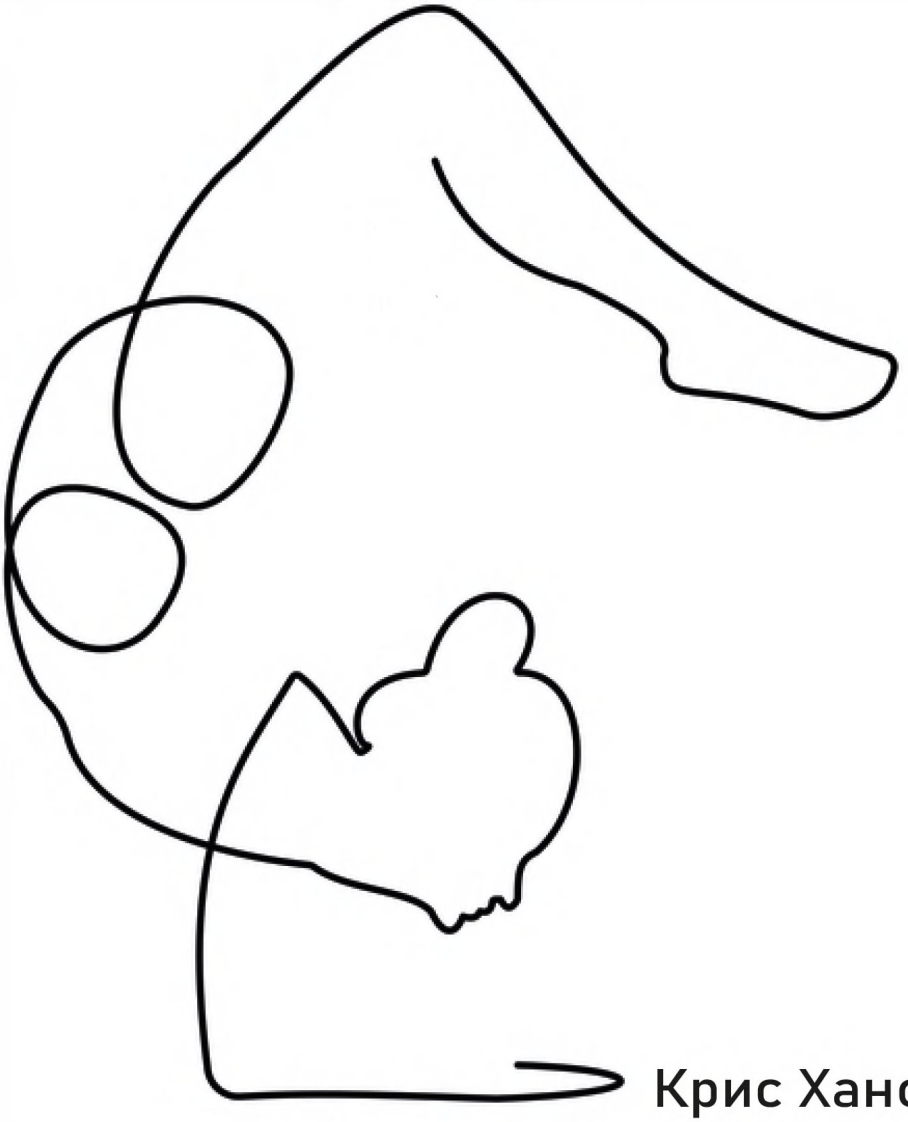


Крис Хансон
Джеральд Джей Сассман

Проектирование гибких программ

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Software Design for Flexibility

How to Avoid Programming Yourself into a Corner

Chris Hanson and Gerald Jay Sussman
foreword by Guy L. Steele Jr.

Проектирование гибких программ

Как не загнать себя в угол

Крис Хансон, Джеральд Джей Сассман

The MIT Press
Cambridge, Massachusetts
London, England

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2022

УДК 004.4
ББК 32.372
Х19

Х19 Хансон К., Сассман Д. Дж.

Проектирование гибких программ / пер. с англ. Ю. Бронникова. – М.: ДМК Пресс, 2022. – 374 с.: ил.

ISBN 978-5-97060-955-2

Бывает так, что при написании программы вы попадаете в тупик. Возможно, это потому, что вы, как оказалось, не учли некоторые особенности исходной задачи. Однако до обидного часто дело в том, что на начальной стадии проектирования вы приняли какое-то решение, выбрали какую-то структуру данных или способ организации кода, который затем оказался слишком ограниченным, а теперь его трудно заменить.

Эта книга служит мастер-классом по стратегиям организации программ, которые позволяют сохранить гибкость. В каждой главе можно видеть, как два эксперта демонстрируют тот или иной передовой метод, шаг за шагом разрабатывая работающую подсистему, объясняют на ходу стратегию своей работы и время от времени указывают на подводный камень или способ обойти то или иное ограничение.

Издание предназначено для разработчиков, стремящихся создавать адаптивные системы, которые можно менять с минимальными усилиями.

ISBN 978-0-26204-549-0 (англ.)
ISBN 978-5-97060-955-2 (рус.)

Компьютер подобен скрипке. Представьте себе новичка, который сначала испытывает проигрыватель, затем скрипку. Скрипка, говорит он, звучит ужасно. Именно этот аргумент мы слышали от наших гуманитариев и специалистов по информатике. Компьютеры, говорят они, хороши для определенных целей, но они недостаточно гибки. Так же и со скрипкой, и с пишущей машинкой, пока Вы не научились их использовать.

Марвин Минский. «Почему программирование — хороший способ выражения малопонятных и туманно сформулированных идей», *Проектирование и планирование*, 1967

Оглавление

Предисловие

Бывает так, что при написании программы вы попадаете в тупик. Возможно, это потому, что вы, как оказалось, не учли некоторые особенности исходной задачи. Однако до обидного часто дело в том, что на начальной стадии проектирования вы приняли какое-то решение, выбрали какую-то структуру данных или способ организации кода, который затем оказался слишком ограниченным, а теперь его трудно изменить.

Эта книга служит мастер-классом по стратегиям организации программ, которые позволяют сохранять гибкость. Все мы уже давно знаем, что, хотя для хранения входных данных легко объявить массив фиксированного размера, такое программное решение может оказаться неприятно ограниченным и привести к тому, что нельзя станет обрабатывать строки больше какой-то длины или наборы записей свыше некоторого фиксированного количества. Многие дыры в безопасности, особенно в сетевом коде, случились потому, что программист выделял буфер фиксированного размера и забывал проверить, что обрабатываемые данные помещаются в этот буфер. Динамически выделяемая память (получаемая от библиотеки вроде вызова `malloc` языка C или от автоматического сборщика мусора), будучи несколько сложнее устроена, тем не менее дает намного больше гибкости и, в качестве дополнительного преимущества, позволяет легче избегать ошибок (особенно если язык программирования всегда проверяет обращения к массивам и убеждается, что индекс не выходит за границы). Это всего лишь простейший пример.

Некоторые ранние языки программирования, в сущности, закладывались на способ проектирования, отражающий стиль аппаратной организации, называемый *Гарвардская архитектура*: код программы находится *здесь*, данные — *там*, и задача кода состоит в том, чтобы манипулировать данными. Однако такое жесткое разделение между несмешиваемыми кодом и данными, как оказалось, слишком сильно ограничивает организацию программ. Уже задолго до конца XX в. из опыта функциональных языков программирования (например, ML, Scheme и Haskell) и объектно ориентированных языков (например, Simula, Smalltalk и C++) мы поняли, что у стиля, позволяющего рассматривать данные как код, код как данные и связывать небольшие блоки кода и относящихся к нему данных в единое целое, есть преимущества перед стилем, разделяющим код и данные как отдельные монолитные области. Самый гибкий вид данных — структура-запись, которая может содержать не только «элементарные ячейки» вроде чисел и символов, но и ссылки на выполняемый код, скажем, на функции. Самая мощная разновидность кода — такой, который порождает другой код, объединенный с точно определенным количеством специально отобранных данных; такая связка уже не просто «указатель на функцию», но

замыкание (в функциональном языке) или *объект* (если язык объектно ориентированный).

Джерри Сассман и Крис Хансон используют свой более чем вековой совместный опыт программирования и представляют набор методов, разработанных и проверенных за десятилетия преподавания в Массачусетском технологическом институте, которые позволяют еще более расширить эту базовую стратегию достижения гибкости. Используйте не просто функции, а *полиморфные функции*, открытые для расширения так, как обычным функциям недоступно. Функции должны оставаться маленькими. Часто лучший вариант возвращаемого значения для функции — другая функция (уточненная соответствующими данными). Будьте готовы относиться к данным как к разновидности кода; иногда, если необходимо, это приводит к созданию специализированного встроенного языка внутри вашей программы. (Это один из способов рассказать, откуда взялся язык Scheme: лисповский диалект MacLisp не поддерживал достаточно общий вид замыкания функций, так что мы с Джерри просто написали на нем встроенный вариант Lisp, где такие замыкания были.) Будьте готовы заменить структуру данных структурой более общего вида, которая включает исходную как частный случай и расширяет ее возможности. С помощью автоматического распространения ограничений можно избежать преждевременного решения, какие данные подаются на вход, а какие получаются на выходе.

Эта книга не является обзорной монографией или учебником — скажу повторно, это мастер-класс. В каждой главе можно видеть, как два эксперта демонстрируют тот или иной передовой метод, шаг за шагом разрабатывая работающую подсистему, объясняют на ходу стратегию своей работы и время от времени указывают на подводный камень или способ обойти то или иное ограничение. Будьте готовы, когда вас попросят показать свою способность работать самостоятельно, расширить структуру данных или дописать дополнительный код — а затем подключите свое воображение и идите дальше, чем показали вам авторы. Идеи этой книги богаты и глубоки; внимание как к словам, так и к программам будет вознаграждено.

Гай Стил
Лексингтон, Массачусетс.
Август 2020

Предисловие авторов

Всем нам приходилось тратить уйму лишнего времени за попытками уломать кусок старого кода на выполнение задач, которые мы не представляли себе, когда его исходно писали. Это ужасная трата времени и сил. К сожалению, на нас действует множество сил, побуждающих нас писать программы, хорошо приспособленные к конкретной узкой задаче, где почти нет многократно используемых частей. Однако мы считаем, что так быть не должно.

Строить системы, приемлемо работающие в более широком классе ситуаций, чем их создатели исходно имели в виду, трудно. Лучшие из таких систем способны к развитию: их можно приспособить к новым задачам ценой лишь небольших изменений. Как мы можем проектировать системы, гибкие в этом смысле?

Было бы замечательно, если бы для добавления новой возможности к программе нам нужно было бы только добавить немного кода, без изменения существующего массива программы. Часто этого можно добиться, если использовать определенные организационные принципы при построении этого массива и включать в него соответствующие зацепки.

Многое о построении гибких, способных к развитию систем можно узнать из наблюдения за биологическими системами. Приемы, исходно разработанные для поддержки символического искусственного интеллекта, можно рассматривать как способы увеличить гибкость и приспособляемость программ и других инженерных систем. Напротив, многие практики, принятые в информатике, активно препятствуют построению систем, которые легко перестроить для использования в новых условиях.

Нам часто случалось при написании программ загонять себя в угол и тратить огромные усилия на рефакторинг кода, позволяющий выбраться из тупика. Поэтому мы считаем, что набрали достаточный опыт и способны выявить, определить и продемонстрировать найденные нами стратегии и методы, эффективные при построении больших систем, способных адаптироваться к задачам, которые создатели исходно не имели в виду. В этой книге мы делимся некоторыми плодами своего более чем векового опыта программирования.

Эта книга

Эта книга появилась в процессе преподавания компьютерного программирования в Массачусетском технологическом институте (MIT). Мы начали вести этот курс много лет назад, желая показать старшекурсникам и аспирантам методы и технологии, полезные при решении основных задач в области искусственного интеллекта, таких как символическая математика и системы, основанные на правилах. Мы хотели,

чтобы студенты были способны строить эти системы гибко, чтобы их можно было легко сочетать при построении еще более мощных программ. Кроме того, мы хотели обучить студентов рассуждать о зависимостях — как их отслеживать, использовать для объяснения поведения и как с их помощью управлять перебором.

Несмотря на то что наш курс был и остается успешным, оказалось, что в начале мы понимали материал не настолько хорошо, как сами об этом думали. Поэтому пришлось потратить немало сил на полировку своих инструментов и уточнение идей. Теперь мы понимаем, что эти методы работают далеко не только в области искусственного интеллекта. Мы считаем, что нашим опытом может воспользоваться любой, кто строит сложные системы, например компиляторы и интегрированные среды разработки. Книга построена на основе лекций и упражнений, которые мы сейчас используем в своем курсе.

Содержание

В этой книге намного больше материала, чем можно пройти за односеместровый курс. Поэтому каждый год мы отбираем некоторую часть для использования в классе. Глава 1 представляет собой введение в нашу философию программирования. Здесь мы демонстрируем понятие *гибкости* в общем контексте природы и техники. Мы стараемся показать, что гибкость столь же важна, как эффективность и правильность поведения. В каждой последующей главе мы вводим некоторый общий метод и иллюстрируем его набором упражнений. Это важный организующий принцип всей книги.

В главе 2 исследуем некоторые универсально применимые способы построения систем, способных к росту. Мощный метод организации гибкой системы — построить ее в виде набора языков специального назначения, каждый из которых приспособлен для описания некоторой подсистемы. Здесь мы разрабатываем основной инструментарий для построения специализированных языков: показываем, как подсистемы можно организовать из взаимозаменяемых частей, как их можно гибко соединять с помощью *комбинаторов*, как части системы обобщаются через *обертки* и как программу часто можно упростить, абстрагируя модель предметной области.

В главе 3 мы вводим чрезвычайно мощный, хотя потенциально опасный метод использования *полиморфных процедур*, управляемых предикатами. В начале мы строим обобщение арифметики для работы с символьными алгебраическими выражениями. Затем показываем, как такое обобщение можно сделать эффективным с помощью меток типа на данных, и демонстрируем силу нашего метода на примере простой, но легко расширяемой игры-бродилки.

В главе 4 мы вводим понятие *сопоставления с образцом*, сначала для использования в системах переписывания термов, затем через *унификацию* показываем, как легко можно построить вывод типов. Сегментные переменные заставляют нас использовать *перебор с возвратами*. Унификация — первый случай, где мы видим силу представления и комбинирования структур с *частичной информацией*. В конце главы мы расширяем эту идею на задачу сопоставления в произвольных графах.

В главе 5 мы исследуем возможности *интерпретации* и *компиляции*. Мы считаем, что программисты должны быть способны выйти за рамки своего языка программирования, каков бы он ни был, и использовать язык, лучше приспособленный для решения конкретной задачи. Мы также показываем, как естественным образом встроить в язык перебор с возвратами путем реализации в составе интерпре-

татора/компилятора оператора недетерминистского выбора `amb` и как использовать *продолжения*.

В главе 6 мы показываем построение систем *многослойных данных* и *многослойных процедур*, где к каждому элементу данных могут в качестве аннотации быть прикреплены метаданные различного рода. Метаданные никак не влияют на обработку нижележащих данных, и код для обработки нижележащих данных не обязан даже знать о них и на них ссылаться. Однако метаданные обрабатываются своими собственными процедурами, в сущности, параллельно основным данным. В качестве иллюстрации мы присоединяем к числовым данным информацию о единицах измерения, а также демонстрируем отслеживание зависимостей, так что каждый элемент данных оказывается снабжен родословной, показывающей, как он получен из элементарных источников.

Все это собирается вместе в главе 7, где мы вводим *распространение*, чтобы освободиться от парадигмы компьютерных языков, ориентированной на выражения. Здесь мы смотрим на соединение модулей как на монтажную схему. Это позволяет нам гибко подключать различные источники частичной информации. Поскольку мы отслеживаем зависимости с помощью многослойных данных, то способны организовать *перебор, управляемый зависимостями*, который значительно уменьшает пространство поиска в больших и сложных системах.

С помощью этой книги можно построить несколько вариантов университетского курса продвинутого уровня. Идея комбинаторов из главы 2 и полиморфные процедуры из главы 3 используются во всех последующих главах. Однако образцы и сопоставление из главы 4, а также вычислители из главы 5 в дальнейших главах не встречаются. Единственный материал из главы 5, необходимый в дальнейшем, — оператор `amb` из разделов 5.4 и 5.4.1. Идея многослойности из главы 6 близко связана с идеей полиморфных процедур, но с некоторым новым поворотом. Использование многослойных данных для отслеживания зависимостей, которое в главе 6 служит примером, становится важным компонентом системы в работе над распространением в главе 7, где зависимости служат для оптимизации поиска с возвратами.

Scheme

Программы в этой книге написаны на Scheme, функциональном по преимуществу языке, который является вариантом Lisp. Несмотря на то что Scheme не популярный язык и не используется широко в промышленном программировании, для этой книги он — правильный выбор[1]. Цель этой книги — демонстрация и объяснение идей. По многим причинам представление кода, воплощающего и иллюстрирующего эти идеи, на Scheme оказывается короче и проще, чем на более популярных языках. Некоторые из наших идей на других языках было бы почти невозможно выразить.

Языки из других семейств, кроме Lisp, требуют множества церемоний, чтобы сказать простые вещи. В нашем же случае единственное, что удлинняет наш код, — описательные имена для вычислительных объектов.

Благодаря тому, что синтаксис Scheme чрезвычайно прост — это всего лишь представление естественного дерева разбора, и требуется лишь минимальный синтаксический анализ, — легко оказывается писать программы, работающие с текстами других программ, такие как интерпретаторы, компиляторы и обработчики алгебраических выражений.

[1] Мы даем краткое введение в Scheme в приложении B.

Важно и то, что Scheme — язык снисходительный, а не нормативный. Он не пытается запретить программисту делать что-то «глупое». Это позволяет нам проделывать некоторые мощные трюки, например динамически перестраивать значение арифметических выражений. В языке, накладывающем на программиста более строгие правила, нам бы это не удалось.

Scheme разрешает использовать присваивание, но поощряет функциональное программирование. Язык Scheme не содержит статических типов, но в нем очень строгая динамическая типизация, что позволяет иметь безопасное выделение памяти и сборку мусора: пользовательская программа не может создать указатель из ничего или получить доступ к произвольной ячейке памяти. Тут дело не в том, что мы считаем статическую типизацию плохой идеей. Она, безусловно, полезна для изгнания многих видов ошибок из программы на раннем этапе. Более того, система типов вроде той, что в языке Haskell, может быть полезна при продумывании стратегий программирования. Однако в этой книге интеллектуальный вес статических типов помешал бы рассматривать потенциально опасные стратегии повышения гибкости.

Кроме того, Scheme имеет некоторые особые свойства, например явную поддержку продолжений и динамического связывания, которых нет в большинстве языков. Эти свойства позволяют нам реализовать такие мощные механизмы, как недетерминистский оператор `amb`, в самом языке (не используя отдельный уровень интерпретации).

Благодарности

Создание этой книги было бы невозможно без помощи множества студентов MIT, прошедших через наш курс. Они решали все упражнения и часто показывали, где мы сделали неверный выбор или ошиблись. Мы чрезвычайно благодарны студентам, которые в течение многих лет работали ассистентами на нашем курсе. Особенно полезны были Майкл Блэр, Алексей Радул, Павел Панчеха, Роберт Л. Макинтайр, Ларс Э. Джонсон, Эли Дэвис, Мика Бродский, Манушаке Муко, Кенни Чен и Лейлани Хендрина Гилпин.

Многие из представленных здесь идей были разработаны с помощью наших друзей и бывших студентов. Ричард Столлмен, Джон Дойл, Дэвид Макаллистер, Рамин Забих, Йохан Деклер, Кен Форбус и Джефф Сискинд помогли нам развить наше понимание перебора с возвратами, управляемого зависимостями. Наше понимание распространения из главы 7 является результатом многих лет совместной работы с Ричардом Столлменом, Гаем Льюисом Стилом мл. и Алексеем Радулом.

Мы крайне благодарны за помощь и поддержку сообществу функциональных программистов, а особенно Команде Scheme. Еще в 70-е годы Гай Стил придумал Scheme вместе с Джеральдом Джеем Сассманом, а на нашем курсе он почти каждый год выступал с гостевой лекцией. Артур Глекер, Гильермо Хуан Росас, Джо Маршалл, Джеймс С. Миллер и Генри Маньян Ву участвовали в разработке MIT/GNU Scheme. Важный вклад в эту досточтимую систему внесли Тейлор Кэмпбелл и Мэтт Биркхольц. Мы также хотели бы поблагодарить Уилла Бёрда и Майкла Баллантайна за помощь в понимании унификации с сегментными переменными.

Хал Абельсон и Джули Сассман, соавторы Джеральда Джея Сассмана в книге *Структура и интерпретация компьютерных программ* (SICP), помогли нам сформировать идеи этой книги. Во многом эта книга служит продвинутым продолжением SICP. Дэн Фридман и его многочисленные замечательные друзья и студенты внесли глубокий вклад в наше понимание программирования. У нас было множество разговоров об искусстве программирования с некоторыми из величайших волшебников этого дела, включая Уильяма Кахана, Ричарда Столлмена, Ричарда Гринблатта, Билла Госпера и Тома Найта. Многолетняя совместная работа с Джеком Уиздомом в области математической динамики позволила прояснить многие из задач, которыми мы занимаемся в этой книге.

Джеральд Джей Сассман особенно хочет отметить влияние своих учителей: идеи, взятые из обсуждений с Марвином Минским, Сеймуром Пейпертом, Джеромом Леттвиным, Джоэлом Мозесом, Полом Пенфилдом и Эдвардом Фредкиным, занимают в тексте книги важное место. Присутствуют здесь также и идеи Карла Хьюитта, Дэвида Уолтца и Патрика Уинстона, которые были студентами Минского

и Пейперта одновременно с Сассманом. Джефф Сискинд и Алексей Радул нашли и помогли уничтожить некоторые очень нетривиальные ошибки.

Крис многое узнал о программировании больших систем, работая в Гугле, а затем в компании Датера; этот опыт повлиял на некоторые разделы нашей книги. Артур Глекер предоставил важную обратную связь во время регулярных совместных обедов. Во время наших частых встреч в Гугле Майк Солсбери всегда был рад слышать последние новости о продвижении работы над текстом. Хунтао Хуанг и Пиюш Джанавадкар прочитали ранние черновики книги. Особую благодарность следует принести Рику Дьюксу, однокурснику Криса в MIT, который показал ему статьи о лямбде и таким образом направил на долгий путь, приведший к созданию этой книги.

Мы благодарны кафедре электротехники и информатики MIT, а также лаборатории информатики и искусственного интеллекта MIT (CSAIL) за радушие и логистическую поддержку. Джеральд Джей Сассман отмечает спонсорство корпорации Panasonic (ранее электропромышленная корпорация Matsushita) в форме профессорского гранта. Крис Хансон также частично получал финансирование от CSAIL, а затем от Гугла.

Джули Сассман внимательно прочла текст и предоставила серьезную критику, которая заставила нас существенно переработать и переписать большие его части. Кроме того, в течение долгих лет она развивала и сопровождала Джеральда Джея Сассмана.

Элизабет Викерс, супруга Криса, в течение многих лет создавала стабильную и уютную атмосферу для него и для их детей, Алана и Эрики. Кроме того, во время долгих рабочих встреч в штате Мэн Элизабет приготовила для обоих авторов множество замечательных обедов и ужинов. Алан нерегулярно, но с энтузиазмом читал некоторые из черновиков.

Крис Хансон и Джеральд Джей Сассман

Глава 1

Гибкость в природе и в проектировании

Спроектировать общеполезный механизм, который будет хорошо справляться с любой работой, очень трудно. Поэтому большинство инженерных систем предназначено для выполнения какой-то конкретной задачи. Изобретения общего назначения, такие как винт с резьбой, случаются редко и оказывают большое влияние. В этом отношении цифровой компьютер является прорывом, поскольку это универсальная машина, способная сымитировать любое другое устройство для обработки информации[1]. Мы пишем программы, которые конфигурируют наш компьютер, чтобы он выполнял такую имитацию с прицелом на конкретные нужные нам задачи.

Мы научились разрабатывать программные системы, отлично решающие конкретные задачи, в качестве расширения обычной инженерной практики прошлых лет. Каждый фрагмент программы предназначен для выполнения некоторой относительно узкой задачи. Если решаемая задача изменяется, требуется модифицировать и программу. Однако не всегда небольшие изменения в задаче приводят лишь к небольшим изменениям в программе. Программа слишком плотно подогнана под конкретный вид работы, чтобы оставалось место для гибкости. Вследствие этого программные системы оказываются неспособны развиваться, сохраняя изящество. Они хрупки, и, если область приложения изменяется, такой проект требуется заменить на совершенно новый[2]. Выходит медленно и дорого.

Инженерные системы не обязательно должны быть хрупкими. Например, интернет выдержал рост от небольшой системы до глобальной. Города способны органично развиваться, воспринимая новые модели бизнеса, стили жизни, виды транспорта и связи. Глядя на биологические системы, мы видим, что возможно построить структуры, приспосабливающиеся к изменениям в окружении, как индивидуально, так и

[1] Открытие универсальных машин Тьюрингом [124], а также то,что множество функций, вычислимых машинами Тьюринга, эквивалентно как множеству функций, представимых в λ-исчислении Алонсо Чёрча [17, 18, 16], так и общерекурсивным функциям Курта Гёделя [45] и Жака Эрбрана [55], — одно из величайших интеллектуальных достижений XX в.

[2] Разумеется, существуют замечательные исключения. Например, *Emacs* — расширяемый редактор кода, изящно приспособившийся как к изменениям в вычислительном окружении, так и к изменениям в ожиданиях пользователя. В мире информатики только начинают исследовать «инженерные фреймворки», например `.net` от Микрософт или Java компании Sun. Такие конструкции предназначены быть инфраструктурой, поддерживающей развивающиеся системы.

в качестве эволюционного ансамбля. Почему же большинство программ не проектируется и не пишется таким образом? На то есть исторические причины, однако главная причина состоит в том, что мы не знаем, как этого добиться в общем случае. До сих пор если система оказывается способной выдержать изменение в требованиях к ней, то это лишь счастливое совпадение.

Аддитивное программирование

Нашей целью в этой книге будет исследовать, как можно строить вычислительные системы, которые можно легко приспособить к новым задачам. Работающую программу не следует модифицировать. Должно быть возможным добавить к ней реализацию новой функциональности либо подстроить старые функции под новые требования. Мы называем такую цель *аддитивным программированием*. Мы будем исследовать методы, позволяющие добавить функциональность к существующей программе, не ломая ее. Это не гарантирует, что добавления будут правильно работать: их тоже надо отлаживать; однако они не должны нечаянно разрушать существующую функциональность.

Многие из методов, исследуемых нами в этой книге, не новы: некоторые существуют с самых ранних лет работы с компьютерами! Они также не составляют единого согласованного набора, это просто коллекция приемов, которые кажутся нам полезными. Мы не столько хотим рекомендовать использование этих конкретных методов, сколько стимулировать стиль мышления, направленный на гибкость.

Для того чтобы сделать аддитивное программирование возможным, необходимо минимизировать предположения о том, как программа работает и как она будет использоваться. Решения, принятые на этапе проектирования и написания программы, могут сузить ее будущие возможные расширения. Вместо того чтобы делать такие предположения, мы строим программы так, чтобы решения принимались точно вовремя на основе конкретного окружения, где им предстоит работать. Мы рассмотрим несколько приемов, поддерживающих такой стиль проектирования.

Программы всегда можно сочетать так, чтобы получилось объединение режимов поведения, обеспечиваемых каждой из них. Однако нам хочется, чтобы целое было больше, чем сумма его частей; нужно, чтобы части общей системы сотрудничали и чтобы в результате появлялись возможности, которые ни одна из компонент сама по себе предоставить не может. Однако здесь необходимы компромиссы: части, из которых мы составляем систему, должны четко разделять зоны ответственности. Если модуль программы хорошо решает одну задачу, его проще заново использовать и проще отлаживать, чем такой, который объединяет несколько различных задач. Если нам требуется аддитивное построение, важно, чтобы отдельные части сочетались с минимумом нежелательных взаимодействий.

Для достижения аддитивного программирования необходимо, чтобы компоненты нашей системы были как можно проще и обладали максимальной общностью. К примеру, компонента, принимающая более широкое множество возможных входных данных, чем строго необходимо для исходно поставленной задачи, будет применима в большем числе случаев, чем та, где этим не озаботились. Семейства модулей, построенных на основе стандартизированного интерфейса, можно сочетать и смешивать, получая великое множество разных систем. Важно также правильно выбрать уровень абстракции наших компонент, выделив для них общую предметную область,

и затем построить семейство, поддерживающее эту предметную область. Мы начнем рассматривать эти требования в главе 2.

Чтобы добиться максимальной гибкости, множество порождаемых значений компоненты должно быть как можно уже и как можно лучше определено — намного меньше, чем множество приемлемых входов для любого модуля, который будет считывать эти данные. Это аналогично статической дисциплине в цифровой абстракции, которой мы обучаем студентов на вводных курсах по вычислительным системам [126]. Сущность цифровой абстракции состоит в том, что выход предыдущего модуля всегда должен быть лучше, чем приемлемое качество входа следующего, так что шум подавляется.

В программной инженерии этот принцип известен как «закон Постела», названный в честь одного из основателей интернета Джона Постела. Он писал в RFC760 [97], описывающем протокол IP: «Реализация протокола должна быть гибкой и разумной. Каждая реализация должна предполагать интероперабельность с продукцией других разработчиков. Хотя целью данной спецификации является четкое и строгое описание протокола, существует вероятность различных интерпретаций стандарта. При передаче дейтаграмм следует строго следовать спецификации, сохраняя в то же время готовность к восприятию любых дейтаграмм, которые можно интерпретировать»*. Обычно это формулируется так: «Будь консервативным в собственном поведении, но либеральным при восприятии чужого».

При использовании модулей, приспособленных к более общим задачам, чем кажется строго необходимым, вся структура нашей системы приобретает дополнительную гибкость. Она выдерживает небольшие нарушения исходных требований, поскольку каждая компонента готова воспринять нарушенные (шумные) входные данные.

Семейство компонент, способных сочетаться друг с другом, служит основой *специализированного языка* для некоторой предметной области. Часто наилучшим способом решить сразу несколько сложных задач является создание нового языка — набора элементарных конструкций, средств их сочетания и абстракции, — который облегчает формулировку решений для этих задач. Поэтому мы хотим научиться производить специализированные языки по необходимости, а также гибко сочетать их. Мы начинаем рассматривать специализированные языки в главе 2. Еще более мощным средством является реализация таких языков методом прямого вычисления. Эта идея рассматривается в главе 5.

Одна из стратегий увеличения гибкости, знакомая многим программистам, называется *полиморфные вызовы*. Мы тщательно исследуем ее в главе 3. Полиморфный вызов часто позволяет расширить область применимости процедуры путем добавления дополнительных обработчиков (методов) в зависимости от того, какие аргументы передаются процедуре. Если мы потребуем, чтобы множества аргументов, на которые реагируют обработчики, не пересекались, мы избежим поломки программы при добавлении каждого нового обработчика. Однако, в отличие от полиморфного вызова в типичном объектно ориентированном контексте, наша концепция полиморфизма не предполагает таких понятий, как классы, экземпляры или наследование. Эти понятия ослабляют разграничение задач, поскольку вносят излишние онтологические обязательства.

Совершенно другая стратегия, рассматриваемая в главе 6, — разделение *слоев* как данных, так и процедур. Здесь используется идея, что данные обычно снабжены

*Перевод Н. Малых. — *Прим. перев.*

метаданными, которые можно обрабатывать отдельно от них. К примеру, числовые данные обычно ассоциируются с единицами измерения. Мы увидим, как гибкое добавление слоев к готовой программе может расширить ее функциональность без всякого изменения исходного кода.

Кроме того, возможно построение систем, сочетающих различные источники *частичной информации* и получающих более полные результаты. Это работает особенно хорошо, если вклады разных источников независимы друг от друга. В главе 4 мы увидим, что вывод типов сводится к сочетанию различных источников частичной информации. Локально выводимые свидетельства о типе значений, например что численное сравнение требует числа на входе и получает на выходе булево значение, можно скомбинировать с другими локальными ограничениями и получить нелокальные ограничения.

В главе 7 мы увидим еще один способ сочетания частичной информации. Расстояние до соседней звезды можно оценить геометрическим методом параллакса: измерить угол, на который образ звезды сдвигается относительно более далеких объектов, когда Земля вращается вокруг Солнца. Расстояние до звезды также можно оценить, рассматривая ее яркость и спектр, используя наши знания о структуре и эволюции звезд. Эти оценки можно сравнить друг с другом и получить новую оценку, которая будет точнее, чем сделанная каждым отдельным методом.

Двойственная идея состоит в *дублировании*, когда есть несколько способов получения некоторых данных, которые мы можем комбинировать и сравнивать. Существует множество способов использовать дублирование, включая поиск ошибок, управление производительностью и распознавание несанкционированного доступа. Важно также то, что система с дублированием аддитивна: каждая ее компонента самодостаточна и может получить результат сама по себе. Один из интересных способов использовать дублирование состоит в том, чтобы динамически выбирать среди нескольких реализаций алгоритма в зависимости от контекста. При этом можно избежать предположений о том, как именно будет использоваться каждая отдельная реализация.

Проектирование и встраивание гибкости в систему влечет определенные затраты. Процедура, способная воспринять больше видов данных, чем необходимо для конкретной задачи, будет содержать больше кода, чем нужно для решения прямо сейчас, и потребует от программиста больше размышлений, чем абсолютно необходимо прямо сейчас. То же самое относится к полиморфным вызовам, многослойной организации и дублированию; каждое из них требует постоянных затрат памяти, времени вычислений и/или времени работы программиста.

Однако основные затраты на программное обеспечение — усилия, затраченные программистами за все время жизни продукта, включая поддержку и адаптации, необходимые при изменяющихся требованиях. Методы проектирования, которые минимизируют переписывание и рефакторинг, снижают общую стоимость, сводя ее к последовательным добавкам функциональности вместо полной переработки. Другими словами, долговременные затраты становятся аддитивными вместо мультипликативных.

1.1 Архитектура вычислений

Для систем того рода, что мы имеем в виду, может быть полезна архитектурная метафора. После того как исследованы природа участка, на котором предполагается

строить, и требования к возводимой постройке, процесс проектирования начинается с *эскизного проекта*: принципов организации работы[3]. Как правило, эскизный проект представляет собой набросок геометрического расположения частей здания. Он может также выражать некоторые абстрактные идеи, например разделение между «рабочими пространствами» и «обслуживающими пространствами», как в работах Луиса Исидора Кана [130]. Такая декомпозиция служит для того, чтобы разделить архитектурную задачу на части, отделив инфраструктурную поддержку вроде коридоров, туалетов, серверных или лифтов, от пространств, которые нужно поддерживать, таких как лаборатории, учебные классы и офисы учебного здания.

Эскизный проект является моделью, однако обычно это еще не полностью работоспособная структура. Требуется развить его, дополнив функциональными элементами. Где проходят воздуховоды вентиляции, водопроводные трубы, системы распределения электричества и связи? Где проложить дорогу, чтобы обеспечить удобство доставки грузов и обслуживание постройки? Эти уточнения могут потребовать некоторых изменений изначального эскиза, однако он продолжает служить как каркас, на который опираются конкретные решения.

В программировании эскизный проект — это абстрактный план вычисления, которое надо произвести. В небольшом масштабе это может быть абстрактный алгоритм или описание структуры данных. В более крупных системах эскиз может содержать абстрактную композицию фаз и параллельных ветвей вычисления. В еще более крупных проектах в него может входить распределение задач по логическим (или даже физическим) локациям.

Исторически сложилось так, что у программистов обычно нет возможности строить эскизные проекты подобно архитекторам. В подробных языках вроде Java эскиз тесно перемешан с проработкой деталей. «Рабочие пространства», выражения, напрямую описывающие требуемое поведение, никак не отделены от «обслуживающих пространств» вроде объявлений типов и классов, а также импорта и экспорта библиотек[4]. Более свободные языки, такие как Lisp и Python, почти не оставляют места для обслуживающих пространств, и попытки добавить к ним объявления типов, даже в виде аннотаций, встречают сопротивление, поскольку они затемняют красоту и ясность эскиза.

В архитектуре эскизный проект должен быть достаточно полон, чтобы на его основе строить модели, доступные для анализа и критики. Скелетная часть программы также должна быть достаточно адекватна для анализа и критики, но она также должна быть исполнима, чтобы производить с ней эксперименты и отлаживать. Так же, как архитектор должен дополнить эскизный проект, чтобы полностью реализовать проектируемую структуру, программист должен дополнить базовый план, чтобы воплотить требуемую вычислительную систему. Многослойная разработка (описываемая нами в главе 6) — один из способов построения систем, позволяющих такую постепенную детализацию.

[3] Эскизный проект представляет собой основную идею архитектурного произведения: это «представленная в целом композиция, детали которой должны быть проработаны позже».[62]

[4] В Java есть интерфейсы, которые можно рассматривать как разновидность эскизного проекта, поскольку они являются абстрактным представлением программы. Однако в архитектуре эскизный проект сочетает абстрактные и конкретные компоненты, а интерфейсы языка Java целиком абстрактны. К тому же многие программисты полагают, что чрезмерное использование интерфейсов «плохо пахнет».

1.2 Гибкость через умные компоненты

Большие системы строятся из множества компонент меньшего размера, и каждая из них участвует в выполнении общей задачи, либо напрямую решая ее часть, либо через взаимодействие с другими компонентами по схеме, определенной архитектором системы. В проектировании систем центральной проблемой является разработка интерфейсов, позволяющих связывать компоненты, чтобы функции этих компонент сочетались в построении составной функции.

В относительно простых системах системный архитектор может указать формальные спецификации различных интерфейсов, которые производители соединяемых компонент должны выполнить. Поразительный успех электроники основывается на том, что в этой области возможно составить такие спецификации и выполнить их. Высокочастотные аналоговые устройства связываются между собой с помощью коаксиальных кабелей со стандартизованным сопротивлением и стандартизованными семействами разъемов [4]. Как назначение устройства, так и поведение его интерфейса обычно можно описать с помощью лишь нескольких параметров [60]. В цифровых системах все еще проще: имеются статические описания значений сигналов (цифровая абстракция), динамическое описание временных характеристик сигналов [126], и, наконец, есть механические описания форм-факторов компонент[5].

К сожалению, подобная априорная спецификация становится все труднее по мере возрастания сложности систем. Можно указать, что программа для игры в шахматы должна играть *по правилам* — не должна их нарушать, — но как нам удастся задать требование, чтобы она играла в шахматы *хорошо*? Программные системы строятся из большого количества уникальных, сильно специализированных частей. Сложность спецификации программных компонент усиливается от того, что многие из них обладают индивидуальными характеристиками.

В противоположность этому биология способна создавать системы умопомрачительной сложности без особенно больших описаний (если даже считать проблему описаний решенной!). Каждая клетка нашего тела происходит от единственной зиготы. Все клетки содержат идентичную наследственность (около 1 Гб постоянной памяти!). Однако существуют клетки кожи, нейроны, мускульные клетки и т. д. Клетки самоорганизуются в ткани, органы и системы органов. В итоге 1 Гб ПЗУ описывает, как построить чрезвычайно сложную машину (человека) из большого числа ненадежных компонент. В этой информации задается, как работают эти компоненты и как их конфигурировать. Кроме того, описывается, как надежно управлять этой машиной в разнообразных неблагоприятных обстоятельствах в течение долгого времени жизни и как эту машину защищать от других, которые бы хотели ее съесть!

Если бы наши программные компоненты были проще или более широко применимы, их спецификации тоже были бы проще. Если бы компоненты умели при-

[5] *Книга данных TTL для инженеров-проектировщиков*[123] может служить классическим примером набора спецификаций компонент для цифровых систем. В TTL описываются несколько внутренне согласованных «семейств» интегральных схем малого и среднего масштаба. Эти семейства различаются своими скоростными и температурными характеристиками, но не функциями. Спецификация указывает статические и динамические характеристики каждого семейства, функции, доступные в каждом семействе, и физические параметры упаковки компонент. Компоненты согласованы не только внутренне, но и между собой, т. е. каждая функция доступна в каждом семействе, в одинаковой упаковке и с единой номенклатурой описания. Таким образом, инженер может спроектировать некоторую составную функцию и только затем выбрать семейство для ее реализации. Всякий хороший инженер (и даже биолог!) должен усвоить уроки TTL.

способливаться к окружению, точность спецификаций была бы не так важна. При помощи обеих этих стратегий биологическим системам удается построить надежные сложные организмы. Разница состоит в том, что биологические клетки конфигурируются динамически и могут приспосабливаться к контексту. Это оказывается возможным, поскольку то, как клетка дифференцируется и специализируется, зависит от ее окружения. Обычно наши программы не обладают этим свойством, и поэтому нам приходится настраивать каждый их модуль вручную. Как же в принципе способна работать биология?

Рассмотрим еще один пример. Известно, что различные части мозга соединяются огромными пучками нервов, и в геноме даже близко не достаточно информации, чтобы во всех деталях описать эти связи. Вероятно, различные части мозга учатся взаимодействовать друг с другом, основываясь на сходстве своего опыта[6]. Таким образом, интерфейсы должны самоконфигурироваться на основании некоторых правил непротиворечивости, информации о среде и активного исследовательского поведения. Это имеет высокую цену во время начальной загрузки (конфигурация полноценного человека занимает несколько лет), но в результате система обладает устойчивостью, недостижимой для наших современных инженерных систем.

Одна из идей состоит в том, что биологические системы используют не императивные, а информативные контекстные сигналы[7]. Не существует управляющего блока, который говорит, что делать каждой из частей; вместо этого части самостоятельно выбирают роли на основе своего окружения. Поведение клеток не кодируется в сигналах; оно отдельно выражено в геноме. Комбинации сигналов только включают одни модели поведения и отключают другие. Такое слабое связывание обеспечивает вариативность способов поведения, срабатывающих в различных точках организма, без того, чтобы изменять механизм, определяющий сами эти точки. Организованные таким образом системы способны к развитию, поскольку они могут запускать адаптивное варьирование в некоторых точках, не меняя поведение подсистем в других точках.

Традиционные программные системы строятся по императивной модели, где существует иерархия управления, однозначно определяемая структурой. Отдельные части системы играют роль безвольных подчиненных, которые делают то, что им сказано. Адаптация в таких системах становится сложной, потому что любые изменения должны отражаться во всей структуре управления. В общественных системах нам известно, какие проблемы происходят из-за строгих структур власти и централизованного управления. Однако наши программы следуют этой порочной модели. Можно сделать лучше: если наши компоненты будут умнее и при этом будут по отдельности отвечать за свое поведение, им будет проще приспосабливаться, поскольку на каждое изменение придется реагировать только тем частям системы, которых оно непосредственно касается.

Планы строения

Все позвоночные имеют почти одинаковый план строения, однако вариация в его деталях громадна. Действительно у всех животных с двусторонней симметрией имеются одни и те же гомеобоксные гены, такие как комплекс Нох. Эти гены создают

[6] Простейшую версию такого самоконфигурирующегося поведения продемонстрировал Джейкоб Бил в магистерской диссертации [9].

[7] Ее исследуют Киршнер и Герхарт [70].

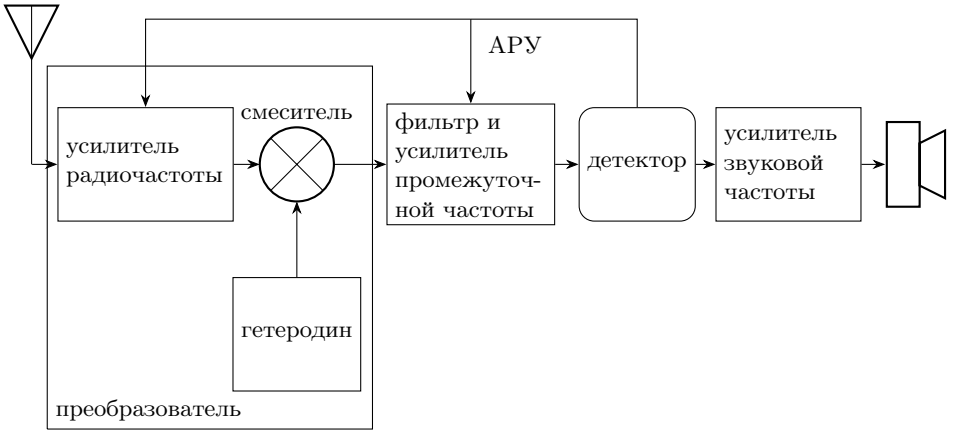


Рис. 1.1. Супергетеродинная схема, которую в 1918 году изобрел майор Эдвин Армстронг, по-прежнему остается преобладающим «планом строения» радиоприемника

приблизительную систему координат при развитии животного, разделяя формирующееся тело на различные области[8]. Каждая область служит контекстом для дифференциации клеток. Информация, выводимая из контакта с соседними клетками, создает дополнительный контекст, на основе которого выбирается конкретное поведение среди моделей возможного поведения, заложенных в генетическую программу клетки[9]. Даже методы построения тканей используются одни и те же — морфогенез проточных желез и органов типа легких и почек основываются на одном эмбриологическом приеме: внесение эпителия в мезенхиму автомагически[10] порождает ветвящийся лабиринт слепых трубочек, окруженных дифференцирующейся мезенхимой[11].

Качественная инженерная работа отличается тем же стилем: хороший проект всегда разбивается на модули. Рассмотрим устройство радиоприемника. Было изобретено несколько общих «планов строения»: приемник прямого преобразования, прямого усиления и супергетеродинный. Каждый из них содержит последовательность локаций, определяемых инженерным эквивалентом комплекса Нох, который накладывает шаблон на систему от антенны до выходного преобразователя. Например, супергетеродинный приемник (рис. 1.1) разбивается на стандартные локации, от носа до хвоста.

Каждый из модулей, указанных на этом плане, сам делится на более мелкие модули — генераторы сигналов, смесители, фильтры и усилители и т. д. вплоть до отдельных электронных деталей. Кроме того, каждый модуль может быть реализован несколькими различными способами: блок радиочастоты может быть просто

[8]Это лишь приблизительно описывает сложный процесс, включающий градиенты морфогенов. Мы здесь не стремимся к большей точности, поскольку речь идет не о биологии, а о том, как знание биологии может помочь инженеру.

[9]Мы исследовали некоторые вопросы программирования при развитии такого рода в проекте Аморфных вычислений [2].

[10]Автомагически: «Автоматически, но таким способом, который автор почему-то (как правило из-за чрезмерной сложности, или чрезмерного уродства, или даже тривиальности) не считает нужным объяснять». Из *Словаря хакера* [117, 101].

[11]Один из хорошо изученных примеров такого механизма — формирование подчелюстной железы у мыши. См., например, исследование [11] или краткое описание в [7], раздел 3.4.3.

фильтром, а может представлять собой сложную комбинацию фильтра и усилителя. В аналоговом телевизионном приемнике часть выхода смесителя обрабатывается видеоцепью как сигнал с амплитудной модуляцией, а часть — аудиоцепью как сигнал с частотной модуляцией. Некоторые блоки, например преобразователь, могут разворачиваться рекурсивно (как если бы часть Нох-комплекса была продублирована!), так что получаются приемники с множественным преобразованием.

В биологических системах структура разделения блоков поддерживается и на более высоких уровнях организации. Имеются ткани, специально предназначенные разделять отделы тела, и трубки, соединяющие их. Органы разграничиваются такими тканями и связываются этими трубками, а вся структура упакована в целомы — в высших организмах это полости, выстеленные специальными тканями.

Подобные же приемы можно применять и в построении программ. План строения — это всего лишь обертка, соединяющая частично специфицированные компоненты. Это разновидность *комбинатора*: нечто, связывающее блоки в более крупный блок. Можно создавать *комбинаторные языки*, где компоненты и составленное из них целое имеют одну и ту же спецификацию интерфейса. В комбинаторном языке можно создать композитные блоки произвольного размера, сочетая между собой небольшое количество видов элементарных компонент. Самоподобные структуры облегчают построение композитов. Мы начнем строить программы на основе комбинаторов в главе 2, а затем эта тема будет сопровождать нас на протяжении всей книги.

Нечто подобное можно сделать с помощью специализированных языков. Абстрагировав предметную область, можно применять один и тот же предметно-независимый код в разных областях. Например, численные интеграторы полезны всюду, где используются числа, независимо от предметной области. Другим примером может служить сопоставление с образцом из главы 4, которое также применимо в большом количестве ситуаций.

Биологические механизмы универсальны в том смысле, что, в принципе, любая компонента может работать как любая другая. Аналоговые электронные компоненты не являются в этом смысле универсальными. Они не могут приспосабливаться к своему окружению на основе локальных сигналов. Однако существуют универсальные электронные строительные блоки (например, программируемый компьютер с аналоговыми интерфейсами!)[12]. Для низкочастотных применений можно строить аналоговые системы из таких блоков. Если бы в каждом блоке содержался весь код, требуемый для построения любого блока системы, но он бы получал специализацию путем взаимодействия с соседями, и если бы в общем наборе были неспециализированные «стволовые клетки», мы могли бы представить самоконфигурирующиеся и саморемонтирующиеся аналоговые системы. Но в наше время мы пока что проектируем и строим части системы по отдельности.

В программировании у нас есть понятие универсального элемента: *вычислитель*. Вычислитель принимает на вход описание некоторого подлежащего исполнению вычисления и входные данные для него. Он выдает выходные значения, которые были бы получены, если бы входные данные были скормлены специализированному компоненту, предназначенному выполнять это вычисление. В области вычислений у нас есть возможность следовать мощной в своей гибкости стратегии эмбриональ-

[12]Петр Митрос разработал новую стратегию проектирования для построения аналоговых цепей из потенциально универсальных строительных блоков. См. [92].

ного развития. Мы подробнее рассмотрим использование стратегии вычислителя в главе 5.

1.3 Избыточность и дублирование

Биологические системы обладают немалой прочностью. Одна из характеристик биологических систем — они всегда *избыточны*. Органы вроде печени или почек обладают этой избыточностью в большой степени: их мощность значительно превосходит необходимый для жизни минимум, так что человек без одной из почек или части печени не теряет в качестве жизни. Кроме того, в биологических системах развито *дублирование*: как правило, любое требование можно выполнить множеством различных способов[13]. Например, если поврежден палец, можно сложить остальные пальцы и схватить ими объект. Необходимую для жизни энергию можно получить из множества различных видов сырья: метаболизму подвергаются углеводы, жиры и белки, хотя механизмы пищеварения и извлечения энергии для каждого из этих источников сильно различаются.

Генетический код сам по себе обладает свойством дублирования, поскольку кодоны (тройки нуклеотидов) отображаются на аминокислоты не один к одному: 64 возможных кодона задают всего лишь около 20 возможных аминокислот [86, 54]. В результате многие точечные мутации (замены одного нуклеотида) не изменяют белок, задаваемый кодирующей областью. Кроме того, замена одной аминокислоты на похожую часто не ухудшает биологическую активность белка. Такое дублирование обеспечивает способы накопления изменчивости без очевидных фенотипических последствий. Более того, если сам ген дублируется (что тоже случается нередко), его копии могут молчаливо разойтись, позволяя таким образом развиться вариантам, которые могут оказаться полезны в будущем, без влияния на текущую жизнеспособность организма. Вдобавок копии могут оказаться под управлением различных механизмов транскрипционного контроля.

Дублирование является результатом эволюции, но оно и помогает эволюции. Скорее всего, дублирование само по себе поддерживается отбором, поскольку только существа со значительным дублированием обладают достаточной приспособляемостью, чтобы выжить при изменениях среды обитания[14]. Допустим, например, что у нас есть некоторое существо (или инженерная система), обладающее дублированием в том смысле, что для некоторой существенной функции у него есть несколько различных независимых механизмов. Если среда обитания (или технические требования) изменится так, что один из способов исполнения этой функции перестанет работать, существо продолжит жить и размножаться (система продолжит работать согласно спецификации). Однако подсистема, которая стала нерабочей, теперь доступна для мутации (или ремонта) без влияния на жизнеспособность (текущую работоспособность) системы в целом.

Теоретическое строение физики глубоко дублировано. Например, задачи классической механики могут решаться несколькими способами. Существует ньютоновская формулировка векторной механики и лагранжевская или гамильтоновская формулировка вариационной механики. Если применимы и ньютоновская, и какая-либо из

[13]Различие, проводимое биологами между избыточностью и дублированием, ясно проявляется в типичных случаях, но в пограничных условиях расплывается. Более подробно см. [32].

[14]Специалисты по информатике исследовали эволюцию приспособляемости методами имитационного моделирования [3].

форм вариационной формулировки, они дают эквивалентные уравнения движения. Для анализа систем с диссипативными силами (такими, как трение) эффективна векторная механика; для таких систем вариационные методы плохо приспособлены. Лагранжева механика работает намного лучше, чем векторная, для систем с жесткими ограничениями, а гамильтонова механика дает канонические преобразования, которые позволяют лучше понять системы с помощью структуры фазового пространства. Как лагранжевская, так и гамильтоновская формулировки отражают глубокую интуицию по поводу роли симметрий и законов сохранения. То, что существует три перекрывающихся способа описания механической системы, согласованных между собой в тех случаях, когда все они применимы, позволяет решать каждую конкретную задачу наиболее удобным образом [121].

Инженерные системы могут обладать некоторой избыточностью в случае критически важных систем, где цена сбоя очень высока. Однако они почти никогда не включают намеренное дублирование такого типа, как в биологических системах, кроме как в виде побочного результата неоптимального проектирования[15].

Дублирование может приносить пользу в наших системах: как и в случае избыточности, можно сравнивать результаты дублированных вычислений, что повышает надежность. Однако дублированные вычисления не просто избыточны, они *отличаются* друг от друга, а значит, программная ошибка в одном из них, скорее всего, не влияет на остальные. Это положительно влияет не только на надежность, но и на безопасность, поскольку успешная атака должна будет скомпрометировать несколько продублированных модулей.

В случае, когда дублированные модули порождают частичную информацию, комбинация результатов может оказаться лучше, чем каждый отдельный результат. Некоторые навигационные системы используют эту идею, порождая точную оценку положения на основе нескольких оценок. Мы исследуем идею комбинирования частичных описаний в главе 7.

1.4 Исследовательское поведение

Исследовательское поведение — один из самых мощных механизмов повышения надежности биологических систем[16]. Идея состоит в том, что желаемый результат достигается при помощи процесса порождения и фильтрации гипотез (см. рис. 1.2). Такая организация позволяет порождающему механизму (генератору) быть достаточно общим и работать независимо от проверяющего механизма (тестера), который принимает либо отвергает каждый отдельный порожденный результат.

Например, важным компонентом жесткого скелета, поддерживающего форму клетки, служит массив микротрубочек. Каждая трубочка состоит из белковых единиц, формирующих ее. В живой клетке трубочки постоянно создаются и разрушаются и растут во всех направлениях. Однако стабильны только те из них, которые встречаются с кинетохором или другим стабилизатором в мембране клетки, и таким образом поддерживается форма, определяемая позицией стабилизаторов [71]. Поэтому механизм роста и поддержания формы относительно независим от механиз-

[15]Напротив, часто можно слышать аргументы против встраивания дублирования в техническую систему. Например, философия компьютерного языка Python утверждает: «Должен существовать один — и желательно только один — очевидный способ сделать это» [95].

