
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 обозреть 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 дүхах. 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-новички 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 сообщество пользователей Лиспа традиционно отвергало попытки провозгласить какое-либо «официальное» описание языка. Вместе с гибкостью и изяществом первоначального замысла такая эволюция позволила

¹Руководство 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.

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 символической 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 различие «пассивных» данных и «активных» процессов. Как мы обнаружим, способность Лиспа рассматривать процедуры в качестве данных делает его одним из самых удобных языков для

²Большинство крупных Лисп-программ 1970х, были написаны на одном из двух диалектов: MacLisp (Moore 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 (Moore 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 Джейм 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 1990). 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).

³Одним 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).

исследования этих методов. Способность 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.

⁴Называть 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, напечатав⁵

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:

⁵Здесь 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

⁶Как 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.

⁷Лисп 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».

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

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

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

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 виде вершины, 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).

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

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.

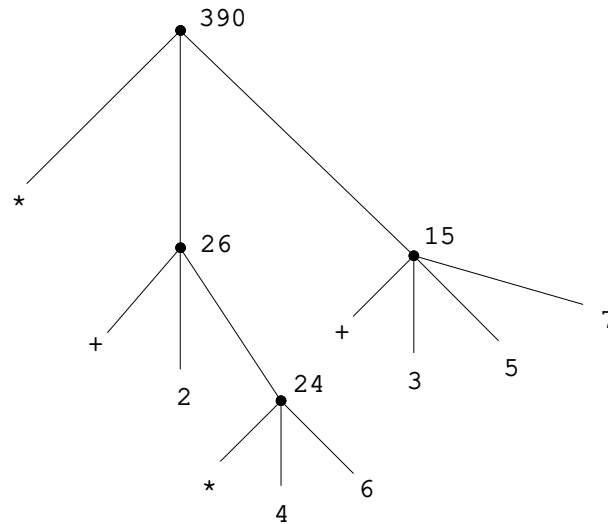


Рис. 1.1: Вычисление, представленное в виде дерева.

• значением остальных имен являются те объекты, с которыми эти имена связаны в окружении.

Мы можем рассматривать второе правило как частный случай третьего, установив, 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

¹¹Особые синтаксические формы, которые представляют собой просто удобное альтернативное поверхностное представление для того, что можно выразить более унифицированным способом,

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 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.

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».

¹²Заметьте, 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.

¹³На всем протяжении этой книги мы будем описывать обобщенный синтаксис выражений, используя курсив в угловых скобках — напр. *<имя>*, чтобы обозначить «дырки» в выражении, которые нужно заполнить, когда это выражение используется в языке.

$\langle \text{Тело} \rangle$ — это выражение, которое вычислит результат применения процедуры, когда формальные параметры будут заменены аргументами, к которым процедура будет применяться.¹⁴ $\langle \text{Имя} \rangle$ и $\langle \text{формальные-параметры} \rangle$ 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:

```
(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).

¹⁴В 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.

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

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

за которыми следуют 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)

¹⁵Несмотря 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 Stoу 1977.

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

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

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>))
```

¹⁶В 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.

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:

```
(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 $\langle p \rangle$. Это заставляет `cond` вернуть в качестве значения значение соответствующего $\langle e \rangle$ в случае, если все предыдущие ветви были пропущены. На самом деле, здесь на месте $\langle p \rangle$ 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

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.

$\langle \text{следствие} \rangle$ и возвращает его значение. В противном случае он вычисляет $\langle \text{альтернативу} \rangle$ 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:

- $(\text{and } \langle e_1 \rangle \ . \ . \ . \ \langle e_n \rangle)$

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

- $(\text{or } \langle e_1 \rangle \ . \ . \ . \ \langle e_n \rangle)$

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

- $(\text{not } \langle e \rangle)$

Значение выражения **not** — истина, если значение выражения $\langle e \rangle$ ложно, и ложь в противном случае.

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)
```

¹⁹Небольшая разница между **if** и **cond** состоит в том, что в **cond** каждое $\langle e \rangle$ может быть последовательностью выражений. Если соответствующее $\langle p \rangle$ оказывается истинным, выражения из $\langle e \rangle$ вычисляются по очереди, и в качестве значения **cond** возвращается значение последнего из них. Напротив, в **if** как $\langle \text{следствие} \rangle$, так и $\langle \text{альтернатива} \rangle$ обязаны состоять из одного выражения.


```

(- 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 выводиться 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

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

Как вычисляются квадратные корни? Наиболее часто применяется Ньютон-ов 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 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))
```

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 называемых языков 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.

