого уровня [very high-level language] 40*n*
ячейка, в реализации сериализатора [cell, in serializer implementation] 295