[16]Это утверждение исследуется в книге Киршнера и Герхарта [70].

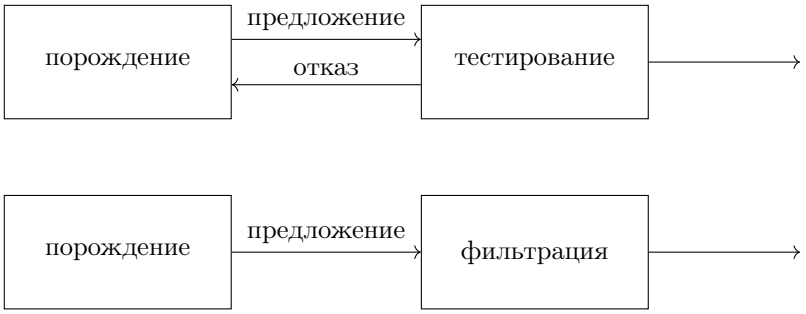


Рис. 1.2. Исследовательское поведение может быть организовано двумя способами. В одном случае порождающий механизм предлагает некоторое действие (или результат), а тестирующий компонент может явным образом его отвергнуть. В таком случае генератор должен породить альтернативу. Во втором случае генератор порождает все альтернативы без какой-либо обратной связи, а фильтр выбирает из них одну или более приемлемую

ма, определяющего эту форму. Такой механизм отчасти определяет форму клеток сложного организма и существует почти у всех животных.

Исследовательское поведение наличествует на всех уровнях детализации биологических систем. Нервная система растущего эмбриона порождает значительно больше нейронов, чем будет нужно взрослому организму. Те из них, которые найдут подходящие цели в других нейронах, органах чувств или мышцах, выживут, а те, которые цели не найдут, самоуничтожатся. Рука образуется через рост пластинки с последующим уничтожением, методом апоптоза (программируемой смерти клеток), ненужного материала между пальцами [131]. Кости в нашем организме постоянно пересоздаются остеобластами (которые строят костный материал) и остеокластами (которые его разрушают). Форма и размер костей определяются ограничениями, задаваемыми их окружением: органами, с которыми им нужно связываться, а именно мышцами, связками, сухожилиями и другими костями.

Поскольку генератор не обязан знать, как тестер принимает или отвергает его предложения, а тестеру не обязательно знать, как формирует предложения генератор, эти два механизма могут развиваться независимо друг от друга. Эволюция и адаптация от этого становятся эффективнее, поскольку мутация в каждой из этих подсистем не обязана сопровождаться парной мутацией в другой. Однако, с другой стороны, такая изоляция может стоить дорого из-за цены, которую приходится платить за порождение и оценку неудачных предложений[17].

В сущности, порождение и проверка являются метафорой эволюции в целом. Механизмы биологического варьирования — случайные мутации, модификации генетических программ. Большинство этих мутаций нейтрально в том смысле, что они прямо не влияют на приспособленность по причине дублирования в системах. Естественный отбор служит фазой проверки. Он не зависит от того, как работает варьирование, а метод варьирования не знает заранее результатов отбора.

[17]Эти расходы можно значительно сократить, если имеется достаточная информация, позволяющая сократить на раннем этапе число подлежащих тестированию кандидатов. Мы рассмотрим изящный пример такой оптимизации в главе 7.

Существуют еще более удивительные явления: даже в близкородственных существах компоненты, которые выглядят почти одинаково во взрослом организме, могут на стадии зародыша порождаться совершенно разным образом[18]. В случае дальнего родства различие механизмов построения сходных структур можно приписать «конвергентной эволюции», однако для близких родственников более разумно считать его свидетельством разделения уровней детализации, где результат определяется отчасти независимо от способа его достижения.

Инженерные системы могут иметь подобную же структуру. Мы стараемся отделять спецификацию от реализации: часто есть множество способов удовлетворить спецификацию, и проекты могут выбирать различные реализации. Лучший способ отсортировать набор данных зависит от размера этого набора и от вычислительной стоимости сравнения элементов. Наилучший метод представления многочлена зависит от того, плотный он или разреженный. Однако, даже если подобный выбор производится динамически (что само по себе необычно), он остается детерминистским: мы почти не знаем систем, одновременно применяющих несколько способов решения задачи и использующих тот ответ, который получен первым (зачем нам вообще все эти процессорные ядра?). Редки даже системы, пробующие несколько методов последовательно: если один потерпит неудачу, можно попробовать другой. Мы рассмотрим, как перебор с возвратами используется для реализации механизма генерации и проверок при сопоставлении с образцом, в главе 4. Мы научимся встраивать автоматический перебор с возвратами в язык в главе 5. Наконец, мы научимся строить механизм перебора, управляемого зависимостями, который извлекает как можно больше информации из неудач, в главе 7.

1.5 Цена гибкости

> Программисты на Lisp знают
> значение чего угодно, но
> ничему не знают цену.
>
> Алан Перлис, перефразируя
> Оскара Уайльда

Мы заметили, что в системах, использующих полиморфизм, многослойность, дублирование и исследовательское поведение, повышается общность и способность к развитию. Каждое из этих свойств само по себе достаточно дорого. Механизм, способный обрабатывать множество различных видов входных данных, для получения того же результата должен проделать больше работы, чем механизм, приспособленный к какому-то одному виду. Механизм с избыточностью содержит больше частей, чем эквивалентный ему механизм без избыточности. Механизм с дублированием кажется еще более расточительным. А механизм, демонстрирующий исследовательское поведение методом порождения и фильтрации результатов, легко может уто-

[18] Роговица курицы и роговица мыши почти идентичны, однако их морфогенез совершенно не совпадает; даже порядок формообразующих событий различается. Бард [7], раздел 3.6.1, сообщает, что наличие у разных видов различных методов формирования сходных структур — распространенное явление. Он приводит ряд примеров. Вот один особо впечатляющий: у лягушек *Gastrotheca riobambae* (см. дель Пино и Элинсон [28]) обычная лягушачья морфология развивается из эмбрионального диска, в то время как у остальных лягушек она происходит из приблизительно сферического эмбриона.

нуть в экспоненциальном поиске. Однако в системах, способных развиваться, все это — ключевые способы решения задач. Возможно, при создании действительно устойчивых систем мы должны быть готовы платить за достаточно сложно устроенную и дорогую инфраструктуру.

Часть проблемы тут в том, что мы думаем о стоимости в неправильных терминах. Цена времени и пространства имеет значение, но наша интуиция по поводу того, откуда происходят эти расходы, несовершенна. Каждому инженеру известно, что оценка реальной производительности системы требует множества тщательных измерений, которые часто показывают, что основные расходы случаются в неожиданных местах. При росте сложности задача становится только труднее. Однако мы продолжаем настаивать на преждевременной оптимизации на всех уровнях нашей программы, не зная их реального значения.

Допустим, нам удалось отделить те части нашей программы, которые должны быть быстрыми, от тех, что должны быть умными. При такой политике вся цена общности и способности к развитию ограничивается теми модулями, которые должны быть умными. В компьютерных системах такой взгляд необычен, однако в обычной жизни он встречается на каждом шагу. Когда мы пытаемся научиться чему-то новому, скажем игре на музыкальном инструменте, на ранних стадиях требуются сознательные действия, чтобы связать желаемый эффект с физическими движениями, вызывающими его. Но, по мере того, как наше умение возрастает, большая часть работы уже не требует сознательного внимания. Это обязательно должно произойти, чтобы научиться играть быстро, поскольку сознательные действия слишком медленные.

Похожее рассуждение проходит при разговоре о программном и аппаратном обеспечении. Аппаратура строится из соображений эффективности, и ценой этого является фиксированный интерфейс. На основе этого интерфейса можно написать программу — в сущности, создавая виртуальную машину. У этого дополнительного уровня абстракции есть известная цена, однако такой компромисс оправдан приобретаемой общностью. (Иначе мы бы до сих пор писали на ассемблере!) Мысль здесь состоит в том, что уровневая структура позволяет одновременно достичь и эффективности, и гибкости. Мы считаем, что требование реализовывать всю систему максимально эффективным способом контрпродуктивно, поскольку вредит гибкости, которая нужна для приспособления системы под будущие нужды.

Основная стоимость системы состоит во времени, затраченном программистами на проектирование, понимание, поддержку, модификацию и отладку системы. Поэтому ценность повышенной приспособляемости может быть даже еще больше. Легко изменяемая и приспособляемая система уничтожает один из главных компонентов стоимости: обучение новых программистов работе существующей системы во всех неприглядных деталях, чтобы они знали, куда полезть и где поправить код. В действительности цена нашей хрупкой инфраструктуры, возможно, намного превышает стоимость гибкого проектирования как из-за стоимости катастроф, так и из-за потерянных возможностей от чрезмерного времени, требуемого на перепроектирование и повторную реализацию. А если существенная доля времени, затраченного на переписывание системы под новые требования, заменяется приспособлением старой системы к новой ситуации, выигрыш еще больше.

Проблемы с корректностью

> Для оптимиста стакан наполовину полный. Для пессимиста стакан наполовину пустой. Для инженера стакан вдвое больше, чем надо.
>
> Автор неизвестен

Однако если мы строим системы так, что они применимы в большем числе ситуаций, чем те, что мы рассматриваем во время проектирования, возникает еще большая компонента стоимости. Поскольку мы оказываемся готовы использовать свои системы в контекстах, для которых они не были предназначены, нам неоткуда знать, что они при этом будут работать правильно!

Нас учат в информатике, что «корректность» программного обеспечения превыше всего и что она достигается путем построения формальных спецификаций подсистем и систем из этих подсистем и при помощи доказательства, что спецификации сочетания удовлетворяются на основе спецификаций отдельных подсистем и способа их сочетания[19]. Мы утверждаем, что такая дисциплина делает системы более хрупкими. Более того, для создания действительно устойчивых систем с этой жесткой дисциплиной надо распрощаться.

Проблема с требованием доказательств состоит в том, что, как правило, общие свойства гибких механизмов доказать сложнее, чем конкретные свойства конкретных механизмов в строго определенных условиях. Это побуждает нас как можно сильнее ограничивать спецификации и модули для того, чтобы упростить доказательства. Однако сочетание узкоспециализированных модулей оказывается хрупким — места для изменений не остается[20]!

Мы не собираемся агитировать против доказательств. Когда они есть, это замечательно. Без них нельзя обойтись в критических компонентах систем вроде сборщиков мусора (или рибосом)[21]. Однако даже для критично важных систем безопасности (вроде автопилотов) ограничение применимости ситуациями, когда соответствие поведения спецификации доказуемо, может привести к излишним поломкам в реальной жизни. Нам хотелось бы, чтобы автопилот пытался безопасно вести поврежденный самолет, даже если способ его повреждения не был предусмотрен проектировщиком!

Мы протестуем против дисциплины, когда доказательства *обязательны*: требование, чтобы применимость всего на свете была доказана для конкретной ситуации прежде, чем его можно было в этой ситуации применять, чрезмерно ограничива-

[19] Сложную систему трудно, если не невозможно, специфицировать. Как мы заметили на стр. 22, легко сформулировать требование, чтобы шахматная программа соблюдала правила шахмат, но как мы потребуем, чтобы она играла хорошо? А в отличие от шахмат, где хотя бы правила фиксированы, спецификации большинства систем постоянно изменяются по мере того, как меняются условия их использования. Как составить точное описание бухгалтерской системы в условиях постоянно перерабатываемых налоговых законов?

[20] В сущности, закон Постела (стр. 19) прямо противоречит практике построения систем из точно и узкоспецифицированных частей. Закон Постела рекомендует делать каждую компоненту более широко применимой, чем строго необходимо для каждого конкретного случая.

[21] Малозаметная ошибка в низкоуровневой подсистеме управления памятью, например в сборщике мусора, очень плохо поддается отладке — особенно если в системе есть параллельные процессы! Однако если ограничить сложность и размер таких подсистем, задача их спецификации и даже доказательства «правильности» становится выполнимой.

ет использование методов, повышающих устойчивость проектов. Это в особенности верно для методов, которые позволяют, с осторожностью, использовать какой-то прием за пределами его доказанной области применимости, и методов, которые предусматривают будущие расширения, не указывая заранее этому расширению четких границ.

К сожалению, многие из рекомендуемых нами методов значительно усложняют задачу доказательства, если не делают его практически невозможным. С другой стороны, иногда лучший способ решить задачу — обобщить ее до такой степени, что возникает простое доказательство.

Глава 2

Специализированные языки

Одна из мощных стратегий встраивания гибкости в программный проект состоит в построении *специализированного языка*, отражающего концептуальную структуру предметной области, для которой разрабатывается программа. Специализированный язык — это абстракция, в которой имена и глаголы языка прямо соотносятся с предметной областью. Такой язык позволяет писать прикладную программу непосредственно в терминах предметной области. По самой своей природе специализированный язык реализует достаточно полную модель своей области, более мощную, чем требуется для конкретной задачи[1]. Несмотря на то что это может показаться излишней работой, не относящейся прямо к решаемой задаче, общие усилия часто оказываются меньшими, чем при написании монолитной программы, а получившуюся программу потом легче изменять, отлаживать и расширять.

Итак, уровень специализированного языка строится таким образом, чтобы поддерживать не только разработку конкретной программы. Он обеспечивает общий каркас, позволяющий строить множество разных программ, связанных общей предметной областью. Он упрощает расширение существующих приложений в рамках этой предметной области. Кроме того, он создает среду, позволяющую программам взаимодействовать.

В этой главе мы сначала вводим понятие систем комбинаторов — мощную стратегию организации уровней специализированных языков. Эффективность этой стратегии мы продемонстрируем, переформулировав невнятную кашу регулярных выражений для поиска строк в виде изящного специализированного языка на основе комбинаторов, встроенного в Scheme. Однако иногда у нас имеются компоненты, которые не удается легко включить в стройную систему; в таких случаях требуется система адаптеров. Как иллюстрацию мы используем специализированный язык для построения оберток процедур, выполняющих преобразования единиц измерения, так чтобы процедуры, написанные в рамках одной системы измерения, можно было использовать с другой. Наконец, мы рассматриваем широкую предметную область настольных игр и показываем, как конкретные детали этой области можно абстрагировать, построив интерпретатор правил игры.

[1] Общность модели предметной области является примером «закона Постела». См. стр. 19.

2.1 Комбинаторы

Существенная часть приспособляемости биологических систем получается, оттого что в них используются чрезвычайно гибко устроенные детали (клетки), которые конфигурируются динамически и, следовательно, могут подстроиться под изменение окружающей их среды. Как правило, вычислительные системы этой стратегией не пользуются и вместо этого опираются на иерархии специализированных деталей и их сочетания. В последние годы появление крупных библиотек точно специфицированных высокоуровневых деталей увеличило уровень абстракции, используемый в этой деятельности. Однако средства комбинирования сами по себе редко подвергаются абстракции или разделяются между программами разве что как «шаблоны»[2].

В некоторых случаях эту практику можно улучшить при помощи простых приемов, упрощающих использование общих средств комбинирования. Если создаваемые нами системы будут строиться на основе семейства компонент, которые можно сочетать и смешивать, получая таким образом новые элементы того же семейства, то измененные требования к системе иногда можно будет выполнить путем простого переупорядочения этих компонент.

Система комбинаторов представляет собой множество элементарных деталей и набор средств комбинирования этих деталей, таких, чтобы составные компоненты обладали той же спецификацией интерфейса, что и элементарные. Это позволяет строить системы без случайно возникающей зависимости между компонентами. Классическим примером комбинатороподобной системы может служить TTL [123] — исторически важная библиотека стандартных элементов и их комбинаций, предназначенная для построения сложных цифровых электронных систем.

Системы комбинаторов представляют собой стратегию проектирования для специализированных языков. Элементы системы служат словами языка, а с помощью комбинаторов можно составлять из них словосочетания и предложения. Преимущество комбинаторных систем состоит в том, что их легко строить и о них легко рассуждать, однако у них есть и недостатки, о которых мы поговорим в разделе 3.1.5. Когда они отражают структуру предметной области, комбинаторы могут быть отличным выбором.

Однако как можно построить систему на основе семейства сочетающихся между собой компонент? Требуется выделить множество элементарных деталей и набор *комбинаторов*, сочетающих свои аргументы и порождающих новые детали с тем же интерфейсом, что и у элементарных. Подобные наборы комбинаторов часто встречаются, иногда в явном, но чаще в неявном виде, в математических системах записи.

2.1.1 Комбинаторы функций

Использование функциональной записи в математике представляет собой комбинаторную дисциплину. У функции есть область определения (или домен), откуда берутся ее аргументы, и область ее возможных значений (или кодомен). Существуют комбинаторы, которые порождают новые функции на основе старых. Например, композиция $f \circ g$ функций f и g есть новая функция, которая принимает аргументы из области определения g и порождает результаты из области значений f.

[2]Существуют важные исключения: функциональные расширения, введенные в Java 8, выражают полезные способы комбинирования непосредственно. В функциональных языках программирования, таких как Lisp или Haskell, имеются библиотеки механизмов комбинирования.

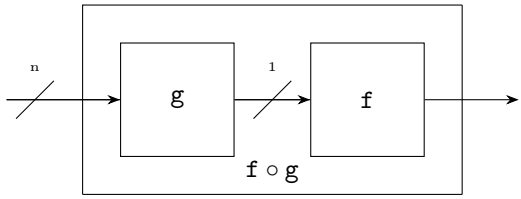


Рис. 2.1. Композиция $f \circ g$ функций f и g представляет собой функцию, определяемую этой диаграммой. Входные данные $f \circ g$ направляются на вход g. Выход g перенаправляется в f, которая порождает общий результат $f \circ g$

Если у двух функций одни и те же область определения и область значений и если на их общей области значений определены арифметические операции, то можно определить сумму (или произведение) функций как функцию, которая, получив аргумент из общей области определения, выдает сумму (или произведение) значений двух функций, примененных к этому аргументу. В языках, где процедуры являются полноправными объектами, есть механизм, поддерживающий эти средства комбинирования, но главное здесь — качество элементарных деталей.

У организации системы в виде семейства комбинаторов есть несколько преимуществ. Детали системы можно переставлять и смешивать произвольным образом. Любое их сочетание будет правильно построенной программой, и поведение этой программы прозрачным образом зависит только от поведения составных частей и от способа их соединения. Контекст, в котором находится компонента, не влияет на ее поведение, и всегда можно взять составную компоненту и поставить ее в новый контекст, не заботясь, как она будет себя вести именно в этом конкретном контексте. Такие программы легко писать, легко читать и легко проверять. Система, построенная на комбинаторах, без труда поддается расширению, поскольку введение новых элементарных деталей или новых комбинаторов не влияет на поведение существующих программ.

На функциональные комбинаторы можно смотреть как на диаграммы, показывающие, как функция строится путем соединения ее частей. Например, функциональная композиция представляет собой блок, составленный из двух подблоков, так что выход одного из них попадает на вход другому, как показано на рис. 2.1. Программа, реализующая эту идею, устроена очевидным образом:

```
(define (compose f g)
  (lambda args
    (f (apply g args))))
```

(Задача становится интереснее, если мы хотим проверять, что у функций правильная арность: что функция, представляемая процедурой `f`, принимает только один аргумент, соответствующий результату `g`. Еще забавнее дела обстоят в случае, если `g` может возвращать несколько значений, а `f` должна принимать их все. Можно также проверять, что композиция функций получает столько аргументов, сколько требуется для `g`. Но все эти детали мы рассмотрим позже.)

Композицию можно продемонстрировать на простом примере:

```
(compose (lambda (x) (list 'foo x))
         (lambda (x) (list 'bar x)))
```

```
 'z)
(foo (bar z))
```

Иногда имеет смысл дать имя процедуре, возвращаемой комбинатором. Например, `compose` можно записать как

```
(define (compose f g)
  (define (the-composition . args)
    (f (apply g args)))
  the-composition)
```

Имя `the-composition` за пределами тела `compose` не определено, так что никакого очевидного преимущества у такого способа записи нет. Мы часто используем в программах анонимные процедуры, определяемые через выражения `lambda`, как в первой версии `compose`, приведенной выше. Выбор варианта записи в основном зависит от стиля[3].

Даже имея один только комбинатор `compose`, уже можно писать довольно изящный код. Рассмотрим пример вычисления n-й степени функции $f^n(x) = f(f^{n-1}(x))$. Она может быть красиво записана в виде программы:

```
(define ((iterate n) f)
  (if (= n 0)
    identity
    (compose f ((iterate (- n 1)) f))))

(define (identity x) x)
```

Результатом вычисления `((iterate n) f)` будет новая функция того же типа, что `f`. Ее можно использовать всюду, где можно использовать `f`. Таким образом, выражение `(iterate n)` само по себе является комбинатором функций. Мы можем с его помощью определить результат многократного возведения в квадрат:

```
(((iterate 3) square) 5)
390625
```

Обратите внимание на аналогию: композиция функций подобна умножению, так что возведение функции в степень подобно возведению в степень числа.

Существует множество простых комбинаторов, широко применимых в программировании. Мы здесь приведем лишь некоторые из них, чтобы дать вам почувствовать пространство возможностей.

Можно применить две функции параллельно, а затем соединить результаты с помощью указанного комбинатора (см. рис. 2.2). Такая параллельная комбинация реализуется процедурой:

```
(define (parallel-combine h f g)
  (define (the-combination . args)
    (h (apply f args) (apply g args)))
  the-combination)
```

[3] В данном случае все просто, но в сложных программах с большим количеством внутренних процедур описательные имена могут упростить чтение и понимание кода. В реализации MIT/GNU Scheme есть еще небольшое дополнительное преимущество у варианта с именованием возвращаемой процедуры, поскольку отладчик может показать это имя.

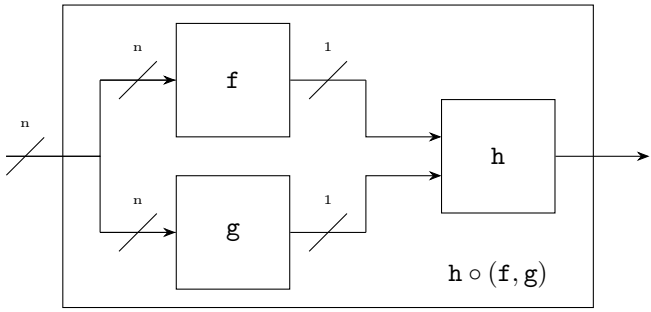


Рис. 2.2. В комбинаторе `parallel-combine` функции f и g принимают одинаковое количество аргументов. Вход «параллельной комбинации» направляется каждой из них. Их выходы сочетаются с помощью двухаргументной функции h

```
((parallel-combine list
                   (lambda (x y z) (list 'foo x y z))
                   (lambda (u v w) (list 'bar u v w)))
 'a 'b 'c)
((foo a b c) (bar a b c))
```

`Parallel-combine` может быть полезен при организации сложного процесса. Пусть, например, у нас есть некоторый источник картинок, изображающих части растений. У нас может быть одна процедура, которая по картинке оценивает цвет растения, и другая, способная выдать описание конкретного органа (лист, корень, стебель...). Третья процедура по двум этим описаниям сможет определить, какое именно растение изображено на картинке. Все три можно понятным образом объединить с помощью `parallel-combine`.

Арность

Существуют целые семейства комбинаторов, о которых мы обычно не думаем, но которые можно с успехом использовать в программировании. Многие из них часто встречаются в математическом контексте. Например, тензоры представляют собой расширение линейной алгебры на линейные операторы с многими аргументами. Однако можно сделать эту идею более общей: «тензорная комбинация» двух процедур представляет собой новую процедуру, которая принимает на вход структуру данных, сочетающую аргументы исходных двух процедур. Она распределяет аргументы по процедурам и порождает структуру данных, объединяющую результаты их применения. Такая потребность развернуть структуру данных, обработать ее компоненты по отдельности и свернуть обратно результаты в программировании встречается на каждом шагу. Диаграмма на рис. 2.3 показывает комбинатор `spread-combine`. Он представляет собой обобщение тензорного произведения из полилинейной алгебры. В случае математического тензорного произведения f и g должны быть линейными преобразованиями своих аргументов, а h является следом по некоторым общим индексам; однако тензоры являются лишь особым случаем, подсказавшим нам этот комбинатор.

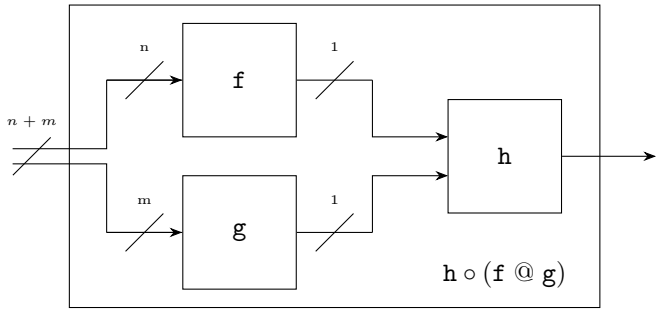


Рис. 2.3. В функции `spread-combine` $n+m$ аргументов распределяются по функциям f и g. Первые n аргументов направляются на вход f, а оставшиеся — на вход g. Результаты потом объединяются с помощью двухаргументной функции h

Программа, реализующая `spread-combine`, несколько сложнее, чем `parallel-combine`, поскольку она должна правильно распределить аргументы для `f` и `g`. Вот первый набросок ее кода:

```
(define (spread-combine h f g)
  (let ((n (get-arity f)))
    (define (the-combination . args)
      (h (apply f (list-head args n))
         (apply g (list-tail args n))))
    the-combination))
```

Этой процедуре требуется способ определить, сколько аргументов принимает некоторая процедура (ее *арность*), поскольку надо знать, сколько аргументов нужно отдать `f`, чтобы остальные направить в `g`.

Эта версия `spread-combine` не очень хороша. Главная проблема в том, что `the-combination` принимает произвольное число аргументов, поэтому у нее нет определенной арности, а следовательно, ее нельзя передать другому комбинатору, если ему нужна арность. Например, результат `spread-combine` не может служить вторым аргументом другого `spread-combine`. Таким образом, нужно как-то снабдить `the-combination` информацией об арности. Вот второй черновой вариант:

```
(define (spread-combine h f g)
  (let ((n (get-arity f)) (m (get-arity g)))
    (let ((t (+ m n)))
      (define (the-combination . args)
        (h (apply f (list-head args n))
           (apply (g (list-tail args n)))))
      (restrict-arity the-combination t))))
```

Здесь для возвращаемой процедуры `the-combination` указывается арность, так что она может передаваться в другие комбинаторы, которым эта арность требуется. Процедура `restrict-arity` принимает на вход процедуру, помечает ее арность так, что потом ее можно считать при помощи `get-arity`, и возвращает помеченную процедуру-аргумент.

Это все замечательно, но самые лучшие программы пишут параноики! Желательно отлавливать ошибки как можно раньше, прежде чем их становится трудно

найти или чем они приведут к серьезным неприятностям. Так что давайте снабдим наш код *контрольным утверждением* (assertion) как отличительным признаком *Параноидального стиля программирования*, чтобы оно проверяло нашу комбинацию на правильное число аргументов.

```
(define (spread-combine h f g)
  (let ((n (get-arity f)) (m (get-arity g)))
    (let (( t (+ m n)))
      (define (the-combination . args)
        (assert (= (length args) t))
        (h (apply f (list-head args n))
           (apply (g (list-tail args n)))))
      (restrict-arity the-combination t))))

((spread-combine list
                 (lambda (x y) (list 'foo x y))
                 (lambda (u v w) (list 'bar u v w)))
 'a 'b 'c 'd 'e)
((foo a b) (bar c d e))
```

Особая форма `assert` — всего лишь удобный способ сигнализировать об ошибке, если значение ее аргумента не истинно.

Один из способов написать `restrict-arity` и `get-arity` такой:

```
(define (restrict-arity proc nargs)
  (hash-table-set! arity-table proc nargs)
  proc)

(define (get-arity proc)
  (or (hash-table-ref/default arity-table proc #f)
      (let ((a (procedure-arity proc))) ;арности нет в таблице
        (assert (eqv? (procedure-arity-min a)
                      (procedure-arity-max a)))
        (procedure-arity-min a))))

(define arity-table (make-key-weak-eqv-hash-table))
```

Мы используем хеш-таблицу, чтобы прикрепить к процедуре «ярлычок»[4]. Это несложный трюк, позволяющий добавлять информацию к существующему объекту. Однако он требует, чтобы аннотируемый объект был уникален (т. е. чтобы по нему можно было вычислить уникальный ключ), так что используйте с осторожностью.

Если процедура `get-arity` не может найти явно заданное значение в таблице `arity-table`, она его вычисляет с помощью элементарных процедур нижележащей системы MIT/GNU Scheme. Здесь требуется некоторая возня, потому что в элементарных процедурах поддерживается более общее понятие арности: процедура имеет некоторое минимальное количество аргументов, а опционально может иметь еще и максимальное количество аргументов. Наш код, связанный с арностью, предполагает, что число аргументов фиксировано, так что `get-arity` ни с каким другим

[4] Документацию по процедурам работы с хеш-таблицами в MIT/GNU Scheme можно найти в [51].

типом процедур работать не может. К сожалению, это исключает процедуры вроде `+`, принимающей произвольное количество аргументов. Если изменить код, чтобы он имел более общее понятие об арности, он станет сложнее, а наша цель здесь состоит скорее в ясности изложения, а не в максимальной общности решения (но см. упражнение 2.2).

Упражнение 2.1. Починка арности
Написанные нами процедуры `compose` и `parallel-combine` не подчиняются правилу, требующему объявлять арность комбинации. Таким образом, они не являются полноправными членами нашего семейства комбинаторов. Исправьте реализации `compose` и `parallel-combine` так, чтобы они:

- проверяли свои компоненты и убеждались, что их арности совместимы;
- порожденная ими комбинация при вызове проверяла, что ей скормлено нужное число аргументов;
- комбинация правильно объявляла свою арность процедуре `get-arity`.

Упражнение 2.2. Расширение арности
Наша реализация комбинаторов несовершенна, поскольку продемонстрированный нами механизм работы с арностью не поддерживает более общее представление, используемое в MIT/GNU Scheme. Например, процедура сложения, значение глобальной переменной `+`, может принимать произвольное число аргументов:

```
(procedure-arity-min (procedure-arity +)) = 0
(procedure-arity-max (procedure-arity +)) = #f
```

а процедура вычисления арктангенса требует либо 1, либо 2 аргумента:

```
(procedure-arity-min (procedure-arity atan)) = 1
(procedure-arity-max (procedure-arity atan)) = 2
```

Полезно было бы расширить нашу работу с арностью, чтобы комбинаторы справлялись с этими более сложными ситуациями.

a. Сделайте набросок плана, как расширить комбинаторы для работы с более общим понятием арности. Заметим, что не всегда с ними можно будет использовать арифметику. Какие вопросы придется решить при переформулировке `spread-combine`? Например, какие ограничения придется наложить на процедуры-аргументы `f`, `g` и `h`?

b. Действуйте по этому плану и заставьте его работать!

В любом языке имеются элементарные конструкции (примитивы), средства комбинирования и средства абстракции. *Комбинаторный язык* определяет элементарные конструкции и средства комбинирования, а средства абстракции наследуются из нижележащего языка программирования. В нашем примере примитивами служат функции, средства комбинирования — это комбинаторы `compose`, `parallel-combine`, `spread-combine` и другие, которые мы еще можем написать.

Множественные значения

Заметим, что как `parallel-combine`, так и `spread-combine` применяют функцию `h` к результатам функций `f` и `g`. Однако при написании этих комбинаторов мы не использовали `compose`. Для того чтобы абстрагировать эту схему действий, нам нужно уметь возвращать множественные значения из комбинации `f` и `g`, а затем

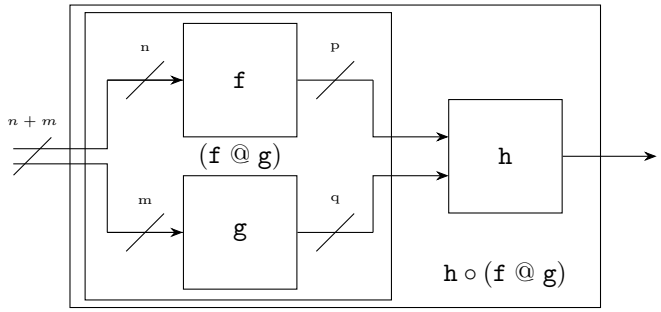


Рис. 2.4. Комбинатор `spread-combine` на самом деле является композицией двух частей. Первая из них, `spread-apply`, представляет собой комбинацию функций f и g, между которыми должным образом распределяются аргументы. Вторая часть, объединитель h, просто присоединяется к первой через композицию. Этот анализ реализуется через механизм множественных возвращаемых значений языка Scheme

использовать эти значения как аргументы `h`. Можно это делать через возврат составной структуры данных, но лучше здесь будет воспользоваться механизмом множественных возвращаемых значений языка Scheme. С его помощью можно определить `spread-combine` как композицию двух частей, `h` и комбинации `f` и `g`[5]:

```
(define (spread-apply f g)
  (let ((n (get-arity f)) (m (get-arity g)))
    (let ((t (+ n m)))
      (define (the-combination . args)
        (assert (= (length args) t))
        (values (apply f (list-head args n))
                (apply g (list-tail args n))))
      (restrict-arity the-combination t))))
```

Процедура `values` языка Scheme возвращает результаты применения `f` и `g` вместе[6].

Затем мы обобщаем `compose` так, чтобы стало можно напрямую определить абстракцию, показанную на рис. 2.4, следующим образом:

```
(define (spread-combine h f g)
  (compose h (spread-apply f g)))
```

Это определение действует так же, как наша исходная версия:

```
((spread-combine list
                 (lambda (x y) (list 'foo x y))
                 (lambda (u v w) (list 'bar u v w)))
 'a 'b 'c 'd 'e)
((foo a b) (bar c d e))
```

Чтобы оно заработало, нам нужно обобщить `compose` и позволить передачу нескольких значений между процедурами, к которым применяется композиция:

[5]Мы благодарны Гаю Л. Стилу-мл., который предложил нам продемонстрировать эту декомпозицию.

[6]Документацию по `values`, `call-with-values` и `let-values` можно найти в [51] и [109].

```
(define (compose f g)
  (define (the-composition . args)
    (call-with-values (lambda () (apply g args))
      f))
  (restrict-arity the-composition (get-arity g)))
```

Здесь второй аргумент `compose` возвращает два значения:

```
((compose (lambda (a b)
            (list 'foo a b))
          (lambda (x)
            (values (list 'bar x)
                    (list 'baz x))))
 'z)
(foo (bar z) (baz z))
```

Теперь мы можем пойти еще дальше. Разрешим всем функциям, с которыми мы работаем, возвращать множественные значения. Если и `f`, и `g` возвращают несколько значений, мы можем собрать их в единый набор значений, который и вернет `the-combination`:

```
(define (spread-apply f g)
  (let ((n (get-arity f)) (m (get-arity g)))
    (let ((t (+ n m)))
      (define (the-combination . args)
        (assert (= (length args) t))
        (let-values ((fv (apply f (list-head args n)))
                     (gv (apply g (list-tail args n))))
          (apply values (append fv gv))))
      (restrict-arity the-combination t))))

((spread-combine list
                 (lambda (x y) (values x y))
                 (lambda (u v w) (values w v u)))
 'a 'b 'c 'd 'e)
(a b e d c)
```

Единственное ограничение состоит в том, чтобы общее число возвращаемых значений подпадало под арность `h`.

Упражнение 2.3. Простой вопрос
Переформулируйте `parallel-combine` в виде композиции двух частей так, чтобы каждая из них могла возвращать несколько значений.

Небольшая библиотека

Многие схемы использования процедур можно сформулировать в виде комбинаторов, и часто с помощью этого приема удается построить весьма изящные программы. Имеет смысл выделить и абстрагировать такие часто встречающиеся схемы. Вот некоторые из них.

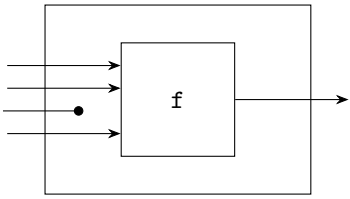


Рис. 2.5. Комбинатор `(discard-argument 2)` принимает трехаргументную функцию f и возвращает новую функцию от четырех аргументов, которая игнорирует свой третий аргумент ($\mathtt{i} = 2$), а остальные аргументы передает в f

Часто у нас есть интерфейс, который для нашей конкретной ситуации обладает избыточной общностью. В таком случае нам бы хотелось его сохранить, но вызвать некоторую более специализированную процедуру, которой нужны не все параметры, передаваемые в общем случае; сделать версию общей процедуры, которая часть аргументов игнорирует.

Процедура `discard-argument` принимает номер `i` аргумента, который надо отбросить, и возвращает комбинатор. Комбинация принимает функцию n аргументов `f` и возвращает функцию от $n+1$ аргумента `the-combination`, которая применяет `f` к n аргументам, остающимся при выкидывании i-го из исходных $n+1$ аргументов. Идея показана на рис. 2.5. Код комбинатора таков:

```
(define (discard-argument i)
  (assert (exact-nonnegative-integer? i))
  (lambda (f)
    (let ((m (+ (get-arity f) 1)))
      (define (the-combination . args)
        (assert (= (length args) m))
        (apply f (list-remove args i)))
      (assert (< i m))
      (restrict-arity the-combination m))))

(define (list-remove lst index)
  (let lp ((lst lst) (index index))
    (if (= index 0)
        (cdr lst)
        (cons (car lst) (lp (cdr lst) (- index 1))))))

(((discard-argument 2)
  (lambda (x y z) (list 'foo x y z)))
 'a 'b 'c 'd)
(foo a b d)
```

Можно этот комбинатор обобщить, чтобы он отбрасывал несколько аргументов.

Часто также встречается ситуация, обратная `discard-argument`. На рис. 2.6 мы видим диаграмму специализации процедуры, где все аргументы, кроме одного, задаются заранее, а этот один указывается при вызове. Это по традиции называется

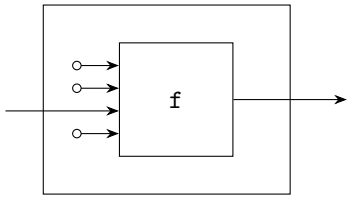


Рис. 2.6. Комбинатор `((curry-argument 2) 'a 'b 'c)` задает три аргумента четырехместной функции f, чтобы оставшийся третий аргумент ($\mathtt{i} = 2$) был указан при вызове получившейся одноместной функции

каррированием в честь логика Хаскелла Карри, раннего исследователя комбинаторной логики[7].

Код функции `curry-argument` не содержит неожиданностей:

```
(define ((curry-argument i) . args)
  (lambda (f)
    (assert (= (length args) (- (get-arity f) 1)))
    (lambda (x)
      (apply f (list-insert args i x)))))

(define (list-insert lst index value)
  (let lp ((lst lst) (index index))
    (if (= index 0)
        (cons value lst)
        (cons (car lst) (lp (cdr lst) (- index 1))))))

((((curry-argument 2) 'a 'b 'c)
  (lambda (x y z w) (list 'foo x y z w)))
 'd)
(foo a b d c)
```

Заметим, что здесь не требуется использовать `restrict-arity`, поскольку у возвращаемой процедуры ровно один аргумент[8]. В упражнении 2.5 мы обобщаем комбинатор каррирования, чтобы можно было оставить незаданными несколько аргументов.

Иногда мы хотим позвать библиотечную процедуру, у которой порядок аргументов не совпадает с соглашениями, принятыми в нашем приложении. Вместо того чтобы строить для этой процедуры отдельный интерфейс, можно использовать общую процедуру перестановки, которая переупорядочит аргументы, как на рис. 2.7. Эта программа также несложна, однако заметим, что процедура `the-combination`, возвращаемая из комбинатора и непосредственно применяемая к `args`, не должна разбирать описание перестановки — это уже один раз сделано внутри выражения `let`, которое окружает `the-combination` и на которое она ссылается. В общем слу-

[7] Комбинаторная логика была изобретена Моисеем Шейнфинкелем [108] и развита Хаскеллом Карри [26] в начале XX в. Их целью были не вычисления, а упрощение оснований математики через возможность избавиться от переменных, связываемых кванторами

[8] Используемая в `get-arity` процедура MIT/GNU Scheme `procedure-arity` выдаст для возвращаемой процедуры числовую арность, так что прикреплять к ней отдельный ярлык незачем.

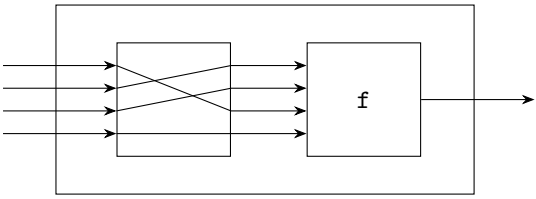


Рис. 2.7. Комбинатор `(permute-arguments 1 2 0 3)` принимает функцию f от четырех аргументов и порождает новую четырехместную функцию, которая переупорядочивает аргументы в соответствии с заданной перестановкой, а затем передает их внутрь f

чае такой способ написания кода позволяет провести глубокие оптимизации путем ранних вычислений даже при очень позднем связывании!

```
(define (permute-arguments . permspec)
  (let ((permute (make-permutation permspec)))
    (lambda (f)
      (define (the-combination . args)
        (apply f (permute args)))
      (let ((n (get-arity f)))
        (assert (= n (length permspec)))
        (restrict-arity the-combination n)))))

(((permute-arguments 1 2 0 3)
  (lambda (x y z w) (list 'foo x y z w)))
 'a 'b 'c 'd)
(foo b c a d)
```

Процедура `make-permutation` простая, но неэффективная:

```
(define (make-permutation permspec)
  (define (the-permuter lst)
    (map (lambda (p) (list-ref lst p))
         permspec))
  the-permuter)
```

2.1.2 Комбинаторы и планы строения

Мораль этой истории в том, что структура, составленная из комбинаторов, представляет собой план строения, подобно планам строения животных или инженерным шаблонам вроде супергетеродинного радиоприемника (рис. 1.1 на стр. 24). Рассмотрим комбинатор `compose`. Он предоставляет набор позиций для процедур `f` и `g`. Эти позиции связаны между собой стандартным образом, но больше от них ничего не требуется. На эти места можно подставить что угодно, лишь бы они принимали нужное количество аргументов и выдавали нужное количество значений. Таким образом, комбинаторы служат организационными принципами, вроде генов Hox: они указывают позиции и их отношение, не ограничивая, что именно происходит в каждой их этих позиций.

Упражнение 2.4. В виде композиций?
Вы могли заметить, что каждый из комбинаторов, задаваемых процедурами `discard-argument`, `curry-argument` и `permute-arguments`, можно рассматривать как композицию некоторого преобразования аргументов и процедуры. Переформулируйте их в виде композиций, используя механизм множественных возвращаемых значений.

Упражнение 2.5. Полезные комбинаторы
Пора еще немного расширить нашу небольшую библиотеку.

a. Можно обобщить `discard-argument` и `curry-argument`, чтобы они позволяли отбрасывать или заранее задавать более одного аргумента. Метод задания аргументов в `permute-arguments` кажется достаточно общим способом указания порядка аргументов вызова (считая с нуля). Постройте обобщенные версии этих процедур с таким интерфейсом. Назовите их `discard-arguments` и `curry-arguments`. Пусть ваш код будет совместим с кодом в тексте книги: вызов `(curry-arguments 2)` должен делать то же самое, что `(curry-argument 2)`.

b. Какие еще комбинаторы, по вашему мнению, были бы полезны? Составьте список с примерами возможного использования в реальной жизни. Реализуйте эти комбинаторы в своей библиотеке.

c. Постройте дальнейшее обобщение комбинатора `compose`, чтобы он мог принимать произвольное число функциональных аргументов. Выражение `(compose f g h)` эквивалентно `(compose f (compose g h))`. Заметим, что оно также эквивалентно `(compose (compose f g) h)`. Осторожно: чему равна композиция нуля аргументов?

2.2 Регулярные выражения

Для поиска информации в строках часто используются регулярные выражения. Несмотря на то что библиотеки для работы с ними основаны на ясном математическом формализме, конкретные решения, принятые при расширении этого формализма, чтобы превратить его в полезные программные системы, часто катастрофически ужасны: соглашения по закавычиванию не отличаются последовательностью; вопиющее правило, что скобки служат одновременно для группировки и для обратных ссылок, представляет собой печальное зрелище. Вдобавок попытки увеличить выразительную силу и исправить недостатки ранних реализаций привели к возникновению множества несовместимых производных языков.

При поверхностном взгляде регулярные выражения выглядят как комбинаторный язык, поскольку фрагменты выражений могут сочетаться и образовывать более сложные выражения. Однако значение фрагмента существенным образом зависит от выражения, куда он вставлен. Например, если у нас есть знак крышки ^ внутри квадратных скобок [...], то он не должен стоять в начале, поскольку, когда крышка не находится в первой позиции, она считается обыкновенным символом, а когда она стоит сразу после скобки, это означает отрицание скобочного выражения. Таким образом, выражение в квадратных скобках не может состоять только из одной крышки.

Итак, синтаксис регулярных выражений гадок, имеется несколько несовместимых диалектов языка, а соглашения по закавычиванию грандиозно нелепы. Регулярные выражения — это специализированный язык, но очень плохой. Отчасти польза от исследования регулярных выражений в том и состоит, чтобы познать на опыте, насколько ужасно могут обстоять дела.

Однако имеется множество полезных программ, начиная с `grep`, определяющих свое поведение с помощью регулярных выражений. Мы изобретем более качественный специализированный комбинаторный язык для задания регулярных выражений, а также создадим инструменты для преобразования этого языка в традиционный синтаксис. В качестве целевого языка нашего транслятора мы выбрали Базовые регулярные выражения POSIX (Basic Regular Expressions, BRE) [96], поскольку они являются подмножеством большинства других вариантов синтаксиса. Стандарт POSIX определяет также Расширенные регулярные выражения; их мы рассмотрим в упражнении.

При помощи нашей библиотеки мы сможем использовать возможности систем вроде `grep` изнутри окружения Scheme. У нас будут все преимущества комбинаторного языка с ясным модульным описанием, и при этом сохранится способность обращаться к существующим инструментам. Пользователям не на что будет жаловаться, если только они не ценят краткость выше, чем читаемость.

Как и в любом языке, у нас будут элементарные конструкции, средства комбинирования и средства абстракции. Наш язык позволяет строить образцы, которые затем утилиты вроде `grep` могут сопоставлять с символьными строками. Поскольку наш язык встроен в Scheme, он наследует мощные возможности языка Scheme: мы сможем сочетать образцы при помощи его конструкций и абстрагировать при помощи его процедур.

2.2.1 Комбинаторный язык регулярных выражений

Образцы составляются на основе следующих элементарных образцов:

`(r:dot)` сопоставляется с любым символом, кроме перевода строки;

`(r:bol)` сопоставляется с началом строки;

`(r:eol)` сопоставляется с концом строки;

`(r:quote` *`строка`*`)` сопоставляется со *строкой*;

`(r:char-from` *`строка`*`)` сопоставляется с любым одним символом из *строки*;

`(r:char-not-from` *`строка`*`)` сопоставляется с любым одним символом, отсутствующим в *строке*.

Образцы можно сочетать между собой и получать составные образцы.

`(r:seq` *`образец ...`*`)`
сопоставляется с каждым *образцом* последовательно, слева направо.

`(r:alt` *`образец ...`*`)`
пытается сопоставиться с каждым из *образцов*-аргументов слева направо, пока один из них не подойдет. Если не подходит ни один из них, сопоставление целиком терпит неудачу.

`(r:repeat` *`min max образец`*`)`
этот образец пытается сопоставиться с `образцом`-аргументом минимум *min* и максимум *max* раз. Если в качестве *max* указана `#f`, значит, никакого ограничения сверху нет. Если *min* равен *max*, сопоставление должно сработать точно указанное число раз.

Вот несколько примеров.

`(r:seq (r:quote "a") (r:dot) (r:quote "c"))`
сопоставляется с любой трехсимвольной строкой, которая начинается на `a` и кончается на `c`. Например, сопоставится с `abc`, `aac`, или `acc`.

`(r:alt (r:quote "foo") (r:quote "bar") (r:quote "baz"))`
сопоставляется либо с `foo`, либо с `bar`, либо с `baz`.

`(r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog")))`
сопоставляется с `catdogcat`, `catcatdogdog`, `dogdogcatdogdog`, но не с `catcatcatdogdogdog`.

Мы реализуем образцы в виде выражений языка Scheme. Таким образом, мы сможем использовать их вместе с любым другим кодом на Scheme, и получим всю выразительную силу этого языка программирования.

2.2.2 Реализация транслятора

Рассмотрим, как наш язык реализуется. Регулярные выражения будут представляться в виде строк в синтаксисе Базовых регулярных выражений POSIX.

```
(define (r:dot) ".")
(define (r:bol) "^")
(define (r:eol) "$")
```

Здесь у нас прямое соответствие синтаксису регулярных выражений.

Выражение `r:seq` позволяет использовать любой набор фрагментов регулярных выражений как самодостаточный замкнутый элемент:

```
(define (r:seq . exprs)
  (string-append "\\(" (apply string-append exprs) "\\)"))
```

При помощи скобок мы изолируем содержимое наших фрагментов выражений от окружающего их контекста. К сожалению, использование `\` в результате неизбежно. В базовом синтаксисе регулярных выражений символы скобок считаются самозакавычивающими. Здесь же они нужны нам в качестве группирующей операции, и для этого нужно перед каждой поставить обратный слеш. Дополнительным неудобством тут оказывается, что, когда мы вставляем это регулярное выражение в строку языка Scheme, каждый обратный слеш нужно защитить *еще одним* обратным слешем. Таким образом, наш пример `(r:seq (r:quote "a") (r:dot) (r:quote "c"))` переводится в `\(\(a\).\(c\)\)`, что в виде строки Scheme выглядит как `"\\(\\(a\\).\\(c\\)\\)"`. Увы нам.

Реализация `r:quote` заметно сложней. В регулярных выражениях большинство символов самозакавычивающие. Однако некоторые символы обозначают операции над регулярными выражениями, и их требуется закавычить явно. Мы оборачиваем результат в `r:seq`, чтобы обеспечить замкнутость результата.

```
(define (r:quote string)
  (r:seq
    (list->string
      (append-map (lambda (char)
```

```
                    (if (memv char chars-needing-quoting)
                        (list #\\ char)
                        (list char)))
                  (string->list string)))))

(define chars-needing-quoting
  '(#\. #\[ #\\ #\^ #\$ #\*))
```

Для реализации выражения-альтернативы мы вставляем между подвыражениями вертикальную черту и оборачиваем результат с помощью `r:seq`:

```
(define (r:alt . exprs)
  (if (pair? exprs)
    (apply r:seq
           (cons (car exprs)
                 (append-map (lambda (expr)
                               (list "\\|" expr))
                             (cdr exprs))))
  (r:seq)))
```

`(r:alt (r:quote "foo") (r:quote "bar") (r:quote "baz"))` переводится в `\(\(foo\)\|\(bar\)\|\(baz\)\)`. Кроме скобок, нам нужно отмечать обратным слешем еще и знак вертикальной черты, который в этом варианте синтаксиса считается самозакавычивающим. Заметим, что выражения-альтернативы, в отличие от остальных конструкций, которые мы здесь используем, не являются частью синтаксиса Базовых выражений: это расширение GNU `grep`, которое поддерживают также многие другие реализации. (Альтернативы *входят* в Расширенный синтаксис.)

Повторение нетрудно реализовать с помощью копий данного регулярного выражения:

```
(define (r:repeat min max expr)
  (apply r:seq
         (append (make-list min expr)
                 (cond ((not max) (list expr "*"))
                       ((= max min) '())
                       (else
                         (make-list (- max min)
                                    (r:alt expr ""))))))))
```

Здесь мы делаем `min` копий выражения `expr`, а затем еще `(- max min)` необязательных копий, где каждая необязательная копия — это альтернативно либо выражение-аргумент, либо пустое выражение. Если максимума нет[9], после выражения ставится звездочка, что обозначает произвольное число повторений. Таким образом, `(r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog")))` переводится в нечто настолько громадное, что оно может вызвать у читателя судороги.

Реализация `r:char-from` и `r:char-not-from` усложняется из-за изощренных правил закавычивания. С ними лучше работать в два приема, сначала обработать различия между двумя функциями, а затем то, что между ними есть общего:

[9]Мы выражаем отсутствие максимума путем вызова `r:repeat` со значением `#f` в качестве `max`.

```
(define (r:char-from string)
  (case (string-length string)
    ((0) (r:seq))
    ((1) (r:quote string))
    (else
      (bracket string
               (lambda (members)
                 (if (lset= eqv? '(#\- #\^) members)
                     '(#\- #\^)
                     (quote-bracketed-contents members)))))))

(define (r:char-not-from string)
  (bracket string
           (lambda (members)
             (cons #\^ (quote-bracketed-contents members)))))

(define (bracket string procedure)
  (list->string
    (append '(#\[)
            (procedure (string->list string))
            '(#\]))))
```

Особые случаи для `r:char-from` включают в себя пустое и одноэлементное множество символов — эта обработка упрощает разбор общего случая. Кроме того, есть особый случай для множества, состоящего из знака крышки и дефиса. В процедуре `r:char-not-from` таких особых случаев нет.

Еще один особый случай обрабатывает закавычивание трех символов, которые внутри скобок интерпретируются особым образом, помещая их в такие позиции, где они не считаются знаками операций. (Мы предупреждали, что будет криво!)

```
(define (quote-bracketed-contents members)
  (define (optional char)
    (if (memv char members) (list char) '()))
  (append (optional #\])
          (remove
            (lambda (c)
              (memv c chars-needing-quoting-in-brackets))
            members)
          (optional #\^)
          (optional #\-)))

(define chars-needing-quoting-in-brackets
  '(#\] #\^ #\-))
```

Чтобы протестировать этот код, можно распечатать соответствующий вызов команды `grep`, и с помощью копирования и вставки запустить его в командной оболочке. Поскольку у разных оболочек разные соглашения по особым символам, нам надо не только закавычить регулярное выражение, но еще и выбрать, с какой оболочкой мы работаем. Командная оболочка Баурна (Bourne shell) работает где угодно, а ее соглашения по закавычиванию относительно просты:

```
(define (write-bourne-shell-grep-command expr filename)
  (display (bourne-shell-grep-command-string expr filename)))

(define (bourne-shell-grep-command-string expr filename)
  (string-append "grep -e "
                 (bourne-shell-quote-string expr)
                 " "
                 filename))
```

Соглашения оболочки Баурна используют одиночные кавычки вокруг строки, которые защищают все внутри этой строки, кроме самого символа одиночной кавычки (закрывающей закавыченную строку). Так что, чтобы закавычить одиночную кавычку, мы завершаем строку, вставляем кавычку под защитой обратного слеша, а затем начинаем новую закавыченную строку. Оболочка рассматривает конкатенацию как единое целое. (Весело, правда?)

```
(define (bourne-shell-quote-string string)
  (list->string
    (append (list #\')
            (append-map (lambda (char)
                          (if (char=? char #\')
                              (list #\' #\\ char #\')
                              (list char)))
                        (string->list string))
            (list \'))))
```

Мораль этой басни

Наш транслятор получился сложным, потому что большинство видов регулярных выражений нельзя составлять, получая более крупные регулярные выражения, если не предпринимать специальные меры для изоляции отдельных фрагментов друг от друга. Наш транслятор проделывает эту работу, но как следствие порождаемые им регулярные выражения содержат много ненужной обвязки. Люди так регулярные выражения не пишут, поскольку они подобную обвязку используют только там, где она нужна. Однако часто они пропускают случаи, где обвязка требуется, что приводит к трудноуловимым ошибкам.