²¹На 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.

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))
```

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

²²Обычно 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.

²³Обратите 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.

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

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:

```
(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.

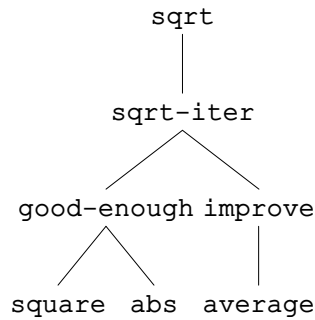
Проверка `good-enough?`, которую мы использовали для вычисления квадратных корней, будет довольно неэффективна для поиска квадратных корней от очень маленьких чисел. Кроме того, в настоящих компьютерах арифметические операции почти всегда вычисляются с ограниченной точностью. Поэтому наш тест оказывается 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 корней.)

Рис. 1.2: Процедурная декомпозиция программы `sqrt`.

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`.

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. Может 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 отразиться 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.

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.

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 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

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

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))
```

²⁷Правило 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.

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, а также абстрагировали получившиеся составные операции путем определения составных процедур. Но всего этого еще недостаточно, чтобы мы могли сказать, что умеем программировать. Положение, в котором мы находимся, похоже на положение человека, выучившего правила передвижения фигур в шахматах, 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), недостает 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 после того, как мы разовьем 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.

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

```

(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 $6!$.

1.2.1 Линейные рекурсия и итерация

Для начала рассмотрим функцию факториал, определяемую уравнением

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

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 $(n-1)!$:

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

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:

```

(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

$$\begin{aligned} \text{произведение} &= \text{счетчик} \cdot \text{произведение} \\ \text{счетчик} &= \text{счетчик} + 1 \end{aligned}$$

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

```

(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

```

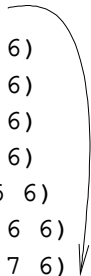


Рис. 1.4: Линейно итеративный процесс для вычисления 6!.

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

```

(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

²⁹В 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.

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 состояниями, 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 по-

³⁰Когда 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).

мощью 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

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)`

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

³¹Довольно 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 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.

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 $5n^2$.

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, ...

Общее правило для чисел Фибоначчи можно сформулировать так:

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

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 $\text{Fib}(n+1)$. Чтобы понять, насколько это плохо, отметим, что значение $\text{Fib}(n)$ растет экспоненциально при увеличении n . Более точно (см. упражнение 1.13), $\text{Fib}(n)$ — это целое число, ближайшее к $\phi^n / \sqrt{5}$, где

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

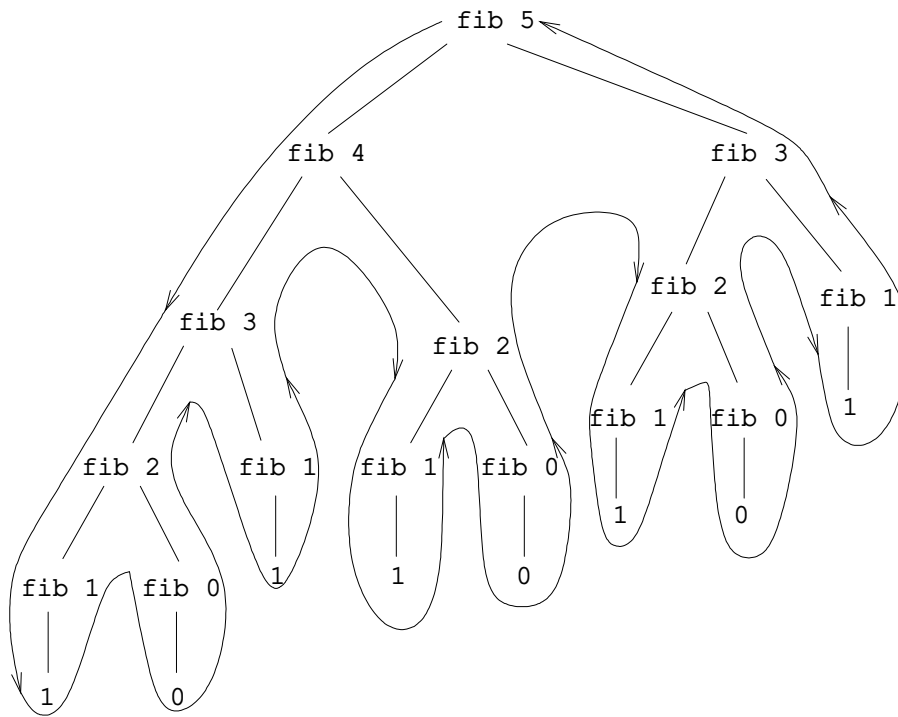


Рис. 1.5: Древовидно-рекурсивный процесс, порождаемый при вычислении (`fib 5`).