Мораль этой истории в том, что регулярные выражения могут служить замечательным примером, как систему *не надо* строить. Использование элементов, легко поддающихся композиции, и комбинаторов, позволяющих делать новые элементы путем сочетания старых, приводит к более простой и надежной реализации.

Упражнение 2.6. Добавление * и + к регулярным выражениям

В традиционных регулярных выражениях оператор звездочка (`*`) после подвыражения обозначает ноль или более копий этого подвыражения. Часто используется также расширение со знаком плюс (`+`). Плюс после подвыражения обозначает одну или более копий этого подвыражения.

Определите процедуры на Scheme `r:*` и `r:+`, которые принимают образец в качестве аргумента и повторяют его, как указано в имени процедуры. Можно выразить эти процедуры в терминах `r:repeat`.

Продемонстрируйте использование этих процедур на реальных данных в каком-нибудь достаточно сложном образце.

Упражнение 2.7. Ошибка, неудачная шутка, две поправки и озарение
Бен Битобор заметил ошибку в нашей реализации `(r:repeat min max expr)`.

Вызов `(r:alt expr "")` в конце процедуры `r:repeat` чересчур хитер. Этот фрагмент кода переводится в `\(expr\|\)`, где *expr* — значение `expr`. Это опирается на соображение, что альтернатива, состоящая из чего-то и ничего, равносильна выражению «один или ноль раз». (В оставшейся части объяснения мы будем пропускать обратные слеши; они нужны, но читать с ними неудобно.) То есть `(expr|)` означает одно или ноль вхождений *expr*. К сожалению, это зависит от недокументированного расширения GNU по отношению к формальному стандарту регулярных выражений POSIX.

А именно, раздел 9.4.3 стандарта POSIX[10] говорит, что вертикальная черта, стоящая непосредственно перед закрывающей скобкой (или непосредственно после открывающей) порождает неопределенное поведение. Иначе говоря, регулярное выражение не может быть пустым.

Реализация GNU `grep` просто «делает, как сказано», если ей дают `(x|)`. Другие реализации не обязаны быть настолько же терпеливыми.

Поэтому Бен просит трех членов своей команды (Хьюго, Лизу и Еву) предложить обходные пути, а в конце концов придумывает свою собственную заплатку, которую вы и реализуете.

- Хьюго Дум считает, что простым и изящным решением будет заменить фрагмент кода `(r:alt expr "")` на простой вызов `(r:repeat 0 1 expr)`.
- Лиза П. Хакер предлагает переписать ветку `else` процедуры `r:repeat` так, чтобы `(r:repeat 3 5 x)` переводился в `xxx|xxxx|xxxxx`, а не в мерзкое нестандартное выражение `xxx(x|)(x|)`, порождаемое нашим кодом. Она ссылается на раздел 9.4.7 документации POSIX[11].
- Ева Лу Атор указывает на операцию `?` из раздела 9.4.6.4[12] и говорит, что правильнее будет реализовать процедуру `r:?` и заменить `(r:alt expr "")` на `(r:? expr)`.
- Тем временем Бен внимательно читает стандарт регулярных выражений и получает озарение. Он предлагает переписать `r:repeat` с помощью интервальных выражений. См. раздел 9.3.6.5 документации POSIX[13]. Постарайтесь, чтобы вас не стошнило.

  Рассмотрим каждое из этих предложений.

  a. Все смеются над глупой шуткой Хьюго. Что здесь такого смешного? А именно, что плохо в этой идее?

     Достаточно ответить одним предложением.

  b. Чем Евин вариант лучше Лизиного, по отношению как к коду, так и к данным? Объясните кратко, в рамках одного абзаца.

  c. В чем преимущество идеи Бена по сравнению с остальными? А именно, посмотрите, на какой раздел стандарта POSIX ссылается он, а на какие остальные, сверьтесь с упражнением 2.10 и подумайте о последствиях. Рассмотрите также размер выходной строки в новом варианте и простоту кода программы.

     Здесь тоже достаточно пары фраз.

  d. Согласно предложению Бена, реализуйте `r:repeat`, чтобы она порождала интервальные выражения. Подсказка: должна пригодиться процедура Scheme `number->string`. Осторожно с обратными слешами!

     Покажите, что получается при вызове `r:repeat`, с помощью нескольких хорошо подобранных примеров. Продемонстрируйте работу процедуры на реальных данных в сложных образцах.

[10] Специальные символы ERE, [96], `#tag_09_04_03`
[11] Альтернатива ERE, [96] `#tag_09_04_07`
[12] Сопоставление с несколькими символами ERE, [96] `#tag_09_04_06`
[13] Сопоставление с несколькими символами BRE, [96] `#tag_09_03_06`

Упражнение 2.8. Слишком много вложенных групп

Наша программа порождает регулярные выражения со слишком большой вложенностью: она создает группы, даже когда они не нужны. Например, следующий простой образец порождает слишком сложное регулярное выражение:

```
(display (r:seq (r:quote "a") (r:dot) (r:quote "c")))
\(\(a\).\(c\)\)
```

Еще одна проблема в том, что базовые регулярные выражения могут включать обратные ссылки (см. раздел 9.3.6.3 документации POSIX по регулярным выражениям[14]). Обратная ссылка указывает на ранее сопоставленное подвыражение, заключенное в скобки. Поэтому важно, чтобы скобками отмечались те подвыражения, где автор действительно имел это в виду. (Ужас! Это одна из худших идей в нашем опыте — группировка, которая нужна для выражения итерации, смешивается с именованием для последующей ссылки.)

Задание. Отредактировать программу и убрать как можно больше ненужных групп. Осторожно: встречаются нетривиальные краевые случаи, за которыми надо следить. Что это за случаи? Покажите, как ваша улучшенная версия программы справляется с этими ситуациями.

Подсказка: в нашей программе как в качестве результата, так и в качестве промежуточного представления используются строки. Возможно, имеет смысл рассмотреть другое промежуточное представление.

Упражнение 2.9. Обратные ссылки

Добавьте процедуру для создания обратных ссылок (см. упражнение 2.8). Удачи вам при разбирательствах с Базовыми выражениями!

Упражнение 2.10. Стандарты?

> Самое замечательное в стандартах — есть из чего выбирать.
>
> Эндрю С. Танненбаум

Помимо Базовых регулярных выражений (BRE), в документации POSIX определяются Расширенные регулярные выражения (ERE) [96]. Некоторые программы, например `egrep`, используют именно эту версию. К сожалению, ERE не являются консервативным расширением BRE. Наоборот, их синтаксис несовместим с синтаксисом BRE! Задача расширить нашу реализацию образцов на Scheme так, чтобы она могла порождать по выбору как BRE, так и ERE, может послужить интересным проектом.

a. В чем состоят существенные отличия между BRE и ERE, причиняющие нам особенную боль? Составьте полный список.

b. Как можно разбить транслятор на части, чтобы наш язык мог переводиться в любую разновидность регулярных выражений в зависимости от нужд пользователя? Как поддержать абстрактное представление, независимое от целевого языка регулярных выражений? Объясните свою стратегию.

c. Реализуйте план, разработанный в пункте b. Продемонстрируйте работу программы, запуская как `grep`, так и `egrep` с одинаковыми результатами, в случаях, где играют роль различия, выделенные вами в пункте a.

[14] Сопоставление с несколькими символами BRE, [96] `#tag_09_03_06`

2.3 Обертки

Иногда нам удается приспособить программу к новым условиям, сделав обертку вместо того, чтобы ее переписывать. Рассмотрим задачу вычисления радиуса сферы, заполненной газом, в зависимости от температуры при постоянном давлении. Вот уравнение состояния идеального газа:

$$PV = nRT,$$

где P — давление, V — объем, n — количество вещества, R — газовая постоянная, а T — температура. Таким образом, объем вычисляется процедурой

```
(define (gas-law-volume pressure temperature amount)
  (/ (* amount gas-constant temperature) pressure))

(define gas-constant 8.3144621)                ; J/(K*mol)
```

а радиус сферы — процедурой

```
(define (sphere-radius volume)
  (expt (/ volume (* 4/3 pi)) 1/3))

(define pi (* 4 (atan 1 1)))
```

(Обратите внимание: `4/3` и `1/3` — это рациональные константы; слеш здесь не является инфиксным символом операции деления.) Выбор газовой постоянной предопределяет использование единиц СИ, так что давление считается в ньютонах на квадратный метр, температура в кельвинах, количество вещества в молях, объем в кубических метрах, а радиус в метрах.

Все это выглядит достаточно естественно, но, если нам нужно работать с другими единицами, возникают сложности. Допустим, мы хотим мерить температуру в градусах Фаренгейта, давление в фунтах на квадратный дюйм, а радиус в дюймах. Определить правильную формулу будет сложнее, чем собственно посчитать ответ. Можно внести в простую формулу изменения, чтобы учесть единицы измерения, но тогда идея программы будет затемнена, а сама она окажется слишком заточена под конкретную задачу. Либо же можно изобрести модульный способ преобразования единиц.

Преобразование единиц — это процедура, связанная с обратной ей процедурой. Можно написать преобразования температуры между традиционными единицами, например градусами Фаренгейта или Цельсия, а также между единицами СИ и традиционными единицами.

```
(define fahrenheit-to-celsius
  (make-unit-conversion (lambda (f) (* 5/9 (- f 32)))
                        (lambda (c) (+ (* c 9/5) 32))))

(define celsius-to-kelvin
  (let ((zero-celsius 273.15)) ; кельвины
    (make-unit-conversion (lambda (c) (+ c zero-celsius))
                          (lambda (k) (- k zero-celsius)))))
```

Доступ к обратной процедуре можно получить через `unit:invert`. Например,

```
(fahrenheit-to-celsius -40)
-40

(fahrenheit-to-celsius 32)
0

((unit:invert fahrenheit-to-celsius) 20)
68
```

Можно выполнять композицию преобразований:

```
((compose celsius-to-kelvin fahrenheit-to-celsius) 80)
299.81666666666666
```

Можно также определять составные преобразования. Например, давление выразить в фунтах на квадратный дюйм или в ньютонах на квадратный метр[15]:

```
(define psi-to-nsm
  (compose pound-to-newton
           (unit:invert inch-to-meter)
           (unit:invert inch-to-meter)))
```

Итак, теперь мы можем вычислить радиус сферы, содержащей 1 моль идеального газа при 68°F и 14.7 фунтах/кв. дюйм, в дюймах.

```
((unit:invert inch-to-meter)
 (sphere-radius
   (gas-law-volume
   (psi-to-nsm 14.7)
   ((compose celsius-to-kelvin fahrenheit-to-celsius) 68)
   1)))
7.049624635839811
```

Выглядит ужасно! Преобразование единиц измерения легко запрограммировать, но результат трудно использовать и трудно читать. С другой стороны, таким образом мы изящно отделяем разные группы подзадач друг от друга. Физические законы поведения газов живут отдельно от геометрии сферы и от единиц измерения. Ни физическое, ни геометрическое описание не замусорены деталями единиц, и читаются они легко.

Можно сделать еще лучше. Можно построить небольшой специализированный язык, нацеленный на единицы измерения. Тогда задача написания новых преобразований упростится, а сами преобразования станут более читаемыми.

2.3.1 Обертки со специализацией

Один из вариантов решения состоит в том, чтобы написать семейство оберток, каждая из которых может принять процедуру вроде `gas-law-volume` и выдать версию

[15] Заметим, что композиция двух экземпляров `meter-to-inch` — разумный способ преобразовать квадратные метры в квадратные дюймы. Не для любых единиц это так. Например, возводить в квадрат преобразование кельвинов в градусы Цельсия бессмысленно, хотя численные процедуры и выдадут некий результат. Это является следствием того, что ноль температуры по Цельсию сдвинут относительно ноля температуры в кельвинах, у которого есть физический смысл. Возведение в квадрат температуры по Цельсию физического смысла не имеет.

этой процедуры, модифицированную преобразованиями единиц для ее входов и выходов. Мы покажем, как это сделать для преобразования единиц измерения, но наш код будет достаточно общим, чтобы его можно было использовать для произвольных операций над данными.

Для нашей текущей задачи можно построить специализирующую обертку процедуры `gas-law-volume`, которая знает ее «родные» единицы измерения (СИ). Обертка определяется с помощью простого языка, который компилируется в необходимые комбинации элементарных преобразований единиц. Это напоминает систему комбинаторов, но только комбинаторы компилируются на основе спецификации на языке высокого уровня. Этот прием мы еще увидим в главе 4, где будем компилировать образцы, сочетая между собой элементарные процедуры сопоставления.

```
(define make-specialized-gas-law-volume
  (unit-specializer
    gas-law-volume
    '(expt meter 3)                    ; выход (объем)
    '(/ newton (expt meter 2))         ; давление
    'kelvin                            ; температура
    'mole))                            ; количество вещества
```

Чтобы получить версию процедуры `gas-law-volume`, которая использует другие единицы измерения, мы указываем, какие именно единицы нужно использовать:

```
(define conventional-gas-law-volume
  (make-specialized-gas-law-volume
    '(expt inch 3)                     ; выход (объем)
    '(/ pound (expt inch 2))           ; давление
    'fahrenheit                        ; температура
    'mole))                            ; количество вещества
```

Затем с помощью этой процедуры можно получить объем в кубических дюймах, а из него посчитать радиус в дюймах:

```
(sphere-radius (conventional-gas-law-volume 14.7 68 1))
7.04962463583981
```

2.3.2 Реализация оберток

Как заставить все это работать? Решение имеет две составные части: процедура `unit-specializer`, которая оборачивает данную ей процедуру необходимыми преобразованиями единиц, а также метод перевода выражений, задающих единицы, в соответствующие преобразования. Первая часть выглядит так:

```
(define (unit-specializer procedure implicit-output-unit
                          . implicit-input-units)
  (define (specializer specific-output-unit
                       . specific-input-units)
    (let ((output-converter
           (make-converter implicit-output-unit
                           specific-output-unit))
          (input-converters
```

```
          (map make-converter
               specific-input-units
               implicit-input-units)))
    (define (specialized-procedure . arguments)
      (output-converter
        (apply procedure
               (map (lambda (converter argument)
                      (converter argument))
                    input-converters
                    arguments))))
    specialized-procedure))
  specializer)
```

Процедура `unit-specializer` принимает на вход процедуру, подлежащую специализации, и ее подразумеваемые родные единицы измерения и возвращает специализатор, который принимает конкретные единицы измерения и возвращает исходную процедуру, приспособленную под эти единицы. Единственная сложность остается в том, чтобы скормить в `make-converter` выражения, задающие единицы, в правильном порядке.

Вторая часть решения — процедура `make-converter`, которая берет два задающих единицы выражения и возвращает процедуру-преобразователь, переводящую данные, заданные в первых из этих единиц, во вторые. Для нашей текущей задачи мы напишем чрезвычайно тупую версию `make-converter`: она рассматривает выражения-единицы как константы, которые можно сравнивать через `equal?`. При таком упрощении `make-converter` может просто искать подходящую процедуру преобразования в таблице, но тогда нам надо явным образом задавать ей все необходимые преобразования, а не выводить их из элементарных преобразований. Вот пример того, как создается таблица:

```
(register-unit-conversion 'fahrenheit 'celsius
                          fahrenheit-to-celsius)

(register-unit-conversion 'celsius 'kelvin
                          celsius-to-kelvin)
```

Так регистрируются преобразования, определенные нами ранее. После того как преобразование зарегистрировано, мы можем найти любое его направление, глядя на порядок аргументов, передаваемых в `make-converter`.

Однако нам нужны не эти преобразования сами по себе, а преобразование из `fahrenheit` в `kelvin`. Поскольку выводить его из имеющихся определений мы не хотим (это интересная, но значительно более сложная задача), нам придется строить составные преобразования из существующих. Чтобы облегчить эту работу, мы введем «алгебру» над преобразованиями единиц следующим образом:

```
(define (unit:* u1 u2)
  (make-unit-conversion (compose u2 u1)
                        (compose (unit:invert u1)
                                 (unit:invert u2))))
```

Процедура `unit:*` в сочетании с `unit:invert` дает нам общий метод комбинации преобразований единиц. Для удобства мы также добавим следующие процедуры, которые легко пишутся на основе `unit:*` и `unit:invert`:

```
(unit:/ u1 u2)
(unit:expt u n)
```

Имея эту алгебру, нетрудно записать нужные нам преобразования:

```
(register-unit-conversion 'fahrenheit 'kelvin
    (unit:* fahrenheit-to-celsius celsius-to-kelvin))

(register-unit-conversion '(/ pound (expt inch 2))
                          '(/ newton (expt meter 2))
    (unit:/ pound-to-newton
            (unit:expt inch-to-meter 2)))

(register-unit-conversion '(expt inch 3) '(expt meter 3)
    (unit:expt inch-to-meter 3))
```

2.3.3 Адаптеры

В этом разделе мы показали один из возможных способов взять существующую программу и расширить область ее возможного применения, не модифицируя исходную программу. Получившийся механизм «адаптеров» сам по себе поддается расширению, и с его помощью можно обобщить множество других типов программ.

Здесь важен следующий принцип: вместо того чтобы переписывать программу, приспосабливая ее для новой цели, лучше начать с простой и достаточно общей базовой программы и обернуть ее, получая специализированную версию для конкретных условий. Сама программа ничего не знает про обертки, а обертки почти не делают предположений об устройстве базовой программы. А процедура `unit-specializer` почти ничего не знает ни о том, ни о другом. Поскольку компоненты связаны настолько слабо, каждую из них можно обобщить во многих направлениях, включая те, о которых мы сейчас даже не думаем. Это разновидность многослойной стратегии, и мы поговорим о ней более подробно в главе 6.

Упражнение 2.11. Реализация преобразований единиц

В этом упражнении мы просим вас заполнить в нашей системе недостающие детали.

a. Для разогрева напишите процедуры `register-unit-conversion` и `make-converter`.

b. Составьте библиотеку преобразований традиционных единиц в единицы СИ. Нужны преобразования единиц массы и длины. (Время в обеих системах измеряется в секундах. Однако вас могут также интересовать минуты, часы, дни, недели, годы и т. п. Не нужно тратить слишком много усилий, пытаясь сделать систему универсальной.)

c. Добавьте полезные составные единицы, например скорость и ускорение.

d. В качестве реального проекта расширьте нашу систему специализации для какого-нибудь преобразования данных в какой-нибудь другой программе, где речь идет не о единицах измерения, а о чем-то другом.

e. Еще одно большое расширение — постройте процедуру `make-converter` так, чтобы она могла выводить составные преобразования по мере необходимости из уже зарегистрированных. Тут потребуется поиск по графу.

2.4 Абстракция предметной области

Рассмотрим, как можно создать уровень специализированного языка, на примере программ, работающих в области настольных игр. У разных настольных игр есть множество общих черт; каждая игра использует некоторые из них. Можно построить *модель предметной области*, которая будет отражать общую структуру некоторого класса игр в терминах абстрактных понятий, описывающих эти игры: фигуры, возможные положения этих фигур и элементы поведения, например движение и взятие фигуры противника.

Программа, работающая с некоторой конкретной игрой, может быть целиком реализована в терминах этой модели предметной области. Если модель обладает достаточной общностью, она будет поддерживать будущие изменения, сама не требуя модификаций.

Рассмотрим такие настольные игры, как шахматы и шашки. И та и другая — игры для двух участников, разыгрываемые на доске в виде прямоугольной сетки. У игроков есть фигуры, расположенные на этой доске. На каждом поле этой доски может в любой момент находиться не более одной фигуры. Игроки ходят по очереди. При каждом ходе игрок выбирает одну фигуру и перемещает ее на другое поле на доске. Иногда при этом происходит взятие фигуры противника. Итак, мы неформально описали модель предметной области некоторого класса настольных игр.

На основе этой модели предметной области мы можем написать *программу-судью* для шашек,* которая будет вычислять все возможные ходы для игрока в определенной игровой позиции. Реализация модели достаточно сложна и включает реализации фигур, координат и игровой доски. Чтобы упростить изложение, мы ограничимся только теми деталями, которые необходимы для написания судьи.

Судья будет устроен так: вычисляются все разрешенные ходы для каждой фигуры по отдельности, а затем они собираются в одну коллекцию. Для этого нам понадобится абстракция, позволяющая отследить результаты перемещения фигуры. Результатом может быть, например, перемена позиции. Кроме того, фигура может сменить тип (например, в шашках — проход в дамки) и может взять фигуру оппонента. Всякий ход состоит из последовательности таких изменений, применяемых к начальному состоянию доски.

Любую хорошую программу требуется переписать несколько раз. К нашим программам это тоже относится. В первом варианте у нас может не получиться ясно разделить аспекты модели, но в процессе его написания программист изучает структуру задачи. Мы продемонстрируем две различных реализации и таким образом покажем, как программа развивается и как исправляются недостатки первоначального варианта.

2.4.1 Монолитная реализация

Мы начнем с простой версии программы-судьи, которую можно написать, разбираясь в сути задачи.

*Здесь будет описываться американский вариант игры, отличающийся от русских шашек. — *Прим. перев.*

Модель предметной области для шашек

Первая реализация будет построена на основе достаточно простой модели предметной области, приспособленной конкретно к шашкам. При второй попытке мы абстрагируемся от деталей, выделяющих именно шашки, а многие детали модели скроем. Наша окончательная модель предметной области будет подходить ко многим другим похожим играм, а возможно, и к другим областям.

В нашей модели будет три абстрактных типа. *Игральная доска* отслеживает участвующие в игре фигуры и помнит, чей сейчас ход (цвет *текущего* игрока). Можно спросить, что за фигура (если она есть) находится на определенном поле. *Фигура* имеет цвет, позицию и помнит, является ли она дамкой. *Поле* определяется *координатами* и считается относительно игрока, который делает ход. Вот операции над досками:

`(current-pieces` *`доска`*`)`
выдает список фигур, принадлежащих текущему игроку;

`(is-position-on-board?` *`координаты доска`*`)`
проверяет, указывают ли *координаты* на поле внутри *доски*. Координаты, не удовлетворяющие этому предикату, при вызове любой другой операции будут приводить к ошибкам;

`(board-get` *`координаты доска`*`)`
выдает фигуру, расположенную на *доске* в поле с *координатами*. Если на этом поле нет никакой фигуры, возвращается `#f`;

`(position-info` *`координаты доска`*`)`
описывает, что находится на поле с *координатами*. Если там ничего нет, возвращается `unoccupied`. Если там стоит фигура текущего игрока, возвращается `occupied-by-self`. Если стоит фигура противника, выдается `occupied-by-opponent`;

`(is-position-unoccupied?` *`координаты доска`*`)`
равносильно тому, что `position-info` возвращает `unoccupied`;

`(is-position-occupied-by-self?` *`координаты доска`*`)`
равносильно тому, что `position-info` возвращает `occupied-by-self`;

`(is-position-occupied-by-opponent?` *`координаты доска`*`)`
равносильно тому, что `position-info` возвращает `occupied-by-opponent`.

Список операций над фигурами также невелик:

`(piece-coords` *`фигура`*`)`
выдает координаты фигуры;

`(should-be-crowned?` *`фигура`*`)`
говорит, нужно ли перевести *фигуру* в дамки — а именно, фигура не должна еще быть дамкой и должна стоять на последнем ряду;

`(crown-piece` *`фигура`*`)`
возвращает новую фигуру, такую же, как эта, только дамку;

`(possible-directions` *фигура*`)`
выдает список *направлений*, в которых в принципе может двигаться *фигура*. Разрешено ли в данный момент движение в каждом из этих направлений, не проверяется.

Система координат проста: указываются горизонталь и вертикаль в виде целых чисел. Когда мы говорим о *координатах*, имеются в виду абсолютные координаты на доске. Для относительных координат мы используем термин *смещение*. Смещение можно добавить к координатам и получить новые координаты либо же можно вычесть две пары координат друг из друга и получить смещение. *Направление* представляет собой смещение, где горизонтальная и вертикальная компоненты равны 0, 1 или -1. В шашках возможные направления — это диагональные ходы вперед, т. е. горизонтальная координата в них равна 1, а вертикальная 1 или -1. Когда шашка становится дамкой, она может двигаться и в обратном направлении, где горизонтальная координата равна -1. В шахматах возможные ходы описываются другими направлениями в зависимости от конкретной фигуры, а ход коня требует более сложного описания. Процедуры для работы с координатами мы не будем явно определять, они должны быть очевидны.

Шашечный судья

Нам потребуется структура данных, чтобы представлять ход в игре. Поскольку в течение одного хода положение фигуры может меняться несколько раз, мы будем использовать список объектов-*шагов*, в каждом из которых содержится фигура, какая она есть в начале шага, фигура после шага, состояние доски после хода и флаг, говорящий, является ли шаг прыжком. Этот список мы будем называть *цепочкой*, и храниться он будет в порядке от последнего шага к первому. Благодаря такому порядку у нас будут разделяться общие подцепочки, которые возникают, когда ход можно продолжить несколькими способами.

`(step-to` *шаг*`)`
состояние фигуры в конце *шага*.

`(step-board` *шаг*`)`
состояние доски в конце *шага*.

`(make-simple-move` *новые-координаты фигура доска*`)`
создает шаг, передвигающий *фигуру* на *доске* в *новые-координаты*.

`(make-jump` *новые-координаты координаты-противника фигура доска*`)`
создает шаг, в котором *фигура* перепрыгивает на *новые-координаты* на *доске* и берет (снимает с доски) чужую фигуру на поле *координаты-противника*.

`(replace-piece` *новая-фигура старая-фигура доска*`)`
создает шаг, который заменяет на *доске старую-фигуру* на *новую-фигуру*.

`(path-contains-jumps?` *цепочка*`)`
проверяет, является ли какой-нибудь из шагов в *цепочке* прыжком.

Давайте теперь напишем программу-судью. Вначале опишем, какие шаги возможны в указанном направлении, начиная с указанной начальной позиции. Процедура *try-step* находит возможный шаг, расширяющий данную ей цепочку. Если такого шага нет, она возвращает `#f`.

```
(define (try-step piece board direction path)
  (let ((new-coords
         (coords+ (piece-coords piece) direction)))
    (and (is-position-on-board? new-coords board)
         (case (position-info new-coords board)
           ((unoccupied)
            (and (not (path-contains-jumps? path))
                 (cons (make-simple-move new-coords
                                         piece
                                         board)
                       path)))
           ((occupied-by-opponent)
            (let ((landing (coords+ new-coords direction)))
              (and (is-position-on-board? landing board)
                   (is-position-unoccupied? landing board)
                   (cons (make-jump landing
                                    new-coords
                                    piece
                                    board)
                         path))))
           ((occupied-by-self) #f)
           (else (error "Непонятное состояние поля"))))))
```

Процедура смотрит на ближайшее поле в заданном направлении; если оно пустое, туда можно пойти. (Мы явным образом проверяем, не является ли шаг продолжением хода-прыжка, потому что такой ход запрещен.) Если на поле стоит собственная фигура игрока, шаг в этом направлении невозможен. Но если на поле расположена фигура противника, а поле вслед за этим пустое, то можно перепрыгнуть через чужую фигуру и взять ее.

Нужно перебрать все возможные направления для фигуры. Процедура `compute-next-steps` возвращает список возможных следующих шагов, расширяющих на один шаг существующую цепочку.

```
(define (compute-next-steps piece board path)
  ;; filter-map отбрасывает ложные значения
  (filter-map (lambda (direction)
                (try-step piece board direction path))
              (possible-directions piece)))
```

По правилам шашек, если возможен хотя бы один прыжок, мы обязаны выбирать среди прыжков.

```
(define (evolve-paths piece board)
  (let ((paths (compute-next-steps piece board '())))
    (let ((jumps (filter path-contains-jumps? paths)))
      (if (null? jumps)
          paths
          (evolve-jumps jumps)))))
```

А если прыжок уже сделан, требуется проверить, можно ли прыгать дальше.

```
(define (evolve-jumps paths)
  (append-map (lambda (path)
                (let ((paths
                       (let ((step (car path)))
                         (compute-next-steps (step-to step)
                                             (step-board step)
                                             path))))
                  (if (null? paths)
                      (list path)
                      ;; если возможно, продолжаем прыгать
                      (evolve-jumps paths))))
              paths))
```

Такова логика порождения ходов для одной фигуры. Судья должен запустить ее для всех фигур и собрать результаты вместе.

```
(define (generate-moves board)
  (crown-kings
   (mandate-jumps
    (append-map (lambda (piece)
                  (evolve-paths piece board))
                (current-pieces board)))))
```

Помимо порождения ходов, эта процедура делает еще две вещи. Во-первых, среди всех ходов для некоторых фигур могут быть прыжки, а для других обычные ходы, и в таком случае разрешены только прыжки.

```
(define (mandate-jumps paths)
  (let ((jumps (filter path-contains-jumps? paths)))
    (if (null? jumps)
        paths
        jumps)))
```

Во-вторых, если какая-то фигура дошла до последнего ряда, она должна сделаться дамкой.

```
(define (crown-kings paths)
  (map (lambda (path)
         (let ((piece (step-to (car path))))
           (if (should-be-crowned? piece)
               (cons (replace-piece (crown-piece piece)
                                    piece
                                    (step-board (car path)))
                     path)
               path)))
       paths))
```

Критика промежуточного решения

Получившийся код вполне симпатичен: он на удивление краток и написан в терминах модели предметной области. Однако правила шашек здесь раскиданы по всему

коду. Процедура `try-step` решает, возможен ли прыжок, но при этом то, что прыжки образуют цепочки, написано в `evolve-jumps`. Кроме того, правило, что если возможен прыжок, то надо прыгать, распределено между процедурами `evolve-paths` и `evolve-jumps`. Более глубокая проблема состоит в том, что структура управления программы-судьи перемешана с правилами игры. Например, накопление изменений (шагов в цепочке) встроено в структуру управления, и так же встроено построение цепочек прыжков. Логика обязательности прыжков оказалась в двух местах потому, что этого требует распределение структуры управления.

2.4.2 Абстрагирование предметной области

Попробуем исправить проблемы, замеченные нами в предыдущей реализации. Сможем ли мы отделить модель предметной области и структуру управления от правил игры в шашки?

Модель предметной области

Реализацию координат, фигур и доски мы можем унаследовать от первой версии, поскольку они практически не меняются. Однако мы уберем конкретное представление о дамках и не-дамках, и вместо него используем символический тип. Появляются две новых операции:

`(piece-type` *`фигура`*`)`
выдает тип *фигуры*;

`(piece-new-type` *`фигура тип`*`)`
создает новую фигуру, которая отличается от *фигуры* только тем, что имеет *тип*.

Мы заново определяем процедуры `should-be-crowned?` и `crown-piece` на основе типа фигур. Они ведут себя так же, как и раньше, но теперь это больше не часть базовой модели предметной области.

Процедура `possible-directions` относится именно к шашкам. Мы продолжим ее использовать, но только внутри определения шашечных правил. Она тоже больше не является частью базовой модели.

Структура данных шага также специфична для шашек, поскольку в ней записывается, является ли шаг прыжком. Мы заменяем ее структурой более общего вида под названием *изменение*:

`(make-change` *`доска фигура флаги`*`)`
создает новый объект-изменение. Аргумент *флаги* представляет собой список символов. С его помощью мы будем представлять изменения состояния, например взятие чужой фигуры. К компонентам объекта-изменения можно обратиться при помощи селекторов `get-board`, `get-piece` и `get-flags`.

Подобно типу фигуры, поле флагов в изменении позволяет нам добавлять к модели предметной области детали, относящиеся к конкретной игре, не встраивая их в базовую модель.

Понятие цепочки мы также заменим на более абстрактное, называемое *частичный ход*. Частичный ход состоит из исходного состояния доски и фигуры в нем, а также нуля или более изменений. В нашем коде частичный ход будет обозначаться идентификатором `pmove`.

`(initial-pmove` *доска фигура*`)`
создает частичный ход без изменений и без флагов.

`(is-pmove-empty?` *ч-ход*`)`
проверяет, является ли *ч-ход* пустым, т. е. содержит ли он какие-то изменения.

`(is-pmove-finished?` *ч-ход*`)`
проверяет, помечен ли *ч-ход* как завершенный.

`(current-board` *ч-ход*`)`
возвращает состояние доски после последнего изменения внутри *ч-хода*. Если изменений нет, возвращает доску, переданную как аргумент `initial-pmove`.

`(current-piece` *ч-ход*`)`
возвращает фигуру после последнего изменения внутри *ч-хода*. Если изменений нет, возвращает фигуру, переданную как аргумент `initial-pmove`.

Следующие несколько функций различным образом расширяют частичные ходы. Говоря «расширить ход *так-то*», мы имеем в виду «расширить ход, добавив к нему изменение, делающее *так-то*».

`(new-piece-position` *координаты* *ч-ход*`)`
расширяет ход, передвигая его фигуру на *координаты*.

`(update-piece` *процедура* *ч-ход*`)`
расширяет ход, заменяя его фигуру на результат применения к ней *процедуры*.

`(finish-move` *ч-ход*`)`
расширяет ход, добавив к нему объект-изменение с флагом, который говорит, что ход завершен. Результат всегда удовлетворяет предикату `is-pmove-finished?`.

В реализации из раздела 2.4.1 мы использовали термины *прыжок* и *взятие фигуры* как взаимозаменяемые. Но взятие в принципе более общее понятие: например, в шахматах фигура снимается, когда ходят на ее поле, а не перепрыгивают ее. Для действия, когда происходит взятие, мы используем флаг внутри изменения, и с ним работают следующие процедуры:

`(captures-pieces?` *ч-ход*`)`
проверяет, содержит ли *ч-ход* взятие каких-либо фигур.

`(capture-piece-at` *координаты* *ч-ход*`)`
расширяет *ч-ход*, убирая с доски фигуру по *координатам*. На поле, куда указывают *координаты*, должна находиться фигура противника. Эта операция также записывает флаг в новый объект-изменение, который запоминает, что произошло взятие фигуры. Получившийся объект всегда удовлетворяет предикату `captures-pieces?`.

Исполнитель

Для того чтобы отделить структуру управления от правил игры в шашки, мы строим программу-исполнитель правил, выражающую структуру управления без отсылки к содержанию конкретных правил. В играх такого рода есть два типа правил. Один тип, который мы назовем *правило развития*, расширяет частичный ход и возвращает, возможно, несколько производных частичных ходов. Второй тип, *агрегатное правило*, применяется к набору частичных ходов и удаляет часть из них, считая запрещенными, либо расширяет часть из них, отражая произошедшие изменения, например производя фигуру в дамки.

Наш исполнитель начинает с набора пустых частичных ходов, по одному на каждую фигуру игрока, и расширяет их, получая набор частичных ходов, которые представляют завершенные ходы. Затем к этому набору применяются агрегатные правила, так что в конце получается набор разрешенных ходов.

Правило развития реализуется как процедура, преобразующая данный частичный ход в набор новых частичных ходов, из которых некоторые могут быть завершенными (удовлетворять предикату `is-pmove-finished?`). Исполнитель рекурсивно применяет к набору частичных ходов все правила развития, пока все частичные ходы не станут завершенными.

Агрегатное правило реализуется как процедура, которая принимает набор завершенных ходов и порождает новый набор. Каждое агрегатное правило применяется только один раз. Никаких ограничений на порядок между агрегатными правилами нет, так что исполнитель может объединить их в одну процедуру — составное агрегатное правило. Если никаких агрегатных правил нет, составное агрегатное правило просто вернет свой аргумент неизменным.

```
(define (execute-rules initial-pmoves evolution-rules
                       aggregate-rules)
  ((reduce compose (lambda (x) x) aggregate-rules)
   (append-map (lambda (pmove)
                 (evolve-pmove pmove evolution-rules))
               initial-pmoves)))

(define (evolve-pmove pmove evolution-rules)
  (append-map (lambda (new-pmove)
                (if (is-pmove-finished? new-pmove)
                    (list new-pmove)
                    (evolve-pmove new-pmove evolution-rules)))
              (append-map (lambda (evolution-rule)
                            (evolution-rule pmove))
                          evolution-rules)))
```

Правила развития регистрируются для использования в исполнителе через процедуру `define-evolution-rule`, а агрегатные правила через процедуру `define-aggregate-rule`. Каждого правило содержит название, указание, к какой игре оно относится, и процедуру, определяющую его поведение.

Правила шашек

Вот правило для простых ходов. Оно ищет свободное прилегающее поле в одном из возможных направлений, затем расширяет частичный ход передвижением в эту позицию. После такого хода продолжать движение нельзя, поэтому мы помечаем частичный ход как завершенный.

```
(define-evolution-rule 'simple-move checkers
  (lambda (pmove)
    (if (is-pmove-empty? pmove)
        (get-simple-moves pmove)
        '())))

(define (get-simple-moves pmove)
  (filter-map
   (lambda (direction)
     (let ((landing (compute-new-position direction 1 pmove))
           (board (current-board pmove)))
       (and (is-position-on-board? landing board)
            (is-position-unoccupied? landing board)
            (finish-move (new-piece-position landing pmove)))))
   (possible-directions (current-piece pmove))))
```

В теле `get-simple-moves` процедура `compute-new-position` выдает место, куда фигура должна переместиться, исходя из направления `direction` и расстояния `distance`. Процедура `offset*` получает смещение и число и перемножает их между собой.

```
(define (compute-new-position direction distance pmove)
  (coords+ (piece-coords (current-piece pmove))
           (offset* direction distance)))
```

Правило для прыжков устроено похожим образом, но здесь нужно найти в нужном направлении занятое поле, за которым следует незанятое. Если никакие прыжки невозможны, частичный ход объявляется завершенным.

```
(define-evolution-rule 'jump checkers
  (lambda (pmove)
    (let ((jumps (get-jumps pmove)))
      (cond ((not (null? jumps))
             jumps)
            ((is-pmove-empty? pmove)
             '()) ; отказаться от этого частичного хода
            (else
             (list (finish-move pmove)))))))

(define (get-jumps pmove)
  (filter-map
    (lambda (direction)
      (let ((possible-jump
             (compute-new-position direction 1 pmove))
```

```
          (landing (compute-new-position direction 2 pmove))
          (board (current-board pmove)))
      (and (is-position-on-board? landing board)
           (is-position-unoccupied? landing board)
           (is-position-occupied-by-opponent? possible-jump
                                              board)
           (capture-piece-at possible-jump
                             (new-piece-position landing
                                                 pmove)))))
  (possible-directions (current-piece pmove))))
```

Производство в дамки работает независимо от остальных ходов: нужно посмотреть на завершенные ходы и перевести в дамки все шашки, не являющиеся дамками и находящиеся при этом на последней горизонтали.

```
(define-aggregate-rule 'coronation checkers
  (lambda (pmoves)
    (map (lambda (pmove)
           (let ((piece (current-piece pmove)))
             (if (should-be-crowned? piece)
                 (update-piece crown-piece pmove)
                 pmove)))
         pmoves)))
```

Наконец, правило, требующее, чтобы при наличии одного или более прыжков выбирался именно прыжок, запускается в конце. Оно проверяет свою применимость, а затем выкидывает все ходы, не являющиеся прыжками.

```
(define-aggregate-rule 'require-jumps checkers
  (lambda (pmoves)
    (let ((jumps (filter captures-pieces? pmoves)))
      (if (null? jumps)
          pmoves
          jumps))))
```

Критика

Реализация судьи, основанная на правилах, решает описанные нами ранее проблемы. Структура управления вынесена из программы в процедуру-исполнитель. Вследствие этого все правила оказываются конкретными; каждое правило шашек выражается отдельным процедурным правилом. В отличие от прошлой реализации, правила не размазаны по всему тексту.

Однако ничто не бывает бесплатным: к каждому правилу приходится добавлять условия, чтобы оно не применялось в неподходящей для него ситуации[16]. Например, правило `simple-move` должно отбросить подаваемые в него непустые частичные ходы, поскольку непустой частичный ход может содержать прыжки, и тогда нельзя расширять его простым передвижением. Это общий недостаток систем, основанных

[16]Впрочем, поскольку исполнитель явным образом проверяет, какие частичные ходы завершены, эту проверку внутри правил проводить не надо.

на правилах: каждое правило должно быть готово получить на вход результат любого другого; обычно с этим справляются через внесение состояния в данные, к которым правило нужно применять.

Упражнение 2.12. Немного шахмат
Та же модель предметной области, которую мы использовали для шашек, может описать и правила шахмат. Однако помимо того, что в шахматах несколько типов фигур, есть еще некоторые различия. Одно из них в том, что расстояние, доступное для хода ладье, слону и ферзю, ограничено только наличием других фигур на пути. Второе состоит в том, что взятие фигуры противника делается перемещением на занятое поле, а не прыжком. В этом упражнении мы смотрим только на ладьи и коней; остальные фигуры рассматриваются в упражнении 2.13.

a. Постройте процедуру — судью, который аналогичен шашечному и порождает ходы ладьи. Правило рокировки не учитывайте.

b. Добавьте в своего судью ходы коня.

Упражнение 2.13. Еще шахматы
Постройте полную реализацию правил шахмат.

Упражнение 2.14. Большой проект
Выберите какую-нибудь другую предметную область, не связанную с настольными играми, и постройте реализацию правил какого-либо процесса в виде процедуры-исполнителя и модели предметной области. Это непростая задача.

2.5 Заключение

Методы и приемы, продемонстрированные в этой главе, могут пригодиться при проектировании и разработке любой системы достаточно большого размера. Почти всегда полезно бывает строить систему из взаимозаменяемых частей с хорошо определенными интерфейсами.

В языках, где есть процедуры высшего порядка и лексические области действия переменных, наподобие Scheme и Java, легко построить систему комбинаторов — стандартных способов сочетания процедур вроде `compose` — для библиотеки взаимозаменяемых компонент. Удобно бывает создавать параметрические компоненты с общим интерфейсом. Например, если интерфейс описывается как процедуры с одним аргументом и одним возвращаемым значением, то

```
(define (make-incrementer dx)
  (lambda (x) (+ x dx)))
```

определяет набор процедур-увеличителей, которые можно менять друг на друга. В языках, где нет процедур высшего порядка с лексической сферой действия, таких как C, создать систему комбинаторов и библиотеку элементов, поддающихся комбинированию, намного сложнее. Однако при должном планировании и некотором усилии можно добиться и этого.

Если нам нужно работать с системой, где не всякий элемент свободно сочетается с другими (вроде регулярных выражений), часто с этими сложностями можно

справиться, применив метапрограммирование. В нашем примере мы построили новый язык на основе комбинаторов и стали компилировать его в язык регулярных выражений, получив таким образом более многословную альтернативу с более приятными свойствами. Наш язык комбинаторов регулярных выражений удобен как специализированный промежуточный язык для программ, которым нужно организовывать поиск в строках, но в качестве скриптового языка для пользователей, которым требуется вводить выражения с клавиатуры, он не так хорош. Для этой цели нам пришлось бы придумать более краткий и чисто выглядящий синтаксис сопоставления строк, который можно было бы компилировать в промежуточный язык на основе комбинаторов[17].

Обертки — это общая стратегия для использования старого кода в новом контексте. Мы показали, как можно использовать программы, предполагающие некоторую систему единиц измерения, с другими системами при помощи системы оберток, которые автоматически производят преобразование единиц. При этом мы построили простой специализированный язык для выражения преобразований единиц, и этот язык компилировался в соответствующие обертки.

Однако полезность оберток не ограничивается адаптерами для старого кода. Можно, например, построить обертку, которая проверяет входные данные на разумность, а также что результат преобразования правдоподобен при данных входных данных. Если проверки не срабатывают, обертка может подать сигнал тревоги. Такое «параноидальное программирование» является важным инструментом для защиты системы от неправильного использования, а также для отладки.

Как показано на примерах с регулярными выражениями и с преобразованиями единиц, часто лучшим способом решить проблему бывает изобретение специализированного языка, на котором конкретное решение будет легко выразить. Для исследования этой стратегии мы рассмотрели, как в задаче порождения возможных ходов в настольной игре выделяются три компонента, каждый из которых можно развивать отдельно: модель предметной области, исполнитель структуры управления и правила конкретной игры. Модель предметной области дает нам набор элементов, участвующих в правилах, и таким образом служит языком для выражения этих правил. Последовательность применения правил определяется исполнителем структуры управления. Такое сочетание служит основой специализированного языка для выражения правил настольных игр, подобных шашкам.

В любом хорошем языке есть элементарные конструкции, средства комбинирования этих конструкций и средства абстракции комбинаций. Примеры из этой главы встроены в язык Scheme, и, таким образом, с ними можно использовать мощные средства комбинирования и абстракции языка Scheme. Однако это лишь начало. В главе 5 мы пойдем дальше, чем эта стратегия встраивания, используя мощную идею металингвистической абстракции.

[17]Интересным примером может служить SRFI 115 [110].

Глава 3

Вариации на арифметическую тему

В этой главе мы вводим в обращение чрезвычайно мощный, но потенциально опасный метод повышения гибкости программ — *полиморфные процедуры*, управляемые предикатами. Начинаем мы с относительно мирной области арифметики и настраиваем значение символов операций. На первом шаге мы обобщаем арифметику, чтобы она работала с символьными алгебраическими выражениями, на втором расширяем ее действие на функции. Для этого мы используем систему комбинаторов, где комбинируемыми элементами служат пакеты арифметических операций.

Однако вскоре нам хочется еще больше гибкости. Поэтому мы изобретаем полиморфные процедуры, расширяемые динамически. Применимость обработчика определяется здесь предикатами, накладываемыми на аргументы функции. Это очень мощный и удобный инструмент. С помощью полиморфных процедур мы расширяем арифметику для работы с «объектами-дифференциалами» и получаем автоматическое дифференцирование с удивительной легкостью!

Управление через предикаты достаточно дорого стоит, так что исследуем способы понизить эту цену. Таким образом, мы изобретаем некую разновидность помеченных данных, где метка служит всего лишь способом мемоизации значения предиката. В конце главы демонстрируем мощность полиморфных процедур, проектируя игру-бродилку, простую, но легко поддающуюся расширению.

3.1 Арифметика на комбинаторах

Допустим, у нас есть программа, вычисляющая какие-то полезные числовые значения. Она зависит от значения арифметических операций, упоминаемых в ее тексте. Мы можем расширить эти операции и применить их не только к числам, которые ожидает увидеть наша программа. С таким расширением программа может научиться делать полезные вещи, которые создатели при ее написании не предвидели. Например, часто встречаются программы, куда вводятся численные веса и другие аргументы и где вычисляются линейные комбинации через взвешенную сумму аргументов. Если мы расширим операции сложения и умножения, чтобы они работали не только с числами, но и с кортежами чисел, получится программа для

построения линейных комбинаций векторов. Такое расширение работает, потому что набор арифметических операций — хорошо определенная и внутренне согласованная сущность. Расширения численных программ будут работать правильно до тех пор, пока новые виды величин повинуются правилам, которые подразумевал автор программы. Например, умножение матриц зависит от их порядка, так что расширение программы, опирающейся на коммутативность умножения, в этом случае работать не будет. На эту проблему мы пока что не обращаем внимания.

3.1.1 Простой интегратор ОДУ

Дифференциальное уравнение — это описание, как ведет себя состояние системы при изменениях независимой переменной. Это называется *эволюцией* состояния системы[1]. Можно приближенно вычислить эволюцию состояния системы, если брать значение независимой переменной в различных точках и приближенно считать изменение состояния в каждой из этих точек. Такое вычисление называется *численным интегрированием*.

Давайте исследуем совокупность числовых операций при численном интегрировании для дифференциальных уравнений второго порядка. Наш интегратор будет смотреть на значения независимой переменной через равные промежутки; каждый такой промежуток называется *шагом*. Рассмотрим уравнение

$$D^2x(t) = F(t, x(t)). \tag{3.1}$$

Основная идея здесь в том, что дискретная аппроксимация второй производной неизвестной функции представляет собой линейную комбинацию вторых производных на некоторых предыдущих шагах. Конкретные коэффициенты определяются численным анализом и не представляют сейчас интереса.

$$\frac{x(t+h) - 2x(t) + x(t-h)}{h^2} = \sum_{j=0}^{k} A(j)F(t - jh, x(t - jh)), \tag{3.2}$$

где h — размер шага, а A — массив волшебных коэффициентов.

Например, интегратор Стёрмера второго порядка выглядит как

$$\begin{aligned} &x(t+h) - 2x(t) + x(t-h) \\ &\quad = \tfrac{h^2}{12}(13F(t, x(t)) - 2F(t-h, x(t-h)) + F(t-2h, x(t-2h))). \end{aligned} \tag{3.3}$$

Чтобы с его помощью вычислять будущие значения x, нам надо написать программу. Из процедуры `stormer-2` возвращается процедура-интегратор для данной функции и размера шага. На основании истории значений x интегратор выдает оценку значения x на следующем шаге, $x(t+h)$. Процедуры `t` и `x` извлекают из истории предыдущие значения времени и функции x: `(x 0 history)` выдает $x(t)$, `(x 1 history)` выдает $x(t-h)$, а `(x 2 history)` выдает $x(t-2h)$. Значение времени на каждом шаге мы получаем подобным же образом: `(t 1 history)` выдает $t-h$.

```
(define (stormer-2 F h)
  (lambda (history)
```

[1] ОДУ значит «обыкновенное дифференциальное уравнение», т. е. дифференциальное уравнение с одной независимой переменной.

```
(+ (* 2 (x 0 history))
   (* -1 (x 1 history))
   (* (/ (expt h 2) 12)
      (+ (* 13 (F (t 0 history) (x 0 history)))
         (* -2 (F (t 1 history) (x 1 history)))
         (F (t 2 history) (x 2 history)))))))
```

Процедура, возвращаемая из `stepper`, принимает на вход историю и возвращает новую историю для данного интегратора, продвинутую на шаг h.

```
(define (stepper h integrator)
  (lambda (history)
    (extend-history (+ (t 0 history) h)
                    (integrator history)
                    history)))
```

`Stepper` используется в процедуре `evolver`, где с его помощью порождается процедура `step`, продвигающая историю на один шаг. Эта процедура `step` затем используется в процедуре `evolve`, которая продвигает историю на заданное количество шагов размера h. Для подсчета шагов мы здесь явным образом используем специализированную арифметику целых чисел (процедуры с именами `n:>` и `n:-`). Это позволит не трогать простой подсчет шагов, когда мы станем подменять арифметику во всех остальных вычислениях[2].

```
(define (evolver F h make-integrator)
  (let ((integrator (make-integrator F h)))
    (let ((step (stepper h integrator)))
      (define (evolve history n-steps)
        (if (n:> n-steps 0)
            (evolve (step history) (n:- n-steps 1))
            history))
      evolve)))
```

Для задания определенной траектории дифференциальное уравнение второго порядка вроде уравнения 3.1 в общем случае требует два начальных значения: для задания $x(t)$ при любом t достаточно знать $x(t_0)$ и $x'(t_0)$. Однако многошаговый интегратор Стёрмера, который мы здесь используем, для вычисления $x(t_0+h)$ требует три значения истории $x(t_0)$, $x(t_0-h)$ и $x(t_0-2h)$. Таким образом, чтобы получить развитие траектории с помощью этого интегратора, нам нужно начать с исходной истории, где есть хотя бы три прошлых значения x.

Рассмотрим очень простое дифференциальное уравнение:

$$D^2x(t) + x(t) = 0.$$

В форме, требуемой уравнением 3.1, правая часть имеет вид:

[2]Поскольку при разработке системы MIT/GNU Scheme мы предвидели, что значение многих операций может изменяться, мы сделали отдельный набор операций, обозначающих элементарные процедуры, к которым хочется сохранять доступ. Этим копиям мы дали имена с префиксом `n:`. Кроме того, в MIT/GNU Scheme исходные элементарные процедуры всегда доступны по своим исходным именам через окружение `system-global-environment`, так что можно было бы получить их оттуда.

```
(define (F t x) (- x))
```

Поскольку все решения этого уравнения представляют собой линейные комбинации синусоид, можно получить функцию синуса, проинициализировав историю тремя значениями:

```
(define numeric-s0
  (make-initial-history 0 .01 (sin 0) (sin -.01) (sin -.02)))
```

где процедура `make-initial-history` принимает следующие аргументы:

`(make-initial-history` $t\ \ h\ \ x(t)\ \ x(t-h)\ \ x(t-2h)$`)`

При использовании встроенной арифметики Scheme через 100 шагов размера $h = .01$ мы получаем неплохое приближение $\sin(1)$:

```
(x 0 ((evolver F .01 stormer-2) numeric-s0 100))
.8414709493275624
(sin 1)
.8414709848078965
```

3.1.2 Настройка арифметических операций

Рассмотрим возможность настроить, что именно мы называем сложением, умножением и т. д. для новых типов данных, которые автор нашего примера не имел в виду. Допустим, мы заставим арифметические операции принимать на вход и создавать на выходе символьные выражения вместо числовых значений. Это может быть полезно при отладке числовых вычислений, поскольку, задав на вход символьные аргументы, мы можем исследовать получившиеся символьные выражения и убедиться, что программа вычисляет именно то, что мы хотим от нее. Кроме того, такая возможность может послужить основой частичного вычислителя для оптимизации численных программ.

Вот один из способов добиться нашей цели. Мы вводим понятие *арифметического пакета*. Арифметический пакет, или просто *арифметика* — это отображение имен операций в сами реализующие их операции. Можно подгрузить арифметику в диалоговую среду пользователя и заменить умолчательные значения операций, упоминаемых в этой арифметике, реализациями из ее отображения.

Процедура `make-arithmetic-1` порождает новый арифметический пакет. Она принимает имя пакета и процедуру порождения операций, которая, получив имя операции, строит саму *операцию* в виде процедуры-обработчика для этого имени. Процедура `make-arithmetic-1` вызывает эту процедуру порождения операций для каждого имени и собирает результаты в новый арифметический пакет. В случае символьной арифметики операция реализуется как процедура, создающая символьное выражение, присоединяя имя операции к списку аргументов.

```
(define symbolic-arithmetic-1
  (make-arithmetic-1 'symbolic
    (lambda (operator)
      (lambda args (cons operator args)))))
```

Чтобы использовать свою свежеопределенную арифметику, мы загружаем ее. Вот выражение, которое переопределяет арифметические операторы, чтобы использовалась символьная арифметика[3]:

```
(install-arithmetic! symbolic-arithmetic-1)
```

Вызов `install-arithmetic!` изменяет глобальные переменные пользователя, являющиеся именами операций в арифметике, и устанавливает их в значения, задаваемые этой арифметикой. Например, после загрузки имеем

```
(+ 'a 'b)
(+ a b)

(+ 1 2)
(+ 1 2)
```

Теперь можно посмотреть результат одного шага эволюции по Стёрмеру[4,5]:

```
(pp (x 0
       ((evolver F 'h stormer-2)
        (make-initial-history 't 'h 'xt 'xt-h 'xt-2h)
        1)))
(+ (+ (* 2 xt) (* -1 xt-h))
   (* (/ (expt h 2) 12)
      (+ (+ (* 13 (negate xt)) (* -2 (negate xt-h)))
      (negate xt-2h))))
```

Можно было бы без труда порождать упрощенные выражения, если заменить `cons` в теле `symbolic-arithmetic-1` на процедуру алгебраического упрощения, и тогда у нас получилась бы программа работы с символьной алгеброй. (Мы исследуем упрощение алгебраических выражений в разделе 4.2.)

Проведенное нами преобразование было до смешного простым; при этом наша исходная программа не была никак специально подготовлена к символьным вычислениям. Точно так же легко мы могли бы поддержать векторную арифметику или матричную, и т. д.

Сложности, возникающие при переопределении операций

Способность переопределять операции *после того, как программа написана*, дает нам чрезвычайную гибкость. Но она же служит источником нового большого класса ошибок. (Мы предвидели эту проблему в процедуре `evolver` и избежали ее, используя для подсчета шагов специальные арифметические операции `n:>` и `n:-`.)

[3]Недавно новая редакция языка Scheme [109] ввела понятие «библиотек», которые позволяют указать связывания для ссылок на свободные переменные в программе. Мы могли бы связывать арифметику с кодом, который ее использует, при помощи этих библиотек. Однако здесь мы демонстрируем свои идеи путем модификации интерактивной среды исполнения.

[4]Процедура `pp` «красиво» распечатывает список, отображая его структуру при помощи разбиения на строки и отступов.

[5]Вы могли заметить, что в этих символьных выражениях сложение и умножение используются как двухместные операции, притом что в Scheme они могут принимать произвольное число аргументов; наша программа-загрузчик реализует n-арные версии в виде вложенных вызовов двухместных операций. Подобным образом одноместный вызов `-` преобразуется в `negate`. Вычитание и деление с несколькими аргументами также реализуются в виде вложенных двухместных вызовов.

Есть и более тонкие проблемы. Программа, полагающаяся на точность результатов операций над целыми числами, может неправильно работать с неточными числами с плавающей точкой. Это ровно тот же самый риск, с которым сталкиваются при эволюции биологические и технические системы — некоторые мутации оказываются смертельными! С другой стороны, некоторые мутации будут чрезвычайно полезными. Но этот риск нужно взвешивать по сравнению со стоимостью узкоприменимой и хрупкой конструкции.

В сущности, про программу почти ничего невозможно доказать, если в ней могут переопределяться элементарные процедуры, кроме того, что она будет работать с теми типами, для которых исходно написана. Этот путь обобщения легкий, но опасный.

3.1.3 Сочетание разных арифметик

Символьная арифметика не может производить арифметические вычисления, так что, заменив определения операций в нашем примере с интегрированием, мы его сломали. На самом деле мы хотим, чтобы действие операции зависело от ее аргументов: например, числовое сложение для `(+ 1 2)`, но построение списка для `(+ 'a 'b)`. Так что арифметические пакеты должны уметь определять, какой из обработчиков для каких аргументов требуется.

Улучшенная арифметическая абстракция

Снабдив каждую операцию *спецификацией ее применимости* (этот термин часто сокращают просто до *применимости*), мы учимся сочетать разные виды арифметики. Например, можно использовать и символьную, и численную арифметику так, чтобы каждая составная операция сама определяла, какая реализация подходит к ее аргументам.

Спецификация применимости — это всего лишь список *случаев*, каждый из которых является списком предикатов вроде `number?` или `symbolic?`. Процедура объявляется применимой к последовательности аргументов, если аргументы удовлетворяют одному из случаев — т. е. если каждый предикат этого случая истинен относительно соответствующего аргумента. Например, для двухместных арифметических операций нам хотелось бы, чтобы численные операции были применимы только в случае `(number? number?)`, а символьные в случаях `((number? symbolic?) (symbolic? number?) (symbolic? symbolic?))`.

Для создания операции, которая включает в себя применимость для процедуры-обработчика, мы используем `make-operation`:

```
(define (make-operation operator applicability procedure)
  (list 'operation operator applicability procedure))
```

У такой операции можно получить применимость:

```
(define (operation-applicability operation)
  (caddr operation))
```

Мы вводим абстракцию, позволяющую записывать для операции информацию о ее применимости. Процедура `all-args` принимает два аргумента, первый из которых — число аргументов, ожидаемых операцией (ее *арность*, как на стр. 38), а второй —

предикат, который должен быть верен для каждого аргумента. Она возвращает спецификацию применимости, с помощью которой можно определить, применима ли операция к данным ей аргументам. В численной арифметике все операции в качестве всех аргументов принимают числа.

Через `all-args` мы можем реализовать конструктор для простейших операций:

```
(define (simple-operation operator predicate procedure)
  (make-operation operator
                  (all-args (operator-arity operator)
                            predicate)
                  procedure))
```

Полезно будет иметь также *предикат области действия*, истинный для объектов (скажем, функций или матриц), которые операции данной арифметики принимают как аргументы — например, в случае численной арифметики это `number?`. Чтобы поддержать эту более сложную идею, мы создаем конструктор арифметических пакетов `make-arithmetic`. Эта процедура подобна `make-arithmetic-1` (см. стр. 74), но у нее больше аргументов.

```
(make-arithmetic имя
                 предикат-области
                 базовые-арифметические-пакеты
                 отображение-имен-констант-в-константы
                 отображение-имен-операций-в-операции)
```

У арифметического пакета, созданного через `make-arithmetic`, есть имя, что полезно для отладки. Есть предикат области, как описано выше. Есть список арифметических пакетов, называемых *базовыми*, на основе которых создается текущий пакет. Кроме того, арифметика содержит множество именованных констант и набор имен операций вместе с их реализациями. Для порождения этих множеств служат два последних аргумента.

Примером использования базовой арифметики могут служить векторы. Вектор представляется как упорядоченная последовательность координат; следовательно, арифметика векторов определяется на основе арифметики их координат. Таким образом, для векторной арифметики подходящая арифметика над координатами будет базовой. Векторная арифметика с числовыми координатами использует в качестве базы численную арифметику, а векторная арифметика с символическими координатами использует в качестве базы символьную арифметику. Для краткости мы часто будем указывать базу через предлог «над», например, «векторы над числами» или «векторы над символами».

Базовая арифметика задает также константы и операции, определяемые производной арифметикой. Определяемые константы — объединение констант баз, а определяемые операции — объединение их операций. Если никаких баз нет, будет определен стандартный набор имен констант и операций.

С этими новыми возможностями мы определяем численную арифметику с информацией о применимости. Поскольку численная арифметика строится на основе Scheme, подходящим обработчиком для числовых аргументов Scheme будет просто значение соответствующего символа операции в реализации Scheme. Кроме того, некоторые символы, например нейтральные элементы сложения и умножения, задаются отдельно явным образом.

```
(define numeric-arithmetic
  (make-arithmetic 'numeric number? '()
    (lambda (name)                ; генератор констант
      (case name
        ((additive-identity) 0)
        ((multiplicative-identity) 1)
        (else (default-object))))
    (lambda (operator)            ; генератор операций
      (simple-operation operator number?
         (get-implementation-value
            (operator->procedure-name operator))))))
```

Последние две строки этого фрагмента ищут в реализации Scheme процедуру с заданным именем операции[6].

Подобным образом мы можем написать конструктор `symbolic-extender`, который создает символьную арифметику поверх данной.

```
(define (symbolic-extender base-arithmetic)
  (make-arithmetic 'symbolic symbolic? (list base-arithmetic)
    (lambda (name base-constant)        ; генератор констант
      base-constant)
    (let ((base-predicate
           (arithmetic-domain-predicate base-arithmetic)))
      (lambda (operator base-operation) ; генератор операций
        (make-operation operator
                        (any-arg (operator-arity operator)
                                 symbolic?
                                 base-predicate)
                        (lambda args
                          (cons operator args)))))))
```

Одно из отличий этой арифметики от численной состоит в том, что символьная арифметика применима всегда, когда *хотя бы один* из аргументов операции является символьным выражением[7]. Это видно по использованию `any-arg` вместо `all-args`; процедура `any-arg` проверяет, что хотя бы один из аргументов удовлетворяет предикату, переданному как второй аргумент, а все остальные — предикату, переданному как третий аргумент[8]. Заметим также, что символьная арифметика базируется на указанной арифметике `base-arithmetic`, что позволит нам создать несколько таких арифметик.

Спецификации применимости не служат для обработчиков защитной проверкой: они не запрещают применять обработчик к неправильным аргументам. Они исполь-

[6]Процедура `default-object` порождает объект, отличающийся от любой возможной константы. Процедура `default-object?` распознает этот объект.

[7]Еще одно различие, которое вы можете заметить, в том, что процедуры порождения констант и операций для численной арифметики имеют только по одному формальному параметру, а процедуры порождения символьного расширения — по два. Символьная арифметика строится на основе базовой, так что константа или операция базовой арифметики передается в генератор.

[8]Вызов `(any-arg 3 p1? p2?)` породит спецификацию применимости с семью случаями, поскольку применимость можно удовлетворить семью способами: `((p2? p2? p1?) (p2? p1? p2?) (p2? p1? p1?) (p1? p2? p2?) (p1? p2? p1?) (p1? p1? p2?) (p1? p1? p1?))`.

зуются только для выбора между различными возможными реализациями данной операции, когда арифметики сочетаются между собой, как описано дальше.

Комбинатор арифметик

Символьная и численная арифметики по своему построению имеют одну и ту же форму. Процедура `symbolic-extender` порождает арифметику с теми же именами операций, что и в данной ей базовой арифметике. Хорошим подходом может быть создание комбинаторного языка для построения композитных арифметик.

Процедура `add-arithmetics`, приведенная ниже, является комбинатором арифметик. Она строит новую арифметику, чей предикат области — дизъюнкция предикатов области арифметик-аргументов, а все операции отображаются на объединение операций арифметик-аргументов[9].

```
(define (add-arithmetics . arithmetics)
  (add-arithmetics* arithmetics))

(define (add-arithmetics* arithmetics)
  (if (n:null? (cdr arithmetics))
      (car arithmetics)            ; только одна арифметика
      (make-arithmetic 'add
                       (disjoin*
                        (map arithmetic-domain-predicate
                        arithmetics))
                       arithmetics
                       constant-union
                       operation-union)))
```

Третьим аргументом процедуры `make-arithmetic` служит список сочетаемых арифметических пакетов. Пакеты должны быть совместимы между собой, т. е. определять одни и те же имена операций. Четвертый аргумент — процедура `constant-union`, сочетающая несколько констант. Здесь она выбирает для комбинированной арифметики одну из констант-аргументов; позже мы это обсудим[10].

```
(define (constant-union name . constants)
  (let ((unique
         (remove default-object?
                 (delete-duplicates constants eqv?))))
    (if (n:pair? unique)
        (car unique)
        (default-object))))
```

Последним аргументом служит процедура `operation-union`, строящая операцию в создаваемой арифметике для данного имени. Операция применима, если она применима хотя бы в одной из комбинируемых арифметик.

[9] Процедура `disjoin*` — это комбинатор предикатов. Он принимает список предикатов и возвращает предикат-дизъюнкцию.

[10] Произвольный выбор здесь вряд ли имеет смысл. Например, нулевой вектор не только отличается от числового нуля, но различен и для векторов разной размерности. Здесь мы решили на эту проблему не обращать внимания.

```
(define (operation-union operator . operations)
  (operation-union* operator operations))

(define (operation-union* operator operations)
  (make-operation operator
                  (applicability-union*
                   (map operation-applicability operations))
                  (lambda args
                    (operation-union-dispatch operator
                                              operations
                                              args))))
```

Процедура `operation-union-dispatch` должна определить, какую из операций нужно применить, на основании данных ей аргументов. Она выбирает одну из операций исходных арифметик, подходящую для аргументов, и применяет ее. Если подходящая операция есть больше, чем в одной исходной арифметике, берется операция из первого аргумента процедуры `add-arithmetics`.

```
(define (operation-union-dispatch operator operations args)
  (let ((operation
         (find (lambda (operation)
                 (is-operation-applicable? operation args))
               operations)))
    (if (not operation)
        (error "Неприменимая операция:" operator args))
        (apply-operation operation args)))
```

Часто встречается шаблон, когда базовая арифметика сочетается со своим собственным расширением. Например, так происходит, когда численная арифметика сочетается с символьной арифметикой на основе численной. Для такого шаблона мы даем отдельную абстракцию:

```
(define (extend-arithmetic extender base-arithmetic)
  (add-arithmetics base-arithmetic
                   (extender base-arithmetic)))
```

С помощью `extend-arithmetic` мы можем сочетать численную арифметику и символьную. Поскольку случаи применимости не пересекаются — для численной арифметики все аргументы должны быть числами, а для символьной хотя бы один должен быть символьным выражением, — порядок аргументов `add-arithmetics` здесь не имеет значения с точностью до скорости работы.

```
(define combined-arithmetic
  (extend-arithmetic symbolic-extender numeric-arithmetic))

(install-arithmetic! combined-arithmetic)
```

Давайте опробуем получившуюся композитную арифметику:

```
(+ 1 2)
3
```

```
(+ 1 'a)
(+ 1 a)

(+ 'a 2)
(+ a 2)

(+ 'a 'b)
(+ a b)
```

Интегратор продолжает работать в численных случаях (сравните со стр. 74):

```
(define numeric-s0
  (make-initial-history 0 .01 (sin 0) (sin -.01) (sin -.02)))

(x 0 ((evolver F .01 stormer-2) numeric-s0 100))
.8414709493275624
```

Работает и символически (сравните со стр. 75):

```
(pp (x 0
       ((evolver F 'h stormer-2)
        (make-initial-history 't 'h 'xt 'xt-h 'xt-2h)
        1)))
(+ (+ (* 2 xt) (* -1 xt-h))
   (* (/ (expt h 2) 12)
      (+ (+ (* 13 (negate xt)) (* -2 (negate xt-h)))
      (negate xt-2h))))
```

Наконец, он работает в смешанных случаях, с численной историей, но символьным шагом `h`:

```
(pp (x 0 ((evolver F 'h stormer-2) numeric-s0 1)))
(+ 9.999833334166664e-3
   (* (/ (expt h 2) 12)
      -9.999750002487318e-7))
```

Заметим, какая мощная система у нас получилась. Мы объединили код, способный проводить символьные арифметические операции, с кодом, проводящим числовые операции. Мы создали систему, которая может сочетать арифметику на основе обеих этих способностей. Это не просто объединение двух способностей, а совместная работа двух механизмов, достигающая результатов, которые ни один из них не может получить сам по себе.

3.1.4 Арифметика на функциях

Традиционная математика распространяет понятие арифметики с чисел на многие другие виды объектов. На протяжении веков «арифметика» стала применяться к комплексным числам, векторам, линейным преобразованиям и их представлениям в виде матриц и т. д. Одно из особенно важных расширений — на функции. Функции

одного типа можно сочетать с помощью арифметических операций:

$$\begin{aligned}(f+g)(x) &= f(x)+g(x)\\ (f-g)(x) &= f(x)-g(x)\\ (fg)(x) &= f(x)g(x)\\ (f/g)(x) &= f(x)/g(x).\end{aligned}$$

У сочетаемых функций должны быть одинаковые области определения и области значений, и на области значения должна быть определена арифметика.

Расширение на функции не представляет сложности. Имея арифметический пакет для области значений функций, подлежащих сочетанию, и считая, что функции реализуются в виде процедур, можно создать арифметический пакет, реализующий арифметику на функциях.

```
(define (pure-function-extender codomain-arithmetic)
  (make-arithmetic 'pure-function function?
                   (list codomain-arithmetic)
    (lambda (name codomain-constant)    ; *** см. ниже
      (lambda args codomain-constant))
      (lambda (operator codomain-operation)
        (simple-operation operator function?
          (lambda functions
            (lambda args
              (apply-operation codomain-operation
                               (map (lambda (function)
                                      (apply function args))
                                    functions))))))))
```

Заметим, что генератор констант (с комментарием ***) должен порождать по константной функции на каждую константу области значений. Например, нейтральный элемент сложения для функций — это функция произвольного числа аргументов, возвращающая нейтральный элемент сложения области значений.

Сочетание функциональной арифметики с арифметикой, работающей в области значений, является полезным пакетом.

```
(install-arithmetic!
  (extend-arithmetic pure-function-extender
                     numeric-arithmetic))

((+ cos sin) 3)
-.8488724885405782

(+ (cos 3) (sin 3))
-.8488724885405782
```

Имея в качестве основы `combined-arithmetic`, мы получаем более интересные результаты:

```
(install-arithmetic!
  (extend-arithmetic pure-function-extender
                     combined-arithmetic))
```

```
((+ cos sin) 3)
-.8488724885405782

((+ cos sin) 'a)
(+ (cos a) (sin a))

(* 'b ((+ cos sin) (+ (+ 1 2) 'a)))
(* b (+ (cos (+ 3 a)) (sin (+ 3 a))))
```

Кроме того, математическая традиция позволяет сочетать числовые величины с функциями, рассматривая числовые величины как константные функции того же типа, как функции, с которыми они сочетаются.

$$(f + 1)(x) = f(x) + 1 \tag{3.4}$$

Это преобразование числовых величин в константные функции без труда реализуется ценой небольших изменений в процедуре `pure-function-extender`:

```
(define (function-extender codomain-arithmetic)
  (let ((codomain-predicate
         (arithmetic-domain-predicate codomain-arithmetic)))
    (make-arithmetic 'function
                     (disjoin codomain-predicate function?)
                     (list codomain-arithmetic)
      (lambda (name codomain-constant)
        codomain-constant)
      (lambda (operator codomain-operation)
        (make-operation operator
                        (any-arg (operator-arity operator)
                                 function?
                                 codomain-predicate)
          (lambda things
            (lambda args
              (apply-operation codomain-operation
                 (map (lambda (thing)
                        ;; вот преобразование:
                        (if (function? thing)
                            (apply thing args)
                            thing))
                      things)))))))))
```

Чтобы было разрешено преобразование элементов области значения, например чисел, в константные функции, область определения нашей новой арифметики над функциями должна включать и функции, и элементы их области значений (возможных результатов применения функций). Реализация операции применима, если хотя бы один из аргументов является функцией; в этом случае функции применяются к имеющимся аргументам. Заметим, что генератор констант в `make-arithmetic` уже не обязан преобразовывать константы области значений в функции, поскольку теперь мы можем использовать константы напрямую.

С этой версией начинает работать

```
(install-arithmetic!
  (extend-arithmetic function-extender combined-arithmetic))

((+ 1 cos) 'a)
(+ 1 (cos a))

(* 'b ((+ 4 cos sin) (+ (+ 1 2) 'a)))
(* b (+ 4 (cos (+ 3 a)) (sin (+ 3 a))))
```

Возникает интересный вопрос: у нас есть символы, такие как `a` и `b`, представляющие литералы чисел, но у нас нет ничего, что представляло бы литералы функций. Например, если мы пишем

```
(* 'b ((+ 'c cos sin) (+ 3 'a)))
```

то наша арифметика будет считать `c` числовым литералом. Но мы можем хотеть, чтобы `c` была функциональным литералом, который сочетается с другими объектами как функция. В нашем нынешнем проекте это сделать трудно, поскольку при `c` нет никакой информации о типе, а контекста здесь недостаточно, чтобы различить эти два возможных вида употреблений.

Однако можно сделать функцию, у которой нет никаких свойств, кроме имени. Такая функция просто прикрепляет свое имя к списку аргументов.

```
(define (literal-function name)
  (lambda args
    (cons name args)))
```

При этом определении функция-литерал `c` будет правильно сочетаться с другими функциями:

```
(* 'b ((+ (literal-function 'c) cos sin) (+ (+ 1 2) 'a)))
(* b (+ (+ (c (+ 3 a)) (cos (+ 3 a))) (sin (+ 3 a))))
```

Это решение узкоприменимое, но оно работает в важном частном случае.

3.1.5 Сложности с комбинаторами

Арифметические структуры, которые мы до сих пор строили, были примером того, как строить сложные структуры из простых при помощи комбинаторов. Однако построение системы на основе комбинаторов имеет свои недостатки. Во-первых, некоторые свойства структуры определяются средствами комбинирования. Например, как мы указывали, процедура `add-arithmetics` задает приоритет на своих аргументах, так что их порядок имеет значение. Во-вторых, уровневая структура, встроенная в такой план строения, что арифметика области значений обязана создаваться раньше, чем арифметика функций, делает невозможным расширение арифметики для области значений после того, как арифметика функций построена. Наконец, мы можем захотеть определить арифметику функций, возвращающих функции. В нашей нынешней структуре это невозможно сделать в общем случае, если не вводить отдельный механизм ссылок на себя, а организовать ссылки на себя сложно и неудобно.

Комбинаторы — мощное и полезное средство, но система на их основе получается не очень гибкой. Одна из проблем в том, что форма деталей должна быть

продумана заранее: общность всей системы зависит от точного устройства ее частей, и нужно заранее иметь детальный план, как именно они будут сочетаться. Для хорошо изученной предметной области вроде арифметики такое требование не является большой проблемой, но для исследовательской системы оно не подходит. В разделе 3.2 мы увидим, как новые виды арифметики можно добавлять по очереди, не решая заранее, как они выстраиваются в иерархию и не изменяя уже готовые работающие детали.

Еще одна проблема с комбинаторами состоит в том, что поведение каждой части комбинаторной системы должно быть независимо от ее контекста. Для проектировщика системы возможность встроить зависимость от контекста может быть важным источником гибкости. Изменяя контекст, можно добиться вариативного поведения системы. Это опасная стратегия, поскольку часто бывает трудно предсказать, как именно изменится поведение. Однако тщательно ограниченная вариативность может быть полезна.

Упражнение 3.1. Разминка: булева арифметика

В проектировании цифровых схем булевы операции *и*, *или* и *не* принято записывать символами, соответственно, `*`, `+` и `-`.

В языке Scheme существует предикат `boolean?`, истинный только для `#t` и `#f`. Постройте с его помощью булев арифметический пакет, который можно сочетать с имеющейся у нас арифметикой. Заметим, что все остальные операции для булевых величин не определены, так что правильным результатом применения к булевой величине чего-то вроде `cos` будет сообщение об ошибке.

Можно воспользоваться следующей заготовкой:

```
(define boolean-arithmetic
  (make-arithmetic 'boolean boolean? '()
    (lambda (name)
      (case name
        ((additive-identity) #f)
        ((multiplicative-identity) #t)
        (else (default-object))))
    (lambda (operator)
      (let ((procedure
             (case operator
               ((+) <...>)
               ((-) <...>)
               ((*) <...>)
               ((negate) <...>)
               (else
                 (lambda args
                   (error "Операция не определена для булевых величин"
                          operator))))))
        (simple-operation operator boolean? procedure)))))
```

В проектировании схем операция `-` обычно используется только как одноместная и реализуется в виде `negate`. При загрузке арифметики двухместные операции `+`, `*`, `-` и `/` обобщаются на n-местный случай. Унарное применение (`- `*операнд*`)` загрузчик преобразует в (`negate `*операнд*`)`. Таким образом, для того чтобы `-` правильно заработал, вам нужно определить одноместную булеву операцию для `negate`.

Упражнение 3.2. Векторная арифметика

Построим и загрузим арифметический пакет для геометрических векторов. Это большое задание, и оно покажет многие сложности и недостатки системы, как мы ее до сих пор строили.

a. Будем представлять вектор в языке Scheme как `vector` с числовыми элементами. Элементами будут служить координаты относительно каких-то декартовых осей. Здесь есть несколько вопросов. Сложение (и вычитание) определено только относительно векторов с одинаковой размерностью, так что ваша арифметика обязана знать про размерности. Сначала создайте арифметику, где определено только сложение, обращение и вычитание векторов относительно базовой арифметики, чьи операции применимы к координатам векторов. Попытка применить любую другую операцию должна приводить к ошибке. Подсказка: пригодятся следующие процедуры:

```
(define (vector-element-wise element-procedure)
  (lambda vecs    ; Внимание: здесь передаются несколько векторов
    (ensure-vector-lengths-match vecs)
    (apply vector-map element-procedure vecs)))

(define (ensure-vector-lengths-match vecs)
  (let ((first-vec-length (vector-length (car vecs))))
    (if (any (lambda (v)
               (not (n:= (vector-length v)
                         first-vec-length)))
             vecs)
        (error "Не совпадают размерности:" vecs))))
```

Вызов `apply` в этом коде требует внимания. Один из способов понять его — представить, как будто язык поддерживает многоточие:

```
(define (vector-element-wise element-procedure)
  (lambda (v1 v2 ...)
    (vector-map element-procedure v1 v2 ...)))
```

Постройте эту арифметику и убедитесь, что она работает на числовых векторах и на векторах, где в координатах смешаны числовые и символьные выражения.

b. Ваша векторная арифметика опиралась на сложение координат. Процедура сложения может быть значением операции `+`, загруженной в окружение пользователя вызовом `install-arithmetic!`, либо же она может быть операцией сложения из базовой арифметики вашего векторного расширения. В любом из этих случаев многие тесты будут работать, и на самом деле использование загруженной арифметики может обладать большей общностью. Какой из вариантов вы использовали? Покажите, как реализовать другой вариант. Как сделанный выбор влияет на возможность будущих добавлений к системе? Аргументируйте свои ответы.

Подсказка: неплохим способом управлять интерпретацией символов операций внутри процедуры может быть задать процедуру для каждого символа как аргумент в «процедуре-фабрике», возвращающей окончательную процедуру. Например, для управления арифметическими операциями в процедуре нахождения модуля вектора `vector-magnitude` можно написать

```
(define (vector-magnitude-maker + * sqrt)
  (let ((dot-product (dot-product-maker + *)))
    (define (vector-magnitude v)
      (sqrt (dot-product v v)))
    vector-magnitude))
```

c. Что будем делать с умножением? Разумно определить результат умножения двух векторов как их скалярное произведение. Однако здесь возникает небольшая проблема. Нужно использовать операции сложения и умножения, скорее всего, из арифметики для координат. Эту проблему решить несложно. Добавьте в свою векторную арифметику умножение двух векторов как их скалярное произведение. Покажите, что скалярное произведение работает.

d. Добавьте к векторной арифметике операцию взятия модуля, расширив численную операцию `magnitude` так, чтобы для вектора она возвращала длину. Приведенный выше код почти все уже делает!

e. Умножение вектора на скаляр и умножение скаляра на вектор должны давать в результате вектор, у которого каждая координата умножена на скаляр. Таким образом, знак умножения может означать либо скалярное произведение, либо произведение вектора на скаляр, в зависимости от типов аргументов. Сделайте так, чтобы ваша арифметика с этим справлялась. Покажите, что оба вида умножения работают. Подсказка: процедура `operation-union` со стр. 80 позволяет изящно решить эту задачу.

Упражнение 3.3. Порядок расширений

Рассмотрим два возможных порядка сочетания векторного расширения (упражнение 3.2) с существующей арифметикой:

```
(define vec-before-func
  (extend-arithmetic
   function-extender
   (extend-arithmetic vector-extender combined-arithmetic)))

(define func-before-vec
  (extend-arithmetic
   vector-extender
   (extend-arithmetic function-extender combined-arithmetic)))
```

Как влияет порядок расширений на свойства полученной арифметики? Вот процедура, выдающая точки единичной окружности:

```
(define (unit-circle x)
  (vector (sin x) (cos x)))
```

При выполнении каждого из двух следующих выражений в окружении, полученном при загрузке либо `vec-before-func`, либо `func-before-vec`

```
((magnitude unit-circle) 'a)

((magnitude (vector sin cos)) 'a)
```

должно получаться (без упрощений)

```
(sqrt (+ (* (sin a) (sin a)) (* (cos a) (cos a))))
```

Однако при каждом из двух порядков расширения одно из двух выражений ломается. Можно ли создать арифметику, которая правильно обрабатывает оба выражения? Объясните.

3.2 Расширяемые полиморфные процедуры

Системы, построенные на комбинаторах, как в разделе 3.1, получаются красивыми, как драгоценные камни. Иногда именно это и нужно, и мы еще увидим тому при-

меры, но к драгоценному камню очень трудно что-то добавить. Если же система строится как комок грязи, то добавить в нее еще грязи совсем несложно[11].

Один из возможных способов организовать себе комок грязи — построить систему на основе расширяемых полиморфных процедур. Современные динамически типизированные языки программирования вроде Lisp, Scheme или Python обычно обладают встроенной арифметикой, полиморфной относительно множества числовых типов, таких как целые, числа с плавающей точкой, рациональные и комплексные числа [115, 64, 105]. Однако обычно системы на основе этих языков после своего написания легко не расширяются.

Проблемы, описанные нами в разделе 3.1.5, проистекают из использования комбинатора `add-arithmetics`. Чтобы с ними справиться, мы откажемся от этого комбинатора. Однако понятие арифметического пакета и идея расширения по-прежнему остаются полезными. Мы построим арифметический пакет, где операции используют полиморфные процедуры, которым можно добавлять новое поведение динамически. Тогда мы сможем добавлять расширения к полиморфной арифметике[12].

Сначала мы реализуем механизм полиморфных процедур. Полиморфная процедура — это процедура, которую можно динамически расширять добавлением обработчиков после того, как она определена. Полиморфная процедура представляет собой диспетчер в сочетании со списком *правил*, каждое из которых описывает обработчик, пригодный для определенного набора аргументов. Такое правило связывает обработчик с условиями его применимости.

Чтобы рассмотреть, как такое устройство может работать, определим полиморфную процедуру с именем `plus`, работающую как сложение для числовых и символических данных:

```
(define plus (simple-generic-procedure 'plus 2 #f))

(define-generic-procedure-handler plus
  (all-args 2 number?)
  (lambda (a b) (+ a b)))

(define-generic-procedure-handler plus
  (any-arg 2 symbolic? number?)
  (lambda (a b) (list '+ a b)))
```

[11] Утверждается, что Джоэл Мозес сказал на конференции APL-79: «Язык APL подобен прекрасному драгоценному камню — он безупречен, изящен и симметричен. Однако к нему ничего нельзя добавить. Если приклеить еще один алмаз, то камня большей величины вы не получите. Lisp подобен комку грязи. Добавьте еще, и у вас по-прежнему будет комок грязи, и он будет выглядеть как Lisp». Однако сам Джоэл отрицает, что когда-либо говорил такое.

[12] Возможности такого рода подразумеваются в большинстве «объектно ориентированных языков» но там они обычно тесно привязаны к онтологическим механизмам вроде наследования. Базовая идея расширяемых полиморфных процедур присутствует в SICP [1]. Ее практические реализации имеются в системах `tinyCLOS` [66] и `SOS` [52].

Основанная на управлении предикатами система расширяемых полиморфных процедур используется в реализации системы математических представлений в книге SICM [121]. Отличное описание управления предикатами дает Эрнст [33].

Мысль, что полиморфные процедуры могут быть мощным инструментом, вращалась в сообществе Lisp десятилетиями. Наиболее полное воплощение она нашла в Объектной системе Common Lisp (CLOS) [42]. Структура, на которой эта система построена, изящно выражается Метаобъектным протоколом [68]. Дальнейшее развитие она нашла в движении «аспектно-ориентированного программирования» [67].

```
(plus 1 2)
3

(plus 1 'a)
(+ 1 a)

(plus 'a 2)
(+ a 2)

(plus 'a 'b)
(plus a b)
```

Процедура `simple-generic-procedure` принимает три аргумента. Первый из них — произвольное имя, помечающее процедуру при отладке; второй — арность процедуры. Третий аргумент задает обработчик по умолчанию. Если такого обработчика нет (что обозначается как `#f`), то в случае, когда никакой конкретный обработчик не подходит, выдается сообщение об ошибке. В нашем примере новая полиморфная процедура, возвращаемая при вызове `simple-generic-procedure`, присваивается переменной `plus`. Это процедура языка Scheme, которую можно позвать с заданным количеством аргументов.

Процедура `define-generic-procedure-handler` добавляет к существующей полиморфной процедуре правило. Первый ее аргумент — полиморфная процедура, подлежащая расширению; второй аргумент — спецификация применимости (как на стр. 76) для добавляемого правила; третий — обработчик для аргументов, которые удовлетворяют спецификации.

```
(define-generic-procedure-handler полиморфная-процедура
                                  применимость
                                  процедура-обработчик)
```

Часто требуется определить правило, где у разных аргументов разные типы. Например, когда мы строим пакет векторной арифметики, нужно указать интерпретацию операции `*`. Если оба аргумента векторы, обработчик должен вычислить скалярное произведение. Если один из аргументов скаляр, а другой — вектор, то обработчик домножает элементы вектора на скаляр. Аргумент-применимость осуществляет этот выбор.

Конструктор `simple-generic-procedure`, использованный нами выше для создания полиморфной процедуры `plus`, создается процедурой `generic-procedure-constructor`:

```
(define simple-generic-procedure
  (generic-procedure-constructor make-simple-dispatch-store))
```

где `make-simple-dispatch-store` — процедура, выражающая стратегию создания, поиска и выбора обработчиков.

Процедура `generic-procedure-constructor` берет конструктор хранилищ обработчиков и выдает конструктор полиморфных процедур, который сам принимает три аргумента — имя для отладки, арность и умолчательный обработчик для случая, когда ни один другой не подходит. Если аргумент-умолчательный обработчик равен `#f`, умолчательный обработчик сообщает об ошибке:

```
((generic-procedure-constructor конструктор-хранилища-обработчиков)
 имя
 арность
 обработчик-по-умолчанию)
```

Полиморфные процедуры создаются именно так потому, что нам потребуются семейства полиморфных процедур, где хранилища обработчиков устроены по-разному.

В разделе 3.2.3 мы увидим, как этот механизм можно реализовать. Однако сначала давайте посмотрим, как он используется.

3.2.1 Полиморфная арифметика

С помощью нового механизма полиморфных процедур мы можем строить арифметические пакеты, где знаки операций отображаются на операции, реализованные в виде полиморфных процедур. Это позволит нам строить структуры со ссылками на себя. Например, можно сделать полиморфную арифметику, включающую в себя арифметику для векторов, где и векторы, и их компоненты обрабатываются одними и теми же полиморфными процедурами. С помощью одного только механизма `add-arithmetics`, введенного нами ранее, мы такую структуру построить бы не смогли.

```
(define (make-generic-arithmetic dispatch-store-maker)
  (make-arithmetic 'generic any-object? '()
    constant-union
    (let ((make-generic-procedure
           (generic-procedure-constructor
            dispatch-store-maker)))
      (lambda (operator)
        (simple-operation operator
                          any-object?
                          (make-generic-procedure
                           operator
                           (operator-arity operator)
                           #f))))))
```

Процедура `make-generic-arithmetic` создает новую арифметику. Для каждого из знаков арифметических операций она создает операцию, реализованную полиморфной процедурой и применимую к любым аргументам. (Предикат `any-object?` выполняется для чего угодно.) Эту арифметику можно установить обычным образом.

Но сначала определим несколько обработчиков в наших полиморфных процедурах. Теперь, когда у нас есть объект-полиморфная арифметика, это совсем несложно. Например, можно вынуть операции и константы из любой уже существующей арифметики.

```
(define (add-to-generic-arithmetic! generic-arithmetic
                                    arithmetic)
  (add-generic-arith-constants! generic-arithmetic
                                arithmetic)
  (add-generic-arith-operations! generic-arithmetic
                                 arithmetic))
```

Этот код берет полиморфный арифметический пакет и обыкновенный арифметический пакет с теми же операциями. Он вливает константы в полиморфную арифметику через `constant-union`. Для каждого знака операции входной арифметики он добавляет в соответствующую полиморфную процедуру по обработчику.

При добавлении обработчика для каждого конкретного знака операции используется стандартный механизм полиморфных процедур. Применимость и процедура-обработчик извлекаются из операции исходной арифметики.

```
(define (add-generic-arith-operations! generic-arithmetic
                                       arithmetic)
  (for-each
   (lambda (operator)
     (let ((generic-procedure
            (simple-operation-procedure
             (arithmetic-operation operator
                                   generic-arithmetic)))
           (operation
            (arithmetic-operation operator arithmetic)))
       (define-generic-procedure-handler
           generic-procedure
           (operation-applicability operation)
           (operation-procedure operation))))
   (arithmetic-operators arithmetic)))
```

Процедура `add-generic-arith-operations!` для каждого знака операции поданной на вход арифметики находит полиморфную процедуру, подлежащую расширению. Затем для этой полиморфной процедуры определяется обработчик, который вызывает обработчик этой операции во входной арифметике, с применимостью этого обработчика из входной арифметики.

Код для добавления констант входной арифметики в полиморфную устроен подобным же образом. Для каждого имени константы полиморфной арифметики он находит запись о связывании этого имени со значениями констант полиморфной арифметики. Затем имя константы заменяется на результат процедуры `constant-union` от существующей константы и константы с тем же именем из входной арифметики.

```
(define (add-generic-arith-constants! generic-arithmetic
                                      arithmetic)
  (for-each
   (lambda (name)
     (let ((binding
            (arithmetic-constant-binding name
                                         generic-arithmetic))
           (element
            (find-arithmetic-constant name arithmetic)))
       (set-cdr! binding
                 (constant-union name
                                 (cdr binding)
                                 element))))
   (arithmetic-constant-names generic-arithmetic)))
```

Игры с полиморфной арифметикой

Чтобы у полиморфной арифметики было интересное поведение, добавим в нее несколько арифметических пакетов:

```
(let ((g
       (make-generic-arithmetic make-simple-dispatch-store)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (add-to-generic-arithmetic! g
    (function-extender numeric-arithmetic))
  (add-to-generic-arithmetic! g
    (symbolic-extender numeric-arithmetic))
  (install-arithmetic! g))
```

Получается полиморфная арифметика, содержащая численную арифметику, символьную арифметику поверх численной и арифметику функций поверх численной.

```
(+ 1 3 'a 'b)
((+ (+ 4 a) b)
```

Можно даже запускать более сложные вычисления, как на стр. 81:

```
(pp (x 0 ((evolver F 'h stormer-2) numeric-s0 1)))
(+ 9.999833334166664e-3
   (* (/ (expt h 2) 12)
      -9.999750002487318e-7))
```

Как и раньше, можно смешивать символы с функциями:

```
(* 'b ((+ cos sin) 3))
(* b -.8488724885405782)
```

но следующий пример, где мы пытаемся сложить символьные объекты `(cos a)` и `(sin a)`, приводит к ошибке:

```
(+ 'b ((+ cos sin) 'a))
```

Ошибка получается оттого, что `cos` и `sin` — знаки численных операций подобно `+`. Поскольку у нас есть символьная арифметика поверх численной, эти операции расширяются так, чтобы для символьного входа, в данном случае `a`, выдавать символьный ответ `(cos a)` и `(sin a)`. У нас есть также арифметика функций, так что, если функции сочетаются численно (здесь через `+`), их результаты могут комбинироваться, только если они числа. Но символьные результаты нельзя численно сложить. Это следует из того, как мы построили арифметику `g`.

Но в полиморфной арифметике присутствует магия. Ее можно замкнуть: все расширения полиморфной арифметики можно делать поверх нее самой!

```
(let ((g
       (make-generic-arithmetic make-simple-dispatch-store)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (extend-generic-arithmetic! g symbolic-extender)
  (extend-generic-arithmetic! g function-extender)
  (install-arithmetic! g))
```

Здесь мы пользуемся новой процедурой `extend-generic-arithmetic!`, воплощающей общий шаблон.

```
(define (extend-generic-arithmetic! generic-arithmetic
                                    extender)
  (add-to-generic-arithmetic! generic-arithmetic
      (extender generic-arithmetic)))
```

Теперь у нас работают сложные смешанные выражения, поскольку функции определены над полиморфной арифметикой:

```
(* 'b ((+ 'c cos sin) (+ 3 'a)))
(* b (+ (+ c (cos (+ 3 a))) (sin (+ 3 a))))
```

Можно даже использовать функции, возвращающие функции:

```
(((+ (lambda (x) (lambda (y) (cons x y)))
     (lambda (x) (lambda (y) (cons y x))))
  3)
 4)
(+ (3 . 4) (4 . 3))
```

Может быть, мы достигли нирваны?

3.2.2 Структура зависит от порядка!

К сожалению, остается неприятная зависимость от порядка добавления правил к полиморфным процедурам. Это неудивительно, поскольку система полиморфных процедур строится через присваивание. Можно это увидеть, поменяв порядок построения:

```
(let ((g
       (make-generic-arithmetic make-simple-dispatch-store)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (extend-generic-arithmetic! g function-extender)   ;*
  (extend-generic-arithmetic! g symbolic-extender)   ;*
  (install-arithmetic! g))
```

И тогда окажется, что пример

```
(* 'b ((+ 'c cos sin) (+ 3 'a)))
```

который с предыдущей арифметикой работал, теперь ломается, потому что символьная арифметика захватывает `(+ 'c cos sin)` и порождает символьное выражение, а оно не является функцией, которую можно применить к `(+ 3 a)`. Проблема в том, что применимость символьной арифметики для `+` принимает аргументы, где хотя бы один элемент символьный, а остальные подпадают под предикат области действия базы. Но символьная арифметика была создана на базе полиморфной, а предикат области действия полиморфной арифметики принимает что угодно! Есть еще операция для `+` над функциями, применимая к тем же самым аргументам, но из-за произвольно установленного порядка расширений выбрана не она. К сожалению, выбор правила здесь неоднозначен. Лучше было бы, если бы применима была только одна операция.

Один из способов справиться с этой неприятностью — ограничить значение символьных величин числами. Можно добиться этого, строя символьную арифметику поверх численной, как на стр. 92, а не поверх всей полиморфной арифметики:

```
(let ((g
       (make-generic-arithmetic make-simple-dispatch-store)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (extend-generic-arithmetic! g function-extender)
  (add-to-generic-arithmetic! g
      (symbolic-extender numeric-arithmetic))
  (install-arithmetic! g))
```

Такой вариант работает независимо от порядка, поскольку в выборе правил нет неоднозначности. Теперь `'c` интерпретируется как константа, которую функциональное расширение преобразует в константную функцию.

```
(* 'b ((+ 'c cos sin) (+ 3 'a)))
(* b (+ (+ c (cos (+ 3 a))) (sin (+ 3 a))))
```

К сожалению, нам могут потребоваться символьные выражения не только поверх чисел. Пока что у нас нет общего решения проблемы. Однако, если действительно нужна функция-литерал с именем `c`, можно, как и раньше, использовать `literal-function`:

```
(* 'b ((+ (literal-function 'c) cos sin) (+ 3 'a)))
(* b (+ (+ (c (+ 3 a)) (cos (+ 3 a))) (sin (+ 3 a))))
```

Это работает независимо от порядка построения полиморфной арифметики.

С таким механизмом мы способны вычислить интегратор Стёрмера от функции-литерала:

```
(pp (x 0 ((evolver (literal-function 'F) 'h stormer-2)
          (make-initial-history 't 'h 'xt 'xt-h 'xt-2h)
          1))
(+ (+ (* 2 xt) (* -1 xt-h))
   (* (/ (expt h 2) 12)
      (+ (+ (* 13 (f t xt))
            (* -2 (f (- t h) xt-h)))
         (f (- t (* 2 h)) xt-2h))))
```

Получается довольно уродливо, а если мы посмотрим на вывод двух шагов интегрирования, станет еще хуже. Однако интересно посмотреть, на результат упрощения двух шагов интегрирования. После применения волшебного упростителя символьных выражений получается вполне читаемый результат. Он может быть полезен при отладке численного процесса.

```
(+ (* 2 (expt h 2) (f t xt))
   (* -1/4 (expt h 2) (f (+ (* -1 h) t) xt-h))
   (* 1/6 (expt h 2) (f (+ (* -2 h) t) xt-2h))
   (* 13/12
      (expt h 2)
      (f (+ h t)
```

```
          (+ (* 13/12 (expt h 2) (f t xt))
             (* -1/6 (expt h 2) (f (+ (* -1 h) t) xt-h))
             (* 1/12 (expt h 2) (f (+ (* -2 h) t) xt-2h))
             (* 2 xt)
             (* -1 xt-h))))
   (* 3 xt)
   (* -2 xt-h))
```

Заметим, например, что у нас только четыре различных верхнеуровневых вызова функции ускорения `f`. Второй аргумент четвертого верхнеуровневого вызова использует три уже вычисленных вызова `f`. Удалив общие подвыражения, получаем:

```
(let* ((G84 (expt h 2)) (G85 (f t xt)) (G87 (* -1 h))
       (G88 (+ G87 t)) (G89 (f G88 xt-h)) (G91 (* -2 h))
       (G92 (+ G91 t)) (G93 (f G92 xt-2h)))
  (+ (* 2 G84 G85)
     (* -1/4 G84 G89)
     (* 1/6 G84 G93)
     (* 13/12 G84
        (f (+ h t)
           (+ (* 13/12 G84 G85)
              (* -1/6 G84 G89)
              (* 1/12 G84 G93)
              (* 2 xt)
              (* -1 xt-h))))
     (* 3 xt)
     (* -2 xt-h)))
```

Здесь ясно видно, что есть только четыре различных вызова функции `f`. Несмотря на то что каждый шаг базового интегратора вызывает `f` трижды, два шага перекрываются по двум промежуточным вызовам. Хотя в таком простом примере это очевидно и так, мы видим, как символьное вычисление может помочь понять численный процесс.

3.2.3 Реализация полиморфных процедур

При помощи полиморфных процедур мы добились удивительной гибкости. Но как заставить их работать?

Построение конструкторов полиморфных процедур

На стр. 89 мы сделали конструктор простых полиморфных процедур:

```
(define simple-generic-procedure
  (generic-procedure-constructor make-simple-dispatch-store))
```

Процедура `generic-procedure-constructor` принимает «стратегию диспетчеризации» в виде процедуры и возвращает конструктор полиморфных процедур, принимающий имя, арность и спецификацию умолчательного обработчика. Когда этот конструктор вызывается с этими тремя аргументами, он возвращает полиморфную процедуру и связывает ее с новым хранилищем метаданных для этой процедуры.

Хранилище метаданных содержит имя, арность, экземпляр стратегии диспетчеризации и, возможно, умолчательный обработчик. Экземпляр стратегии диспетчеризации будет поддерживать информацию об обработчиках, их применимости и содержит механизм, решающий, какой обработчик позвать при конкретных аргументах полиморфной процедуры.

Код, реализующий `generic-procedure-constructor`, выглядит так:

```
(define (generic-procedure-constructor dispatch-store-maker)
  (lambda (name arity default-handler)
    (let ((metadata
           (make-generic-metadata
             name arity (dispatch-store-maker)
             (or default-handler
                 (error-generic-procedure-handler name)))))
    (define (the-generic-procedure . args)
      (generic-procedure-dispatch metadata args))
    (set-generic-procedure-metadata! the-generic-procedure
                                     metadata)
    the-generic-procedure)))
```

Эта реализация использует `the-generic-procedure`, обыкновенную процедуру языка Scheme, для представления полиморфной процедуры и хранилище метаданных (правил и т. п.), определяющее ее поведение. Хранилище привязывается к процедуре через «ярлык» (как на стр. 39). Его потом можно будет получить через вызов `generic-procedure-metadata`. Это позволяет процедурам вроде `define-generic-procedure-handler` изменять метаданные конкретной полиморфной процедуры.

В качестве аргумента в `generic-procedure-constructor` передается процедура, создающая диспетчерское хранилище для запоминания и поиска обработчиков. Это хранилище воплощает стратегию выбора обработчика.

Вот простой конструктор диспетчерских хранилищ, который мы до сих пор использовали. Хранилище реализовано как процедура, принимающая сообщения:

```
(define (make-simple-dispatch-store)
  (let ((rules '()) (default-handler #f))
    (define (get-handler args)
      ;; тело будет показано ниже в тексте.
      ...)
    (define (add-handler! applicability handler)
      ;; тело будет показано ниже в тексте.
      ...)
    (define (get-default-handler) default-handler)
    (define (set-default-handler! handler)
      (set! default-handler handler))
    (lambda (message)
      ; простое диспетчерское хранилище
      (case message
        ((get-handler) get-handler)
        ((add-handler!) add-handler!)
        ((get-default-handler) get-default-handler)
        ((set-default-handler!) set-default-handler!)
```

```
        ((get-rules) (lambda () rules))
        (else (error "Неизвестное сообщение:" message))))))
```

Это диспетчерское хранилище содержит просто список правил, каждое из которых связывает применимость и обработчик. Когда внутренняя процедура `get-handler` вызывается с аргументами полиморфной процедуры, она последовательно проходит по этому списку, находит обработчик с применимостью, подходящей к данным аргументам, и возвращает его. Если такого нет, она возвращает `#f`.

```
(define (get-handler args)
  (let ((rule
         (find (lambda (rule)
                 (predicates-match? (car rule) args))
               rules)))
    (and rule (cdr rule))))
```

Есть много возможных стратегий выбора нужного обработчика. Код выше возвращает первый применимый обработчик в списке. Другой стратегией будет вернуть все применимые обработчики. Если таких больше одного, возможно, имеет смысл запустить их все (параллельно?) и сравнить результаты! То, что мы передаем конструктор диспетчерского хранилища аргументом в `generic-procedure-constructor`, позволяет выбирать стратегию при создании конструктора полиморфных процедур, а не задавать ее в реализации жестко.

Добавление обработчиков к полиморфным процедурам

Процедура определения обработчика (см. ниже) добавляет новые правила, вызывая внутреннюю процедуру диспетчерского хранилища `add-handler`. В случае `make-simple-dispatch-store`, приведенного выше, `add-handler` добавляет новое правило в начало списка правил. (Однако, если уже было правило для этой применимости, она просто заменяет обработчик.)

```
(define (add-handler! applicability handler)
  (for-each (lambda (predicates)
              (let ((p (assoc predicates rules)))
                (if p
                    (set-cdr! p handler)
                    (set! rules
                          (cons (cons predicates handler)
                                rules)))))
            applicability))
```

Процедура `define-generic-procedure-handler` получает метаданные для данной полиморфной процедуры из таблицы метаданных. Она запрашивает в диспетчерском хранилище процедуру `add-handler!` и с помощью этой процедуры добавляет в метаданные правило, связывающее применимость с обработчиком. Экземпляр диспетчерского хранилища достается из метаданных полиморфной процедуры вызовом `generic-metadata-dispatch-store`.

```
(define (define-generic-procedure-handler generic-procedure
                                          applicability
```

```
                                        handler)
  (((generic-metadata-dispatch-store
     (generic-procedure-metadata generic-procedure))
    'add-handler!)
   applicability
   handler))
```

Наконец, сердцем всего механизма является диспетчер, запускаемый из полиморфной процедуры (`the-generic-procedure` со стр. 96), который находит подходящий обработчик и вызывает его. Если ни одного применимого обработчика нет, зовется умолчательный обработчик, заданный при создании полиморфной процедуры[13].

```
(define (generic-procedure-dispatch metadata args)
  (let ((handler
         (get-generic-procedure-handler metadata args)))
    (apply handler args)))

(define (get-generic-procedure-handler metadata args)
  (or ((generic-metadata-getter metadata) args)
      ((generic-metadata-default-getter metadata))))
```

Сила расширяемых полиморфных процедур

Построение системы на основе расширяемых полиморфных процедур — мощная идея. В нашем примере оказывается возможным определить, что означает сложение, умножение и т. д. для новых типов, которые создатель языка не имел в виду. Например, если арифметические операции реализованы в системе как полиморфные процедуры, пользователь может расширить их и распространить на кватернионы, векторы, матрицы, целые по модулю простого числа, функции, тензоры, дифференциальные формы... Это не просто добавляет новые возможности; при этом расширяются старые программы. Таким образом, программа, написанная для обработки простых числовых значений, может оказаться полезной при работе с функциями, возвращающими скаляры.

Мы видели, что при использовании расширимых полиморфных процедур возможны проблемы. С другой стороны, некоторые «мутации» будут чрезвычайно полезными. Например, арифметику можно расширить для работы с символьными величинами. Самый простой способ сделать это — добавить ко всем операциям полиморфное расширение, принимающее в качестве аргументов символьные величины и возвращающее структуру данных, представляющую применение данной операции к аргументам. Если добавить процедуру упрощения арифметических выражений, в наших руках внезапно окажется программа для символьных манипуляций. Это полезно и при отладке численных вычислений, поскольку, подав им на вход символьные аргументы, можно исследовать получившиеся символьные выражения и убедиться, что программа действительно вычисляет то, что должна. Кроме того,

[13] Процедуры `generic-metadata-getter` и `generic-metadata-default-getter` извлекают из экземпляра диспетчерского хранилища, хранимого в метаданных полиморфной процедуры, процедуры `get-handler` и `get-default-handler`.

символьный манипулятор может служить основой частичного вычислителя для оптимизации численных программ. Функциональное дифференцирование можно рассматривать как полиморфное расширение арифметики на составной тип данных (см. раздел 3.3). Система `scmutils`, при помощи которой мы преподаем классическую механику, именно так реализует взятие производной.

Упражнение 3.4. Функциональные значения

Структура полиморфной арифметики позволяет нам замкнуть систему, так что заработают функции, возвращающие функции, как в следующем примере:

```
(((* 3
     (lambda (x) (lambda (y) (+ x y)))
     (lambda (x) (lambda (y) (vector y x))))
  'a)
 4)
(* (* 3 (+ a 4)) #(4 a))
```

a. Насколько сложно заставить это работать в арифметике, основанной на чистых комбинаторах, как в разделе 3.1? Почему?

b. В упражнении 3.3 мы спрашивали, как на работу системы влияет порядок между векторным и функциональным расширениями. Способна ли полиморфная система поддержать оба обсуждаемых там выражения (они повторены ниже)? Объясните.

```
((magnitude unit-circle) 'a)
((magnitude (vector sin cos)) 'a)
```

c. Существует ли вообще какой-нибудь способ заставить работать вот такое?

```
((vector cos sin) 3)
#(-.9899924966004454 .1411200080598672)
```

Покажите код, реализующий это, или объясните, какие здесь сложности.

Упражнение 3.5. Странная ошибка

Рассмотрим приведенную ниже процедуру `+-like` («плюсоподобная») из файла `arith.scm`, которая как шаг в процессе установки арифметики реализует n-местные процедуры `+` и `*`. Она возвращает пару, состоящую из имени и процедуры; программа установки привяжет имя к процедуре.