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

$$\begin{aligned} a &\leftarrow a + b \\ b &\leftarrow a \end{aligned}$$

Нетрудно показать, что после того, как мы сделаем эту трансформацию n раз, a и b будут соответственно равны $\text{Fib}(n + 1)$ и $\text{Fib}(n)$. 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 $\text{Fib}(n)$, — 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 состояниями.

Размен денег

Чтобы сочинить итеративный алгоритм для чисел Фибоначчи, нужно совсем немного смекалки. Теперь для контраста рассмотрим следующую задачу: сколь-

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

кими 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
- 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)))
```

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

(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
```

`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).

			1		
		1		1	
	1		2		1
1		3		3	
	1	4		6	
1		4		6	
	1	6		10	
1		6		10	
	1	10		15	
1		10		15	
	1	15		21	
1		15		21	
	1	21		28	
1		21		28	
	1	28		36	
1		28		36	
	1	36		45	
1		36		45	
	1	45		55	
1		45		55	
	1	55		66	
1		55		66	
	1	66		78	
1		66		78	
	1	78		91	
1		78		91	
	1	91		105	
1		91		105	
	1	105		120	
1		105		120	
	1	120		136	
1		120		136	
	1	136		153	
1		136		153	
	1	153		171	
1		153		171	
	1	171		190	
1		171		190	
	1	190		210	
1		190		210	
	1	210		231	
1		210		231	
	1	231		253	
1		231		253	
	1	253		276	
1		253		276	
	1	276		300	
1		276		300	
	1	300		325	
1		300		325	
	1	325		351	
1		325		351	
	1	351		378	
1		351		378	
	1	378		406	
1		378		406	
	1	406		436	
1		406		436	
	1	436		467	
1		436		467	
	1	467		500	
1		467		500	
	1	500		533	
1		500		533	
	1	533		567	
1		533		567	
	1	567		602	
1		567		602	
	1	602		638	
1		602		638	
	1	638		675	
1		638		675	
	1	675		713	
1		675		713	
	1	713		752	
1		713		752	
	1	752		792	
1		752		792	
	1	792		833	
1		792		833	
	1	833		875	
1		833		875	
	1	875		918	
1		875		918	
	1	918		962	
1		918		962	
	1	962		1007	
1		962		1007	
	1	1007		1053	
1		1007		1053	
	1	1053		1100	
1		1053		1100	
	1	1100		1148	
1		1100		1148	
	1	1148		1197	
1		1148		1197	
	1	1197		1247	
1		1197		1247	
	1	1247		1298	
1		1247		1298	
	1	1298		1350	
1		1298		1350	
	1	1350		1403	
1		1350		1403	
	1	1403		1457	
1		1403		1457	
	1	1457		1512	
1		1457		1512	
	1	1512		1568	
1		1512		1568	
	1	1568		1625	
1		1568		1625	
	1	1625		1683	
1		1625		1683	
	1	1683		1742	
1		1683		1742	
	1	1742		1802	
1		1742		1802	
	1	1802		1863	
1		1802		1863	
	1	1863		1925	
1		1863		1925	
	1	1925		1988	
1		1925		1988	
	1	1988		2052	
1		1988		2052	
	1	2052		2117	
1		2052		2117	
	1	2117		2183	
1		2117		2183	
	1	2183		2250	
1		2183		2250	
	1	2250		2318	