Кажется, что она должна звать процедуру `get-identity`, вычисляющую процедуру тождественности, каждый раз, когда операция вызывается с нулем аргументов.

```
(define (+-like operator identity-name)
  (lambda (arithmetic)
    (let ((binary-operation
           (find-arithmetic-operation operator arithmetic)))
      (and binary-operation
           (let ((binary
                  (operation-procedure binary-operation))
                 (get-identity
                  (identity-name->getter identity-name
                                         arithmetic)))
             (cons operator
                   (lambda args
                     (case (length args)
                       ((0) (get-identity))
```

```
                    ((1) (car args))
                    (else (pairwise binary args)))))))))))
```

Возможно, стоит вызывать процедуру тождественности для операции только один раз, а не при каждом вызове обработчика. Таким образом, предлагается изменить код так:

```
(define (+-like operator identity-name)
  (lambda (arithmetic)
    (let ((binary-operation
           (find-arithmetic-operation operator arithmetic)))
      (and binary-operation
           (let ((binary
                  (operation-procedure binary-operation))
                 (identity
                  ((identity-name->getter identity-name
                                          arithmetic))))
             (cons operator
                   (lambda args
                     (case (length args)
                       ((0) identity)
                       ((1) (car args))
                       (else (pairwise binary args)))))))))))
```

Однако здесь есть нетривиальная ошибка! Сможете ли вы ее найти и объяснить?

Упражнение 3.6. Матрицы

В научных и технических вычислениях матрицы встречаются на каждом шагу.

a. Создайте и установите арифметический пакет для матриц чисел с операциями `+`, `-`, `negate` и `*`. Арифметика должна будет знать, сколько в матрице строк и столбцов, поскольку умножение матриц определено только тогда, когда число столбцов первой матрицы равно числу строк второй.

   Проследите, чтобы ваш умножитель матриц мог домножить матрицу на скаляр или вектор. Чтобы матрицы хорошо взаимодействовали с векторами, вероятно, вам потребуется различать векторы-строки и векторы-столбцы. Как это влияет на проектирование векторного пакета (см. упр. 3.2 на стр. 86)?

   Можете предполагать, что размерность матриц и векторов небольшая, так что работать с разреженными представлениями не надо. Разумным представлением матрицы будет вектор языка Scheme, каждый элемент которого тоже вектор, представляющий строку.

b. Векторы и матрицы могут содержать символьные численные значения. Заставьте это работать.

c. В этой арифметике имеет смысл обращение матриц. Если символьная матрица плотная, при взятии обратного значения может потребоваться память пропорционально факториалу размерности матрицы. Почему?

   Пояснение. Мы не просим вас реализовывать обращение матриц.

Упражнение 3.7. Векторные и матричные литералы

Можно также построить арифметику на векторных и матричных литералах с алгеброй символьных выражений векторов и матриц. Можете ли вы сделать так, чтобы символьная алгебра таких структур правильно работала с векторами и матрицами, где элементами служат символьные численные выражения? Предупреждение: это сложная задача. Возможно, она должна быть частью долговременного проекта.

3.3 Пример: автоматическое дифференцирование

Одно из замечательных применений расширяемых полиморфных процедур — автоматическое дифференцирование[14]. Это красивый способ получить программу, которая считает производную функции, вычисляемой другой программой[15]. В наше время автоматическое дифференцирование является важной составляющей в приложениях машинного обучения.

Как мы увидим, простой способ реализовать автоматическое дифференцирование — расширить полиморфные арифметические операции, чтобы они могли работать с новым составным типом данных, *объектами-дифференциалами*. Таким образом, мы получим автоматическое дифференцирование и символьных, и численных функций. Это также позволит нам производить автоматическое дифференцирование функций высшего порядка — процедур, возвращающих в качестве значения другие процедуры.

Вот простой пример автоматического дифференцирования, показывающий, о чем мы говорим:

```
((derivative (lambda (x) (expt x 3))) 2)
12
```

Заметим, что производная функции, вычисляющей куб аргумента, — это новая функция, которая, будучи применена к аргументу 2, возвращает значение 12.

Если мы расширим арифметику на символьные выражения и применим немного алгебраических упрощений к результату, получим

```
((derivative (lambda (x) (expt x 3))) 'a)
(* 3 (expt a 2))
```

Доступна вся мощь нашего языка программирования, включая процедуры высшего порядка. Системы такого рода полезны при работе с очень большими выражениями, которые встречаются в интересных физических задачах[16].

Рассмотрим простое приложение — вычисление корней уравнения методом Ньютона. Требуется найти значения x, для которых $f(x) = 0$. Если функция f достаточно гладкая и у нас есть достаточно хорошее начальное приближение x_0, мы можем его улучшить, вычисляя новое значение x_1 по формуле:

$$x_{n+1} = x_n - \frac{f(x_n)}{Df(x_n)}.$$

Эту операцию можно повторять по необходимости, пока не получится достаточно точная оценка. Элементарная программа, выполняющая это, такова:

[14]Термин *автоматическое дифференцирование* ввел в 1964 г. Венгерт [129].

[15]Под производной здесь понимается производная функции, а не выражения. Если есть функция f, то ее производная Df — это новая функция, которая при применении к значению x возвращает $Df(x)$. С производной выражения она соотносится как

$$Df(t) = \left.\frac{d}{dx} f(x)\right|_{x=t}$$

[16]Программа автоматического дифференцирования, которую мы здесь представляем, создана на основе кода, написанного нами в рамках поддержки курса классической механики для старшекурсников, который Сассман читает в MIT совместно с Джеком Уиздомом [121, 122].

```
(define (root-newton f initial-guess tolerance)
  (let ((Df (derivative f)))
    (define (improve-guess xn)
      (- xn (/ (f xn) (Df xn))))
    (let loop ((xn initial-guess))
      (let ((xn+1 (improve-guess xn)))
        (if (close-enuf? xn xn+1 tolerance)
            xn+1
            (loop xn+1))))))
```

Заметим, что локальная процедура под именем `Df` в `root-newton` — это процедура, вычисляющая производную функции, которую считает процедура, переданная в качестве параметра `f`.

Допустим, например, что мы хотим знать угол θ в первом квадранте, для которого $\cos(\theta) = \sin(\theta)$. (Ответом будет $\pi/4 \approx .7853981633974484$.) Можно написать:

```
(define (cs theta)
  (- (cos theta) (sin theta)))

(root-newton cs 0.5 1e-8)
.7853981633974484
```

Результат верен с полной точностью машинного представления.

3.3.1 Как работает автоматическое дифференцирование

Программа автоматического дифференцирования прямо основана на математическом определении производной. Допустим, что, имея функцию f и точку x в ее области определения, мы хотим знать значение функции в близлежащей точке $f(x+\Delta x)$, где Δx — небольшое приращение. Производная функции f определяется как функция Df, чье значение в конкретной точке x можно «умножить» на приращение аргумента Δx и получить наилучшее линейное приближение приращения значения функции f:

$$f(x + \Delta x) \approx f(x) + Df(x)\Delta x.$$

Мы реализуем это определение через тип данных, который мы называем *объект-дифференциал*. Объект-дифференциал $[x, \delta x]$ можно рассматривать как число с небольшим приращением, $x+\delta x$. Однако мы рассматриваем его как новую числовую величину наподобие комплексного числа: у дифференциала есть две компоненты, *конечная часть* и *бесконечно малая часть*[17]. Каждую элементарную арифметическую функцию мы расширяем для работы с объектами-дифференциалами; каждая элементарная арифметическая функция f должна знать свою производную Df, так что

$$[x, \delta x] \xmapsto{f} [f(x), Df(x)\delta x]. \tag{3.5}$$

[17] Объекты-дифференциалы такого вида иногда называют *двойственными числами*. Двойственные числа, введенные Клиффордом в 1873 г. [20], — это расширение вещественных чисел, к которым присоединяется новый элемент ϵ, имеющий свойство $\epsilon^2 = 0$. Однако для того, чтобы удобно было вычислять производные высшего порядка (а также производные функций нескольких аргументов), лучше ввести бесконечно малую часть у каждой независимой переменной. Таким образом, наша алгебра дифференциалов намного сложнее, чем пространство двойственных чисел с единственным ϵ. Наши объекты-дифференциалы похожи также на гипервещественные числа, изобретенные Эдвином Хьюиттом в 1948 г. [59]

Заметим, что производная функции f в точке x, $Df(x)$, служит коэффициентом при δx в бесконечно малой части получающегося объекта-дифференциала.

Теперь важная идея: если теперь мы передадим значение $f([x, \delta x])$ (уравнение 3.5) на вход другой функции g, получится результат по правилу производной сложной функции, как нам и хотелось:

$$[f(x), Df(x)\delta x] \stackrel{g}{\longmapsto} [g(f(x)), Dg(f(x))Df(x)\delta x].$$

Таким образом, если мы умеем вычислять все элементарные функции на объектах-дифференциалах, то мы умеем вычислять на дифференциалах и все их сочетания. Имея этот результат, можем извлечь производную сочетания: производная равна коэффициенту при бесконечно малом приращении получившегося дифференциала.

Чтобы расширить полиморфную арифметическую операцию для работы с дифференциалами, нужно только задать процедуру, вычисляющую производную элементарной арифметической функции, которую обозначает этот знак операции. После этого мы сможем использовать обычные средства композиции языка Scheme и получать производные любого сочетания арифметических функций[18].

Имея на входе процедуру, реализующую одноместную функцию `f`, процедура `derivative` создает новую процедуру `the-derivative`, которая считает производную функции, вычисляемой `f`[19]. Будучи применена к некоторому аргументу `x`, функция-производная создает бесконечно малое приращение `dx` и добавляет его к аргументу, получая новый объект-дифференциал $[x, \delta x]$, который представляет $x + \delta x$. Затем процедура f применяется к этому дифференциалу, и производная f извлекается как коэффициент при бесконечно малом приращении dx получившегося значения:

```
(define (derivative f)
  (define (the-derivative x)
    (let* ((dx (make-new-dx))
           (value (f (d:+ x (make-infinitesimal dx)))))
      (extract-dx-part value dx)))
  the-derivative)
```

Процедура `make-infinitesimal` создает объект-дифференциал с нулевой конечной частью и бесконечно малой частью `dx`. Процедура `d:+` складывает дифференциалы. Детали мы разъясним в разделе 3.3.3.

Расширение элементарных операций

Нужно написать процедуры-обработчики, расширяющие полиморфные арифметические процедуры для работы с дифференциалами. В каждой одноместной процедуре надо создать конечную часть результата, бесконечно малую часть результата и

[18] Эту идею «изобрели» в 1992 г. Дэн Зурас (который тогда работал в корпорации Hewlett Packard) и Джеральд Джей Сассман во время ночного приступа программирования. Мы тогда предположили, что ее уже открыли много раз до того. Так оно и есть [129, 12], но как здорово было придумать ее самим! Формальное введение в автоматическое дифференцирование можно найти в [94].

[19] До двухместных функций мы тоже скоро доберемся. Пока что мы просто хотим разъяснить главную идею, прежде чем заниматься сложными деталями. Расширение системы на n-местные функции обсуждается в разделе 3.3.2.

собрать их вместе, как показано в уравнении 3.5. Обработчик одноместной элементарной арифметической процедуры, вычисляющей функцию f, порождается процедурой `diff:unary-proc`, принимающей процедуру `f` как f и процедуру `df` как ее производную Df. Они склеиваются через особые процедуры сложения и умножения `d:+` и `d:*` для дифференциалов, которые мы разъясним в разделе 3.3.3.

```
(define (diff:unary-proc f df)
  (define (uop x)
    ; x --- объект-дифференциал
    (let ((xf (finite-part x))
          (dx (infinitesimal-part x)))
      (d:+ (f xf) (d:* (df xf) dx))))
  uop)
```

Например, обработчик процедуры `sqrt` для дифференциалов — это просто

```
(define diff:sqrt
  (diff:unary-proc sqrt (lambda (x) (/ 1 (* 2 (sqrt x))))))
```

Первым аргументом в `diff:unary-proc` передается процедура `sqrt`, а вторым — процедура, вычисляющая производную `sqrt`.

Новый обработчик полиморфной процедуры `sqrt` добавляется через

```
(assign-handler! sqrt diff:sqrt differential?)
```

где `differential?` — предикат, истинный только относительно объектов-дифференциалов. Процедура `assign-handler!` — всего лишь краткая запись полезной схемы:

```
(define (assign-handler! procedure handler . preds)
  (define-generic-procedure-handler procedure
    (apply match-args preds)
    handler))
```

Процедура `match-args` создает из последовательности предикатов спецификацию применимости.

Обработчики остальных одноместных элементарных процедур очевидны[20]:

```
(define diff:exp (diff:unary-proc exp exp))

(define diff:log (diff:unary-proc log (lambda (x) (/ 1 x))))

(define diff:sin (diff:unary-proc sin cos))

(define diff:cos
  (diff:unary-proc cos (lambda (x) (* -1 (sin x)))))
```

⋮

Двухместные арифметические операции устроены несколько сложнее:

$$g(x + \Delta x, y + \Delta y) \approx g(x, y) + \partial_0 g(x, y)\Delta x + \partial_1 g(x, y)\Delta y. \tag{3.6}$$

[20]Здесь мы приводим определения обработчиков. Мы не показываем, как они устанавливаются в систему.

где $\partial_0 f$ и $\partial_1 f$ — функции частных производных f по двум аргументам. Пусть f будет двухместной функцией; тогда $\partial_0 f$ — это функция двух аргументов, вычисляющая частную производную f по первому аргументу:

$$\partial_0 f(x,y) = \frac{\partial}{\partial u} f(u,v)\bigg|_{u=x,v=y}.$$

Таким образом, правило для двухместных операций будет:

$$([x,\delta x],[y,\delta y]) \xmapsto{f} [f(x,y), \partial_0 f(x,y)\delta x + \partial_1 f(x,y)\delta y].$$

Казалось бы, при реализации двухместных операций можно просто следовать тому же плану, что и для одноместных, где `d0f` и `d1f` — две функции частных производных:

```
(define (diff:binary-proc f d0f d1f)
  (define (bop x y)
    (let ((dx (infinitesimal-part x))
          (dy (infinitesimal-part y))
          (xf (finite-part x))
          (yf (finite-part y)))
      (d:+ (f xf yf)
           (d:+ (d:* dx (d0f xf yf))
                (d:* (d1f xf yf) dy)))))
  bop)
```

Этот план хорош, но не совсем верен: он не обеспечивает выбор совместимых конечных и бесконечно малых частей для двух аргументов. Эту техническую деталь мы объясним и исправим в разделе 3.3.3, а сейчас давайте продолжим работу с этим приблизительно верным кодом.

Сложение и умножение пишутся несложно, поскольку частные производные устроены просто; деление и возведение в степень более интересны. Мы записываем установку обработчиков только для `diff:+`, поскольку с остальными все устроено точно так же.

```
(define diff:+
  (diff:binary-proc +
                    (lambda (x y) 1)
                    (lambda (x y) 1)))

(assign-handler! + diff:+ differential? any-object?)
(assign-handler! + diff:+ any-object? differential?)

(define diff:*
  (diff:binary-proc *
                    (lambda (x y) y)
                    (lambda (x y) x)))

(define diff:/
  (diff:binary-proc /
```

```
                    (lambda (x y)
                      (/ 1 y))
                    (lambda (x y)
                      (* -1 (/ x (square y))))))
```

Обработчик для возведения в степень $f(x, y) = x^y$ несколько сложнее. Частная производная по первому аргументу проста: $\partial_o f(x, y) = yx^{y-1}$. Однако частная производная по второму аргументу обычно равна $\partial_1 f(x, y) = x^y \log x$, за исключением некоторых особых случаев:

```
(define diff:expt
  (diff:binary-proc expt
    (lambda (x y)
      (* y (expt x (- y 1))))
    (lambda (x y)
      (if (and (number? x) (zero? x))
          (if (number? y)
              (if (positive? y)
                  0
                  (error "Производная не определена: EXPT"
                         x y))
              0)
          (* (log x) (expt x y))))))
```

Извлечение значения производной

Чтобы посчитать значение производной функции, мы применяем функцию к объекту-дифференциалу и получаем результат. Нужно извлечь из этого результата значение производной. Требуется рассмотреть несколько случаев. Если результат является дифференциалом, нужно взять значение производной из него. Если же результат не является дифференциалом, значение производной равно нулю. Есть еще несколько случаев, которые мы тут не упомянули. Таким образом, нам нужна полиморфная процедура, чей обработчик по умолчанию выдает ноль.

```
(define (extract-dx-default value dx) 0)

(define extract-dx-part
  (simple-generic-procedure 'extract-dx-part 2
                            extract-dx-default))
```

В случае, когда возвращается объект-дифференциал, коэффициентом при `dx` будет нужная нам производная. Тут возникнут некоторые сложности, но базовую идею можно выразить так:

```
(define (extract-dx-differential value dx)
  (extract-dx-coefficient-from (infinitesimal-part value) dx))

(define-generic-procedure-handler extract-dx-part
  (match-args differential? diff-factor?)
  extract-dx-differential)
```

Это не совсем верно потому, что по техническим причинам структура объекта-дифференциала сложнее, чем мы до сих пор показали. Мы разберемся с этим в разделе 3.3.3.

Заметим, что мы сделали процедуру извлечения производной полиморфной, чтобы позволить дальнейшие расширения на функции, возвращающие другие функции или составные объекты вроде векторов, матриц или тензоров (см. упражнение 3.12 на стр. 118).

С точностью до того, что у нас может быть больше элементарных операций и структур данных, это все, что нужно для реализации автоматического дифференцирования! Все процедуры, упоминаемые в обработчиках, — это обыкновенные полиморфные процедуры арифметики; они могут включать в себя символьную и функциональную арифметику.

3.3.2 Производные n-местных функций

Для функции нескольких аргументов требуется вычислить частные производные по каждому аргументу. Один из способов сделать это таков[21]:

```
(define ((partial i) f)
  (define (the-derivative . args)
    (if (not (< i (length args)))
        (error "Недостаточно аргументов PARTIAL" i f args))
        (let* ((dx (make-new-dx))
               (value
                (apply f (map (lambda (arg j)
                                (if (= i j)
                                    (d:+ arg
                                         (make-infinitesimal dx))
                                    arg))
                              args (iota (length args))))))
          (extract-dx-part value dx)))
  the-derivative)
```

Здесь мы извлекаем коэффициент при бесконечно малом `dx` из результата применения `f` к данным аргументам, где i-й аргумент взят с приращением `dx`[22].

Рассмотрим теперь функцию g от двух аргументов. Развернув уравнение 3.6, мы видим, что производная Dg домножается на вектор приращений аргументов:

$$\begin{aligned} g(x+\Delta x, y+\Delta y) &\approx g(x,y) + \Delta g(x,y)\cdot(\Delta x, \Delta y) \\ &= g(x,y) + [\partial_0 g(x,y), \partial_1 g(x,y)]\cdot(\Delta x, \Delta y) \\ &= g(x,y) + \partial_0 g(x,y)\Delta x + \partial_1 g(x,y)\Delta y \end{aligned}$$

Производная Dg функции g в точке x, y — это пара частных производных в квадратных скобках. Скалярное произведение этого *ковектора* частных производных с *вектором* приращений является приращением функции g. Этот результат вычисляет функция `general-derivative`:

```
(define (general-derivative g)
  (define ((the-derivative . args) . increments)
```

[21] Альтернативную стратегию можно увидеть в упражнении 3.8 на стр. 108.

[22] Процедура `iota` возвращает список последовательных целых чисел от 0 до `(length args)`.

```
    (let ((n (length args)))
      (assert (= n (length increments)))
      (if (= n 1)
          (* ((derivative g) (car args))
             (car increments))
          (reduce (lambda (x y) (+ y x))
                  0
                  (map (lambda (i inc)
                         (* (apply ((partial i) g) args)
                            inc))
                       (iota n)
                       increments)))))
  the-derivative)
```

К сожалению, `general-derivative` не возвращает структуру частных производных. Во многих контекстах полезно иметь процедуру вычисления производной `gradient`, которая выдает ковектор частных производных (см. упражнение 3.10).

Упражнение 3.8. Частные производные

Можно также рассмотреть частные производные через понятие каррирования из λ-исчисления. Нарисуйте диаграмму потока данных. С помощью каррирования закрепите аргументы, которые должны быть постоянными, и получите одноместную процедуру, к которой будет применено обычное вычисление производной. Запишите эту версию процедуры вычисления частной производной.

Упражнение 3.9. Добавление обработчиков

Для некоторых арифметических процедур мы не добавили обработчики объектов-дифференциалов; например, `tan`.

a. Добавьте обработчики для `tan` и `atan1` (`atan1` — функция одного аргумента).

b. Было бы хорошо, если бы `atan` мог опционально принимать два аргумента, как в «Сообщении о языке Scheme» [109], поскольку обычно требуется сохранить квадрант, в котором мы работаем. Исправьте полиморфную процедуру `atan`, чтобы позволить такое поведение; в случае одного аргумента вызывается `atan1`, а в случае двух — `atan2`. Установите также обработчик `atan2` для дифференциалов. Помните, что он должен сосуществовать с обработчиком `atan1`.

Упражнение 3.10. Векторы и ковекторы

Как описано выше, можно обобщить идею производной на функции с несколькими аргументами. *Градиент* многоместной функции — это ковектор частных производных по каждому аргументу.

a. Разработайте типы данных для векторов и ковекторов, чтобы значением $Dg(x, y)$ был ковектор частных производных. Напишите процедуру `gradient`, вычисляющую это значение. Помните, что при умножении вектора и ковектора должно получаться их скалярное произведение — сумма почленных произведений элементов.

b. Заметим, что если на вход функции подается вектор, то это подобно нескольким входам, так что на выходе градиента должен получаться ковектор. Заметим также, что если входом функции служит ковектор, то на выходе градиента должен быть вектор. Сделайте так, чтобы это работало.

3.3.3 Технические детали

Несмотря на то что основная идея, лежащая в основе автоматического дифференцирования, несложна, имеется несколько тонких технических деталей, которым нужно уделить внимание, чтобы все заработало правильно.

Алгебра дифференциалов

Чтобы вычислить вторую производную, требуется взять производную производной функции. Вычисление такой функции потребует работы с двумя бесконечно малыми величинами. Для того чтобы стало возможным вычисление кратных производных и производных от функций с несколькими аргументами, нужно определить алгебру объектов-дифференциалов в «пространстве бесконечно малых величин». Объектами здесь служат степенные последовательности с многими переменными, где ни одно бесконечно малое приращение не имеет степень выше единицы[23].

Объект-дифференциал представляется как помеченный список членов степенной последовательности. Каждый ее член содержит коэффициент и список бесконечно малых множителей. Список хранится в отсортированном виде по убыванию. (Порядок определяется числом приращений. Таким образом, $\delta x \delta y$ стоит выше, чем δx и δy.) Вот незамысловатая реализация[24]:

```
(define differential-tag 'differential)

(define (differential? x)
  (and (pair? x) (eq? (car x) differential-tag)))

(define (diff-terms h)
  (if (differential? h)
      (cdr h)
      (list (make-diff-term h '()))))
```

Список членов последовательности — это просто `cdr` объекта-дифференциала. Однако если нам дан объект, который не является дифференциалом в явном виде, например число, то мы преобразуем его в дифференциал с единственным членом без множителей-приращений. При создании дифференциала из (сортированного) списка членов мы всегда пытаемся вернуть упрощенную версию, возможно, просто число, которое не представлено явно как объект-дифференциал:

```
(define (make-differential terms)
  (let ((terms                      ; Ненулевые члены
         (filter
          (lambda (term)
            (let ((coeff (diff-coefficient term)))
              (not (and (number? coeff) (= coeff 0)))))
          terms)))
    (cond ((null? terms) 0)
```

[23] Формальные алгебраические детали этой картины прояснил Хал Абельсон в 1994 г. в процессе исправления программной ошибки. Затем в 1997 г. код тщательно переработал Сассман при содействии Харди Майера и Джека Уиздома.

[24] Было бы правильнее использовать структуры-записи, но тогда было бы труднее отлаживать без возможности красиво их распечатать.

```
    ((and (null? (cdr terms))
          ;; Только конечная часть:
          (null? (diff-factors (car terms))))
     (diff-coefficient (car terms)))
    ((every diff-term? terms)
     (cons differential-tag terms))
    (else (error "Некорректеый список членов")))))
```

В этой реализации каждый член также представлен как помеченный список, содержащий коэффициент и упорядоченный список множителей.

```
(define diff-term-tag 'diff-term)

(define (make-diff-term coefficient factors)
  (list diff-term-tag coefficient factors))

(define (diff-term? x)
  (and (pair? x) (eq? (car x) diff-term-tag)))

(define (diff-coefficient x)
  (cadr x))

(define (diff-factors x)
  (caddr x))
```

При вычислении производных нужно складывать и умножать дифференциалы:

```
(define (d:+ x y)
  (make-differential
   (+diff-termlists (diff-terms x) (diff-terms y))))

(define (d:* x y)
  (make-differential
   (*diff-termlists (diff-terms x) (diff-terms y))))

(define (make-infinitesimal dx)
  (make-differential (list (make-diff-term 1 (list dx)))))
```

Сложение членов — это место, где мы отслеживаем и используем их порядок, причем члены высшего порядка идут в списке первыми. Два члена можно сложить, только если при них стоят одинаковые множители. А если сумма коэффициентов равна нулю, то соответствующий член незачем включать в список.

```
(define (+diff-termlists l1 l2)
  (cond ((null? l1) l2)
        ((null? l2) l1)
        (else
         (let ((t1 (car l1)) (t2 (car l2)))
           (cond ((equal? (diff-factors t1) (diff-factors t2))
                  (let ((newcoeff (+ (diff-coefficient t1)
                                     (diff-coefficient t2))))
```

```
                (if (and (number? newcoeff)
                         (= newcoeff 0))
                    (+diff-termlists (cdr l1) (cdr l2))
                    (cons
                     (make-diff-term newcoeff
                                     (diff-factors t1))
                     (+diff-termlists (cdr l1)
                                      (cdr l2))))))
            ((diff-term>? t1 t2)
             (cons t1 (+diff-termlists (cdr l1) l2)))
            (else
             (cons t2
                   (+diff-termlists l1 (cdr l2)))))))))
```

Умножение списков членов не представляет труда при условии, что мы умеем перемножать отдельные члены. Произведение двух списков членов `l1` и `l2` — это список членов, возникающий при суммировании списков членов, возникающих при умножении каждого члена `l1` на каждый член `l2`.

```
(define (*diff-termlists l1 l2)
  (reduce (lambda (x y)
            (+diff-termlists y x))
          '()
          (map (lambda (t1)
                 (append-map (lambda (t2)
                               (*diff-terms t1 t2))
                             l2))
               l1)))
```

Каждый член содержит коэффициент и список множителей (бесконечно малых). Ни один член в дифференциале не должен содержать бесконечно малую величину в степени больше первой, поскольку $\delta x^2 = 0$. Поэтому при перемножении двух членов требуется проверить, что списки множителей, которые мы сливаем, не имеют общих элементов. По этой причине `*diff-terms` возвращает либо список, содержащий член-произведение, либо пустой список. Это значение нужно присоединить к `*diff-termlists`. При слиянии двух списков множителей результат сохраняется в отсортированном виде; это помогает отсортировать члены.

```
(define (*diff-terms x y)
  (let ((fx (diff-factors x)) (fy (diff-factors y)))
    (if (null? (ordered-intersect diff-factor>? fx fy))
        (list (make-diff-term
               (* (diff-coefficient x) (diff-coefficient y))
               (ordered-union diff-factor>? fx fy)))
        '())))
```

Конечная и бесконечно малая часть

Дифференциал содержит конечную и бесконечно малую часть. Процедура `diff:binary-proc` со стр. 105 неправильно работает с объектами-дифференциалами, в

которых больше одного бесконечно малого. Для того чтобы части аргументов `x` и `y` получались совместимыми, мы на самом деле используем:

```
(define (diff:binary-proc f d0f d1f)
  (define (bop x y)
    (let ((factor (maximal-factor x y)))
      (let ((dx (infinitesimal-part x factor))
            (dy (infinitesimal-part y factor))
            (xe (finite-part x factor))
            (ye (finite-part y factor)))
        (d:+ (f xe ye)
             (d:+ (d:* dx (d0f xe ye))
                  (d:* (d1f xe ye) dy))))))
  bop)
```

где множитель выбирается через `maximal-factor`, так чтобы и `x`, и `y` содержали его в члене с максимальным количеством множителей.

Конечная часть объекта-дифференциала состоит из всех его членов, кроме содержащих максимальный множитель в члене самого высокого порядка, а бесконечно малая часть состоит из оставшихся членов, которые все содержат этот множитель.

Рассмотрим следующее вычисление:

$$f(x+\delta x, y+\delta y) = \\ f(x,y) + \partial_0 f(x,y)\cdot\delta x + \partial_1 f(x,y)\cdot\delta y + \partial_0\partial_1 f(x,y)\cdot\delta x\delta y$$

Член самого высокого порядка здесь $\partial_0\partial_1 f(x,y)\cdot\delta x\delta y$. Он симметричен относительно x и y. Здесь ключевое соображение состоит в том, что мы можем разбить объект-дифференциал произвольно, считая первичным любой из максимальных множителей (здесь это δx и δy). Какой именно множитель выбран, не имеет значения, поскольку смешанные частные производные $\mathbf{R}^n \to \mathbf{R}$ независимы от порядка[25].

```
(define (finite-part x #!optional factor)
  (if (differential? x)
      (let ((factor (default-maximal-factor x factor)))
        (make-differential
         (remove (lambda (term)
                   (memv factor (diff-factors term)))
                 (diff-terms x))))
      x))

(define (infinitesimal-part x #!optional factor)
  (if (differential? x)
      (let ((factor (default-maximal-factor x factor)))
        (make-differential
         (filter (lambda (term)
                   (memv factor (diff-factors term)))
                 (diff-terms x))))
      0))
```

[25] То, что можно использовать любой множитель члена наивысшего порядка, было главным соображением в модификации идеи частного дифференцирования Халом Абельсоном в 1994 г.

```
(define (default-maximal-factor x factor)
  (if (default-object? factor)
      (maximal-factor x)
      factor))
```

Как работает извлечение производной

Как разъясняется на стр. 109, для работы с производными высших порядков и функциями нескольких аргументов дифференциал представляется как степенная последовательность с несколькими переменными, где ни одно бесконечно малое приращение не имеет степени больше единицы. Каждый член этой последовательности содержит коэффициент и список бесконечно малых множителей-приращений. Такое устройство усложняет извлечение производной относительно каждого приращения.

Вот как все организовано.

В случае, когда результатом является объект-дифференциал, требуется найти его члены, содержащие бесконечно малый множитель `dx` для той производной, которую мы вычисляем. Собираем эти члены, убирая из них `dx`. Если после отбрасывания множителей `dx` ничего не осталось, значение производной равно нулю. Если остался ровно один член без дифференциальных множителей, то коэффициент при этом члене и есть значение производной. Но если остались члены с другими дифференциальными множителями, то в качестве значения производной нужно вернуть объект-дифференциал с этими оставшимися членами.

```
(define (extract-dx-differential value dx)
  (let ((dx-diff-terms
         (filter-map
          (lambda (term)
            (let ((factors (diff-factors term)))
              (and (memv dx factors)
                   (make-diff-term (diff-coefficient term)
                                   (delv dx factors)))))
          (diff-terms value))))
    (cond ((null? dx-diff-terms) 0)
          ((and (null? (cdr dx-diff-terms))
                (null? (diff-factors (car dx-diff-terms))))
           (diff-coefficient (car dx-diff-terms)))
          (else (make-differential dx-diff-terms)))))

(define-generic-procedure-handler extract-dx-part
  (match-args differential? diff-factor?)
  extract-dx-differential)
```

Функции высших порядков

Во многих приложениях требуется, чтобы автоматическое дифференцирование правильно работало с функциями, возвращающими функции в качестве значений:

```
(((derivative
   (lambda (x)
```

```
     (lambda (y z)
       (* x y z))))
  2)
 3
 4)
12
```

С подключением функций-литералов и частных производных дела становятся еще интереснее:

```
((derivative
  (lambda (x)
    (((partial 1) (literal-function 'f))
     x 'v)))
 'u)
(((partial 0) ((partial 1) f)) u v)
```

Можно и еще сложнее:

```
(((derivative
   (lambda (x)
     (derivative
      (lambda (y)
        ((literal-function 'f)
         x y)))))
 'u)
 'v)
(((partial 0) ((partial 1) f)) u v)
```

Работа с такими случаями делает процедуру `extract-dx-part` достаточно хитро устроенной.

Если в результате применения функции к дифференциалу получается функция — производная производной, например, — требуется отложить извлечение производной до тех пор, пока эта функция будет вызвана с аргументами. В случае, когда возвращается функция, например

```
(((derivative
   (lambda (x)
     (derivative
      (lambda (y)
        (* x y)))))
  'u)
 'v)
1
```

производную нельзя извлечь, пока функция не применена к аргументам. Таким образом, мы откладываем вычисление, пока не получим значение в результате такого применения. Расширяем нашу полиморфную процедуру извлечения:

```
(define (extract-dx-function fn dx)
  (lambda args
```

```
    (extract-dx-part (apply fn args) dx)))

(define-generic-procedure-handler extract-dx-part
  (match-args function? diff-factor?)
  extract-dx-function)
```

К сожалению, эта версия `extract-dx-function` содержит нетривиальную ошибку[26]. Наше исправление состоит в том, чтобы завернуть тело новой отложенной процедуры в код, переименовывающий множитель `dx` во избежание нежелательного конфликта. Итак, обработчик для функций заменяется на:

```
(define (extract-dx-function fn dx)
  (lambda args
    (let ((eps (make-new-dx)))
      (replace-dx dx eps
       (extract-dx-part
        (apply fn
          (map (lambda (arg)
                 (replace-dx eps dx arg))
               args))
        dx)))))
```

Здесь создается новый множитель `eps`, и он используется вместо `dx` в аргументах, чтобы избежать конфликта с другими экземплярами `dx`.

Сама по себе замена множителя устроена сложнее, поскольку приходится рыться в различных структурах данных. Мы делаем замену полиморфной процедурой, чтобы иметь возможность расширять ее на новые типы данных. По умолчанию замена просто возвращает исходный объект:

```
(define (replace-dx-default new-dx old-dx object) object)

(define replace-dx
  (simple-generic-procedure 'replace-dx 3
                            replace-dx-default))
```

В случае объекта-дифференциала нам нужно просмотреть и заменить старый множитель на новый, сохраняя при этом список множителей отсортированным:

```
(define (replace-dx-differential new-dx old-dx object)
  (make-differential
   (sort (map (lambda (term)
                (make-diff-term
```

[26]На ошибку такого рода нам указал в 2011 г. Алексей Радул. Общую проблему впервые описали Сискинд и Перлмуттер в 2005 г. [111]: ярлыки дифференциалов, созданные для различения бесконечно малых, приращающих аргументы при вычислении производной, могут смешиваться при вычислении производной от функции, значением которой является функция. Отложенная процедура вычисления производной может вызываться более одного раза и использовать ярлык, созданный при вычислении внешней производной. Недавно Джефф Сискинд показал нам еще одну ошибку в нашем исправлении первой ошибки: существовала потенциальная коллизия между меткой, содержащейся в аргументе, и меткой, унаследованной из лексической сферы действия производной функции. Эти весьма сложные ошибки объясняются в замечательной статье Манзюка и др. [87] Там также дается тщательный анализ способов их исправления.

```
                (diff-coefficient term)
                (sort (substitute new-dx old-dx
                                  (diff-factors term))
                      diff-factor>?)))
            (diff-terms object))
         diff-term>?)))

(define-generic-procedure-handler replace-dx
  (match-args diff-factor? diff-factor? differential?)
  replace-dx-differential)
```

Наконец, если объект сам по себе является функцией, требуется отложить ее, пока не появятся аргументы для вычисления значения:

```
(define (replace-dx-function new-dx old-dx fn)
  (lambda args
    (let ((eps (make-new-dx)))
      (replace-dx old-dx eps
        (replace-dx new-dx old-dx
          (apply fn
            (map (lambda (arg)
                   (replace-dx eps old-dx arg))
                 args)))))))

(define-generic-procedure-handler replace-dx
  (match-args diff-factor? diff-factor? function?)
  replace-dx-function)
```

Получилось намного сложнее, чем можно было ожидать. Выполняется целых три замены дифференциальных множителей. Все это проделывается для того, чтобы избежать коллизий с множителями, которые могут быть свободными в теле `fn`, будучи унаследованными из лексического окружения, где определяется функция `fn`[27].

Упражнение 3.11. Ошибка!
До того, как мы узнали об ошибке, указанной в примечании 26 на стр. 115, процедура `extract-dx-function` записывалась как:

```
(define (extract-dx-function fn dx)
  (lambda args
    (extract-dx-part (apply fn args) dx)))
```

Продемонстрируйте необходимость использования обертки `replace-dx`, построив функцию, чья производная с исходной версией `extract-dx-part` вычисляется неправильно, а с исправленной — правильно. Это нелегко! Возможно, полезно будет прочитать работы, на которые ссылается примечание 26.

3.3.4 Функции-литералы от дифференциальных аргументов

В случае простых аргументов применение функционального литерала сводится к построению выражения, применяющего функциональный литерал к этим аргументам.

[27] Это подробно объясняется в работе Манзюка и др. [87]

Однако функциональные литералы должны также уметь принимать в качестве аргументов дифференциалы. Когда такое случается, функциональный литерал обязан построить выражения (частных) производных от тех аргументов, которые являются дифференциалами. Выражение производной для `i`-го аргумента `n`-местной функции таково:

```
(define (deriv-expr i n fexp)
  (if (= n 1)
      `(derivative ,fexp)
      `((partial ,i) ,fexp)))
```

Некоторые аргументы могут быть дифференциалами, так что функция-литерал для каждого аргумента должна выбрать конечную часть и бесконечно малую часть. Как и для обработчиков двухместных арифметических операций, требуется последовательным образом выбрать максимальный множитель. Функции-литералы могут принимать несколько аргументов, так что этот процесс может показаться сложным, однако уже написанная нами процедура `maximal-factor` умеет обрабатывать такие ситуации. Это объясняется в разделе 3.3.3.

Если среди аргументов нет дифференциалов, мы просто собираем искомое выражение в виде списка. Если дифференциалы есть, требуется построить производную функционального литерала. Для этого мы находим максимальный множитель всех аргументов и выделяем в аргументах конечные части — все члены, которые не содержат этот множитель. (Бесконечно малые части состоят из членов, где этот множитель содержится.) Частные производные сами являются функциями-литералами, чьи выражения включают индекс аргумента. Результатом будет объект-дифференциал в виде скалярного произведения частных производных на конечных частях аргументов с бесконечно малыми частями аргументов.

Все это собрано вместе в следующей процедуре:

```
(define (literal-function fexp)
  (define (the-function . args)
    (if (any differential? args)
        (let ((n (length args))
              (factor (apply maximal-factor args)))
          (let ((realargs
                 (map (lambda (arg)
                        (finite-part arg factor))
                      args))
                (deltargs
                 (map (lambda (arg)
                        (infinitesimal-part arg factor))
                      args)))
            (let ((fxs (apply the-function realargs))
                  (partials
                   (map (lambda (i)
                          (apply (literal-function
                                  (deriv-expr i n fexp))
                                 realargs))
                        (iota n))))
              (fold d:+ fxs
```

```
                (map d:* partials deltargs)))))
        `(,fexp ,@args)))
  the-function)
```

Упражнение 3.12. Функции со структурным значением
Мы сделали процедуру `extract-dx-part` полиморфной (стр. 106), чтобы можно было расширить ее на другие значения, помимо дифференциалов и функций. Расширьте `extract-dx-part`, чтобы она работала с производными функций, возвращающих векторы. Замечание: придется расширить также полиморфную процедуру `replace-dx` (стр. 115) в программе извлечения производной.

3.4 Эффективность полиморфных процедур

В разделе 3.2.3 переход к нужному обработчику происходит через поиск применимого правила в диспетчерском хранилище, содержащемся в метаданных:

```
(define (generic-procedure-dispatch metadata args)
  (let ((handler
         (get-generic-procedure-handler metadata args)))
    (apply handler args)))
```

Реализация диспетчерского хранилища (стр. 96), которую мы использовали в конструкторе `simple-generic-procedure` (стр. 89), устроена довольно грубо. Правила в простом диспетчерском хранилище содержатся в виде списка. Каждое из этих правил представляется в виде пары из применимости и обработчика. Применимость представляет собой список списков предикатов, которые нужно применить к заданным при вызове аргументам. Полиморфная процедура, построенная через `simple-generic-procedure`, выбирает подходящий обработчик через последовательный просмотр списка правил в поисках применимости, подходящей к аргументам.

Это очень неэффективно, поскольку у многих правил применимость может в некоторой позиции аргумента содержать один и тот же предикат. Например, у процедуры умножения в системе численной и символьной арифметики может быть много правил с предикатом `number?` в первой позиции. Таким образом, прежде чем мы найдем подходящее правило, процедура `number?` будет вызвана многократно. Было бы лучше устроить поиск применимого правила так, чтобы избежать излишних проверок. Обычно этого добиваются с помощью индекса.

3.4.1 Префиксные графы

Один из простых механизмов индексирования основан на *префиксном графе*[28].

Обычно префиксный граф представляет собой дерево, но в более общем случае это может быть любой направленный граф. Из каждой вершины исходят ветви, направленные к вершинам-потомкам. С каждой ветвью связан предикат. Ключом для поиска данных служит последовательность признаков, в данном случае аргументов полиморфной процедуры.

Начиная с корня графа, из последовательности извлекается первый признак, и к нему применяются все предикаты с ветвей, выходящих из корневой вершины. По

[28]Префиксные графы изобрел Эдвард Фредкин в начале 1960-х гг.

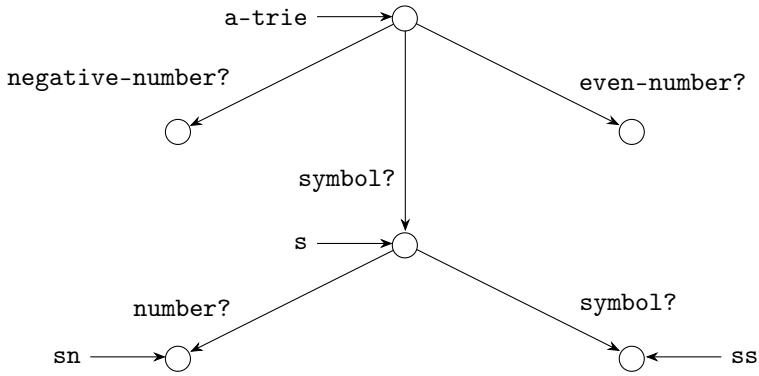


Рис. 3.1. Последовательности признаков можно классифицировать с помощью префиксного графа. Это направленный граф, где каждой ветви приписан предикат. Начиная с корня, первый признак проверяется каждым предикатом на ветвях, исходящих из корневой вершины. Если предикат успешен, процесс переходит в вершину на другом конце ветви, и проверяется следующий признак. Так продолжается шаг за шагом со всеми признаками. Последовательности приписывается тот класс, который указан при последней достигнутой вершине

ветви, предикат которой оказался успешен, происходит переход, и процесс повторяется с оставшимися элементами списка признаков. Когда признаки закончатся, текущая вершина будет содержать значение, подходящее к последовательности, в данном случае применимый к аргументам обработчик.

На некоторых вершинах может срабатывать больше одного предиката. В таких случаях нужно переходить по всем успешным ветвям. Поэтому у нас может оказаться больше одного применимого обработчика, и для таких ситуаций требуется отдельный механизм, решающий, что делать.

Вот как можно использовать префиксный граф. Следующая последовательность команд пошагово создаст граф, показанный на рис. 3.1.

```
(define a-trie (make-trie))
```

К графу можно добавить ветвь:

```
(define s (add-edge-to-trie a-trie symbol?))
```

где возвращаемым значением `add-edge-to-trie` служит новая вершина на конце добавляемой ветви. Эта вершина будет достигнута, если аргумент окажется символом.

Можно создавать последовательности вершин, указывая пути к ним в виде списков предикатов на ветвях.

```
(define sn (add-edge-to-trie s number?))
```

Вершина `sn` будет достигнута из корня через путь `(list symbol? number?)`. Можно использовать сразу весь путь; это более простой способ создать цепочку вершин, чем многократный вызов `add-edge-to-trie`.

```
(define ss (intern-path-trie a-trie (list symbol? symbol?)))
```

Можно добавить к любой вершине значение (здесь мы используем символьные значения, но в дальнейшем в качестве значений будут выступать процедуры-обработчики):

```
(trie-has-value? sn)
#f

(set-trie-value! sn '(symbol number))

(trie-has-value? sn)
#t

(trie-value sn)
(symbol number)
```

Для задания значений можно также использовать интерфейс на основе путей.

```
(set-path-value! a-trie (list symbol? symbol?)
                 '(symbol symbol))

(trie-value ss)
(symbol symbol)
```

Заметим, что и `intern-path-trie`, и `set-path-value!` используют, где возможно, существующие вершины и ветви и добавляют новые там, где это необходимо.

Теперь можно сопоставить последовательность признаков с построенным графом:

```
(equal? (list ss) (get-matching-tries a-trie '(a b)))
#t

(equal? (list s) (get-matching-tries a-trie '(c)))
#t
```

Можно также сочетать поиск и извлечение значений. Процедура `get-a-value` находит все подходящие вершины, выбирает одну, содержащую значение, и возвращает это значение.

```
(get-a-value a-trie '(a b))
(symbol symbol)
```

Однако не любая последовательность признаков имеет значение:

```
(get-a-value a-trie '(-4))
#f
```

Можно последовательно добавлять значения в вершины префиксного графа:

```
(set-path-value! a-trie (list negative-number?)
                 '(negative-number))
(set-path-value! a-trie (list even-number?)
                 '(even-number))

(get-all-values a-trie '(-4))
((even-number) (negative-number))
```

где процедура get-all-values находит все вершины, соответствующие данной последовательности признаков, и возвращает их значения.

На основе этой реализации префиксного графа мы можем создать диспетчерское хранилище, использующее граф в качестве индекса:

```
(define (make-trie-dispatch-store)
  (let ((delegate (make-simple-dispatch-store))
        (trie (make-trie)))
    (define (get-handler args)
      (get-a-value trie args))
    (define (add-handler! applicability handler)
      ((delegate 'add-handler!) applicability handler)
      (for-each (lambda (path)
                  (set-path-value! trie path handler))
                applicability))
    (lambda (message)
      (case message
        ((get-handler) get-handler)
        ((add-handler!) add-handler!)
        (else (delegate message))))))
```

Новую реализацию мы делаем простой, перенаправляя большинство операций в простое диспетчерское хранилище. Две операции, которые не перенаправляются, — это add-handler!, которая сохраняет обработчик одновременно в простом диспетчерском хранилище и в префиксном графе, и get-handler, которая для доступа использует только префиксный граф. Простое диспетчерское хранилище отвечает за обработчик по умолчанию и за хранение множества правил, что полезно при отладке. Это простой пример, как можно для расширения интерфейса использовать перенаправление, в отличие от более известной идеи наследования.

Упражнение 3.13. Правила в префиксном графе

Чтобы легче было экспериментировать с различными диспетчерскими хранилищами, мы дали конструктор диспетчерских хранилищ в качестве параметра процедурам generic-procedure-constructor и make-generic-arithmetic. Например, можно построить полную полиморфную арифметику, как на стр. 94, но с использованием make-trie-dispatch-store:

```
(define trie-full-generic-arithmetic
  (let ((g (make-generic-arithmetic make-trie-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (add-to-generic-arithmetic! g
       (symbolic-extender numeric-arithmetic))
    g))

(install-arithmetic! trie-full-generic-arithmetic)
```

a. Влияет ли это как-то на зависимость от порядка, с которой мы боролись в разделе 3.2.2?

b. Какие характеристики предикатов в общем случае могут привести к ситуации, где к одной последовательности аргументов подходит несколько обработчиков?

c. Есть ли такие ситуации в нашем арифметическом коде?

Мы написали несложный механизм для измерения эффективности стратегий диспетчеризации. Обернув произвольное вычисление вызовом процедуры `with-predicate-counts`, можно узнать, сколько раз при его выполнении зовутся предикаты диспетчеризации. Например, вычисление `(fib 20)` в полиморфной арифметике с диспетчерским хранилищем на основе префиксного графа может выдать что-то вроде[29]:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

(with-predicate-counts (lambda () (fib 20)))
(109453 number)
(109453 function)
(54727 any-object)
(109453 symbolic)
6765
```

Упражнение 3.14. Эффективность диспетчеризации: измеряй!
Имея инструмент измерения производительности, полезно посмотреть на выполнение

```
(define (test-stormer-counts)
  (define (F t x) (- x))
  (define numeric-s0
    (make-initial-history 0 .01 (sin 0) (sin -.01) (sin -.02)))
  (with-predicate-counts
   (lambda ()
     (x 0 ((evolver F 'h stormer-2) numeric-s0 1)))))
```

для диспетчера на основе списка правил `make-simple-dispatch-store` в арифметике, создаваемой через

```
(define full-generic-arithmetic
  (let ((g (make-generic-arithmetic make-simple-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (add-to-generic-arithmetic! g
    (symbolic-extender numeric-arithmetic))
    g))

(install-arithmetic! full-generic-arithmetic)
```

и для версии на основе префиксного графа (упражнение 3.13) в арифметике, получаемой через

```
(install-arithmetic! trie-full-generic-arithmetic)
```

В некоторых задачах префиксный граф должен работать намного лучше, чем простой список. Можно ожидать, что граф будет выигрывать в производительности, когда есть много правил с одинаковым начальным сегментом.

[29] Распечатываемые имена предикатов в `with-predicate-counts` не содержат в конце вопросительного знака. Например, для предиката `number?` печатается просто `number`. Причина этого закопана глубоко в систему; мы приглашаем любопытных найти ее в коде.

Это важно понимать, потому что иначе кажется контринтуитивным, что иногда префиксный граф не помогает производительности. Мы явным образом ввели его, чтобы избежать повторных проверок. Кратко объясните это явление (примерно на абзац).

Дополнительную пищу для размышлений может дать производительность `(fib 20)` в этих двух реализациях.

Если к данной последовательности аргументов применимо больше одного обработчика, не очень понятно, как с ними обращаться. Решение этого вопроса — задача *политики разрешения*. При проектировании политики разрешения есть несколько соображений. Например, часто хорошей политикой будет выбрать наиболее специализированный обработчик; однако, чтобы реализовать ее, нам нужна дополнительная информация. Иногда имеет смысл запустить все подходящие обработчики и сравнить результаты. Это помогает отлавливать ошибки и дает некоторую степень дублирования. Или же, если каждый обработчик сообщает нам частичную информацию, например числовой интервал, результаты обработчиков можно скомпоновать и получить информацию лучшего качества.

3.4.2 Кеширование

При помощи префиксных графов мы избавились от излишнего вычисления предикатов на аргументах. Можно выиграть еще больше, если использовать абстракцию и вообще устранить вычисление этих предикатов. Предикат выделяет множество объектов, отличающихся от всех остальных объектов; другими словами, в сущности, предикат и множество, им выделяемое, — это одно и то же. В нашей реализации префиксного графа мы избегали избыточных вычислений, основываясь на равенстве процедур, вычисляющих предикаты; если бы этого равенства не было, получились бы избыточные ветви графа и не было бы никакого выигрыша. Поэтому, кстати, с реализацией префиксного графа плохо сочетаются комбинации предикатов.

Проблема здесь в том, что нам хочется строить индекс, различающий объекты в соответствии с предикатами, но непрозрачность процедур делает их ненадежными в качестве ключей индекса. На самом деле хотелось бы присваивать множеству, выделяемому предикатом, имя. Если бы мы могли получить это имя из каждого данного объекта при поверхностном его рассмотрении, можно было бы вообще избежать вычисления предиката. Это имя представляет собой «тип»; однако, чтобы избежать путаницы, мы его будем называть *меткой*.

Если есть способ получить из объекта его метку, можно создать кеш, сохраняющий обработчик от предыдущего диспетчирования и применяющий его заново для новых диспетчирований, где метки аргументов имеют тот же вид. Однако, если явно приклеенных к объектам меток у нас нет, этот подход оказывается ограниченным, поскольку различать можно только объекты с одинаковым представлением, которое зависит от конкретной реализации. Например, легко отличить число от символа, но трудно выделить простое число, поскольку, как правило, в реализации языка нет для них особого представления.

К проблеме явных меток мы вернемся в разделе 3.5, а пока что все же можно получить полезный кеш на основе меток представления из реализации Scheme. Имея в нашей реализации Scheme процедуру `implementation-type-name`, выдающую метку представления для объекта, можно сделать кешированное диспетчерское хранилище:

```
(define a-cached-dispatch-store
  (cache-wrapped-dispatch-store (make-trie-dispatch-store)
                                implementation-type-name))
```

Это хранилище оборачивает кешем диспетчерское хранилище на основе префиксного графа, но точно так же можно обернуть и простое диспетчерское хранилище.

В сердце кеширующего диспетчерского хранилища лежит мемоизатор на основе хеш-таблицы. Ключом в этой таблице служит список меток представления, извлеченный применением к аргументам процедуры `implementation-type-name`. Поскольку мы передаем процедуру `implementation-type-name` в нашу обертку аргументом (как `get-key`), мы вскоре сможем использовать ее для построения диспетчерских хранилищ с более мощными механизмами выбора меток.

```
(define (cache-wrapped-dispatch-store dispatch-store get-key)
  (let ((get-handler
         (simple-list-memoizer
          eqv?
          (lambda (args) (map get-key args))
          (dispatch-store 'get-handler))))
    (lambda (message)
      (case message
        ((get-handler) get-handler)
        (else (dispatch-store message))))))
```

При вызове `simple-list-memoizer` на его последний аргумент накладывается кеш, и получается его мемоизированная версия. Второй аргумент указывает, как получить ключ для кеширования из аргументов процедуры. Аргумент `eqv?` говорит, как сравнивать метки в кеше.

Упражнение 3.15. Производительность кеша
Используя тот же инструмент оценки производительности, что мы ввели в упражнении 3.14 на стр. 122, измерьте выполнение `(test-stormer-counts)` и `(fib 20)` с кешированной версией диспетчера и той же самой полиморфной арифметикой, что и в упражнении 3.14. Сравните результаты.

3.5 Эффективные типы, определяемые пользователем

В разделе 3.4.2 мы ввели в качестве детали кеширующего механизма для диспетчера метки. Каждый аргумент отображается в метку, а затем список меток используется как ключ в кеше, чтобы получить обработчик. Если кеш содержит обработчик, связанный с данным списком меток, он и используется. Если нет, подходящий обработчик ищется при помощи префиксного дерева предикатов и заносится в кеш, привязанный к списку меток.

Этот механизм достаточно груб: в спецификациях применимости можно использовать только те предикаты, которые для любых двух объектов с одинаковой меткой всегда дают одно и то же булево значение. Таким образом, нельзя различать типы более тонко, чем позволяют имеющиеся метки. Метками у нас служат символы,

определяемые реализацией, например `pair`, `vector` или `procedure`. Это накладывает на них жесткие ограничения. Например, невозможно написать правила, которые отличают целые числа, выполняющие предикат `even-integer?` (четные) от выполняющих предикат `odd-integer?` (нечетные).

Нам нужна система меток, позволяющая вычислительно дешево получить метку объекта данных, но где при этом метки не ограничены узким множеством значений, определяемым реализацией. Можно этого добиться, если привязывать метку к каждому объекту, либо с помощью явной структуры данных, либо через ассоциативную таблицу.

Здесь переплетается несколько проблем: хочется использовать предикаты в спецификациях применимости; хочется иметь эффективный механизм диспетчеризации; и хочется также уметь указывать отношения между предикатами, используемыми в диспетчере. Например, хотелось бы иметь возможность сказать, что предикат `integer?` является дизъюнкцией предикатов `even-integer?` и `odd-integer?` и одновременно дизъюнкцией предикатов `positive-integer?`, `negative-integer?` и `zero?`.

Чтобы отобразить такие отношения, нам нужно будет снабдить предикаты метаданными. Однако устроить ассоциативный поиск для получения метаданных предиката, как мы это делали с арностью функций (на стр. 39), будет слишком дорого, поскольку метаданные будут содержать ссылки на другие метки, и отслеживание этих ссылок должно работать эффективно.

Один из способов решить задачу — ввести регистрацию предикатов. Регистрация создает новый тип меток, структуру данных, привязанную к предикату. Эту метку легко будет связывать с объектами, которые принимаются этим предикатом. В метке удобно будет хранить метаданные.

Мы создадим систему, где у каждого объекта может быть только одна метка и где можно объявлять отношения между предикатами. Может показаться, что такое устройство слишком простое, но для наших целей его достаточно.

3.5.1 Предикаты как типы

Начнем с нескольких *простых предикатов*. Например, элементарная процедура `exact-integer?` заранее зарегистрирована в нашей системе как простой предикат:

```
(predicate? exact-integer?)
#t
```

Определим теперь новый предикат, не являющийся элементарным. Построим его на основе следующего чрезвычайно медленного теста на простоту числа.

```
(define (slow-prime? n)
  (and (n:exact-positive-integer? n)
       (n:>= n 2)
       (let loop ((k 2))
         (or (n:> (n:square k) n)
             (and (not (n:= (n:remainder n k) 0))
                  (loop (n:+ k 1)))))))
```

Заметим, что все знаки арифметических операций мы используем с префиксом `n:`, чтобы это были операции нижележащего языка Scheme.

Мы строим абстрактный предикат `prime-number?`, давая ему имя для использования в сообщениях об ошибке и критерий, определяющий, что число считается простым, `slow-prime?`:

```
(define prime-number?
  (simple-abstract-predicate 'prime-number slow-prime?))
```

Процедура `simple-abstract-predicate` строит *абстрактный предикат*. Это хитрый трюк для мемоизации результата дорогого предиката (в нашем случае `slow-prime?`). С абстрактным предикатом связывается конструктор, создающий *помеченный объект*, состоящий из метки абстрактного предиката и самого объекта. Конструктор требует, чтобы помечаемый объект удовлетворял дорогому предикату. Получающийся помеченный объект удовлетворяет абстрактному предикату и несет его метку. Следовательно, помеченный объект можно проверить на свойство, определяемое дорогим предикатом, с помощью быстрого абстрактного предиката (или, что то же самое, через диспетчеризацию по метке).

Например, с помощью абстрактного предиката мы помечаем объекты, про которые мы знаем, что они простые числа, и получаем эффективную реализацию полиморфной диспетчеризации. Это важно потому, что нам не хочется выполнять медленную процедуру `slow-prime?`, чтобы проверить число на простоту во время диспетчеризации. Поэтому мы строим новый *помеченный объект*, состоящий из *метки* (метка для `prime-number?`) и *данных* (само простое число). Когда полиморфная процедура получает помеченный объект, она может эффективно взять его метку и использовать ее как ключ поиска в кеше.

Чтобы создавать помеченные объекты, мы получаем конструктор, связанный с абстрактным предикатом, с помощью процедуры `predicate-constructor`:

```
(define make-prime-number
  (predicate-constructor prime-number?))

(define short-list-of-primes
  (list (make-prime-number 2)
        (make-prime-number 7)
        (make-prime-number 31)))
```

Конструктор `make-prime-number` требует, чтобы его аргумент был простым, что определяется через `slow-prime?`: помечаемые конструктором объекты обязаны быть простыми числами.

```
(make-prime-number 4)
; Неподходящие данные для prime-number: 4
```

3.5.2 Отношения между предикатами

Множества, которые мы определяем через абстрактные предикаты, могут иметь отношения между собой. Например, простые числа являются подмножеством целых. Положительные числа, четные и нечетные, — тоже подмножества целых. Это важно, потому что любая операция, применимая к целым числам, применима и к любому элементу любого их подмножества. Существуют, однако, операции, применимые к элементам подмножества, но неприменимые к произвольным элементам объемлющего множества. Например, четные числа можно без остатка делить на два, но не со всеми целыми числами можно так делать.

Когда мы определили `prime-number?`, мы, в сущности, определили множество объектов. Однако это множество не имеет никакого отношения к множеству, определяемому предикатом `exact-integer?`:

```
(exact-integer? (make-prime-number 2))
#f
```

Хотелось бы, чтобы эти множества были правильным образом связаны между собой. Это делается путем добавления метаданных к самим предикатам:

```
(set-predicate<=! prime-number? exact-integer?)
```

Процедура `set-predicate<=!` изменяет метаданные предикатов-аргументов и отмечает, что множество, выделяемое первым аргументом, является (нестрогим) подмножеством множества, выделяемого вторым аргументом. В нашем случае множество, определяемое предикатом `prime-number?`, объявляется подмножеством множества, определяемого предикатом `exact-integer?`. После этого `exact-integer?` станет узнавать наши объекты:

```
(exact-integer? (make-prime-number 2))
#t
```

3.5.3 Предикаты как ключи для диспетчеризации

Определяемые нами предикаты подходят для использования при диспетчеризации полиморфных процедур. Более того, их можно использовать как ключи поиска в кеше, чтобы диспетчеризация была эффективной. Как описано выше, когда предикат регистрируется, создается новая метка и связывается с этим предикатом. Нам нужен только способ получить метку данного объекта. Это делает процедура `get-tag`.

Если мы передадим `get-tag` в качестве аргумента `get-key` процедуре `cache-wrapped-dispatch-store`, мы получим работающую реализацию. Однако, поскольку множество, определяемое предикатом, может иметь подмножества, нужно рассмотреть ситуацию, когда для некоторого набора аргументов есть несколько потенциальных обработчиков. Имеется несколько способов разрешить эту ситуацию, однако чаще всего как-то выделяют «самый специализированный» обработчик и вызывают его. Поскольку отношение подмножества лишь частичный порядок, необязательно будет ясно, какой обработчик самый специализированный, так что реализация обязана решить этот вопрос независимым способом.

Вот одна из возможных реализаций. При помощи процедуры `rule<` подходящие правила сортируются в нужном порядке, затем из результата выбирается обработчик[30].

```
(define (make-subsetting-dispatch-store-maker choose-handler)
  (lambda ()
    (let ((delegate (make-simple-dispatch-store)))
      (define (get-handler args)
        (let ((matching
```

[30] Процедура `is-generic-handler-applicable?` абстрагирует проверку обработчиков, которую мы раньше в `get-handler` на стр. 97 выполняли с помощью `predicates-match?`. Это дает нам зацепку для дальнейшей настройки.

```
            (filter (lambda (rule)
                      (is-generic-handler-applicable?
                       rule args))
                    ((delegate 'get-rules)))))
        (and (n:pair? matching)
             (choose-handler    ; выбор из отсортированных обработчиков
              (map cdr (sort matching rule<))
              ((delegate 'get-default-handler))))))
    (lambda (message)
      (case message
        ((get-handler) get-handler)
        (else (delegate message))))))
```

Процедура `make-most-specific-dispatch-store` выбирает для запуска первый из отсортированных обработчиков:

```
(define make-most-specific-dispatch-store
  (make-subsetting-dispatch-store-maker
    (lambda (handlers default-handler)
      (car handlers))))
```

Другим возможным выбором было бы создать «цепочечное» диспетчерское хранилище, когда каждому обработчику передается аргумент, с помощью которого можно вызвать следующий обработчик в отсортированной последовательности. Это полезно, когда обработчик для подмножества хочет расширить поведение обработчика для объемлющего множества, а не заменить его. Мы увидим пример этого в обработчике времени игры-бродилки из раздела 3.5.4.

```
(define make-chaining-dispatch-store
  (make-subsetting-dispatch-store-maker
   (lambda (handlers default-handler)
     (let loop ((handlers handlers))
       (if (pair? handlers)
           (let ((handler (car handlers))
                 (next-handler (loop (cdr handlers))))
             (lambda args
               (apply handler (cons next-handler args))))
           default-handler)))))
```

К любому из этих диспетчерских хранилищ можно добавить кеширование с помощью обертки:

```
(define (make-cached-most-specific-dispatch-store)
  (cache-wrapped-dispatch-store
     (make-most-specific-dispatch-store)
     get-tag))

(define (make-cached-chaining-dispatch-store)
  (cache-wrapped-dispatch-store
     (make-chaining-dispatch-store)
     get-tag))
```

Затем мы создаем соответствующие конструкторы полиморфных процедур:

```
(define most-specific-generic-procedure
  (generic-procedure-constructor
   make-cached-most-specific-dispatch-store))

(define chaining-generic-procedure
  (generic-procedure-constructor
   make-cached-chaining-dispatch-store))
```

3.5.4 Пример: игра-бродилка

Один из традиционных способов моделирования мира — «объектно ориентированное программирование». Идея состоит в том, что моделируемый мир состоит из объектов, каждый из которых обладает независимым внутренним состоянием, а связи между объектами слабые. Предполагается, что у каждого объекта есть определенное поведение. Объект может принимать сообщения от других объектов, изменять свое состояние и посылать сообщения другим объектам. Такое построение программы очень естественно в случаях, когда поведение, подлежащее моделированию, не зависит от взаимодействия различных источников информации: каждое сообщение приходит из какого-то одного источника. Это жесткое ограничение на организацию программы.

Существуют другие способы разбиения задачи на части. Мы уже достаточно работали с «арифметикой», чтобы понять, что значение операции вроде * может зависеть от свойств нескольких аргументов. Например, произведение числа и вектора отличается от произведения двух векторов или двух чисел. Задачи такого рода естественным образом формулируются в терминах полиморфных процедур[31].

Рассмотрим задачу моделирования мира, состоящего из «мест», «вещей» и «людей», с помощью полиморфных процедур. Как должны представляться и храниться переменные состояния, которые считаются внутренними для разных сущностей? Какие операции должны быть полиморфными и по каким видам сущностей? Поскольку естественно будет группировать сущности по типам (или множествам) и считать, что некоторые операции подходят для всех членов объемлющего множества, как нам работать с подтипами? Любой вариант объектно ориентированной точки зрения будет предписывать некоторые конкретные ответы на эти вопросы. У нас здесь больше свободы, и поэтому нам нужно самим придумать соглашения, которым мы будем следовать.

Для иллюстрации этого процесса мы построим мир простой игры-бродилки. Имеется сеть комнат, связанных проходами и населенных различными существами. Некоторые из существ *автономны* и могут перемещаться с места на место. Есть *аватара*, управляемая игроком. Имеются вещи, часть из которых существа могут брать и перемещать. Есть способы взаимодействия существ: тролль может укусить кого-нибудь и нанести ему вред; любое существо может отобрать у другого существа несомую им вещь.

У каждой сущности нашего мира будет сколько-то именованных свойств. Некоторые из этих свойств неизменны; другие могут меняться. Например, в комнате

[31] В языках вроде Haskell или Smalltalk множественные аргументы обрабатываются через диспетчеризацию по первому аргументу, которое выдает объект, диспетчирующий по второму, и т. д.

есть проходы в другие комнаты. Они задают топологию сети и изменяться не могут. Кроме того, у комнаты есть содержимое, а именно существа, находящиеся сейчас в комнате, и вещи, которые можно взять. Содержимое комнаты меняется, когда существа переходят из одной комнаты в другую и носят с собой вещи. Мы сделаем вычислительную модель множества именованных свойств в виде таблицы, сопоставляющей именам значения.

Для этого мира имеет смысл множество полиморфных процедур. Например, некоторые объекты — книги, существа и аватара игрока — могут перемещаться. В каждом таком случае перемещение объекта требует исключение его из содержимого исходной точки, добавление к содержимому конечной точки и изменение свойства местоположения. Эта операция одна и та же для книг, людей и троллей — все это элементы множества подвижных вещей.

Книгу можно прочитать; персонаж может что-то сказать; тролль может укусить кого-нибудь. Для реализации этого поведения требуется, чтобы у книг набор свойств отличался от свойств людей или троллей. Однако у всех этих подвижных вещей есть и общие свойства, например местоположение. Таким образом, при создании вещь должна организовать таблицу всех своих свойств, включая унаследованные от более широких множеств. Правила, реализующие поведение операций вроде перемещения, должны уметь в каждом случае найти обработчики для нужных свойств.

Игра

Наша игра происходит на приблизительной топологической карте MIT. В ней есть различные автономные агенты (неигровые персонажи), например студенты и административные работники. Скажем, замдекана — тролль. Есть подвижные и неподвижные объекты, причем подвижные объекты может взять и перемещать как автономный агент, так и аватара игрока. В игре сейчас мало деталей, но ее можно расширить и сделать интересной.

Мы создаем игровую сессию с аватарой по имени `gjs`, появляющейся в случайном месте. Игра сообщает игроку об окружении аватары.

```
(начать-приключение 'gjs)
вы в: коридор-общежития
вы видите: замдекана
вы можете пойти на: восток
```

Поскольку здесь замдекана, разумно будет уйти. (Он может укусить, а после нескольких укусов аватара умрет!)

```
(идти 'восток)
gjs выходит на: восток
gjs входит в: вестибюль-7
вы в: вестибюль-7
вы видите: вестибюль-10 бесконечный-коридор
вы можете пойти на: верх запад восток
алиса-хакер входит в: вестибюль-7
алиса-хакер говорит: привет gjs
бен-битобор входит в: вестибюль-7
бен-битобор говорит: привет алиса-хакер gjs
замдекана входит в: вестибюль-7
замдекана говорит: привет бен-битобор алиса-хакер gjs
```

Заметим, что вслед за аватарой входят несколько автономных агентов, и входят они по одному. Как видим, сообщения пишутся по мере хода моделируемого времени, а не описывают состояние в какой-то один момент. Это следствие устройства нашей реализации, а не сознательное решение проектировщика.

К сожалению, замдекана пошел вслед за нами, так что надо идти дальше.

```
(сказать "Пойду отсюда")
gjs говорит: Пойду отсюда

(идти 'восток)
gjs выходит на: восток
gjs входит в: вестибюль-10
вы в: вестибюль-10
вы видите: вестибюль-7 бесконечный-коридор большой-двор
вы можете пойти на: восток юг запад верх

(идти 'верх)
gjs выходит на: верх
gjs входит в: 10-250
вы в: 10-250
вы видите: доска
вы можете пойти на: верх низ
```

Комната 10-250 — лекционная аудитория с большой доской. Может быть, ее можно взять?

```
(взять-вещь 'доска)
доска не двигается
```

Жаль — gjs любит доски. Продолжим прогулку.

```
(идти 'верх)
gjs идет на: верх
gjs входит в: библиотека-баркера
вы в: библиотека-баркера
вы видите: инженерный-справочник
вы можете пойти на: верх низ
слышен ужасный, душераздирающий вопль...
```

Похоже, кого-то съел тролль (возможно, замдекана). Однако у нас здесь есть книга, и ее можно взять. Мы берем ее и возвращаемся в аудиторию.

```
(взять-вещь 'инженерный-справочник)
gjs берет: инженерный-справочник

(идти 'низ)
gjs идет на: низ
gjs входит в: 10-250
вы в: 10-250
у вас есть: инженерный-справочник
вы видите: доска
вы можете пойти на: верх низ
```

Из аудитории мы выходим в вестибюль и встречаем там лямбда-мена, который крадет у нас книгу.

```
(идти 'низ)
gjs идет на: низ
gjs входит в: вестибюль-10
gjs говорит: привет лямбда-мен
вы в: вестибюль-10
у вас есть: инженерный-справочник
вы видите: лямбда-мен
вы видите: вестибюль-7 бесконечный-коридор большой-двор
вы можете пойти на: восток юг запад верх
алиса-хакер входит в: вестибюль-10
алиса-хакер говорит: привет gjs лямбда-мен
лямбда-мен берет: инженерный-справочник у gjs
gjs говорит: ох! несчастье!
```

Типы объектов

Для того чтобы создать в игре объект, мы определяем его свойства через `make-property`, предикат типа через `make-type`, получаем связанную с предикатом процедуру создания через `type-instantiator` и вызываем эту процедуру с соответствующими аргументами.

Как сделать тролля? Конструктор `make-troll` принимает аргументы, указывающие значения свойств, принадлежащих конкретному создаваемому троллю. Тролль будет создан в указанном месте с параметрами `restlessness` (подвижность, склонность перемещаться), `acquisitiveness` (хватательность, склонность брать вещи) и `hunger` (голод, склонность кусать людей).

```
(define (create-troll name place restlessness hunger)
  (make-troll 'name name
              'location place
              'restlessness restlessness
              'acquisitiveness 1/10
              'hunger hunger))
```

Мы создаем двух троллей: `грендель` и `замдекана`. Они помещаются в случайные места со случайными значениями склонностей.

```
(define (create-trolls places)
  (map (lambda (name)
         (create-troll name
                       (random-choice places)
                       (random-bias 3)
                       (random-bias 3)))
       '(грендель замдекана)))
```

Процедура `random-choice` случайным образом выбирает элемент из переданного ей списка. Процедура `random-bias` выбирает число (в нашем случае 1, 2 или 3) и берет обратное к нему.

Тип троллей определяется предикатом, который истинен только для троллей. Процедуре `make-type` передается имя типа и описание свойств, характерных для троллей. (Только у троллей есть голод.)

```
(define troll:hunger
  (make-property 'hunger 'predicate bias?))

(define troll?
  (make-type 'troll (list troll:hunger)))
```

Тролль — конкретный тип автономного агента. Таким образом, множество троллей является подмножеством (`<=`) множества автономных агентов.

```
(set-predicate<=! troll? autonomous-agent?)
```

Конструктор троллей напрямую выводится из предиката, определяющего тип. То же самое и с процедурой-аксессором для свойства `hunger`.

```
(define make-troll
  (type-instantiator troll?))

(define get-hunger
  (property-getter troll:hunger troll?))
```

Автономным объектам иногда «часы» сообщают, что надо произвести какое-то действие. Действие, характерное для троллей, — кусать людей.

```
(define-clock-handler troll? eat-people!)
```

Бросается монета со смещением, и она решает, голоден ли тролль в настоящий момент. Если да, тролль ищет вокруг себя других людей (а тролли тоже люди!), и, если такие имеются, он выбирает одного как цель для укуса. Жертва терпит при этом определенный ущерб. Комментатор сообщает о событии.

```
(define (eat-people! troll)
  (if (flip-coin (get-hunger troll))
      (let ((people (people-here troll)))
        (if (n:null? people)
            (narrate! (list "У" (possessive troll) "бурчит живот")
                      troll)
        (let ((victim (random-choice people)))
          (narrate! (list troll "откусывает кусок от"
                          victim)
                    troll)
          (suffer! (random-number 3) victim))))))
```

Процедура `flip-coin` порождает случайное число между 0 и 1. Если это число больше ее аргумента, она возвращает истину. Процедура `random-number` возвращает положительное число, меньшее или равное аргументу.

Процедура `narrate!` добавляет комментарий к истории событий. Второй аргумент `narrate!` (в этом куске кода это `troll`) может быть чем угодно, лишь бы у него было местоположение. Комментатор выводит первый аргумент-сообщение в этом месте. Сообщение можно услышать, только если в этом месте находишься.

Мы сказали, что тролль — разновидность автономного агента. Тип автономного объекта определяется своим предикатом, указывающим свойства, характерные для такого агента. Мы говорим также, что множество автономных агентов является подмножеством множества людей.

```
(define autonomous-agent:restlessness
  (make-property 'restlessness 'predicate bias?))

(define autonomous-agent:acquisitiveness
  (make-property 'acquisitiveness 'predicate bias?))

(define autonomous-agent?
  (make-type 'autonomous-agent
             (list autonomous-agent:restlessness
                   autonomous-agent:acquisitiveness)))

(set-predicate<=! autonomous-agent? person?)
```

Конструктор для троллей указывает значения свойств `restlessness` и `acquisitiveness`, необходимых автономному агенту, в дополнение к свойству `hunger`, которое есть только у троллей. Поскольку тролли — автономные агенты, а автономные агенты — люди, нужны еще значения для свойств людей и их надмножеств. В нашей системе почти у всех свойств есть умолчательные значения, которые присваиваются, если другое явно не указано. Например, у всех объектов должно быть имя; в конструкторе тролля это имя указывается. Однако у человека есть также свойство `health` (здоровье), которое требуется, чтобы запомнить нанесенный ему вред, и в конструкторе троллей значение этого свойства явно не задавалось.

Полиморфные процедуры

Теперь, рассмотрев, как строятся объекты, мы покажем, как реализуется их поведение. А именно, убедимся, что полиморфные процедуры служат эффективным инструментом для описания сложного поведения.

Мы определили процедуру `get-hunger`, которая используется в `eat-people!`, при помощи `property-getter`. Процедура доступа для свойств объекта данного типа реализуется как полиморфная процедура, принимающая объект как аргумент и выдающая значение свойства.

```
(define (property-getter property type)
  (let ((procedure    ; процедура доступа
         (most-specific-generic-procedure
          (symbol 'get- (property-name property))
          1           ; арность
          #f)))       ; обработчик по умолчанию
    (define-generic-procedure-handler procedure
      (match-args type)
      (lambda (object)
        (get-property-value property object)))
    procedure))
```

Здесь показано построение полиморфной процедуры с порождаемым именем (например, `get-hunger`), принимающей один аргумент, и добавление обработчика, ответственного за сам доступ. Последним аргументом `most-specific-generic-procedure` должен быть обработчик по умолчанию; мы ставим здесь `#f`, и это значит, что в таком случае должна выдаваться ошибка.

Кроме того, для описания действия, происходящего при тиканьи часов, мы использовали процедуру `define-clock-handler`. Эта процедура просто добавляет обработчик в уже существующую полиморфную процедуру `clock-tick!`.

```
(define (define-clock-handler type action)
  (define-generic-procedure-handler clock-tick!
    (match-args type)
    (lambda (super object)
      (super object)
      (action object))))
```

В этой полиморфной процедуре поддерживаются «цепочки», когда каждый обработчик принимает дополнительный аргумент (здесь он называется `super`), при вызове которого зовутся все обработчики, определенные на надмножествах данного объекта. Аргументы, передаваемые в `super`, имеют то же значение, что и аргументы текущей процедуры; в нашем случае такой аргумент только один, и он передается дальше. По существу, это тот же механизм, что используется в языках вроде Java, но там это делается через волшебное ключевое слово, а не через аргумент.

Процедура `clock-tick!` зовется для того, чтобы произвести действие, а не чтобы вернуть результат. Заметим, что действие, за которое сейчас отвечаем мы, будет проделано после действий, определяемых надмножествами. Можно было бы произвести сначала текущее действие, а уже потом остальные; для этого достаточно было бы просто поменять порядок вызовов.

Настоящая сила организации программы в виде полиморфных процедур видна в механизме перемещения объектов. Например, когда мы берем инженерный справочник, мы его перемещаем из комнаты в свою сумку. За это отвечает процедура `move!`.

```
(define (move! thing destination actor)
  (generic-move! thing
                 (get-location thing)
                 destination
                 actor))
```

Процедура `move!` реализована через вызов более общей процедуры `generic-move!`, принимающей четыре аргумента: вещь, подлежащую перемещению, ее текущее положение, желаемое положение и агента, производящего перемещение. Процедура эта полиморфна, поскольку поведение при перемещении может зависеть от типов всех аргументов.

При создании `generic-move!` мы указываем обработчик общего вида, отвечающий за случаи, не покрываемые более частными обработчиками (для конкретных типов аргументов).

```
(define generic-move!
  (most-specific-generic-procedure 'generic-move! 4 #f))
```

```
(define-generic-procedure-handler generic-move!
  (match-args thing? container? container? person?)
  (lambda (thing from to actor)
    (tell! (list thing "не двигается")
           actor)))
```

Процедура `tell!` посылает сообщение (первый аргумент) агенту `actor`, который пытается переместить вещь `thing`. Если агент — аватара игрока, сообщение выводится на экран.

В нашем сеансе игры мы взяли книгу. Мы проделали это путем вызова процедуры `взять-вещь` с именем `инженерный-справочник` в качестве аргумента. Эта процедура соотносит имя с вещью и затем зовет `take-thing!`, которая в свою очередь вызывает `move!`.

```
(define (взять-вещь name)
  (let ((thing (find-thing name (here))))
    (if thing
        (take-thing! thing my-avatar)))
        'done)

(define (take-thing! thing person)
  (move! thing (get-bag person) person))
```

Здесь две процедуры. Первая из них — процедура пользовательского интерфейса, позволяющая игроку указывать на вещь удобным образом, называя ее по имени. Она вызывает вторую внутреннюю процедуру, которая зовется также из других мест в программе.

Чтобы все это заработало, мы добавляем в `generic-move!` обработчик, отвечающий за перемещение подвижных объектов из локаций в сумки:

```
(define-generic-procedure-handler generic-move!
  (match-args mobile-thing? place? bag? person?)
  (lambda (mobile-thing from to actor)
    (let ((new-holder (get-holder to)))
      (cond ((eqv? actor new-holder)
             (narrate! (list actor
                             "берет" mobile-thing)
                       actor))
            (else
             (narrate! (list actor
                             "берет" mobile-thing
                             "и отдает" new-holder)
                       actor)))
      (if (not (eqv? actor new-holder))
          (say! new-holder (list "Спасибо!")))
      (move-internal! mobile-thing from to))))
```

Если агент `actor` берет вещь, `actor` равен `new-holder`. Однако может случиться, что `actor` берет вещь `thing` в месте `place` и кладет ее в чью-то еще сумку!

Процедура `say!` указывает, что кто-то что-то говорит. Первый ее аргумент — говорящий персонаж, а второй — текст, который произносится. Наконец, процедура `move-internal!` занимается собственно перемещением объекта из одного места в другое.

Чтобы бросить вещь, мы с помощью процедуры `drop-thing` перемещаем ее из сумки в место, где находимся:

```
(define (бросить-вещь name)
  (let ((thing (find-thing name my-avatar)))
    (if thing
        (drop-thing! thing my-avatar)))
        'done)

(define (drop-thing! thing person)
  (move! thing (get-location person) person))
```

Следующий обработчик для `generic-move!` позволяет бросить вещь. Агент `actor` может бросить вещь `thing` из своей собственной сумки `bag`, а может взять и бросить что-то из сумки другого персонажа.

```
(define-generic-procedure-handler generic-move!
  (match-args mobile-thing? bag? place? person?)
  (lambda (mobile-thing from to actor)
    (let ((former-holder (get-holder from)))
      (cond ((eqv? actor former-holder)
             (narrate! (list actor
                             "бросает" mobile-thing)
                       actor))
            (else
             (narrate! (list actor
                             "берет" mobile-thing
                             "у" former-holder
                             "и бросает на пол")
                       actor)))
      (if (not (eqv? actor former-holder))
          (say! former-holder
                (list "Ты зачем это делаешь?")))
      (move-internal! mobile-thing from to))))
```

Еще один обработчик `generic-move!` обеспечивает возможность что-то подарить или украсть, переместив вещь из одной сумки в другую. Здесь поведение зависит от отношений между агентом, исходным владельцем вещи и адресатом перемещения.

```
(define-generic-procedure-handler generic-move!
  (match-args mobile-thing? bag? bag? person?)
  (lambda (mobile-thing from to actor)
    (let ((former-holder (get-holder from))
          (new-holder (get-holder to)))
      (cond ((eqv? from to)
             (tell! (list mobile-thing "и так у" new-holder)
```

```
                actor))
           ((eqv? actor former-holder)
            (narrate! (list actor
                            "дает" mobile-thing
                            new-holder)
                      actor))
           ((eqv? actor new-holder)
            (narrate! (list actor
                            "берет" mobile-thing
                            "у" former-holder)
                      actor))
           (else
            (narrate! (list actor
                            "берет" mobile-thing
                            "у" former-holder
                            "и дает" new-holder)
                      actor)))
     (if (not (eqv? actor former-holder))
         (say! former-holder (list "ох! несчастье!")))
     (if (not (eqv? actor new-holder))
         (say! new-holder
               (list "ого! откуда у тебя такое?")))
     (if (not (eqv? from to))
         (move-internal! mobile-thing from to)))))
```

Еще один интересный случай — перемещение персонажа из одного места в другое. Это реализует следующий обработчик:

```
(define-generic-procedure-handler generic-move!
  (match-args person? place? place? person?)
  (lambda (person from to actor)
    (let ((exit (find-exit from to)))
      (cond ((or (eqv? from (get-heaven))
                 (eqv? to (get-heaven)))
             (move-internal! person from to))
            ((not exit)
             (tell! (list "нет прохода из" from
                          "в" to)
                    actor))
            ((eqv? person actor)
             (narrate! (list person "выходит на:" (get-direction exit))
                       from)
             (move-internal! person from to))
            (else
             (tell! (list "нельзя двигать" person "!")
                    actor))))))
```

Можно добавить еще много обработчиков. Важно видеть, что поведение процедуры перемещения зависит от типов всех ее аргументов. Поэтому поведение ясным образом разбивается на фрагменты, каждый из которых понятен по отдельности. В

традиционном объектно ориентированном проектировании такой изящной декомпозиции достичь трудно, поскольку требуется выбрать один из аргументов как основной центр диспетчеризации. Должен ли это быть перемещаемый предмет? Исходное положение? Конечное положение? Агент? При любом выборе ситуация оказывается сложнее, чем следовало.

Как писал Алан Перлис, «лучше, если 100 функций работают с одной структурой данных, чем 10 функций с 10 структурами».

Реализация свойств

Как мы видели, чтобы создать объект в нашей игре, нужно определить его свойства через `make-property`, определить предикат типа через `make-type`, получить процедуру создания объекта, связанную с типом, через `type-instantiator` и вызвать эту процедуру с нужными аргументами. За этим простым описанием скрывается сложная реализация, и она стоит подробного рассмотрения.

Интересная особенность этого кода в том, что он обеспечивает простой и гибкий механизм работы со свойствами, привязанными к экземпляру типа, устойчивый в условиях подтипирования. Свойства представлены как абстрактные объекты, а не по именам, чтобы избежать конфликта имен при подтипировании. Например, тип млекопитающего `mammal` может обладать свойством `forelimb`, указывающим на обычную переднюю ногу. Его подтип `bat`, изображающий летучую мышь, может иметь свойство с тем же именем, которое указывает на совершенно другой объект, крыло! Если бы свойства определялись именем, пришлось бы поменять имя одному из этих типов. В нашей же реализации свойства обозначают сами себя, и два свойства с одинаковыми именами отличны друг от друга.

Процедура `make-property` создает тип данных, содержащий имя, предикат и процедуру-поставщик умолчательного значения. Первый аргумент `make-property` — это имя свойства, а остальные представляют собой список свойств с дополнительными метаданными о создаваемом свойстве. Примером может служить определение `troll:hunger` на стр. 133. Мы не будем разбирать обработку этого списка, поскольку это не слишком интересно[32].

```
(define (make-property name . plist)
  (guarantee n:symbol? name)
  (guarantee property-list? plist)
  (%make-property name
                  (get-predicate-property plist)
                  (get-default-supplier-property plist)))
```

Свойство реализуется в виде *записи* языка Scheme [65], т. е. структуры данных, состоящей из именованных полей. Запись определяется через сложную синтаксиче-

[32]Процедура `make-property` проверяет свои аргументы через вызовы вспомогательной процедуры `guarantee`. Первым аргументом `guarantee` является предикат (желательно, зарегистрированный), а вторым — объект, подлежащий проверке. Может быть еще третий аргумент, идентифицирующий место вызова. Если объект не удовлетворяет предикату, `guarantee` сообщает об ошибке. Процедура `guarantee-list-of` работает подобным же образом, только вместо объекта должен быть список объектов, удовлетворяющих предикату.

Ранее в тексте мы использовали `assert`. Процедура `assert` удобна для задания условий, которые должны выполняться в конкретном месте программы. В более узко ограниченном случае, когда проверяется тип аргументов, предпочтительна `guarantee`.

ческую конструкцию, где указывается конструктор, предикат типа и процедуры доступа к каждому полю:

```
(define-record-type <property>
    (%make-property name predicate default-supplier)
    property?
  (name property-name)
  (predicate property-predicate)
  (default-supplier property-default-supplier))
```

Элементарной процедуре создания записи `%make-property` мы дали имя, начинающееся со знака процента (%). Знак процента в начале часто используется, чтобы отметить процедуру, которая нужна только для поддержки более высокоуровневой абстракции. Единственный вызов `%make-property` делается из `make-property`, а уже она используется в других частях системы.

Имея набор свойств, мы можем создать предикат типа:

```
(define (make-type name properties)
  (guarantee-list-of property? properties)
  (let ((type
         (simple-abstract-predicate name instance-data?)))
    (%set-type-properties! type properties)
    type))
```

Предикат типа состоит из обыкновенного абстрактного предиката (см. стр. 126) и указанных при нем свойств, и они сохраняются вместе через вызов `%set-type-properties!`. Указанные свойства используются не сами по себе, а вместе со свойствами надмножеств данного типа. Помечаемый объект удовлетворяет предикату `instance-data?`. Он связывает свойства своего типа со значениями.

```
(define (type-properties type)
  (append-map %type-properties
              (cons type (all-supertypes type))))
```

Наконец, процедура `type-instantiator` порождает процедуру создания объекта, которая принимает список свойств, где ключами служат имена этих свойств, разбирает этот список и в результате создает экземпляр типа, где каждое свойство связано со своим значением. Кроме того, она зовет процедуру `set-up!`, которая позволяет провести дополнительную инициализацию для данного типа.

```
(define (type-instantiator type)
  (let ((constructor (predicate-constructor type))
        (properties (type-properties type)))
    (lambda plist
      (let ((object
             (constructor (parse-plist plist properties))))
        (set-up! object)
        object))))
```

Упражнение 3.16. Знакомство с бродилкой

Загрузите игру-бродилку и запустите ее командой `(start-adventure ваше-имя)`. Походите своей аватарой по комнатам. Найдите какой-нибудь объект и возьмите его. Бросьте на пол предмет, подобранный где-нибудь еще.

Упражнение 3.17. Здоровье
Измените представление здоровья, чтобы у него было больше значений, чем доступно в исходной игре. Настройте свое представление так, чтобы вероятность смерти от укуса тролля оставалась такой же, как до ваших изменений. Сделайте так, чтобы от несмертельного укуса тролля или другого повреждения можно было излечиться, отдохнув несколько циклов.

Упражнение 3.18. Медицинская помощь
Создайте новую локацию, поликлинику. Сделайте ее доступной из здания Гринов и из башни Гейтса. Если персонаж, страдающий от ран (вызванных, например, укусом тролля), добирается до поликлиники, его здоровье должно восстанавливаться.

Упражнение 3.19. Палантир
Создайте новый объект, *палантир* («видящий камень», как во *Властелине колец* Толкина). Каждый экземпляр палантира способен связываться с любым другим. Так что если один палантир находится в вестибюле-10, а другой — в коридоре общежития, то можно смотреть, что происходит в коридоре общежития, глядя в палантир из вестибюля-10. (В сущности, палантир — это магическая камера наблюдения с дисплеем.)

Поставьте несколько неподвижных палантиров в разных местах кампуса и позвольте своей аватаре ими пользоваться. Можете ли вы отслеживать положение своих друзей? А троллей?

Можете ли вы устроить так, чтобы кто-то еще, кроме аватары, использовал палантир для каких-нибудь осмысленных целей? Может быть, президент университета?

Упражнение 3.20. Невидимость
Создайте «плащ-невидимку», который делает любого персонажа (включая вашу аватару) невидимым и недоступным для атак троллей. Однако этот плащ нужно снимать (бросать) спустя какое-то время, потому что владение им медленно съедает здоровье обладателя.

Упражнение 3.21. Ваша очередь
Теперь, поиграв в нашем «мире», состоящем из персонажей, локаций и предметов, расширьте его каким-то более существенным образом. Вы ограничены только своей фантазией. Можно, например, иметь подвижные локации, вроде лифтов, у которых входы и выходы меняются со временем, и, может быть, они управляются персонажами. Но это только одно из возможных предложений — изобретайте сами!

Упражнение 3.22. Многопользовательская игра
Это не простое упражнение, а довольно большой проект.

a. Дополните игру так, чтобы в нее могли играть несколько человек, каждый со своей аватарой.

b. Сделайте так, чтобы игроки могли заходить в игру с разных терминалов.

3.6 Заключение

Полиморфные процедуры, описанные в этой главе, — одновременно мощное и опасное средство; они не для слабаков. То, что мы позволяем программисту динамически

изменять значение элементарных операций языка, может привести к коду, который будет невозможно поддерживать. Однако если мы будем расширять операции осторожно, только на новые типы аргументов, и не будем менять значение на исходных типах, можно будет получить важные расширения, не испортив старые программы. В большинстве языков программирования нет свободы изменять существующее поведение существующих операций, и тому есть веские причины. Однако многие идеи из этой главы переносимы, и их можно безопасно использовать. К примеру, во многих языках, столь разных как C++ и Haskell, разрешается перегружать знаки операций и придавать им новое значение на типах, определенных пользователем.

Расширения арифметики достаточно просты и естественны, однако и здесь следует помнить о проблемах, которые могут возникнуть, и о хитрых ошибках, которые легко совершить: сложение целых чисел ассоциативно, но сложение чисел с плавающей точкой не ассоциативно; умножение чисел коммутативно, а умножение матриц нет. Если же мы перегрузим знак сложения и разрешим ему обозначать конкатенацию строк, такое расширение окажется некоммутативным. Прекрасно, впрочем, что можно без труда расширить арифметику, чтобы она работала над символьными выражениями, содержащими числовые литералы, наряду с числовыми значениями. Также несложно, хотя очень трудоемко, расширить ее дальше на функции, векторы, матрицы и тензоры. Однако в конце концов мы окажемся в тупике по поводу упорядочения выражений — символьный вектор не то же самое, что вектор с символическими координатами! Сложности будут также представлять типы символических функций.

Один из красивых примеров силы расширяемых полиморфных функций — почти тривиальная реализация автоматического дифференцирования в прямом режиме путем расширения каждой арифметической операции для работы с объектами-дифференциалами. Однако заставить эту систему правильно работать с функциями высших порядков, возвращающими функции в качестве результата, оказалось нелегкой задачей. (Разумеется, большинство программистов, которые пишут приложения, нуждающиеся в автоматическом дифференцировании, заботиться об этой сложности не должны.)

В нашей системе «тип» представляется предикатом, верным относительно элементов этого типа. Чтобы это оказалось эффективным, мы ввели регистрацию предикатов и систему меток, которая позволила нам добавлять объявления об отношениях между типами. Например, мы смогли заявить, что простые числа являются подмножеством целых, так что числа, удовлетворяющие определенному пользователем предикату `prime?`, автоматически удовлетворяют предикату `integer?`.

Создав типы, определяемые пользователем, и отношения вложенности между ними, мы получаем целый спектр новых возможностей. Мы показали это на примере простой, но элегантно расширяемой игры-бродилки. Поскольку у нас полиморфные процедуры диспетчируются по типам всех своих аргументов, описание поведения сущностей в нашей игре оказывается намного проще и модульнее, чем было бы, если бы мы диспетчировали по первому аргументу, получали процедуру, диспетчирующую по второму и т. д. Таким образом, моделирование такого же поведения в типичной объектно ориентированной системе с диспетчированием по одному аргументу было бы сложнее и запутаннее.

Мы реализовали эффективные полиморфные процедуры с помощью помеченных данных. К данным присоединяется ярлык с информацией, которая требуется для решения, с помощью каких процедур реализуются указанные операции. Однако,

раз уж мы получили возможность снабжать данные ярлыками, эти ярлыки можно использовать и для других целей. Например, можно помечать данные сведениями об их происхождении, о процедуре, их породившей, или о предположениях, на которых они основываются. Такой след можно использовать для контроля доступа, для отслеживания обращений к чувствительным данным либо для отладки сложных систем [128]. Способность присоединять произвольные метки к любым данным несет в себе дополнительные возможности, помимо их использования для выбора обработчика в полиморфных процедурах.

Глава 4

Сопоставление с образцом

Сопоставление с образцом — это технология, поддерживающая создание специализированных языков и других систем, которым необходима аддитивность.

Сопоставление с образцом является обобщением проверки на равенство. При проверке на равенство мы сравниваем два объекта и проверяем, что у них одинаковы структура и содержимое. При сопоставлении с образцом мы обобщаем эту операцию, позволяя не указывать некоторые фрагменты структуры и содержимого. *Образец* — это описание, которое точно задает некоторые части данных, но в нем могут содержаться «пропуски» (*переменные образца*), которые сопоставляются с недоописанными частями данных. Можно указывать ограничения, задающие, с чем разрешается сопоставляться переменной. Можно также требовать, чтобы различные вхождения одной и той же переменной образца сопоставлялись одинаковым данным.

Образец может сопоставляться с частью более крупной структуры данных; контекст такого сопоставления никак не ограничивается. Способность работать с частичной информацией означает, что только заданные фрагменты образца содержат предположения о сопоставляемых данных; про незаданные фрагменты предположений нет или почти нет.

Это свойство сопоставления с образцом позволяет строить очень гибкие системы, основанные на правилах. В таких системах можно расширять функциональность путем добавления новых правил; существуют, однако, сложности, связанные с тем, как правила определяются и как они взаимодействуют между собой. Например, если применимо более одного правила, результат работы может зависеть от порядка их применения. Мы уже встречались с проблемами взаимодействия правил в интерпретаторе правил настольной игры (см. критику на стр. 68).

Помимо поиска данных, удовлетворяющих частичному описанию, образцы можно сами по себе использовать для представления частично известной информации. Слияние таких образцов (*унификация*) может породить более конкретную информацию, чем в любом из исходных образцов.

Еще один способ использовать образцы — как обобщение полиморфных процедур. Полиморфные процедуры позволяют нам творить волшебные вещи, настраивая значение свободных переменных в программе. Можно считать что программа, которая использует полиморфную процедуру, например обозначенную как +, ищет обработчик, способный выполнить задачу «плюсования» указанных аргументов. Обработчик, прикрепленный к полиморфной процедуре, оказывается применим, если

данный набор аргументов удовлетворяет предикатам, заданным во время прикрепления. Но наш язык для описания задач довольно ограничен — у нас есть только один символ, в этом случае `+`. Если вместо этого мы будем объявлять задачи, подлежащие решению, в виде образцов, а процедуры, способные решить эти задачи, с помощью других образцов, то получится намного более богатый язык: *вызов, управляемый образцами.*

4.1 Образцы

Элементарные законы алгебры можно выразить в виде эквивалентностей образцов:

$$a \times (b + c) = a \times b + a \times c.$$

Это распределительный закон умножения относительно сложения. Он говорит, что одну сторону уравнения можно заменить другой, и значение выражения не изменится. Каждая сторона представляет собой образец с переменными a, b и c и константами $\times$ и $+$. Закон говорит, что любое алгебраическое выражение, представляющее собой произведение чего-либо и суммы, мы можем заменить на сумму произведений, и наоборот.

Рассмотрим теперь, как можно организовывать программы на основе сопоставления с образцом. Здесь основной идеей будет компиляция образцов в комбинаторы, из которых строится сопоставитель. В разделе 4.2 мы продемонстрируем это на примере системы переписывания термов для элементарной алгебры.

Язык образцов

Наша первая задача — построить язык образцов. Начнем с простого. Будем строить образцы в виде списков языка Lisp (Scheme). В отличие от математических примеров, приведенных выше, у нас не будет зарезервированных символов вроде $\times$ и $+$, так что придется отличать переменные образцов от констант. Переменную образца можно представить как список, начинающийся с символа вопроса, `?`. Такова традиция. Таким образом, в этом языке образцы, из которых состоит распределительный закон, можно представить следующими списками (мы предполагаем, что работаем с математическими выражениями в лисповском префиксном виде):

```
(* (? a) (+ (? b) (? c)))

(+ (* (? a) (? b)) (* (? a) (? c)))
```

Можно пожаловаться, что было бы короче использовать выделенные символы, например, `?a` вместо длинного выражения `(? a)`. Однако такой вариант было бы сложнее расширять, например если понадобится переменная, сопоставляющаяся только с числами. Разумеется, если это будет нужным, мы можем добавить синтаксис позднее. Однако напомним афоризм Алана Перлиса: «Синтаксический сахар вызывает рак точки с запятой». В нашем простом списочном представлении можно ограничить значения переменной образца `(? a)` числами путем добавления предиката, описывающего ограничение, в список: `(? a ,number?)`.

Одно из ограничений нашего сопоставителя состоит в том, что со вторым образцом должно сопоставляться выражение

```
(+ (* (cos x) (exp y)) (* (cos x) (sin z)))
```

где `a=(cos x)`, `b=(exp y)` и `c=(sin z)`.

Но с ним не должно сопоставляться

```
(+ (* (cos x) (exp y)) (* (cos (+ x y)) (sin z)))
```

поскольку не существует подходящего присваивания переменной `(? a)`, если только почему-то не окажется `x = (+ x y)`[1]. Такие ситуации мы разберем при изучении унификации в разделе 4.4; пока что мы решаем, что сопоставление невозможно.

Еще одно ограничение, которое окажет важное влияние на структуру сопоставителя, состоит в том, что иногда требуется сопоставить неизвестное число последовательных элементов списка. Допустим, например, что нам нужно правило, заменяющее сумму квадратов синуса и косинуса на 1, даже если эти квадраты идут в сумме не подряд:

```
(+ ... (expt (sin theta) 2) ... (expt (cos theta) 2) ...)
(+ 1 ... ... ...)
```

Здесь `...` обозначает последовательность термов. Для сопоставления с несколькими термами мы будем использовать *сегментные переменные*, которые пишутся с префиксом `??`. Таким образом, мы записываем наш образец в виде:

```
(+ (?? t1)
   (expt (sin (? x)) 2)
   (?? t2)
   (expt (cos (? x)) 2)
   (?? t3))
```

Здесь нам потребовалось три сегментных переменных. Сегментная переменная `(?? t1)` сопоставляется с термами перед синусом, `(?? t2)` — с термами между синусом и косинусом, а `(?? t3)` — с термами после косинуса.

Сегментные переменные сильно влияют на систему, потому что мы не знаем, какой длины у нас сегмент, пока не найдем следующий сопоставляющийся фрагмент, а также потому что у нас может быть несколько способов сопоставить образцу выражение. Например, в одной и той же сумме могут встречаться квадраты синуса и косинуса двух различных углов. Или, еще проще, образец

```
(a (?? x) (?? y) (?? x) c)
```

соответствует выражению

```
(a b b b b b b c)
```

четырьмя различными способами. (Заметим, что сегментная переменная `x` должна съесть одинаковое количество `b` в обоих местах, где она встречается в образце.) Таким образом, сопоставитель должен производить поиск в пространстве возможных присваиваний переменных.

[1] Разумеется, очень умный сопоставитель, зная, что мы работаем с числами, мог бы вывести $y = 0$.

4.2 Переписывание термов

Системы переписывания термов представляют собой мощный инструмент создания специализированных языков для обработки информации, имеющей форму выражений. Если у нас есть синтаксическая система, где может требоваться заменить некоторые подвыражения другими «эквивалентными» подвыражениями, часто эти трансформации можно описать с помощью системы переписывания термов, основанной на правилах. Например, многие оптимизации в компиляторах выразимы как локальные переписывания фрагментов программы в более широком контексте. Основными свойствами систем переписывания термов являются поиск информации, подлежащей переписыванию, с помощью сопоставителя, а также система шаблонов для порождения замен.

Имеется целая линия исследований [72] задачи построения сходящихся систем переписывания термов для эквациональных теорий (систем «эквивалентных» выражений). Однако здесь мы ее рассматривать не будем. Кроме того, существует поверхностное сходство между образцами, с которыми происходит сопоставление, и шаблонами для замен. Можно считать, что это позволяет строить двунаправленные правила; эту задачу мы также не будем рассматривать. Сначала разработаем простую однонаправленную систему, где образцы служат для распознавания входов, а шаблоны для порождения выходов.

Вот некоторое приближение к законам алгебраического упрощения:

```
(define algebra-1
  (rule-simplifier
   (list
    ;; Ассоциативный закон для сложения
    (rule '(+ (? a) (+ (? b) (? c)))
          `(+ (+ ,a ,b) ,c))
    ;; Коммутативный закон для умножения
    (rule '(* (? b) (? a))
          (and (expr<? a b)
               `(* ,a ,b)))
    ;; Дистрибутивный закон для умножения над сложением
    (rule '(* (? a) (+ (? b) (? c)))
          `(+ (* ,a ,b) (* ,a ,c))))))
```

В системе `algebra-1` три правила. Первое реализует ассоциативный закон сложения, второе — коммутативный закон умножения, а третье — дистрибутивный закон умножения относительно сложения.

Каждое правило состоит из двух частей, это образец для сопоставления с подвыражением и выражение-*следствие*. Если образец сопоставляется, следствие вычисляется. Если значением следствия будет `#f`, правило неприменимо. Иначе же результат вычисления заменяет сопоставленное подвыражение. Заметим, что для упрощения выражений-следствий мы используем механизм обратной кавычки, описанный на стр. 341.

Правила собираются в список и подаются на вход процедуре `rule-simplifier`. Получается процедура упрощения, которую можно применить к алгебраическому выражению:

```
(algebra-1 '(* (+ y (+ z w)) x))
(+ (+ (* x y) (* x z)) (* w x))
```

Обратите внимание на предикат-ограничитель `expr<?` в следствии правила, выражающего коммутативный закон:

```
(rule '(* (? b) (? a))
      (and (expr<? a b)
           `(* ,a ,b)))
```

Если выражение-следствие возвращает `#f`, считается, что сопоставление не удалось. Система откатывается обратно в сопоставитель и ищет другое соответствие; если таковое не найдется, правило не применимо. В коммутативном законе ограничительный предикат `expr<?` задает порядок на алгебраических выражениях. Причину такого ограничения предлагается выяснить в упражнении 4.1.

Упражнение 4.1. Выражения-ограничители
Зачем в коммутативном законе нужен ограничитель `(expr<? a b)`? Что сломается, если его не будет?

4.2.1 Сегментные переменные в алгебре

Система правил `algebra-2` намного интереснее. Она строится в предположении, что сложение и умножение являются n-местными операциями. Чтобы это заработало, требуются сегментные переменные. Кроме того, чтобы ввести правила для численных упрощений, мы используем предикат `number?` в ограничителях переменных.

```
(define algebra-2
  (rule-simplifier
   (list
    ;; Суммы
    (rule `(+ (? a))
          a)                        ; одноместный + ничего не делает
    (rule `(+ (?? a) (+ (?? b)) (?? c))
          `(+ ,@a ,@b ,@c))         ; ассоциативность: используем n-местный +
    (rule `(+ (?? a) (? y) (? x) (?? b))
          (and (expr<? x y)         ; коммутативность
               `(+ ,@a ,x ,y ,@b)))

    ;; Произведения
    (rule `(* (? a))
          a)                        ; одноместный * ничего не делает
    (rule `(* (?? a) (* (?? b)) (?? c))
          `(* ,@a ,@b ,@c))         ; ассоциативность: используем n-местный *
    (rule `(* (?? a) (? y) (? x) (?? b))
          (and (expr<? x y)         ; коммутативность
               `(* ,@a ,x ,y ,@b)))

    ;; Распределительный закон
    (rule `(* (?? a) (+ (?? b)) (?? c))
```

```
        `(+ ,@(map (lambda (x) `(* ,@a ,x ,@c)) b)))

  ;; Численные упрощения
  (rule `(+ 0 (?? x))
        `(+ ,@x))
  (rule `(+ (? x ,number?) (? y ,number?) (?? z))
        `(+ ,(+ x y) ,@z))
  (rule `(* 0 (?? x))
        0)
  (rule `(* 1 (?? x))
        `(* ,@x))
  (rule `(* (? x ,number?) (? y ,number?) (?? z))
        `(* ,(* x y) ,@z))
  )))
```

В системе `algebra-2`, помимо работы с множественными аргументами сумм и произведений, мы реализовали некоторые численные упрощения. Обратите внимание, что механизм обратной кавычки используется не только при построении выражений-следствий, но и в образцах, где в качестве ограничения на переменные выступает предикат `number?`. Чтобы лучше понять работу упростителя, см. упражнение 4.2.

Теперь мы имеем следующие результаты:

```
(algebra-2 '(* (+ y (+ z w)) x))
(+ (* w x) (* x y) (* x z))

(algebra-2 '(+ (* 3 (+ x 1)) -3))
(* 3 x)
```

Уже видно, как такую систему можно расширить и получить упроститель для широкого класса алгебраических выражений.

Упражнение 4.2. Упорядочивание термов

Предикат `expr<?` говорит, что всякое выражение, явным образом представляющее число, меньше выражения, которое не является явным числом.

a. Как в системе `algebra-2` порядок, задаваемый законами коммутативности, помогает правилам численного упрощения?

b. Допустим, в законах коммутативности мы бы не требовали упорядочения. Как бы нам тогда пришлось выражать правила численного упрощения? Объясните, почему в таком случае эти правила оказались бы крайне дорогими.

Упражнение 4.3. Эффективность сортировки

Порядок, задаваемый в правилах коммутативности, приводит к тому, что термы в суммах и произведениях сортируются пузырьком, что имеет сложность n^2. В серьезных алгебраических задачах, где термов может быть много, это может оказаться слишком дорого. Можно ли в этой системе добиться более эффективной сортировки? Если нет, почему? Если да, как бы вы ее организовали?

Упражнение 4.4. Группировка подобных термов

Описанная нами система не группирует подобные термы. Например,

```
(algebra-2 '(+ (* 4 x) (* 3 x)))
(+ (* 3 x) (* 4 x))
```

Создайте новую систему `algebra-3`, где были бы правила, собирающие подобные термы и преобразующие их в суммы термов. Продемонстрируйте работу этой системы. Вы должны быть способны решать задачи вроде

```
(algebra-3
  '(+ y (* x -2 w) (* x 4 y) (* w x) z (* 5 z) (* x w) (* x y 3)))
(+ y (* 6 z) (* 7 x y))
```

4.2.2 Реализация систем правил

Мы получили некоторое представление, как пользоваться системами правил, основанными на сопоставлении. Рассмотрим теперь, как такие системы работают.

Мы реализуем правило в виде процедуры, которая сопоставляет образец данному на вход выражению. Если сопоставление удачно, правило выполняет следствие в окружении, где переменные правила связываются с сопоставленными данными. Процедура правила принимает продолжение успеха и продолжение неудачи, и таким образом организуется перебор с вызовом следствий и сопоставлением частей правила[2].

Использованная нами процедура `rule-simplifier` является конструктором для простого рекурсивного упростителя. Она создает процедуру `simplify-expression`, которая принимает на вход выражение, и с помощью правил его упрощает. Сначала она рекурсивно упрощает все подвыражения, а затем применяет правила к тому, что получилось. Этот процесс повторяется, пока не сойдется. Таким образом, возвращаемое выражение является неподвижной точкой процесса упрощения.