1		2250		2318	
	1	2318		2387	
1		2318		2387	
	1	2387		2457	
1		2387		2457	
	1	2457		2528	
1		2457		2528	
	1	2528		2600	
1		2528		2600	
	1	2600		2673	
1		2600		2673	
	1	2673		2747	
1		2673		2747	
	1	2747		2822	
1		2747		2822	
	1	2822		2900	
1		2822		2900	
	1	2900		2979	
1		2900		2979	
	1	2979		3059	
1		2979		3059	
	1	3059		3140	
1		3059		3140	
	1	3140		3222	
1		3140		3222	
	1	3222		3305	
1		3222		3305	
	1	3305		3390	
1		3305		3390	
	1	3390		3476	
1		3390		3476	
	1	3476		3563	
1		3476		3563	
	1	3563		3651	
1		3563		3651	
	1	3651		3740	
1		3651		3740	
	1	3740		3830	
1		3740		3830	
	1	3830		3921	
1		3830		3921	
	1	3921		4013	
1		3921		4013	
	1	4013		4106	
1		4013		4106	
	1	4106		4200	
1		4106		4200	
	1	4200		4295	
1		4200		4295	
	1	4295		4391	
1		4295		4391	
	1	4391		4488	
1		4391		4488	
	1	4488		4586	
1		4488		4586	
	1	4586		4685	
1		4586		4685	
	1	4685		4785	
1		4685		4785	
	1	4785		4886	
1		4785		4886	
	1	4886		4988	
1		4886		4988	
	1	4988		5091	
1		4988		5091	
	1	5091		5195	
1		5091		5195	
	1	5195		5300	
1		5195		5300	
	1	5300		5406	
1		5300		5406	
	1	5406		5513	
1		5406		5513	
	1	5513		5621	
1		5513		5621	
	1	5621		5730	
1		5621		5730	
	1	5730		5840	
1		5730		5840	
	1	5840		5951	
1		5840		5951	
	1	5951		6063	
1		5951		6063	
	1	6063		6176	
1		6063		6176	
	1	6176		6290	
1		6176		6290	
	1	6290		6405	
1		6290		6405	
	1	6405		6521	
1		6405		6521	
	1	6521		6638	
1		6521		6638	
	1	6638		6756	
1		6638		6756	
	1	6756		6875	
1		6756		6875	
	1	6875		6995	
1		6875		6995	
	1	6995		7116	
1		6995		7116	
	1	7116		7238	
1		7116		7238	
	1	7238		7361	
1		7238		7361	
	1	7361		7485	
1		7361		7485	
	1	7485		7610	
1		7485		7610	
	1	7610		7736	
1		7610		7736	
	1	7736		7863	
1		7736		7863	
	1	7863		7991	
1		7863		7991	
	1	7991		8120	
1		7991		8120	
	1	8120		8250	
1		8120		8250	
	1	8250		8381	
1		8250		8381	
	1	8381		8513	
1		8381		8513	
	1	8513		8646	
1		8513		8646	
	1	8646		8780	
1		8646		8780	
	1	8780		8915	
1		8780		8915	
	1	8915		9051	
1		8915		9051	
	1	9051		9188	
1		9051		9188	
	1	9188		9326	
1		9188		9326	
	1	9326		9465	
1		9326		9465	
	1	9465		9605	
1		9465		9605	
	1	9605		9746	
1		9605		9746	
	1	9746		9888	
1		9746		9888	
	1	9888		10031	
1		9888		10031	
	1	10031		10175	
1		10031		10175	
	1	10175		10320	
1		10175		10320	
	1	10320		10466	
1		10320		10466	
	1	10466		10613	
1		10466		10613	
	1	10613		10761	
1		10613		10761	

и индукции докажите, что $\text{Fib}(n) = (\phi^n - \psi^n)/\sqrt{5}$.

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 выполняемых 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, что требуемый объем памяти растет 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

³⁶В 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.