```
(define (rule-simplifier the-rules)
  (define (simplify-expression expression)
    (let ((subexpressions-simplified
           (if (list? expression)
               (map simplify-expression expression)
               expression)))
      (try-rules subexpressions-simplified the-rules
        (lambda (result fail)   ; A: продолжение успеха
          (simplify-expression result))
        (lambda ()              ; B: продолжение неудачи
          subexpressions-simplified))))
  simplify-expression)
```

Процедура `try-rules` просто проходит по списку правил, сшивая этот процесс с помощью продолжений `succeed` и `fail`.

```
(define (try-rules data rules succeed fail)
  (let per-rule ((rules rules))
    (if (null? rules)
        (fail)              ; конец правил: перейти к точке B
        (try-rule data
```

[2]Подробные примеры и объяснение этого шаблона с обработкой успеха и неудачи можно найти в разделе 5.4.2 на стр. 242.

```
                (car rules)
                succeed    ; если правило успешно, перейти к точке A
                (lambda () ; если правило неудачно, попробовать другие
                  (per-rule (cdr rules)))))))

(define (try-rule data rule succeed fail)
  (rule data succeed fail))
```

Построение правил реализуется процедурой `make-rule`. Она принимает образец и обработчик, реализующий следствие. Например, правило для закона коммутативности на стр. 149 можно напрямую создать через `make-rule`:

```
(make-rule '(* (? b) (? a))
  (lambda (b a)
    (and (expr<? a b)
         `(* ,a ,b))))
```

Обработчику `(lambda (b a) ...)` нужно передать в качестве аргументов значения переменных образца по имени `a` и `b` из словаря, порождаемого процедурой сопоставления. Правило применяет обработчик к списку этих значений в порядке их встречаемости в образце. Таким образом, обработчик должен принимать параметры именно в этом порядке.

Конструктор правила `make-rule` компилирует образец в процедуру сопоставления. Возвращаемое им значение — процедура, которая применяет это сопоставление к данным. При успехе правило применяет обработчик к значениям переменных образца, полученных в результате сопоставления.

Мы увидим, как образец компилируется в процедуру сопоставления, в разделе 4.3; пока что нам достаточно знать, что процедура сопоставления запускается вызовом `run-matcher` и что при успехе третий ее аргумент вызывается со *словарем* в качестве параметра. Словарь `dict` отображает переменные образца на подвыражения, с которыми они сопоставились. Если же сопоставление неудачно, `run-matcher` возвращает `#f`, и правило также неудачно.

```
(define (make-rule pattern handler)
  (let ((match-procedure (match:compile-pattern pattern)))
    (define (the-rule data succeed fail)
      (or (run-matcher match-procedure data
            (lambda (dict)
              (let ((result
                     (apply handler
                            (match:all-values dict))))
                (and result
                     (succeed result
                              (lambda () #f))))))
          (fail)))
    the-rule))
```

Процедура `match:all-values` возвращает значения переменных образца в порядке их встречаемости.

4.2.3 Ремарка: волшебные макросы

Сравним определение правила со стр. 149:

```
(rule '(* (? b) (? a))
      (and (expr<? a b)
           `(* ,a ,b)))
```

с тем, что требуется как аргумент `make-rule`:

```
(make-rule '(* (? b) (? a))
  (lambda (b a)
    (and (expr<? a b)
         `(* ,a ,b))))
```

Имена `a` и `b` повторяются: они встречаются и в образце, и в списке параметров обработчика в одном и том же порядке. Это неприятно и провоцирует ошибки, поскольку требуется, чтобы программист помнил эти имена, и легко ошибиться, если повторить их неверно или перепутать порядок.

Это случай, когда полезна синтаксическая абстракция, называемая также *макрос*. Следующий волшебный фрагмент кода переводит определение правила в соответствующий вызов `make-rule`:

```
(define-syntax rule
  (er-macro-transformer
   (lambda (form rename compare)
     (let ((pattern (cadr form))
           (handler-body (caddr form))
           (r-make-rule (rename 'make-rule))
           (r-lambda (rename 'lambda)))
       `(,r-make-rule ,pattern
                      (,r-lambda
                       ,(match:pattern-names pattern)
                       ,handler-body))))))
```

Можно частично проверить этот макрос с помощью следующего заклинания, раскрывающего макросы, которые встречаются в выражении:

```
(pp (syntax '(rule '(* (? b) (? a))
                   (and (expr<? a b)
                        `(* ,a ,b)))
            (the-environment)))
(make-rule '(* (? b) (? a))
           (lambda (b a)
             (and (expr<? a b)
                  (list '* a b))))
```

Как видим, правило раскрывается в вызов процедуры `make-rule` с образцом и процедурой-обработчиком в качестве аргументов. Это выражение, которое требуется вычислить для создания правила. В более традиционных языках макрос вроде `rule` переводится прямо в код, который подставляется на место вызова этого

макроса. Однако этот процесс не является референциально прозрачным, поскольку развернутый макрос может использовать символы, конфликтующие с символами пользователя. В языке Scheme мы пытаемся избежать этой проблемы, позволяя пользователю писать *гигиенические макросы*, где конфликты возникнуть не могут. Однако развертывание оказывается сложнее,чем простая подстановка одного выражения вместо другого. Здесь мы не будем пытаться объяснить, в чем проблема и как она решается, а просто используем решение, описанное в Справочном руководстве по MIT/GNU Scheme [51].

4.2.4 Вызов, управляемый образцами

Запускатель правил `try-rules` можно также использовать в процедурах, которые с помощью правил организуют диспетчеризацию по свойствам своих входных данных. Аргументы символа операции сопоставляются с образцом, соответствующим этому символу. Следствие сопоставленного правила вычисляет возвращаемое значение.

Например, можно написать традиционную процедуру вычисления факториала, следующим образом распределив условия:

```
(define factorial
  (make-pattern-operator
   (rule '(0) 1)
   (rule `((? n ,positive?))
         (* n (factorial (- n 1))))))

(factorial 10)
3628800
```

С помощью этого механизма можно также строить процедуры, чье поведение зависит от числа заданных аргументов. Например, в Lisp символ - означает смену знака, если он применен к одному аргументу, и отрицание, если к нескольким:

```
(define -
  (make-pattern-operator
   (rule '((? x)) (n:- 0 x))
   (rule '((? x) (?? y)) (n:- x (apply n:+ y)))))
```

Можно позволить операции, управляемой образцом, динамически расширяться добавлением новых правил. Такие операции аналогичны полиморфным процедурам и позволяют программисту нелокально распределять определения правил по тексту программы. Например, в компиляторе, в оптимизаторе последовательностей команд может быть полезно группировать разные оптимизации с различными фрагментами генератора кода:

```
(define peephole (make-pattern-operator))

(attach-rule! peephole
  (rule '((push (? reg1))
          (pop (? reg2)))
        (if (eqv? reg1 reg2)
            '()
            `((move ,reg1 ,reg2)))))
```

```
(attach-rule!
  (rule `((or (? reg) (? const1 ,unsigned-integer?))
          (or (? reg) (? const2 ,unsigned-integer?)))
        '((or ,reg
              ,(bitwise-or const1 const2)))))
```

Первое из этих правил может встретиться в той части оптимизатора, которая заведует потоком управления, а второе — в части, относящейся к арифметике логических операций.

Вот как можно реализовать операции, управляемые образцами. Последнее правило, передаваемое в `make-pattern-operator`, — это *правило по умолчанию*. Оно всегда проверяется в последнюю очередь независимо от того, какие правила будут добавлены впоследствии.

```
(define (make-pattern-operator . rules)
  (let ((rules
         (cons 'rules
               (if (pair? rules)
                   (except-last-pair rules)
                   '())))
        (default-rule
         (and (pair? rules)
              (last rules))))
    (define (the-operator . data)
      (define (succeed value fail) value)
      (define (fail)
        (error "Нет применимой операции:" data))
      (try-rules data
                 (cdr rules)
                 succeed
                 (if default-rule
                     (lambda ()
                       (try-rule data
                                 default-rule
                                 succeed
                                 fail))
                     fail)))
    (set-pattern-metadata! the-operator rules)
    the-operator))
```

С помощью `set-pattern-metadata!` мы прикрепляем список правил как «ярлык» к символу операции, а в последующем коде извлекаем его через вызов процедуры `pattern-metadata`. Имеются процедуры для добавления правил в начало (`override-rule!`) и в конец (`attach-rule!`) списка.

```
(define (attach-rule! operator rule)
  (let ((metadata (pattern-metadata operator)))
    (set-cdr! metadata
              (append (cdr metadata)
```

```
                    (list rule)))))

(define (override-rule! operator rule)
  (let ((metadata (pattern-metadata operator)))
    (set-cdr! metadata
              (cons rule (cdr metadata)))))
```

4.3 Устройство сопоставителя

Теперь, когда мы увидели выразительную силу сопоставления с образцом, пора рассмотреть, как оно работает. Наша система строится на основе семейства элементарных процедур сопоставления (*сопоставителей*) и комбинаторов, позволяющих строить составные сопоставители[3]. Элементарные компоненты образца соответствуют элементарным сопоставителям, а единственная составная конструкция, список, представляется в виде комбинатора, который собирает общий сопоставитель из сопоставителей, представляющих элементы списка.

Все процедуры сопоставления принимают на вход три аргумента: список, содержащий данные, подлежащие сопоставлению, словарь связываний для переменных образца и процедуру продолжения (`succeed`), которую нужно позвать при успешном сопоставлении. Аргументами `succeed` служат новый словарь, полученный в результате сопоставления, и число элементов, использованных во входном списке. С помощью этого числа мы будем определять, что делать дальше после возврата из сопоставления с сегментом. При неудаче процедура сопоставления возвращает `#f`.

Есть три элементарных процедуры сопоставления и один комбинатор. Мы рассмотрим их по очереди. Кроме того, нам потребуется маленькая процедура, которую можно передать сопоставителю в качестве аргумента `succeed`, чтобы она сообщала о результате.

```
(define (result-receiver dict n-eaten)
  `(success ,(match:bindings dict) ,n-eaten))
```

Образцы-константы

Процедура `match:eqv` принимает на вход образец-константу, например `x`, и порождает процедуру сопоставления `eqv-match`, которая удачна тогда и только тогда, когда первый элемент данных равен (в смысле `eqv?`) этой константе. Она ничего не добавляет к словарю. Второй аргумент вызова `succeed` — это число сопоставленных элементов. Для нашей текущей процедуры оно равно 1.

```
(define (match:eqv pattern-constant)
  (define (eqv-match data dictionary succeed)
    (and (pair? data)
         (eqv? (car data) pattern-constant)
         (succeed dictionary 1)))
  eqv-match)
```

Например,

[3] Эта стратегия построения программ сопоставления впервые описана в диссертации Карла Хьюитта [56].

```
(define x-matcher (match:eqv 'x))

(x-matcher '(x) (match:new-dict) result-receiver)
(success () 1)

(x-matcher '(y) (match:new-dict) result-receiver)
#f
```

Элементные переменные

Процедура `match:element` нужна, чтобы создать процедуру сопоставления `element-match` для переменной образца вроде `(? x)`, которая должна соответствовать одному элементу списка данных.

При сопоставлении с *элементной переменной* есть две возможности: либо у переменной уже есть значение, либо еще нет. Если у нее есть значение, оно присутствует в словаре связываний. В этом случае сопоставление удачно тогда и только тогда, когда связанное значение равно (в смысле `equal?`) первому элементу списка данных. Если у переменной нет значения, она не связана. В этом случае сопоставление удачно, а словарь расширяется путем добавления связывания между переменной и первым элементом списка данных. В обоих успешных вариантах мы указываем, что число потребленных нами элементов равно 1.

```
(define (match:element variable)
  (define (element-match data dictionary succeed)
    (and (pair? data)
         (let ((binding (match:lookup variable dictionary)))
           (if binding
               (and (equal? (match:binding-value binding)
                            (car data))
                    (succeed dictionary 1))
               (succeed (match:extend-dict variable
                                           (car data)
                                           dictionary)
                        1)))))
  element-match)
```

Вот несколько примеров. Связывание при сопоставлении представляет собой список: первый его элемент — имя переменной; второй элемент — значение, а третий — тип переменной (здесь все переменные элементные, что обозначается как `?`).

```
((match:element '(? x))
 '(a) (match:new-dict) result-receiver)
(success ((x a ?)) 1)

((match:element '(? x))
 '(a b) (match:new-dict) result-receiver)
(success ((x a ?)) 1)

((match:element '(? x))
 '((a b) c) (match:new-dict) result-receiver)
(success ((x (a b) ?)) 1)
```

Сегментные переменные

Процедура `match:segment` служит для построения процедуры сопоставления `segment-match` для переменной образца вроде `(?? x)`, которая должна сопоставляться с последовательностью элементов данных. Сопоставитель для сегментных переменных сложнее, чем для элементных, поскольку он съедает неизвестное заранее число элементов. Поэтому сегментный сопоставитель должен сообщать вызывающей процедуре не только новый словарь, но и число потребленных элементов входных данных.

При сопоставлении с сегментной переменной есть две возможности: либо у нее уже есть значение, либо пока еще нет. Если у переменной есть значение, оно должно сопоставляться с данными; это проверяет процедура `match:segment-equal?` со стр. 159. Если же у сегментной переменной еще нет значения, его ей надо присвоить.

Сопоставитель `segment-match`, возвращаемый из `match:segment`, оказывается успешным, а возможным значением, присваиваемым сегментной переменной, оказывается некоторый начальный подсписок входных данных `(list-head data i)`. (Мы начинаем с `i` = 0, предполагая, что сегмент не потребляет ни одного элемента входных данных.) Однако, если текущий успех затем приведет к неудаче сопоставления, сопоставитель попробует взять на один элемент больше, чем до сих пор (выполняя `(lp (+ i 1))`). Если у сегментного сопоставителя кончаются данные на входе, он терпит неудачу. Это основной компонент перебора с возвратами, который становится необходим при наличии сегментных переменных.

```
(define (match:segment variable)
  (define (segment-match data dictionary succeed)
    (and (list? data)
         (let ((binding (match:lookup variable dictionary)))
           (if binding
               (match:segment-equal?
                data
                (match:binding-value binding)
                (lambda (n) (succeed dictionary n)))
               (let ((n (length data)))
                 (let lp ((i 0))
                   (and (<= i n)
                        (or (succeed (match:extend-dict
                                      variable
                                      (list-head data i)
                                      dictionary)
                                     i)
                            (lp (+ i 1)))))))))))
  segment-match)
```

Например,

```
((match:segment '(?? a))
 '(z z z) (match:new-dict) result-receiver)
(success ((a () ??)) 0)
```

Разумеется, в такой ситуации пустой сегмент приемлем.

Если нужно увидеть все возможные варианты сопоставления, мы можем после распечатки вернуть из процедуры обработки результата `#f`. Это заставит процедуру-сопоставитель выдать нам следующую альтернативу, если она есть.

```
(define (print-all-results dict n-eaten)
  (pp `(success ,(match:bindings dict) ,n-eaten))
  ;; возвращая #f, мы заставляем работать перебор с возвратами
  #f)

((match:segment '(?? a))
 '(z z z) (match:new-dict) print-all-results)
(success ((a () ??)) 0)
(success ((a (z) ??)) 1)
(success ((a (z z) ??)) 2)
(success ((a (z z z) ??)) 3)
#f
```

Вернемся теперь к случаю, когда у сегментной переменной уже есть значение. Нам нужно убедиться, что это значение соответствует начальному сегменту наших входных данных. Это обрабатывает процедура `match:segment-equal?`. Она сравнивает элементы значения переменной с входными данными. При успехе она вызывает `ok` (процедуру, переданную ей вторым аргументом), передавая ей число потребленных входных элементов (оно должно равняться длине значения переменной); при неудаче возвращается `#f`.

```
(define (match:segment-equal? data value ok)
  (let lp ((data data) (value value) (n 0))
    (cond ((pair? value)
           (if (and (pair? data)
                    (equal? (car data) (car value)))
               (lp (cdr data) (cdr value) (+ n 1))
               #f))
          ((null? value) (ok n))
          (else #f))))
```

Сопоставление списков

Наконец, у нас есть комбинатор `match:list`, который принимает список процедур сопоставления и порождает процедуру сопоставления, которая сопоставляется со списком элементов данных тогда и только тогда, когда данные сопоставители потребляют все элементы входного списка. Этот комбинатор последовательно применяет все сопоставители. Каждый сопоставитель сообщает комбинатору, на сколько элементов надо прыгнуть, прежде чем передать остаток данных следующему сопоставителю.

```
(define (match:list matchers)
  (define (list-match data dictionary succeed)
    (and (pair? data)
         (let lp ((data-list (car data))
                  (matchers matchers)
```

```
                (dictionary dictionary))
           (cond ((pair? matchers)
                  ((car matchers)
                   data-list
                   dictionary
                   (lambda (new-dictionary n)
                     ;; Сущность сопоставления со списком:
                     (lp (list-tail data-list n)
                         (cdr matchers)
                         new-dictionary))))
                 ((pair? data-list) #f) ; несопоставленные данные
                 ((null? data-list) (succeed dictionary 1))
                 (else #f)))))
  list-match)
```

Заметим, что интерфейс сопоставителя `list-match`, возвращаемого комбинатором `match:list`, в точности совпадает с интерфейсом других сопоставителей, и таким образом, его тоже можно включать в комбинации. Именно то, что все базовые процедуры сопоставления имеют один и тот же интерфейс, делает нашу систему системой комбинаторов.

Теперь мы можем построить сопоставитель, который распознает список с произвольным числом элементов, начинающийся с символа `a`, заканчивающийся символом `b`, с сегментной переменной `(?? x)` между ними:

```
((match:list (list (match:eqv 'a)
                   (match:segment '(?? x))
                   (match:eqv 'b)))
 '((a 1 2 b))
 (match:new-dict)
 result-receiver)
(success ((x (1 2)) ??)) 1)
```

Сопоставление было удачным. Возвращаемый словарь имеет ровно один элемент: `x = (1 2)`. Сопоставление потребило ровно один элемент (список `(a 1 2 b)`) из списка входных данных.

```
((match:list (list (match:eqv 'a)
                   (match:segment '(?? x))
                   (match:eqv 'b)))
 '((a 1 2 b 3))
 (match:new-dict)
 result-receiver)
#f
```

Здесь мы потерпели неудачу, потому что элементу 3 после `b` во входных данных ничего не соответствует.

Словарь

В качестве словаря мы будем использовать просто список связываний с заголовком. Каждое связывание является списком из имени переменной, ее значения и типа.

```
(define (match:new-dict)
  (list 'dict))

(define (match:bindings dict)
  (cdr dict))

(define (match:new-bindings dict bindings)
  (cons 'dict bindings))

(define (match:extend-dict var value dict)
  (match:new-bindings dict
                      (cons (match:make-binding var value)
                      (match:bindings dict))))

(define (match:lookup var dict)
  (let ((name
         (if (symbol? var)
             var
             (match:var-name var))))
    (find (lambda (binding)
            (eq? name (match:binding-name binding)))
          (match:bindings dict))))

(define (match:make-binding var value)
  (list (match:var-name var)
        value
        (match:var-type var)))

(define match:binding-name car)
(define match:binding-type caddr)

(define match:binding-value
  (simple-generic-procedure 'match:binding-value 1 cadr))
```

Процедура доступа `match:binding-value` равна просто `cadr`. Однако мы сделали ее полиморфной, чтобы впоследствии ее можно было расширить. Это нам потребуется при поддержке кода из раздела 4.5.

4.3.1 Компиляция образцов

Можно автоматизировать преобразование образцов в сопоставители с помощью примитивного компилятора. Компилятор выдает в качестве результата процедуру сопоставления, соответствующую поданному ему на вход образцу. Эта процедура имеет в точности тот же самый интерфейс, как и элементарные сопоставители, описанные выше.

Интерфейс процедуры-сопоставителя удобен для композиции сопоставителей, но он не очень дружественный для пользователя. Для упражнений и тестов удобнее использовать:

```
(define (run-matcher match-procedure datum succeed)
  (match-procedure (list datum)
                   (match:new-dict)
                   (lambda (dict n)
                     (and (= n 1)
                          (succeed dict)))))
```

При таком интерфейсе мы скрываем несколько деталей процедур сопоставления: оборачиваем входной элемент данных в список; проверяем, что сопоставлен ровно один элемент этого списка (входные данные); порождаем начальное состояние словаря.

Вот несколько простых примеров:

```
(run-matcher
 (match:compile-pattern '(a ((? b) 2 3) (? b) c))
 '(a (1 2 3) 2 c)
 match:bindings)
#f

(run-matcher
 (match:compile-pattern '(a ((? b) 2 3) (? b) c))
 '(a (1 2 3) 1 c)
 match:bindings)
((b 1 ?))
```

Как мы уже видели, образцы, включающие в себя сегментные переменные, могут сопоставляться несколькими способами. Можно получить все варианты сопоставления, если искусственно породить неудачу и вызвать поиск следующего варианта, — и так, пока не будут перебраны все:

```
(run-matcher
 (match:compile-pattern '(a (?? x) (?? y) (?? x) c))
 '(a b b b b b b c)
 print-all-matches)
((y (b b b b b b) ??) (x () ??))
((y (b b b b) ??) (x (b) ??))
((y (b b) ??) (x (b b) ??))
((y () ??) (x (b b b) ??))
#f
```

Во всех вариантах здесь требуется, чтобы оба экземпляра `(?? x)` сопоставлялись с одинаковыми значениями.

Процедура `print-all-matches` распечатывает связывания переменных и порождает неудачу:

```
(define (print-all-matches dict)
  (pp (match:bindings dict))
  #f)
```

В нашем компиляторе требуется определить синтаксис переменных образца. У нас пока синтаксис очень простой: переменная представляет собой список из значка типа (`?` или `??`) и имени.

```
(define (match:var-type var)
  (car var))

(define (match:var-type? object)
  (memq object match:var-types))

(define match:var-types '(? ??))

(define (match:named-var? object)
  (and (pair? object)
       (match:var-type? (car object))
       (n:>= (length object) 2)
       (symbol? (cadr object))))

(define (match:element-var? object)
  (and (match:var? object)
       (eq? '? (match:var-type object))))

(define (match:segment-var? object)
  (and (match:var? object)
       (eq? '?? (match:var-type object))))
```

Этот код сложнее, чем можно было бы ожидать, поскольку мы будем расширять синтаксис переменных в разделе 4.5 и в некоторых упражнениях.

```
(define match:var-name
  (simple-generic-procedure 'match:var-name 1
    (constant-generic-procedure-handler #f)))

(define-generic-procedure-handler match:var-name
  (match-args match:named-var?)
  cadr)
```

Обработчик по умолчанию всегда возвращает ложь. Пока что у нас только один осмысленный обработчик, возвращающий имя переменной.

Кроме того, мы делаем полиморфным предикат, определяющий, является ли аргумент переменной сопоставления, хотя пока что этому предикату удовлетворяют только переменные с именами:

```
(define match:var?
  (simple-generic-procedure 'match:var? 1
    (constant-generic-procedure-handler #f)))

(define-generic-procedure-handler match:var?
  (match-args match:named-var?)
  (constant-generic-procedure-handler #t))
```

Компилятор преобразует образец в соответствующий сопоставитель:

```
(define (match:compile-pattern pattern)
  (cond ((match:var? pattern)
```

```
         (case (match:var-type pattern)
           ((?) (match:element pattern))
           ((??) (match:segment pattern))
           (else (error "Неизвестный тип переменной:" pattern))))
        ((list? pattern)
         (match:list (map match:compile-pattern pattern)))
        (else   ; константа
         (match:eqv pattern))))
```

Изменяя этот компилятор, мы можем варьировать синтаксис образцов любым угодным нам способом.

```
(run-matcher
 (match:compile-pattern '(a ((? b) 2 3) (? b) c))
 '(a (1 2 3) 1 c)
 match:bindings)
((b 1 ?))
```

Упражнение 4.5. Перебор с возвратами
В примере на стр. 162 мы получили несколько вариантов сопоставления, возвращая `#f` из процедуры обработки успеха `print-all-matches`. Это поведение может показаться таинственным. Как оно работает? Коротко и ясно объясните, как порождается последовательность сопоставлений (примерно на один абзац).

4.3.2 Ограничения на переменные сопоставления

Часто нам хочется ограничить, с какими видами объектов может сопоставляться переменная образца. Например, может потребоваться образец, где переменная соответствует только положительным целым числам. Один из способов этого добиться, который мы использовали в системе переписывания термов в разделе 4.2.1, — позволить переменной содержать в себе предикат, проверяющий данные на приемлемость. Например, пусть нас интересуют положительные целые степени функции синус. Можно записать требуемый образец так:

```
`(expt (sin (? x)) (? n ,exact-positive-integer?))
```

В таком образце нельзя использовать простую кавычку, потому что выражение-предикат (здесь это `exact-positive-integer?`) требуется вычислить, прежде чем включать его в образец. Как и при переписывании термов (раздел 4.2), мы для этого пользуемся механизмом обратной кавычки.

Чтобы создать сопоставитель, способный проверять, удовлетворяют ли данные такому предикату, нужно добавить к `match:element` одну строчку:

```
(define (match:element variable)
  (define (element-match data dictionary succeed)
    (and (pair? data)
         (match:satisfies-restriction? variable (car data))
         (let ((binding (match:lookup variable dictionary)))
           (if binding
               (and (equal? (match:binding-value binding)
```

```
                              (car data))
                         (succeed dictionary 1))
                   (succeed (match:extend-dict variable
                                               (car data)
                                               dictionary)
                            1)))))
  element-match)

(define (match:satisfies-restriction? var value)
  (or (not (match:var-has-restriction? var))
      ((match:var-restriction var) value)))
```

Упражнение 4.6. Альтернативы в образцах: хорошо иметь выбор
Интересный способ расширить наш язык образцов — добавить операцию выбора:

`(?:choice` *образец* `...)`

Такое выражение должно компилироваться в сопоставитель, который пытается по очереди слева направо сопоставить каждый из данных ему *образцов* и возвращает первый удачный вариант либо `#f` при неудаче. (Это должно вам напомнить «альтернативу» в регулярных выражениях (см. стр. 47), но в сопоставлении с образцом чаще используют термин «выбор».)

Например:

```
(run-matcher
 (match:compile-pattern '(?:choice a b (? x) c))
 'z
 match:bindings)
((x z ?))

(run-matcher
 (match:compile-pattern
 `((? y) (?:choice a b (? x ,string?) (? y ,symbol?) c)))
 '(z z)
 match:bindings)
((y z ?))

(run-matcher
 (match:compile-pattern `(?:choice b (? x ,symbol?)))
 'b
 print-all-matches)
()
((x b ?))
#f
```

Задание. Реализуйте новую процедуру-сопоставитель `match:choice` для этой схемы образцов. Дополните нужным образом компилятор образцов.

Упражнение 4.7. Именование образцов
Еще одно возможное расширение — именованные образцы, подобно `letrec` в Scheme.

Именование позволяет строить более короткие и модульные образцы, а также поддерживает рекурсивные подобразцы, включая взаимную рекурсию.

Например, образец

```
(?:pletrec ((odd-even-etc (?:choice () (1 (?:ref even-odd-etc))))
            (even-odd-etc (?:choice () (2 (?:ref odd-even-etc)))))
    (?:ref odd-even-etc))
```

должен сопоставляться со всеми списками следующего вида (включая пустой список):

```
(1 (2 (1 (2 (1 ...)))))
```

Выражение `?:pletrec` вводит блок взаимно рекурсивных определений образцов, а `?:ref` подставляет определенный таким образом образец (нужно отличать такие ссылки от константных символов вроде `a` и от переменных образца вроде `(? x)`).

Что делать. Реализуйте новые схемы `?:pletrec` и `?:ref`. Один из возможных подходов — реализовать новые процедуры-сопоставители `match:pletrec` и `match:ref`, а затем расширить под них компилятор образцов. Могут сработать и другие варианты реализации. Если ваш подход устроен нетривиально, объясните его тонкости.

О чем подумать (до того, как действовать!). В правильной реализации, основанной, подобно `letrec`, на окружениях, вложенные экземпляры `?:pletrec` вводили бы раздельные контуры областей действия. Однако в структуре управления нашего сопоставителя это сложно организовать.

Процедуры сопоставления просматривают образцы и данные в порядке слева направо, связывают текстуально первое вхождение каждой отдельной переменной образца (вроде `(? x)`) с соответствующими данными, а затем считают, что каждое последующее вхождение той же переменной в образце проверяет данные на равенство. Это достигается за счет того, что словарь протаскивается сквозь перебор с возвратами в глубину. Обратите особенное внимание, как в теле `match:list` используется переменная `new-dictionary`. В сущности, такая структура управления диктует, что самое левое, самое глубокое вхождение каждой переменной образца определяет ее значение в неявном глобальном пространстве имен, а все последующие вхождения проверяют это значение.

Давайте поэтому в этом упражнении не будем заботиться о сложностях с областями действия. А именно, как переменные образца живут в одной глобальной области действия, так же пусть будет и с определениями именованных образцов.

Разумеется, если у вас достаточно амбиций, вы можете реализовать лексические сферы действия, переписав все имеющиеся интерфейсы сопоставителей и добавив в них дополнительный параметр-окружение `pattern-environment`. Однако это задача на другой раз (упражнение 4.9).

Упражнение 4.8. Свалились в собственную яму

Казалось бы, должно быть несложно расширить нашу систему сопоставителей, чтобы она могла работать с векторами и образцами, представляющими векторы. Однако при проектировании системы сопоставления мы сделали сильное предположение, что это будет сопоставитель для списков и списочных образцов.

a. Насколько легко будет освободиться от этого предположения и построить сопоставитель, способный работать с обоими видами составных данных? С произвольными последовательностями? Какого рода изменения потребуются? Как надо будет модифицировать интерфейс процедуры-сопоставителя?

b. Сделайте это! (Потребуется очень много работы.)

Упражнение 4.9. Общий язык сопоставления

Даже после добавления в упражнении 4.7 конструкций `?:pletrec` и `?:pref` наш сопоставитель не обладает полноценным языком, поскольку не поддерживает систему областей действия и параметрические образцы. Например, невозможно написать следующий образец, который должен сопоставляться со списками символов, представляющими собой палиндромы:

```
(?:pletrec ((palindrome
             (?:pnew (x)
               (?:choice ()
                         ((? x ,symbol?)
                          (?:ref palindrome)
                          (? x))))))
  (?:ref palindrome))
```

Чтобы это работало сколько-нибудь разумно, конструкция `?:pnew` должна создавать переменные образца с лексической областью действия, и к ним должно быть возможно обращаться только в теле `?:pnew`.

Полноценный язык образцов — замечательно мощная система, но построить ее нелегко.

Что делать. Воплотите эту идею и постройте полный язык образцов для сопоставления. Задача не для слабых духом!

4.4 Унификация

Как мы уже говорили, сопоставление с образцом — это разновидность проверки на равенство для структурированных данных. Это обобщение простой проверки на равенство, поскольку некоторые части структуры мы оставляем незаданными и позволяем с ними сопоставляться переменным образца. Однако мы требуем, чтобы все вхождения определенной переменной соотносились с эквивалентными данными.

Наши сопоставители до сих пор были односторонними: мы сопоставляем образец, содержащий переменные, с данными, где переменных нет, и получаем словарь — отображение переменных образца на соответствующую подструктуру данных. Если мы подставим в исходный образец полученные значения переменных, то получим *экземпляр подстановки* этого образца. Этот экземпляр всегда равен исходным данным.

Теперь мы хотим устранить ограничение, говорящее, что данные переменных не содержат: мы разрешаем переменным встречаться на обеих сторонах сопоставления. Такое важное расширение понятия сопоставления называется *унификация*. В результате успешной унификации также получается словарь, однако значения переменных могут содержать переменные, и словарь не обязательно содержит значения для всех переменных, встречающихся в образцах. Если мы подставим значения, находящиеся в словаре, на место переменных в обоих образцах, мы получим экземпляр подстановки для обоих исходных образцов. Этот экземпляр подстановки, который может содержать переменные, называется *унификатором* этих образцов. Унификатор является наиболее общим совместным экземпляром подстановки двух образцов: любой другой совместный экземпляр подстановки будет экземпляром подстановки унификатора. Унификатор единственен с точностью до переименования переменных, поэтому это понятие правильно определено[4].

Унификация была впервые описана Дж. А. Робинсоном, когда он изобрел знаменитый метод резолюции для доказательства теорем [104][5].

[4]Унификатор единственен для образцов, которые содержат только элементные переменные. Мы докажем эту теорему в книге. Однако в разделе 4.4.4 мы расширим унификацию и позволим использовать сегментные переменные. Если в унификации содержатся сегментные переменные, то в общем случае могут получаться множественные результаты.

[5]Подробный обзор методов и применений унификации см. в [6].

Рассмотрим простой пример. Пусть у нас есть несколько источников частичной информации о датах рождения и смерти Бенджамина Франклина:

```
(define a
  '(((? gn) franklin) (? bdate) ((? dmo) (? dday) 1790)))

(define b
  '((ben franklin) ((? bmo) 6 1705) (apr 17 (? dyear))))

(define c
  '((ben (? fn)) (jan (? bday) 1705) (apr 17 (? dyear))))
```

При унификации двух выражений мы получим словарь значений переменных, выведенных в сопоставлении. С помощью этого словаря строим унификатор двух образцов:

```
(unifier a b)
((ben franklin) ((? bmo) 6 1705) (apr 17 1790))

(unifier a c)
((ben franklin) (jan (? bday) 1705) (apr 17 1790))

(unifier b c)
((ben franklin) (jan 6 1705) (apr 17 (? dyear)))
```

Каждый из этих результатов определен более точно, чем частичная информация из любого источника. Можно сочетать теперь все три источника и получить полную картину:

```
(unifier a (unifier b c))
((ben franklin) (jan 6 1705) (apr 17 1790))

(unifier b (unifier a c))
((ben franklin) (jan 6 1705) (apr 17 1790))

(unifier c (unifier a b))
((ben franklin) (jan 6 1705) (apr 17 1790))
```

Часто на переменные в выражении накладываются ограничения. Например, могут присутствовать несколько вхождений одной и той же переменной, и они должны оставаться согласованными, как в следующем выводе:

```
(define addition-commutativity
  '(= (+ (? u) (? v)) (+ (? v) (? u))))

(unifier '(= (+ (cos (? a)) (exp (? b))) (? c))
         addition-commutativity)
(= (+ (cos (? a)) (exp (? b))) (+ (exp (? b)) (cos (? a))))
```

4.4.1 Как работает унификация

Унификацию можно рассматривать как своего рода решение уравнений. Если считать два образца структурированными частичными описаниями, их унификация

проверяет утверждение, что это частичные описания одного и того же объекта. Чтобы образцы могли унифицироваться, должны унифицироваться их компоненты. Таким образом, унификация состоит в том, чтобы построить уравнения для компонент и найти неизвестные (фрагменты информации, оставленные незаданными в каждом из образцов).

Этот процесс аналогичен тому, как решаются численные уравнения. Наша цель — избавиться от как можно большего числа переменных в уравнениях. В результате получаем список подстановок подвыражений вместо исключенных переменных. Подставляемым выражениям не разрешается ссылаться на исключенные переменные. Мы просматриваем уравнения в поисках такого, где значение одной из переменных можно найти. Решение уравнения относительно одной из переменных изолирует ее и находит эквивалентное выражение, где эта переменная не содержится. Мы исключаем эту переменную, заменяя все ее вхождения на найденное значение (эквивалентное выражение), которое переменную не содержит. Проделываем это и в оставшихся уравнениях, и в значениях, связанных с уже ранее исключенными переменными в списке подстановок. Затем мы добавляем новую подстановку в список. Этот процесс повторяется, пока либо не закончатся уравнения, подлежащие решению, либо не найдется противоречие. В результате получаем либо построенный список подстановок, либо сообщение о противоречии.

В унификации «уравнения» представляют собой сопоставления соответствующих друг другу частей входных образцов, а «список подстановок» — это словарь. Один из способов реализовать унификацию — обход общей структуры входных образцов. Как и в одностороннем сопоставлении, образцы представляются в виде списковой структуры. Если на какой-либо стороне сопоставления стоит переменная, она связывается в словаре с данными с другой стороны.

Если в исходных образцах присутствует больше одного вхождения переменной, каждое следующее вхождение должно сопоставляться с тем значением, с которым было связано первое вхождение. Это обеспечивается тем, что каждый раз, как мы встречаем переменную, она ищется в словаре, и если находится связывание, оно используется вместо переменной. Кроме того, каждый раз, как создается новое связывание, оно заменяет все вхождения исключаемой переменной в значениях других переменных словаря.

Чтобы получить унификатор `unifier` двух образцов, мы зовем для них процедуру `unify` и получаем словарь подстановок, если унификация успешна. Если нет, `unify` сигнализирует об ошибке, возвращая `#f`. Затем словарь используется для подстановки в один из исходных образцов; какой из двух, не имеет значения. Процедура `match:dict-substitution` заменяет внутри образца `pattern1` все вхождения переменных, имеющих связывание в словаре `dict`, на их значения из этого словаря.

```
(define (unifier pattern1 pattern2)
  (let ((dict (unify pattern1 pattern2)))
    (and dict
         ((match:dict-substitution dict) pattern1))))
```

Основным интерфейсом к унификатору является процедура `unify`, которая при удачном сопоставлении возвращает словарь, а при неудачном `#f`.

```
(define (unify pattern1 pattern2)
  (unify:internal pattern1 pattern2
```

```
(match:new-dict)
(lambda (dict) dict)))
```

Точка входа `unify:internal` дает больше возможностей управления процессом сопоставления. Она принимает два образца, подлежащих унификации, словарь, где могут уже содержаться некоторые связывания, и продолжение успеха `succeed`, которое будет вызвано в случае удачного сопоставления. Продолжение успеха, которое нам дает `unify`, выше в тексте, просто возвращает словарь. В разделе 4.4.4, когда будем экспериментировать с добавлением в образцы сегментных переменных, мы сможем получать несколько результатов сопоставления, возвращая из продолжения `succeed` значение `#f` и обозначая таким образом, что текущий результат нас не устраивает. Возможность возобновить перебор с возвратами внутри процесса сопоставления также упрощает другие интересные семантические расширения, например введение в процесс сопоставления алгебраических выражений и решение уравнений в нем[6].

Внутри `unify:internal` образцы `pattern1` и `pattern2`, подлежащие унификации, оборачиваются в списки. Эти списки программа-унификатор будет просматривать элемент за элементом, строя при этом словарь, который делает соответствующие термы равными друг другу. Сопоставление считается удачным, если оба списка термов одновременно заканчиваются. На верхнем уровне, в `unify:internal`, списки термов содержат только два исходных образца, однако центральная процедура унификации `unify:dispatch` рекурсивно спускается внутрь образцов и сравнивает подобразцы как списки термов[7].

```
(define (unify:internal pattern1 pattern2 dict succeed)
  ((unify:dispatch (list pattern1) (list pattern2))
   dict
   (lambda (dict fail rest1 rest2)
     (or (and (null? rest1) (null? rest2)
              (succeed dict))
         (fail)))
     (lambda () #f)))
```

Процедура `unify:dispatch`, принимающая два списка термов, — ядро сопоставителя, построенное на рекурсивном спуске. Процесс сопоставления зависит от содержимого списков термов. Например, если оба этих списка начинаются с константы, скажем, `(ben franklin)` и `(ben (? fn))`, эти константы следует сравнить. Сопоставление можно продолжить, только если они окажутся равными. Если один из списков термов начинается с переменной, второй может начинаться с любого

[6]Внутри нашего унификатора оказывается удобным при неудаче явным образом вызывать продолжение неудачи. Однако в `unify:internal` мы используем другое соглашение по сигнализации об ошибках: возврат `#f` из продолжения успеха. Это мы делаем для того, чтобы правила использования унификатора были такие же, как правила использования сопоставителя из раздела 4.3.

Это интересное изменение. В системе правил из раздела 4.2.2 у нас были явные продолжения успеха и неудачи, так что, чтобы использовать с этой системе сопоставитель, нам пришлось проделать обратный переход: сопоставитель использовал соглашение с `#f`, так что в процедуре `make-rule` (на стр. 152) проводится преобразование к двум продолжениям.

Какое соглашение использовать при реализации неудачи в системе перебора с возвратами — вопрос стиля. Однако часто систему с продолжением неудачи оказывается удобнее расширять. К счастью, эти два стиля перебора легко преобразовать друг в друга.

[7]Подобно системе сопоставления с образцом из раздела 4.3, программа унификации строится вокруг списков термов, что затем позволит нам расширить ее, введя сегментные переменные.

терма, и переменную следует с этим термом связать. (Если переменные на обеих сторонах, одна из них будет исключена в пользу другой.) Так что если один из списков равен `((? bmo) 6 1705)`, а другой — `(jan (? bday) 1705)`, то переменная `(? bmo)` должна быть связана со значением `jan`, прежде чем сопоставление продолжится. Если оба списка термов начинаются со списков, которые не являются переменными, нужно рекурсивно сопоставить соответствующие подсписки, а затем продолжить сопоставление остатков исходных списков термов. Например, в примере с Франклином, при унификации `b` и `c` после того, как сопоставлены первые термы, словарь содержит связывание `fn` с `franklin`, а остающиеся списки термов равны `(((?bmo) 6 1705) (apr 17 (? dyear)))` и `((jan (? bday) 1705) (apr 17 (? dyear)))`. Оба эти списка термов начинаются со списков, так что требуется рекурсивно сравнить подсписки `((? bmo) 6 1705)` и `(jan (? bday) 1705)`.

Процедура `unify-dispatcher`, возвращаемая как значение из `unify:dispatch`, принимает три аргумента: словарь, продолжение успеха и продолжение неудачи. Если оба списка термов пустые, сопоставление успешно. Если есть еще термы, подлежащие сопоставлению, полиморфная процедура `unify:gdispatch` вызывает соответствующую процедуру-сопоставитель в зависимости от содержимого двух списков термов. Если сопоставление удачно, это означает, что изначальные термы двух списков могут быть унифицированы в соответствии с указанным словарем. В этом случае вызывается продолжение успеха, и ему передаются новый словарь `dict*`, новое продолжение неудачи `fail*` и еще не сопоставленные хвосты исходных списков `rest1` и `rest2`. Эти хвосты будут сопоставлены рекурсивным вызовом `unify:dispatch`.

```
(define (unify:dispatch terms1 terms2)
  (define (unify-dispatcher dict succeed fail)
    (if (and (null? terms1) (null? terms2))
        (succeed dict fail terms1 terms2)
        ((unify:gdispatch terms1 terms2)
         dict
         (lambda (dict* fail* rest1 rest2)
           ((unify:dispatch rest1 rest2)
            dict* succeed fail*))
         fail)))
  unify-dispatcher)
```

Полиморфная процедура `unify:gdispatch` содержит обработчики для описанных выше случаев: сопоставление двух констант, двух списков термов и переменной с чем угодно. (Поскольку процедура полиморфная, ее можно будет расширить для новых видов сопоставления.) Умолчательный обработчик, срабатывающий в случаях вроде сопоставления константы со списком термов, зовет `unify:fail`:

```
(define (unify:fail terms1 terms2)
  (define (unify-fail dict succeed fail)
    (fail))
  unify-fail)

(define unify:gdispatch
  (simple-generic-procedure 'unify 2 unify:fail))
```

В этом унификаторе списки термов сопоставляются элемент за элементом, так что задачей обработчика является сопоставить первые элементы двух списков. Таким образом, применимость обработчика зависит только от первого терма в каждом из списков. Чтобы упростить спецификации применимостей, мы вводим процедуру `car-satisfies`, которая принимает предикат и порождает новый предикат, проверяющий, что в списке имеется первый терм и что этот терм удовлетворяет предикату-аргументу.

```
(define (car-satisfies pred)
  (lambda (terms)
    (and (pair? terms)
         (pred (car terms)))))
```

Всякий терм, не являющийся переменной сопоставления или списком, считается константой. Константы сопоставляются между собой только тогда, когда они равны.

```
(define (unify:constant-terms terms1 terms2)
  (let ((first1 (car terms1)) (rest1 (cdr terms1))
        (first2 (car terms2)) (rest2 (cdr terms2)))
    (define (unify-constants dict succeed fail)
      (if (eqv? first1 first2)
          (succeed dict fail rest1 rest2)
          (fail)))
    unify-constants))

(define (constant-term? term)
  (and (not (match:var? term))
       (not (list? term))))

(define-generic-procedure-handler unify:gdispatch
  (match-args (car-satisfies constant-term?)
              (car-satisfies constant-term?))
  unify:constant-terms)
```

Обработчик `unify:list-terms` — это место, где, собственно, происходит рекурсивный спуск. Поскольку в каждом списке термов первый элемент сам является списком, нужно рекурсивно вызвать сопоставитель для унификации этих подсписков. Если сопоставление подсписков оказывается удачным, нужно продолжить сопоставление с остатками входных списков термов. (Заметим, что рекурсивный вызов сопоставления будет удачен, только если сопоставляются все термы в подсписках; таким образом, хвосты — необработанные подсписки, передаваемые в продолжение успеха — будут пусты, и их можно игнорировать.)

```
(define (unify:list-terms terms1 terms2)
  (let ((first1 (car terms1)) (rest1 (cdr terms1))
        (first2 (car terms2)) (rest2 (cdr terms2)))
    (define (unify-lists dict succeed fail)
      ((unify:dispatch first1 first2)
       dict
       (lambda (dict* fail* null1 null2)
```

```
        (succeed dict* fail* rest1 rest2))
      fail))
  unify-lists))

(define (list-term? term)
  (and (not (match:var? term))
       (list? term)))

(define-generic-procedure-handler unify:gdispatch
  (match-args (car-satisfies list-term?)
              (car-satisfies list-term?))
  unify:list-terms)
```

Пока что наш код реализует рекурсивный спуск и сопоставление констант. Очевидные противоречия вроде сопоставления двух различных символов или символа со списком порождают неудачу. Чтобы решать интересные уравнения, нужно уметь сопоставлять термы с переменными. Когда мы встречаем переменные, нужно добавлять новые связывания в словарь. Часть решателя уравнений, отвечающая за работу с переменными, называется `maybe-substitute`.

Процедура `maybe-substitute` принимает на вход список термов `var-first`, который должен начинаться с переменной. Она сопоставляет эту переменную с первым элементом второго списка термов, `terms`.

Если переменная сопоставляется сама с собой, у нас получается тавтология, сопоставление считается успешным, а словарь не меняется. Если у переменной уже есть значение, мы заменяем переменную на это значение и сопоставляем получившийся список со списком термов `terms`. Наконец, если значения у переменной нет, мы можем ее исключить с помощью процедуры `do-substitute`, отвечающей за добавление связывания переменной `var` с термом `term`, когда это возможно.

```
(define (maybe-substitute var-first terms)
  (define (unify-substitute dict succeed fail)
    (let ((var (car var-first)) (rest1 (cdr var-first))
          (term (car terms)) (rest2 (cdr terms)))
      (cond ((and (match:element-var? term)
                  (match:vars-equal? var term))
             (succeed dict fail rest1 rest2))
            ((match:has-binding? var dict)
             ((unify:dispatch
                 (cons (match:get-value var dict) rest1)
                 terms)
              dict succeed fail))
            (else
             (let ((dict* (do-substitute var term dict)))
               (if dict*
                   (succeed dict* fail rest1 rest2)
                   (fail)))))))
  unify-substitute)
```

В процедуре `do-substitute` мы первым делом чистим входной терм `term`, заменяя все уже исключенные переменные на их значения из старого словаря. Затем

смотрим, есть ли ограничения, с какими объектами разрешено сопоставляться переменной `var`. Наконец, проверяем, не встречается ли ссылок на `var` в очищенном терме `term*`. Если да, сопоставление продолжаться не может[8]. Если наше сопоставление проходит все эти проверки, создаем новый словарь, включающий новое связывание переменной `var` с очищенным термом. Остальные значения связываний в новом словаре также требуется очистить от вхождений `var`.

```
(define (do-substitute var term dict)
  (let ((term* ((match:dict-substitution dict) term)))
    (and (match:satisfies-restriction? var term*)
         (or (and (match:var? term*)
                  (match:vars-equal? var term*))
             (not (match:occurs-in? var term*)))
         (match:extend-dict var term*
           (match:map-dict-values
            (match:single-substitution var term*)
            dict)))))
```

Теперь, когда мы умеем обращаться с переменными, нужно только установить обработчик в полиморфную процедуру-диспетчер. Тут единственная тонкость в том, что исключаемая переменная может находиться в любом из списков термов. Нужно убедиться, что список термов, содержащий переменную, служит первым аргументом вызова `maybe-substitute`.

```
(define (element? term)
  (any-object? term))

(define-generic-procedure-handler unify:gdispatch
  (match-args (car-satisfies match:element-var?)
              (car-satisfies element?))
  (lambda (var-first terms)
    (maybe-substitute var-first terms)))

(define-generic-procedure-handler unify:gdispatch
  (match-args (car-satisfies element?)
              (car-satisfies match:element-var?))
  (lambda (terms var-first)
    (maybe-substitute var-first terms)))
```

К этому моменту у нас есть полный правильно работающий и грамотно построенный традиционный унификатор[9]. Он использует полиморфные процедуры и поэтому может легко быть расширен для работы с другими видами данных. И лишь с

[8] В литературе по унификации это называется «проверка на вхождение». Проверка на вхождение предотвращает попытки найти решение в уравнениях вроде $x = f(x)$. В некоторых случаях, если мы что-то знаем о функции f, такие уравнения с неподвижной точкой можно решить, но наш унификатор — программа *синтаксического* сопоставления. Можно было бы оставить в этом месте крючок для более мощной системы решения уравнений, но мы так делать не будем. В большинстве реализаций языка Prolog проверка на вхождение не производится из соображений эффективности.

[9] Поскольку понятие унификации настолько важное, существует множество работ, где разрабатываются эффективные алгоритмы для нее. Больших улучшений можно добиться с помощью мемоизации. Подробный разбор алгоритмов унификации можно найти в [6].

небольшими усилиями можно добавить к нему семантические дополнения, например коммутативность списков, начинающихся с + и *. Как мы вскоре убедимся, можно также добавить поддержку новых видов синтаксических переменных, например сегментных. Однако прежде, чем заняться этим, посмотрим на реальное приложение: вывод типов.

Упражнение 4.10. Унификация векторов
Наш унификатор можно расширить, чтобы он мог обрабатывать данные и образцы, включающие другие типы данных, например векторы. Напишите обработчики для векторов, не преобразуя их предварительно в списки (это было бы слишком просто!).

Упражнение 4.11. Унификация строк
Дополните унификатор так, чтобы он мог работать на строках. Это может быть красиво и полезно, однако вам придется придумать синтаксический механизм для представления строковых переменных внутри строк. Это достаточно тонкая работа, поскольку может потребоваться представлять строковую переменную с помощью строкового выражения. Возникают проблемы с закавычиванием; пожалуйста, постарайтесь не изобретать громоздких механизмов. Попробуйте также избежать предположений, которые потом не дадут вам добавить строковые сегментные переменные (см. упражнение 4.21 на стр. 188).

Упражнение 4.12. Ограничения на переменные
Мы поддержали ограничения на переменные, так же как мы это сделали в одностороннем сопоставителе: добавили проверку в главном условии процедуры `do-substitute`. Однако тут есть сложности.

- Что случится, если ограниченная переменная сопоставляется с другой ограниченной переменной?
- Если переменная сначала связывается с выражением, пройдя проверку на ограничение, все ее вхождения убираются. Но при этом ограничение теряется и не может потом отсечь неподходящее вхождение на более поздней стадии.

Вашей задачей будет понять, в чем состоят эти сложности, и решить, как с ними справиться. Насколько эти проблемы будут важны в применениях унификации, которые вы для себя предвидите? Есть ли решение, хорошо ложащееся в нашу стратегию реализации?

Упражнение 4.13. Унификация с помощью комбинаторов образцов?
В отличие от одностороннего сопоставителя из предыдущего раздела, наша процедура унификации не компилирует образцы в процедуры сопоставления, которые сочетаются и образуют процедуру сопоставления для составного образца. Однако система с процедурами сопоставления потенциально эффективнее, поскольку при этом во время сопоставления не происходит повторного синтаксического анализа образцов. Возможно ли построить подобным образом систему унификации? Если нет, почему? Если да, будет ли хорошей идеей сделать так? Если нет, почему? Если да, реализуйте эту идею! (Это сложная задача!)

4.4.2 Приложение: вывод типов

Одно из классических применений унификации — вывод типов: дана программа и некоторая дополнительная информация о частях этой программы. Требуется вывести информацию о типах в других частях программы. Например, если мы знаем,

что процедура < принимает два числовых аргумента и порождает булево значение, то, анализируя выражение (g (< x (f y))), мы можем заключить, что процедуры f и g одноместные, g принимает булев аргумент, f порождает числовое значение, и переменная x также имеет числовое значение. Если использовать эту информацию, чтобы вывести свойства программы, где встречается наше выражение, о программе можно узнать очень много. Вот анализ этого выражения:

```
(pp (infer-program-types '(g (< x (f y)))))
(t (? type:17)
   ((t (type:procedure ((boolean-type)) (? type:17)) g)
    (t (boolean-type)
       ((t (type:procedure ((numeric-type) (numeric-type))
                           (boolean-type))
           <)
       (t (numeric-type) x)
       (t (numeric-type)
          ((t (type:procedure ((? y:12)) (numeric-type)) f)
           (t (? y:12) y)))))))
```

Это абстрактное дерево данного выражения с аннотациями типов. Каждое подвыражение x заменено на *типизированное выражение* вида (t *тип* x). Например, ссылка на g имеет тип

```
(type:procedure ((boolean-type)) (? type:17))
```

Как и ожидалось, g является процедурой, принимающей булев аргумент, но информации о ее возвращаемом значении у нас нет. Неизвестный нам тип значения представляется переменной образца (? type:17).

Рассмотрим более осмысленный пример:

```
(define foo
  (infer-program-types
   '(define fact
      (lambda (n)
        (begin
          (define iter
            (lambda (product counter)
              (if (> counter n)
                  product
                  (iter (* product counter)
                        (+ counter 1)))))
          (iter 1 1))))))
```

Результат в переменной foo получается довольно многословным. Поэтому воспользуемся упростителем, который переведет его в «человеко-читаемый» вид.