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 роста, 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)))))
```

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

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

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

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

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^{2^n} 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

³⁷Точнее, количество требуемых умножений равно логарифму n по основанию 2 минус 1 и плюс

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 `half`, 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

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 $\Theta(\log(n))$.

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

³⁹Итеративный 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.

⁴⁰Этот 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.

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: $a \leftarrow a+b$ и $b \leftarrow a$. Назовем эту трансформацию T и заметим, что n -кратное применение T , начиная с 1 и 0, дает нам пару $\text{Fib}(n+1)$ и $\text{Fib}(n)$. 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 меньших и меньших пар целых чисел. Например,

⁴¹Это упражнение нам предложил Джо Стой на основе примера из Kaldewaij 1990.

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

сводит $\text{НОД}(206, 40)$ к $\text{НОД}(2, 0)$, 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 растёт пропорционально логарифму чисел-аргументов.

Тот факт, что число шагов, затрачиваемых алгоритмом Евклида, растёт 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, должно выполняться $n \geq \text{Fib}(k) \approx \phi^k / \sqrt{5}$. Следовательно, число шагов k растёт как логарифм n (по основанию ϕ). Следовательно, порядок роста равен $\Theta(\log n)$.

⁴²Алгоритм 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.

⁴³Эту 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 $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ — 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 $a_{k-1} = b_k, b_{k-1} = \text{остаток от деления } a_k \text{ на } b_k$. 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 $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ — 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 даёт $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k-1) = \text{Fib}(k+1)$, что и доказывает теорему Ламэ.

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

⁴⁴Если d — делитель n , то n/d тоже. Но d и n/d не могут оба быть больше \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

⁴⁵Пьер 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 дано Эйлером 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.

⁴⁶Шаги 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 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.)

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 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

⁴⁷Числа, «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.

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 algorithms). 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.

Большая 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, 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?

⁴⁸Одно 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.

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))
```

Правда 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 на альтернативной 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 . Проверять 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 `exptmod`. Однако когда в процедуре `exptmod` 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 `exptmod` 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 `exptmod` 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 возможно-сти 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

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

который (очень медленно) сходится к $\pi/8$:⁴⁹

```
(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:

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


```
(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

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

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

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

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))

(integral cube 0 1 0.01)
.24998750000000042

(integral cube 0 1 0.001)
.2499998750000001
```

(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

⁵⁰Обратите 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 за чем-либо еще. В разделе 1.3.2 мы совсем от них избавимся.

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

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 формуле⁵²

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

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)
```

⁵¹Смысл 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 получают еще более мощные абстракции. Мы увидим, как эти методы сами по себе становятся мощным и изящным подходом к проектированию программ.

⁵²Эту формулу открыл английский математик семнадцатого века Джон Уоллис.

`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`.

b. 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:

a. сумму квадратов простых чисел в интервале от `a` до `b` (в предположении, что процедура `prime?` уже написана);

b. произведение всех положительных целых чисел меньше `n`, которые просты по отношению к `n` (то есть всех таких положительных целых чисел $i < n$, что $\text{НОД}(i, n) = 1$).

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

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

⁵³Было 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.

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

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 $\langle пер_1 \rangle$ имеет значение $\langle выр_1 \rangle$
и $\langle пер_2 \rangle$ имеет значение $\langle выр_2 \rangle$
...
и $\langle пер_n \rangle$ имеет значение $\langle выр_n \rangle$
в $\langle теле \rangle$

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

⁵⁴Если 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.

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 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 растёт 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 `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:

⁵⁵Мы 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.

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

```
(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:

$$f(x), f(f(x)), f(f(f(x))), \dots$$

— 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)
    (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 $y = \sin y + \cos y$:

```
(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 сформулировать вычисление квадратного корня как поиск неподвижной точки. Вычислить квадратного корня из произвольного числа x означает найти такое y , что $y^2 = x$.

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

Переведа это уравнение в эквивалентную форму $y = x/y$, мы обнаруживаем, что должны найти неподвижную точку функции⁵⁸ $y \mapsto x/y$, 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))
```

(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

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 .

`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.

а. Бесконечная *цепная дробь* (continued fraction) есть выражение вида

$$f = \frac{N_1}{D_1 + \frac{N_2}{D_2 + \frac{N_3}{D_3 + \dots}}}$$

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

$$f = \frac{N_1}{D_1 + \frac{N_2}{D_2 + \dots + \frac{N_k}{D_k}}}$$

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, ... 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:

$$\operatorname{tg} x = \frac{x}{1 - \frac{x^2}{3 - \frac{x^2}{5 - \dots}}}$$

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

⁵⁹Заметьте, 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 выражении такого рода — когда нам нужно применить процедуру, полученную в качестве значения из процедуры высшего порядка.

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 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 торжественному усреднению, 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)
```

⁶⁰См. дальнейшее обобщение в упражнении 1.45

⁶¹Вводные 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.

⁶²Метод 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.

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))
```

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.

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

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 Ньютона 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 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

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

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

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)
```

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 $f(f(\dots(f(x))\dots))$. 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 $f(x + dx)$.

⁶⁶Основная 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.

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. Проведите 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 вычисления корней 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`.