```
(pp (simplify-annotated-program foo))
(begin
  (define fact
    (lambda (n)
      (declare-type n (numeric-type))
```

```
    (define iter
      (lambda (product counter)
        (declare-type product (numeric-type))
        (declare-type counter (numeric-type))
        (if (> counter n)
            product
            (iter (* product counter)
                  (+ counter 1)))))
            (declare-type iter
              (type:procedure ((numeric-type) (numeric-type))
                              (numeric-type)))
    (iter 1 1)))
  (declare-type fact
    (type:procedure ((numeric-type)) (numeric-type))))
```

Как видим, программа вывода типов оказалась способна вывести полный тип программы вычисления факториала — то, что это процедура, принимающая на вход одно число и порождающая числовой результат. Об этом сообщает объявление:

```
(declare-type fact
  (type:procedure ((numeric-type)) (numeric-type))))
```

Вывелся и тип внутреннего определения `iter`: эта процедура принимает два числовых аргумента и порождает число.

```
(declare-type iter
  (type:procedure ((numeric-type) (numeric-type))
                  (numeric-type)))
```

Кроме того, были определены типы всех внутренних переменных, и для них составлены соответствующие объявления:

```
(declare-type n (numeric-type))
(declare-type product (numeric-type))
(declare-type counter (numeric-type))
```

4.4.3 Как работает вывод типов

Процесс вывода типов состоит из четырех фаз.

1. Составляется аннотация с типовыми переменными для всех подвыражений программы.

2. На основании семантической структуры программы формулируются ограничения на типовые переменные.

3. Проводится унификация ограничений, чтобы исключить как можно больше переменных.

4. Проводится специализация аннотированной программы с помощью словаря, полученного при унификации ограничений. Результатом будет аннотированная программа, где аннотации типов учитывают ограничения.

План этого процесса выражается следующей процедурой:

```
(define (infer-program-types expr)
  (let ((texpr (annotate-program expr)))
    (let ((constraints (program-constraints texpr)))
      (let ((dict (unify-constraints constraints)))
        (if dict
            ((match:dict-substitution dict) texpr)
            '***type-error***)))))
```

Процедура жалуется, если не выходит построить согласованный набор аннотаций типа. Однако она не объясняет, почему произошла ошибка; этого можно было бы добиться, если передавать информацию через продолжения неудачи.

Аннотация

Процедура `annotate-program` реализуется через вызов полиморфной процедуры `annotate-expr`, чтобы ее можно было без труда расширять, вводя обработку новых конструкций языка.

```
(define (annotate-program expr)
  (annotate-expr expr (top-level-env)))

(define annotate-expr
  (simple-generic-procedure 'annotate-expr 2 #f))
```

Процедура `annotate-expr` принимает окружение для связываний типовых переменных. Начальным значением служит окружение верхнего уровня, создаваемое ниже.

Есть простые обработчики для аннотации простых разновидностей выражений. Если в подвыражении встречается числовая константа, ей дается константный тип, создаваемый вызовом `(numeric-type)`:

```
(define-generic-procedure-handler annotate-expr
  (match-args number? any-object?)
  (lambda (expr env)
    (make-texpr (numeric-type) expr)))
```

Процедура `make-texpr` создает типизированное выражение из типа и выражения. Части его можно извлечь вызовами `texpr-type` и `texpr-expr`.

Однако тип идентификатора, представленного символом, мы заранее можем не знать. Процедура `get-var-type` пытается найти тип идентификатора в окружении, и при неудаче создает типовую переменную, которой будут аннотироваться все вхождения идентификатора в данном лексическом контексте:

```
(define-generic-procedure-handler annotate-expr
  (match-args symbol? any-object?)
  (lambda (expr env)
    (make-texpr (get-var-type expr env) expr)))
```

Типы некоторых идентификаторов могут быть нам известны; например, элементарные процедуры языка. Они содержатся в окружении верхнего уровня. Приведенные здесь элементарные процедуры имеют процедурные типы с типовыми константами (напр., `(numeric-type)`) в качестве аргументов и возвращаемых значений.

```
(define (top-level-env)
  (list (make-top-level-env-frame)))

(define (make-top-level-env-frame)
  (let ((binary-numerical
         (let ((v (numeric-type)))
           (procedure-type (list v v) v)))
        (binary-comparator
         (let ((v (numeric-type)))
           (procedure-type (list v v) (boolean-type)))))
    (list (cons '+ binary-numerical)
          ...
          (cons '= binary-comparator)
          (cons '< binary-comparator)
          ...)))
```

В случае условного выражения создается типовая переменная для результата выражения в целом, и рекурсивно проводится аннотация каждого подвыражения:

```
(define-generic-procedure-handler annotate-expr
  (match-args if-expr? any-object?)
  (lambda (expr env)
    (make-texpr (type-variable)
        (make-if-expr
         (annotate-expr (if-predicate expr) env)
         (annotate-expr (if-consequent expr) env)
         (annotate-expr (if-alternative expr) env)))))
```

Для каждого вида выражений требуется аннотирующий обработчик. Мы не будем демонстрировать их все, но аннотация конструкции `lambda` представляет интерес:

```
(define-generic-procedure-handler annotate-expr
  (match-args lambda-expr? any-object?)
  (lambda (expr env)
    (let ((env* (new-frame (lambda-bvl expr) env)))
      (make-texpr
       (procedure-type (map (lambda (name)
                              (get-var-type name env*))
                            (lambda-bvl expr))
                       (type-variable))
       (make-lambda-expr (lambda-bvl expr)
                         (annotate-expr (lambda-body expr)
                                        env*))))))
```

Как и в интерпретаторе или компиляторе, при аннотации выражения `lambda` создается новый кадр окружения, который будет содержать информацию о связанных переменных; в этом случае для каждой связанной переменной мы создаем типовую переменную. Создаем также процедурный тип для значения выражения `lambda` с только что созданными типовыми переменными для связанных переменных и еще

одной типовой переменной для возвращаемого значения и рекурсивно запускаем аннотацию тела.

Ограничения

Процедура `program-constraints` формулирует ограничения на типовые переменные на основе семантической структуры программы. Она также реализуется через полиморфную процедуру с обработчиками для каждого типа выражений.

```
(define (program-constraints texpr)
  (program-constraints-1 (texpr-type texpr)
                         (texpr-expr texpr)))

(define program-constraints-1
  (simple-generic-procedure 'program-constraints-1 2 #f))
```

Полиморфная процедура принимает два аргумента: тип выражения и само выражение. Она возвращает список ограничений на типы, обнаруженных ею при просмотре выражения. Процедура обходит дерево выражения, находит и формулирует ограничения на типы.

Вот обработчик условных выражений:

```
(define-generic-procedure-handler program-constraints-1
  (match-args type-expression? if-expr?)
  (lambda (type expr)
    (append
     (list (constrain (boolean-type)
                      (texpr-type (if-predicate expr)))
           (constrain type
                      (texpr-type (if-consequent expr)))
           (constrain type
                      (texpr-type (if-alternative expr))))
     (program-constraints (if-predicate expr))
     (program-constraints (if-consequent expr))
     (program-constraints (if-alternative expr)))))
```

Этот обработчик формулирует три типовых ограничения и добавляет их к ограничениям, рекурсивно сформулированным в трех подвыражениях условного выражения. Первое ограничение утверждает, что значение условия имеет булев тип. Второе и третье ограничения говорят, что тип значения условного выражения совпадает с типами значений выражения-следствия и выражения-альтернативы.

Ограничения выражаются в виде уравнений:

```
(define (constrain lhs rhs)
  `(= ,lhs ,rhs))
```

(Идентификаторы `lhs` и `rhs` происходят от английских выражений *left hand side* «левая сторона» и *right hand side* «правая сторона».)

Ограничение для выражения `lambda` говорит, что тип значения, возвращаемого созданной через `lambda` процедурой, равен типу значения ее тела. Это утверждение добавляется к ограничениям, сформулированным для тела.

```
(define-generic-procedure-handler program-constraints-1
  (match-args type-expression? lambda-expr?)
  (lambda (type expr)
    (cons (constrain (procedure-type-codomain type)
                     (texpr-type (lambda-body expr)))
          (program-constraints (lambda-body expr)))))
```

Для вызова процедуры ограничения состоят в том, что тип вызываемого объекта — процедурный, типы выражений-аргументов соответствуют типам аргументов процедуры, а тип значения, возвращаемого процедурой, является типом всего вызова.

```
(define-generic-procedure-handler program-constraints-1
  (match-args type-expression? combination-expr?)
  (lambda (type expr)
    (cons (constrain (texpr-type (combination-operator expr))
                     (procedure-type
                      (map texpr-type
                           (combination-operands expr))
                      type))
          (append
           (program-constraints (combination-operator expr))
           (append-map program-constraints
                       (combination-operands expr))))))
```

Унификация

Каждое ограничение, обнаруженное нами, представляет собой уравнение из двух типовых выражений. Таким образом, теперь нам нужно решить эту систему уравнений. Это делается путем унификации левой стороны с правой стороной каждого уравнения. Унификации должны производиться в одном и том же контексте связывания переменных, чтобы все уравнения были решены одновременно. Поскольку унификация двух списков унифицирует их соответствующие элементы, можно просто собрать все ограничения в одну гигантскую унификацию:

```
(define (unify-constraints constraints)
  (unify (map constraint-lhs constraints)
         (map constraint-rhs constraints)))
```

Словарь, возвращаемый из `unify-constraints`, используется затем в процедуре `infer-program-types` (стр. 178) для получения типизированной программы.

Критика

Несмотря на то что наша небольшая система вывода типов служит изящной демонстрацией унификации, свою задачу она решает не очень хорошо: процедуры могли бы обрабатываться лучше. Рассмотрим, например, простой случай:

```
(pp (infer-program-types
    '(begin (define id (lambda (x) x))
            (id 2))))
```

Казалось бы, все правильно срабатывает, и мы получаем:

```
(t (numeric-type)
   (begin
     (t (type:procedure ((numeric-type)) (numeric-type))
        (define id
          (t (type:procedure ((numeric-type)) (numeric-type))
             (lambda (x) (t (numeric-type) x)))))
     (t (numeric-type)
        ((t (type:procedure ((numeric-type)) (numeric-type))
            id)
         (t (numeric-type) 2)))))
```

Заметим, однако, что процедура `id` получила тип с числовым аргументом и числовым значением, поскольку она с числовым аргументом использовалась. Однако правильный тип для этой процедуры не должен требовать никакого конкретного типа аргумента. Обобщая этот случай, можно сказать, что тип процедуры не должен зависеть от ее использования в конкретном примере. Эта наша недоработка приводит к ошибке типизации в совершенно правильно построенном фрагменте кода:

```
(infer-program-types
  '(begin (define id (lambda (x) x))
          (id 2)
          (id #t)))
***type-error***
```

Упражнение 4.14. Процедуры
Решить конкретную проблему, продемонстрированную выше, несложно, однако справиться с более общим классом случаев не так уж легко. Как мы должны обрабатывать процедуры, передаваемые в качестве аргументов и возвращаемые как значения? Заметим, что в теле процедуры могут встречаться свободные переменные, лексически связанные в точке, где эта процедура определена.

Проработайте решение этой проблемы и сделайте его как можно более общим.

Упражнение 4.15. Параметрические типы
В этом упражнении мы рассматриваем, насколько сложно расширить систему вывода типов на параметрические типы. Например, процедура `map` языка Scheme работает со списками объектов любого типа.

a. Что нужно сделать, чтобы научить нашу систему поддерживать параметрические типы? Требует ли это расширение изменений в программе унификации? Если да, объясните, почему. Если нет, объясните, почему это не нужно.

b. Реализуйте изменения, позволяющие использовать параметрические типы.

Упражнение 4.16. Типы-объединения
Описанная нами система типов не содержит никакой поддержки типов-объединений. Например, операция сложения `+` определена в численной арифметике. Но что, если мы хотим использовать символ `+` и для сложения чисел, и для конкатенации строк?

a. Что нужно сделать, чтобы расширить систему и ввести поддержку типов-объединений? Требуется ли при этом изменять программу унификации? Если да, объясните, почему. Если нет, объясните, почему это не нужно.

b. Реализуйте изменения, позволяющие использовать типы-объединения. Примечание: это нелегко.

Упражнение 4.17. Побочные эффекты
Описанная нами система вывода типов адекватна для чисто функциональных программ. Можно ли ее естественным образом расширить на программы, использующие присваивание? Если вы считаете, что да, объясните и продемонстрируйте свой проект. Если считаете, что нет, изложите свои соображения.

Это не простой вопрос. Задача понять все детали и заставить их работать может быть темой хорошего семестрового проекта.

Упражнение 4.18. Практичность?
Насколько практична наша реализация вывода типов?

a. Оцените порядки роста памяти и времени в задачах аннотации и построения ограничений в зависимости от размера анализируемой программы.

b. Оцените порядки роста памяти и времени большой фазы унификации с использованием приведенного алгоритма. Как обстоят дела в наилучших известных алгоритмах? (Здесь потребуется исследование литературы вопроса.)

c. Возможно ли разбить большую фазу унификации на подзадачи, чтобы улучшить ее асимптотическое поведение?

4.4.4 Эксперимент: добавляем сегментные переменные

Добавление сегментных переменных в унификатор — увлекательное занятие: мы в точности не знаем, что у нас получится[10]. Однако последовательное использование нами полиморфных процедур гарантирует, что никакая программа, которая зависит от поведения унификатора без добавления сегментных переменных (скажем, наш пример вывода типов), не начнет выдавать неверные результаты в результате этого эксперимента. В самом деле, то, что унификатор организован на основе полиморфных процедур, делает подобные эксперименты относительно беспроблемными[11].

Использование предикатов для управления диспетчеризацией задает взаимодействие между элементными и сегментными переменными. Например, сегментная переменная может включать элементную переменную в сегмент, который она захватывает, но элементная переменная не может иметь в качестве значения сегментную. Таким образом, требуется изменить определение предиката `element?` (стр. 174), чтобы оно не включало сегментные переменные:

```
(define (element? term)
  (not (match:segment-var? term)))
```

[10] Другие исследователи добавляли сегментные переменные в программы сопоставления с образцом и унификаторы [5] с некоторым успехом. Существуют вроде бы версии языка Prolog с сегментными переменными [34]. Подробное теоретическое обсуждение алгоритма унификации, включающего переменные последовательностей (другое название сегментных переменных), можно найти в диссертации Куция [79]. Однако здесь мы не пытаемся построить полный и теоретически обоснованный унификатор с сегментными переменными. Мы всего лишь хотим показать, как легко добавить новый вид полезного поведения к уже построенной нами элементарной процедуре унификации.

[11] Добавление сегментных переменных — весьма тонкая работа. Мы благодарны Кенни Чену, Уиллу Бёрду и Майклу Баллантайну за помощь в продумывании этого эксперимента.

Нужны полиморфные обработчики для сегментных переменных. Обработчик функции `unify:gdispatch` для списков термов, начинающихся с сегментной переменной, — это функция `maybe-grab-segment`, которая устанавливается следующим образом. Список, про который известно, что он содержит сегментную переменную, всегда передается в `maybe-grab-segment` как первый аргумент (так же как мы делали с `maybe-substitute`)[12].

```
(define-generic-procedure-handler unify:gdispatch
  (match-args (car-satisfies match:segment-var?)
              (complement (car-satisfies match:segment-var?)))
  (lambda (var-first terms)
    (maybe-grab-segment var-first terms)))

(define-generic-procedure-handler unify:gdispatch
  (match-args (complement (car-satisfies match:segment-var?))
              (car-satisfies match:segment-var?))
  (lambda (terms var-first)
    (maybe-grab-segment var-first terms)))
```

При обработке двух списков термов, которые оба начинаются с сегментной переменной, нужно учесть особый случай: если переменная в начале списков одна и та же, мы можем отбросить тавтологию без дальнейшей обработки. В противном случае можем получить сопоставление, начиная с той или другой переменной. Однако существуют случаи, когда хорошее сопоставление, начиная с одной переменной, получается, а с другой — нет, в зависимости от других вхождений этих переменных в образцы. Чтобы убедиться, что мы ничего не пропускаем, начав с одной из переменных, мы делаем сопоставление симметричным, рассматривая обратный порядок, если прямой терпит неудачу.

```
(define (unify:segment-var-var var-first1 var-first2)
  (define (unify-seg-var-var dict succeed fail)
    (if (match:vars-equal? (car var-first1) (car var-first2))
        (succeed dict fail (cdr var-first1) (cdr var-first2))
        ((maybe-grab-segment var-first1 var-first2)
         dict
         succeed
         (lambda ()
           ((maybe-grab-segment var-first2 var-first1)
            dict
            succeed
            fail)))))
  unify-seg-var-var)

(define-generic-procedure-handler unify:gdispatch
  (match-args (car-satisfies match:segment-var?)
              (car-satisfies match:segment-var?))
  unify:segment-var-var)
```

[12]Процедура `complement` — комбинатор для предикатов: `complement` создает новый предикат, являющийся отрицанием своего аргумента.

Процедура maybe-grab-segment аналогична maybe-substitute (см. стр. 173), используемой для элементных переменных. Случай сопоставления сегментной переменной с самой собой уже обработан в unify:segment-var-var; поэтому maybe-grab-segment начинает с проверки, имеет ли сегментная переменная из начала var-first уже значение. Если да, мы заменяем переменную на это значение и сопоставляем получившийся список со списком термов terms. Поскольку значением сегментной переменной является список, который она съедает, для замены сегментной переменной на значение используется append. Более сложная задача сопоставления несвязанной сегментной переменной передается в процедуру grab-segment.

```
(define (maybe-grab-segment var-first terms)
  (define (maybe-grab dict succeed fail)
    (let ((var (car var-first)))
      (if (match:has-binding? var dict)
          ((unify:dispatch
            (append (match:get-value var dict)
                    (cdr var-first))
            terms)
           dict succeed fail)
          ((grab-segment var-first terms)
           dict succeed fail))))
  maybe-grab)
```

Именно в процедуре grab-segment происходит собственно сопоставление сегментов и перебор с возвратами. Список термов разбивается на две части: начальный сегмент и остальные термы (terms*). Начальный сегмент initial в начале работы пустой, а список terms* равен всему входному списку термов. Сопоставитель пытается продолжить с сегментной переменной, установленной в значение initial. Если это не получается, продолжение неудачи пытается сопоставиться, перенеся один элемент из terms* в initial. Так это продолжается, пока одно из сопоставлений не окажется успешным или пока сопоставление всего списка термов не потерпит неудачу.

```
(define (grab-segment var-first terms)
  (define (grab dict succeed fail)
    (let ((var (car var-first)))
      (let slp ((initial '()) (terms* terms))
        (define (continue)
          (if (null? terms*)
              (fail)
              (slp (append initial (list (car terms*)))
                   (cdr terms*))))
        (let ((dict* (do-substitute var initial dict)))
          (if dict*
              (succeed dict* continue (cdr var-first) terms*)
              (continue))))))
  grab)
```

Казалось бы, это все, что нужно, чтобы построить унификатор с экспериментальной поддержкой сегментных переменных. При наличии сегментных переменных мы

можем ожидать множественные варианты сопоставления. Можно отвергнуть вариант, заставить программу откатиться в переборе с возвратами и получить таким образом альтернативное решение. Заметим, что каждый элемент словаря представляет собой список, состоящий из имени переменной, ее значения и типа переменной.

Можно использовать унификатор как программу одностороннего сопоставления. Например, как мы видим, есть ровно два способа сопоставить образец распределительного закона с данным алгебраическим выражением[13]:

```
(let ((pattern '(* (?? a) (+ (?? b)) (?? c)))
      (expression '(* x y (+ z w) m (+ n o) p)))
  (unify:internal pattern expression (match:new-dict)
    (lambda (dict)
      (pp (match:bindings dict))
      #f)))
((c (m (+ n o) p) ??) (b (z w) ??) (a (x y) ??))
((c (p) ??) (b (n o) ??) (a (x y (+ z w) m) ??))
#f
```

Оба этих словаря порождают один и тот же экземпляр подстановки:

```
(* x y (+ z w) m (+ n o) p)
```

Однако в программе для алгебраических преобразований нам нужны оба этих словаря, поскольку они представляют различные способы применить распределительный закон.

Ситуация становится сложнее и намного запутаннее, когда сегментные переменные сопоставляются со списками, содержащими другие сегментные переменные:

```
(let ((p1 '(a (?? x) (?? y) (?? x) c))
      (p2 '(a b b b (?? w) b b b c)))
  (unify:internal p1 p2 (match:new-dict)
    (lambda (dict)
      (pp (match:bindings dict))
      #f)))

((y (b b b (?? w) b b b) ??) (x () ??))
((y (b b (?? w) b b) ??) (x (b) ??))
((y (b (?? w) b) ??) (x (b b) ??))
((w () ??) (y () ??) (x (b b b) ??))
((w () ??) (y () ??) (x (b b b) ??))
((y ((?? w)) ??) (x (b b b) ??))
((y () ??) (w () ??) (x (b b b) ??))
((w ((?? y)) ??) (x (b b b) ??))
((w () ??) (y ()??) (x (b b b)??))
#f
```

Судя по всему, есть много способов построить сопоставление. Однако многие из этих словарей представляют собой различные способы сконструировать один и тот же экземпляр подстановки. Чтобы ясно это увидеть, построим в каждом случае подстановочный экземпляр:

[13]Это одностороннее сопоставление можно было бы получить и с предыдущей версией сопоставителя, но полезно иметь возможность работать с переменными на обеих сторонах.

```
(let ((p1 '(a (?? x) (?? y) (?? x) c))
      (p2 '(a b b b (?? w) b b b c)))
  (unify:internal p1 p2 (match:new-dict)
    (lambda (dict)
      (and dict
           (let ((subst (match:dict-substitution dict)))
             (let ((p1* (subst p1)) (p2* (subst p2)))
               (if (not (equal? p1* p2*))
                   (error "Bad dictionary"))
               (pp p1*))))
      #f)))

(a b b b (?? w) b b b c)
(a b b b (?? w) b b b c)
(a b b b (?? w) b b b c)
(a b b b b b b c)
(a b b b b b b c)
(a b b b (?? w) b b b c)
(a b b b b b b c)
(a b b b (?? y) b b b c)
(a b b b b b b c)
#f
```

Мы видим, что каждое «решение» действительно является решением задачи найти значения переменных, которые, будучи подставлены обратно в данные образцы, сделают их равными друг другу. В этом случае пять решений эквивалентны. Они представляют собой наиболее общие унификаторы, и они равны друг другу с точностью до переименования переменных. Еще четыре решения не обладают наибольшей возможной общностью. Однако предполагается, что унификация порождает единственный наиболее общий подстановочный экземпляр с точностью до переименования переменных. Таким образом, если у нас есть сегменты, наша действительно полезная программа сопоставления с образцом настоящим унификатором не является.

Проблема в действительности еще хуже. Существуют работающие варианты сопоставления, которые наша программа не находит. Вот пример:

```
;;; Пропущенное решение!
(unify:internal '(((?? x) 3) ((?? x)))
                '((4 (?? y)) (4 5))
                (match:new-dict)
                (lambda (dict)
                  (pp (match:bindings dict))
                  #f))
#f
```

Однако эти два выражения можно сопоставить с такими связываниями:

```
((x (4 5) ??) (y (5 3) ??))
```

Обидно! Ну у этой истории есть мораль. С помощью полиморфных процедур можно сделать расширения, возможно, проблематичные, правильно работающего

алгоритма, и при этом не подвергается опасности его правильность в рамках использования нерасширенного алгоритма. Эти расширения для некоторых целей могут оказаться полезными, даже если они не удовлетворяют требованиям корректности исходного алгоритма.

Упражнение 4.19. Можно ли устранить недостатки?

У нас проблемы с образцами для унификации, содержащими сегментные переменные. Иногда мы пропускаем варианты сопоставления; иногда находим несколько копий одного и того же решения; и некоторые результаты, которые мы находим, хотя и решают задачу сделать образцы одинаковыми, не обладают максимальной общностью. Давайте подумаем, как эти проблемы решить.

a. Напишите обертку, собирающую вместе все решения. Это нетрудно сделать с использованием присваивания, однако интереснее поискать функциональное решение — но не тратьте слишком много сил!

b. Имея решения все вместе, нетрудно убрать дубликаты. Создайте результат подстановки каждого решения во входные образцы. Можно убедиться, что результаты двух подстановок одинаковы — это проверка, что алгоритм решения задачи работает корректно. Теперь создайте для каждого результата пару из одного экземпляра подстановки и результата. Осторожно: имена переменных не имеют значения, так что два словаря относятся к одному и тому же решению, если один из них можно получить из другого последовательным переименованием переменных.

c. Если один из результатов в коллекции является подстановочным экземпляром другого, он не представляет собой максимально общее описание входов. Напишите предикат `substitution-instance?`, чтобы отфильтровать эти случаи. Теперь у вас осталась коллекция наиболее общих решений, порождаемых алгоритмом. Верните ее.

d. Найдите способ не пропускать сопоставления вроде того, которое выше помечено как «пропущенное решение». Существует ли простое расширение имеющегося кода, которое справляется с такими задачами? Примечание: этот вопрос очень сложный.

Упражнение 4.20. Более общие сопоставления

Помимо неприятных проблем, описанных выше, есть еще одна тонкость, с которой наша программа унификации не справляется. Рассмотрим следующую задачу:

```
(unifier '((?? x) 3) '(4 (??y)))
((4 3))
```

Мы получили работающую подстановку, но это не самое общее решение. Проблема в том, что между 4 и 3 может располагаться произвольное число элементов. Более правильным ответом было бы

```
((4 (?? z) 3))
```

Разберитесь, как получить этот ответ. Потребуется существенное расширение унификатора.

Упражнение 4.21. Строки с сегментами

Если вы не решали упражнение 4.11 (стр. 175), решите его теперь. Однако на этот раз мы хотим, чтобы вы добавили строковые сегментные переменные. Это может быть полезно для поиска фрагментов ДНК!

4.5 Сопоставление с образцом на графах

До сих пор мы разрабатывали сопоставление с образцом для списковых структур. Такое структуры отлично приспособлены для представления выражений, например алгебраических выражений или абстрактных синтаксических деревьев выражений компьютерных языков. Однако при помощи сопоставления с образцом можно работать с гораздо более широким классом данных. Если интересующая нас структура может быть охарактеризована отношением доступности, такую структуру имеет смысл описывать как граф, состоящий из *вершин*, которые представляют «места» или «позиции», и *ребер*, которые представляют «фрагменты путей», описывающих связи между этими позициями. Примером такой структуры является электрическая цепь, где точки в этой цепи и электрические элементы служат позициями, а отношение доступности — это просто схема соединений. Настольная игра вроде шахмат — еще один пример; здесь вершинами графа служат поля на доске, а отношение соседства между полями задает ребра.

Мы реализуем граф в виде коллекции, состоящей из вершин и ребер. Наши графы будут неизменяемыми в том смысле, что после добавления вершину или ребро нельзя будет модифицировать: граф можно изменять только добавлением новых вершин и ребер. Следствия этого ограничения можно будет увидеть в разделе 4.5.4.

Вершина содержит коллекцию ребер, а ребро имеет внутри себя *метку* и *значение*. Объект-метка уникален в смысле предиката `eqv?`; обычно это символ или номер. Значением ребра является объект языка Scheme, часто еще одна вершина.

Наша реализация будет работать с конкретными графами (где все вершины и ребра существуют к тому времени, когда мы создаем граф), а также с ленивыми графами (которые расширяются по необходимости в процессе доступа). В более простом мире линейных последовательностей список представляет собой конкретный граф, а поток — ленивый граф, расширяемый по мере обращения к нему.

Сначала мы рассмотрим простой пример, показывающий, как работают графы. Затем используем расширенный пример — шахматного судью, — и с его помощью исследуем более сложные способы работы с графами и сопоставление с образцом на графах.

4.5.1 Списки как графы

Начнем с простого и знакомого мира списков. Ячейки `cons` будут реализовываться как вершины и создаваться через `g:cons`, а `car` и `cdr` этих ячеек реализуются как ребра, помеченные символами `car` и `cdr`; доступ к ним осуществляется процедурами `g:car` и `g:cdr`.

```
(define (g:cons car cdr)
  (let ((pair (make-graph-node 'pair)))
    (pair 'connect! 'car car)
    (pair 'connect! 'cdr cdr)
    pair))

(define (g:car pair) (pair 'edge-value 'car))
(define (g:cdr pair) (pair 'edge-value 'cdr))
```

Чтобы представлять списки как графы, нам потребуется специальная метка конца списка:

```
(define nil (make-graph-node 'nil))

(define (g:null) nil)

(define (g:null? object) (eqv? object nil))
```

Вот преобразование списка в списковый граф:

```
(define (list->graph list)
  (if (pair? list)
      (g:cons (car list) (list->graph (cdr list)))
      (g:null)))
```

Простой пример работает как ожидается:

```
(define g (list->graph '(a b c)))

(and (eqv? 'a (g:car g))
     (eqv? 'b (g:car (g:cdr g)))
     (eqv? 'c (g:car (g:cdr (g:cdr g))))
     (g:null? (g:cdr (g:cdr (g:cdr g)))))
#t
```

Можно преобразовать конструктор, строящий граф из списка, и сделать графы ленивыми, чтобы вершины создавались по мере обращения к графу:

```
(define (list->lazy-graph list)
  (if (pair? list)
      (g:cons (delay (car list))
              (delay (list->lazy-graph (cdr list))))
      (g:null)))
```

Здесь мы с помощью конструкции `delay` языка Scheme [109] создаем *обещание*, которое вычислит задержанное (отложенное) вычисление при вынуждении. Потоки [13] (ленивые списки) обычно строятся с помощью `delay` и `force`.

4.5.2 Реализация графов

Нам нужно создавать *вершины графа* и связывать их между собой *ребрами*. Будем представлять вершину графа как *пакетную процедуру*: набор процедур-делегатов, которые можно вызывать по имени[14].

```
(define (make-graph-node name)
  (let ((edges '()))
    (define (get-name) name)
    (define (all-edges) (list-copy edges))
    (define (%find-edge label)
      (find (lambda (edge)
              (eqv? label (edge 'get-label)))
```

[14]Пример использования вершины графа см. на стр. 189. Более полное описание пакетных процедур см. на стр. 345.

```
        edges))
  (define (has-edge? label)
    (and (%find-edge label) #t)) ; булево значение
  (define (get-edge label)
    (let ((edge (%find-edge label)))
      (if (not edge)
          (error "Нет ребра с такой меткой:" label))
          edge))
  (define (edge-value label)
    ((get-edge label) 'get-value))
  (define (connect! label value)
    (if (has-edge? label)
        (error "Два ребра с одной меткой:" label))
    (set! edges
          (cons (make-graph-edge label value) edges)))
  (define (maybe-connect! label value)
    (if (not (default-object? value))
        (connect! label value)))
  (bundle graph-node? get-name all-edges has-edge?
          get-edge edge-value connect! maybe-connect!)))
```

Аргументом процедуры `make-graph-node` служит имя новой вершины; оно показывается при распечатке объекта-вершины. Первым аргументом макроса `bundle` является предикат, которому будет удовлетворять создаваемый пакет. В нашем случае он определяется как

```
(define graph-node? (make-bundle-predicate 'graph-node))
```

Определения других пакетных предикатов мы приводить не будем; они устроены точно так же.

Ребра также представляются пакетными процедурами. Ребро может содержать конкретное значение, либо же значение может быть обещанием (создаваемым через `delay`), которое создаст значение по запросу. Второй вариант позволяет строить ленивые графовые структуры.

```
(define (make-graph-edge label value)
  (define (get-label) label)
  (define (get-value)
    (if (promise? value)
        (force value)
        value))
  (bundle graph-edge? get-label get-value))
```

Упражнение 4.22. Еще больше ленивых графов

Мы показали, как создавать конкретные списки и ленивые списки. Как насчет более интересных структур?

Возможно, нам понадобится работать с динамически расширяемым деревом. Например, таким образом можно представлять дерево игры: по мере получения ресурсов мы можем расширять это дерево и в глубину, и в ширину. Постройте пример такого дерева, которое можно увеличивать, учитывая больше возможных ходов на каждом уровне либо увеличивая число учитываемых уровней.

4.5.3 Сопоставление на графах

Мы можем искать в графе интересующие нас подграфы. Один из способов это делать — накладывать на граф образцы. Пусть образец для графа описывает чередующуюся последовательность вершин и ребер: *путь*. Такой образец можно сопоставлять, начиная с некоторой вершины и пытаясь проследовать по пути, который образец описывает.

Представим себе, например, что у нас есть шахматная доска и фигуры. Поля доски являются вершинами графа. Вершины, соответствующие соседним полям, связаны с данной вершиной ребрами. Можно помечать эти ребра сторонами света с точки зрения белых: `north` (север), `south` (юг), `east` (восток), `west` (запад), `northeast` (северо-восток), `southeast` (юго-восток), `northwest` (северо-запад), `southwest` (юго-запад). Ход на север направлен в сторону базовой горизонтали черных, а на юг — в сторону базовой горизонтали белых.

Имея такую конструкцию, можно описать ситуацию, где коню разрешен ход на северо-северо-восток:

```
(define basic-knight-move
  `((? source-node ,(occupied-by 'knight))
    north (?)
    north (?)
    east (? target-node ,maybe-opponent)))
```

Этот образец во многом похож на образцы, рассматриваемые нами в предыдущих разделах: элементные переменные вводятся символом `?`; они могут иметь имена (например `source-node`) и ограничения (например `(occupied-by 'knight)`). Мы вводим синтаксическую конструкцию `(?)` как анонимную элементную переменную.

Сопоставление с образцом начинается с исходного поля `source-node`, проходит по двум ребрам с меткой `north` — с вершинами, которые нас не интересуют, — и в конце переходит на восток через ребро `east`, достигая поля `target-node`. Образец такого рода мы называем *образец пути* или, в контексте шахмат, *образец хода*.

Разумеется, это только один из возможных ходов коня. Но остальные ходы можно получить применением симметрий: можно отразить ход по оси восток-запад, повернуть его на 90° или на 180°.

```
(define all-knight-moves
  (symmetrize-move basic-knight-move
                   reflect-ew rotate-90 rotate-180))
```

Процедура `symmetrize-move` применяет все возможные комбинации трех симметрий и получает восемь ходов. Для наших трансформаций порядок, в котором они применяются, не имеет значения.

```
(define (symmetrize-move move . transformations)
  (let loop ((xforms transformations) (moves (list move)))
    (if (null? xforms)
        moves
        (loop (cdr xforms)
              (append moves
                      (map (rewrite-path-edges (car xforms))
                           moves))))))
```

где процедура `rewrite-path-edges` применяет свой аргумент ко всем меткам ребер внутри хода и получает ход с подставленными метками.

Вот пример такого преобразования симметрии:

```
(define (reflect-ew label)
  (case label
    ((east) 'west)
    ((northeast) 'northwest)
    ((northwest) 'northeast)
    ((southeast) 'southwest)
    ((southwest) 'southeast)
    ((west) 'east)
    (else label)))
```

Остальные представляют собой подобные же переименования сторон света.

Получаем такой список ходов коня:

```
((source north (?) north (?) east target)
 (source north (?) north (?) west target)
 (source east (?) east (?) south target)
 (source east (?) east (?) north target)
 (source south (?) south (?) west target)
 (source south (?) south (?) east target)
 (source west (?) west (?) north target)
 (source west (?) west (?) south target))
```

(Мы упростили распечатку, заменив ограниченные переменные, указывающие начальное и конечное поле, на курсивные *source* и *target*.)

Ходы коня в шахматах устроены иначе, чем остальные: конь может на пути к новому месту перепрыгивать через поля, где находятся свои или чужие фигуры. Ладьи, слоны и ферзи не могут проходить через занятое поле, но зато могут пересекать на пути к новому месту любое число пустых полей. Мы выражаем многократное пересечение поля с помощью `(* ...)`:

```
(define basic-queen-move
  `((? source-node ,(occupied-by 'queen))
    (* north (?* ,unoccupied))
    north (? target-node ,maybe-opponent)))
```

Ферзь может идти на север, проходя через любое количество пустых полей на пути к своему назначению. Обозначение `(?* ...)` — новый вид переменной образца; ее можно использовать только внутри конструкции `(* ...)`. Подобно простой переменной образца, она сопоставляется с одним элементом, но сохраняет не единственное сопоставленное значение, а собирает в себя список всех элементов, встреченных при повторении. Теперь мы можем выразить все возможные ходы ферзя через:

```
(define all-queen-moves
  (symmetrize-move basic-queen-move
                   rotate-45 rotate-90 rotate-180))
```

Правила для пешек более сложные. Пешка — (почти) единственная фигура, чьи возможные ходы зависят от ее собственного расположения и от расположения фи-

гуры противника рядом[15]. Из начального положения пешка может идти на север на одно или два поля, но из любого другого положения она может идти только на одно поле на север. Кроме того, пешка может пойти на одно поле к северо-западу или северо-востоку тогда и только тогда, когда при этом ходе она берет фигуру противника. Наконец, пешка на предпоследней горизонтали может передвинуться на последнюю и быть произведена в любую другую фигуру, чаще всего ферзя[16].

Упражнение 4.23. Остальные шахматные ходы
Мы показали, как составить образцы для ходов коня и ферзя, но не создали образцов для остальных фигур.

a. Ходы ладьи и слона подобны ходам ферзя, но более ограничены: ладья не ходит по диагонали, а слон ходит только по диагонали. Постройте образцы для ходов слона и простых ходов ладьи.

b. Ходы пешки намного сложнее. Постройте образцы для всех ходов пешки (кроме взятия на проходе).

c. Сделайте набор образцов для весьма ограниченных ходов короля. Не учитывайте рокировку и правило, что король не может ходить под шах.

d. Наконец, последний особый случай — рокировка. В ней участвуют король и ладья. Создайте для нее набор образцов (см. прим. 15).

4.5.4 Шахматные доски и альтернативные перспективы для графов

Рассмотрение шахматной доски как графа приводит к интересной идее. Нам хочется, чтобы одни и те же образцы работали для обоих игроков. Однако ребра, описывающие направления, для них различаются: то, что для белых север (`north`) — для черных юг (`south`), а запад (`west`) для белых является для черных востоком (`east`)! Для большинства фигур (ладей, коней, слонов, королей и ферзей) это не играет большой роли, поскольку их возможные ходы симметричны, однако белые пешки могут ходить только на север, а черные только на юг. В любом случае полезно будет сделать описания ходов одинаковыми для обоих игроков.

Нам хочется, чтобы игроки смотрели на граф-доску по-разному: нужно, чтобы метки ребер были для игроков различны. Если играющий белыми видит между полем A (вершиной, представляющей его) и полем B ребро с меткой `north`, то для играющего черными должно быть ребро с меткой `north` из поля B в поле A.

Для этого мы вводим *перспективы графов*. Перспектива — это обратимое отображение между метками ребер. Когда перспектива применяется к вершине, возвращается новая вершина с переименованными ребрами.

В случае шахмат нужная нам перспектива — поворот доски на 180°.

```
(define rotate-180-view
  (make-graph-view 'inverse rotate-180 rotate-180))
```

[15] Еще один особый случай — рокировка. Рокировка разрешена в ограниченных условиях: король и ладья должны находиться в начальном положении, поля между ними незаняты, король не находится под шахом, не окажется под шахом на новом месте и не будет проходить через поля, где он был бы под шахом.

[16] Существует еще ход пешки, взятие на проходе, который зависит от предыдущего хода противника.

Процедура `make-graph-view` создает перспективу графа. Процедура `graph-node-view` применяет ее к вершине:

`(graph-node-view` *`перспектива вершина`*`)`

Белые будут смотреть на вершину непосредственно, а черные ту же самую вершину будут видеть пропущенной сквозь `rotate-180-view`. С таким отображением все ходы для белых и черных выглядят одинаково.

С помощью перспективы мы справляемся с *относительной* адресацией, когда требуется смотреть на соседей данной вершины. Однако нужна и *абсолютная* адресация, когда поле определяется горизонталью и вертикалью. Каждый игрок хочет использовать один и тот же вид адресации, где его базовая горизонталь имеет номер 0, а горизонталь соперника —номер 7. Кроме того, для каждого игрока левая вертикаль имеет номер 0, а правая — 7[17]. Адреса с точки зрения белых используются по умолчанию, а для черных они обращаются с помощью процедуры `invert-address`.

Давайте создадим шахматную доску. Код, приведенный ниже, приспособлен конкретно к шахматам, поскольку сейчас мы не заботимся о создании абстрактной предметной области. Мы порождаем массив 8×8 вершин, представляющих поля. У каждого поля есть адрес. Затем проходим по всем адресам и связываем каждое поле с соседями, давая ребрам соответствующие имена. Наконец, размещаем на доске фигуры.

```
(define chess-board-size 8)
(define chess-board-indices (iota chess-board-size))
(define chess-board-last-index (last chess-board-indices))

(define (make-chess-board)
  (let ((board (make-chess-board-internal)))
    (for-each (lambda (address)
                (connect-up-square address board))
              board-addresses)
    (populate-sides board)
    board))
```

Возможными адресами для полей шахматной доски являются все пары целых чисел от 0 до 7:

```
(define board-addresses
  (append-map (lambda (y)
                (map (lambda (x)
                       (make-address x y))
                     chess-board-indices))
              chess-board-indices))
```

Процедура `make-chess-board-internal` создает массив вершин для полей как список горизонталей, где каждый элемент представляет список вертикалей. Возвращается пакетная процедура с набором делегатов для работы с доской.

```
(define (make-chess-board-internal)
  (let ((nodes
```

[17] В отличие от традиционных в шахматах соглашений, мы считаем индексы от нуля, но, кроме как при вводе и выводе для игроков, это не имеет значения.

```
        (map (lambda (x)
               (map (lambda (y)
                      (make-graph-node (string x "," y)))
                    chess-board-indices))
             chess-board-indices)))
    (let loop ((turn 0))
      Определения делегатов приведены ниже.
      (bundle #f node-at piece-at piece-in address-of
              set-piece-at color next-turn))))
```

Переменная `turn` содержит номер текущего хода, начиная с нуля. Белые ходят в четные ходы, черные в нечетные; это отображает процедура-делегат `color`:

```
(define (color) (if (white-move?) 'white 'black))
(define (white-move?) (even? turn))
```

Процедура-делегат `node-at` возвращает вершину по данному адресу. Если ходят черные, она преобразует адрес и применяет перспективу к вершине:

```
(define (node-at address)
  (define (get-node address)
    (list-ref (list-ref nodes (address-x address))
              (address-y address)))
  (if (white-move?)
      (get-node address)
      (graph-node-view (get-node (invert-address address))
                       rotate-180-view)))
```

Процедура `address-of` — обратная к `node-at`. У каждой вершины есть ребро с меткой `address`, значением которого является адрес. Как и в случае `node-at`, при ходе черных возвращаемый адрес надо перевести:

```
(define (address-of node)
  (let ((address (node 'edge-value 'address)))
    (if (white-move?)
        address
        (invert-address address))))
```

Процедура-делегат `next-turn` продвигает состояние доски после того, как сделан ход:

```
(define (next-turn) (loop (+ turn 1)))
```

При создании связей между полями арифметика адресов нужна, чтобы обработать (явно указываемые) краевые случаи. В результате между каждым полем и его соседями создаются помеченные ребра. Кроме того, для каждого поля строится ребро `address`.

```
(define (connect-up-square address board)
  (let ((node (board 'node-at address)))
    (node 'connect! 'address address)
    (for-each-direction
```

```
      (lambda (label x-delta y-delta)
        (let ((x+ (+ (address-x address) x-delta))
              (y+ (+ (address-y address) y-delta)))
          (if (and (<= 0 x+ chess-board-last-index)
                   (<= 0 y+ chess-board-last-index))
              (node 'connect! label
                    (board 'node-at
                           (make-address x+ y+)))))))))

(define (for-each-direction procedure)
  (procedure 'north 0 1)
  (procedure 'northeast 1 1)
  (procedure 'east 1 0)
  (procedure 'southeast 1 -1)
  (procedure 'south 0 -1)
  (procedure 'southwest -1 -1)
  (procedure 'west -1 0)
  (procedure 'northwest -1 1))
```

Адрес представляется как список из номера вертикали и горизонтали:

```
(define (make-address x y) (list x y))
(define (address-x address) (car address))
(define (address-y address) (cadr address))

(define (address= a b)
  (and (= (address-x a) (address-x b))
       (= (address-y a) (address-y b))))

(define (invert-address address)
  (make-address (- chess-board-last-index
                   (address-x address))
                (- chess-board-last-index
                   (address-y address))))
```

Фигура представляется объектом данных, включающим в себя тип фигуры и цвет. Действует соглашение, что на n-м ходу каждая фигура на доске будет связана с вершиной, представляющей поле, где она стоит, через ребро от этой вершины с меткой n. Это следствие неизменяемости графа; иначе бы можно было просто модифицировать связь через побочный эффект. Мы заселяем доску, связывая каждую фигуру с ее начальным положением ребром с меткой 0.

```
(define (populate-sides board)

  (define (populate-side color home-row pawn-row)

    (define (do-column col type)
      (add-piece col home-row type)
      (add-piece col pawn-row 'pawn))
```

```
    (define (add-piece col row type)
      ((board 'node-at (make-address col row))
       'connect! 0 (make-piece type color)))

    (do-column 0 'rook)
    (do-column 1 'knight)
    (do-column 2 'bishop)
    (do-column 3 'queen)
    (do-column 4 'king)
    (do-column 5 'bishop)
    (do-column 6 'knight)
    (do-column 7 'rook))

  (populate-side 'white 0 1)
  (populate-side 'black 7 6))
```

Теперь можно начать игру:

```
(define the-board)

(define (start-chess-game)
  (set! the-board (make-chess-board))
  (print-chess-board the-board))
```

Получаем такую приятную распечатку:

```
;;;     0    1    2    3    4    5    6    7
;;;   +----+----+----+----+----+----+----+----+
;;; 7 | Rb | Nb | Bb | Qb | Kb | Bb | Nb | Rb |
;;;   +----+----+----+----+----+----+----+----+
;;; 6 | Pb | Pb | Pb | Pb | Pb | Pb | Pb | Pb |
;;;   +----+----+----+----+----+----+----+----+
;;; 5 |    |    |    |    |    |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 4 |    |    |    |    |    |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 3 |    |    |    |    |    |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 2 |    |    |    |    |    |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 1 | Pw | Pw | Pw | Pw | Pw | Pw | Pw | Pw |
;;;   +----+----+----+----+----+----+----+----+
;;; 0 | Rw | Nw | Bw | Qw | Kw | Bw | Nw | Rw |
;;;   +----+----+----+----+----+----+----+----+
;;; ход белых
```

4.5.5 Шахматные ходы

Теперь, когда у нас есть доска с фигурами, нужно научиться эти фигуры передвигать. Если фигура на определенном ходу находится на определенном поле, то из

вершины, представляющей это поле, есть ребро, чье значение равно этой фигуре, а метка является номером хода. Это отражается в следующих процедурах-делегатах из `make-chess-board-internal` (стр. 195):

```
(define (piece-at address)
  (piece-in (node-at address)))

(define (piece-in node)
  (and (node 'has-edge? turn)
       (node 'edge-value turn)))

(define (set-piece-at address piece)
  ((node-at address) 'connect! (+ turn 1) piece))
```

С помощью `piece-at` мы получаем фигуру, которую затем пытаемся двигать, по адресу. Разумеется, всегда имеет смысл убедиться, что нет очевидных ошибок:

```
(define (get-piece-to-move board from)
  (let ((my-piece (board 'piece-at from)))
    (if (not my-piece)
        (error "На этом поле нет фигуры:" from))
    (if (not (eq? (board 'color) (piece-color my-piece)))
        (error "Можно передвигать только свои фигуры:"
               my-piece from))
    my-piece))
```

Чтобы собственно передвинуть фигуру, нужно взять ее и поставить на целевое поле. Однако ход разрешен, только если целевое поле пусто или занято фигурой противника, которая в таком случае бьется.

```
(define (simple-move board from to)
  (let ((my-piece (get-piece-to-move board from)))
    (let ((captured (board 'piece-at to)))
      (if (not (no-piece-or-opponent? captured my-piece))
          (error "Нельзя бить собственную фигуру:"
          captured)))
      ;; Ход разрешен; сделать его:
      (board 'set-piece-at to my-piece)
      ;; Теперь переносим все незатронутые фигуры
      ;; в следующее состояние доски:
      (for-each (lambda (address)
                  (if (not (or (address= from address)
                               (address= to address)))
                      (let ((p (board 'piece-at address)))
                        (if p
                            (board 'set-piece-at address p)))))
                board-addresses)
      (board 'next-turn)))
```

Заметим, что мы не вставили проверки, что передвигаемая нами фигура может ходить указанным образом. Описания ходов, разрешенных каждому типу фигур, у

нас есть только в виде графовых образцов, построенных в разделе 4.5.3. Мы решим эту задачу в упражнении 4.24 на стр. 201.

Однако сначала давайте с помощью сопоставления определим, происходит ли при описанном таким образом ходе взятие фигуры:

```
(define (capture? board from path)
  (let* ((my-piece (get-piece-to-move board from))
         (dict
          (graph-match path
                       (match:extend-dict chess-board:var ;**
                                          board
                                          (match:new-dict))
                       (board 'node-at from))))
    (and dict
         (let* ((target (match:get-value 'target-node dict))
                (captured (board 'piece-in target)))
           (and captured
                `(capture ,my-piece
                          ,captured
                          ,(board 'address-of target)))))))
```

В строке, помеченной комментарием `;**`, мы добавляем к начальному словарю специальное связывание, при помощи которого потом в некоторых ограничениях образцов можем обратиться к доске.

Для удобства пользователя `chess-move` обновляет состояние доски, а затем распечатывает доску для игрока, чья очередь ходить следующим:

```
(define (chess-move from to)
  (set! the-board (simple-move the-board from to))
  (print-chess-board the-board))
```

Чтобы продемонстрировать работу этого кода, мы создаем интересную позицию*:

```
(define (giuoco-piano-opening)
  (start-chess-game)
  (chess-move '(4 1) '(4 3))    ; W: P-K4   e4
  (chess-move '(3 1) '(3 3))    ; B: P-K4   e5
  (chess-move '(6 0) '(5 2))    ; W: N-KB3  Kf3
  (chess-move '(6 0) '(5 2))    ; B: N-QB3  Kc6
  (chess-move '(5 0) '(2 3))    ; W: B-QB4  Cc4
  (chess-move '(2 0) '(5 3)))   : B: B-QB4  Cc5

(giuoco-piano-opening)
```

*Авторы приводят ходы в описательной нотации, распространенной в англоязычных странах. Мы добавляем стандартную алгебраическую нотацию здесь и используем ее далее в тексте. — *Прим. перев.*

После нескольких распечаток получаем следующую позицию:

```
;;;     0    1    2    3    4    5    6    7
;;;   +----+----+----+----+----+----+----+----+
;;; 7 | Rb |    | Bb | Qb | Kb |    | Nb | Rb |
;;;   +----+----+----+----+----+----+----+----+
;;; 6 | Pb | Pb | Pb | Pb |    | Pb | Pb | Pb |
;;;   +----+----+----+----+----+----+----+----+
;;; 5 |    |    | Nb |    |    |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 4 |    |    | Bb |    | Pb |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 3 |    |    | Bw |    | Pw |    |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 2 |    |    |    |    |    | Nw |    |    |
;;;   +----+----+----+----+----+----+----+----+
;;; 1 | Pw | Pw | Pw | Pw |    | Pw | Pw | Pw |
;;;   +----+----+----+----+----+----+----+----+
;;; 0 | Rw | Nw | Bw | Qw | Kw |    |    | Rw |
;;;   +----+----+----+----+----+----+----+----+
;;; ход белых
```

К этому моменту белый конь на f3 нападает на белую пешку на e5. Брать ее не стоит, поскольку ее защищает черный конь на c6, а менять коня на пешку невыгодно. Однако при помощи образца для ходов коня мы можем проверить, что такое взятие разрешено:

```
(capture? the-board
          (make-address 5 2)
          `((? source-node ,(occupied-by 'knight))
            north (?) north (?)
            west (? target-node ,maybe-opponent)))
(capture (knight white) (pawn black) (4 4))
```

Это единственный возможный ход этого коня со взятием:

```
(filter-map (lambda (path)
              (capture? the-board
                        (make-address 5 2)
                        path))
            all-knight-moves)
((capture (knight white) (pawn black) (4 4)))
```

Упражнение 4.24. Шахматные ходы

В упражнении 4.23 на стр. 194 мы построили библиотеку образцов для всех разрешенных ходов в шахматах. Измените программу `simple-move` (стр. 199), чтобы она проверяла, что передвигаемая фигура действительно может ходить указанным образом.

4.5.6 Реализация сопоставления на графах

Входной точкой для сопоставления на графах служит процедура:

```
(define (graph-match path dict object)
  ((gmatch:compile-path path) object dict
   (lambda (object* dict*)
     dict*)))
```

Мы компилируем образец пути в процедуру сопоставления, которая принимает стартовую вершину в графе, исходный словарь и продолжение успеха. Если образец успешно сопоставляется с последовательностью ребер, начиная с этой вершины, вызывается продолжение успеха, которому передаются вершина `object*` на конце сопоставленного пути и словарь связываний переменных, накопленных в ходе сопоставления, как описано в разделе 4.3[18]. Если образец не удалось сопоставить с данным объектом, процедура возвращает `#f`.

Образцы, используемые нами для сопоставления с графами, являются *выражениями* небольшого языка, которые мы хотим компилировать в процедуры сопоставления. Синтаксис выражений для сопоставления можно описать грамматикой в форме Бэкуса-Наура. Знак `*` после выражения обозначает 0 или более вхождений, `+` обозначает 1 или более вхождений, а `?` — 0 или 1 вхождение. `|` между выражениями обозначает альтернативу. Элементы в кавычках — строковые константы. Например, переменная образца, соответствующая одному элементу, начинается с `(?`, может иметь имя, может иметь предикат и заканчивается символом `)`.

```
<ребро> = <метка-ребра> <конец>
<метка-ребра> = <символ>
<конец> = <прм-вершины> | <прм-объекта> | <константа>
<прм-вершины> = <одиночная-прм>
<прм-объекта> = <одиночная-прм> | <прм-посл>
<одиночная-прм> = "(?" <имя-прм>? <одноместный-предикат>? ")"
<прм-посл> = "(?*" <имя-прм>? <одноместный-предикат>? ")"
<имя-прм> = <символ>

<путь> = <прм-вершины> <элементы-пути>
<элементы-пути> = <элемент-пути>*

<элемент-пути> =
    <ребро>
  | "(*" <элементы-пути> ")"     ; повторить сколько угодно раз
  | "(+" <элементы-пути> ")"     ; повторить хотя бы один раз
  | "(opt" <элементы-пути> ")"   ; ноль или один раз
  | "(or" <элементы-пути-ск>+ ")"
  | "(and" <элементы-пути-ск>+ ")"
<элементы-пути-ск> = "(" <элементы-пути> ")"
```

В нашем языке графовых образцов любой путь начинается с переменной вершины. Переменная вершины — это одиночная элементная переменная, удовлетворяющая предикату `match:element-var?`. Путь компилируется так:

[18]Заметим, однако, что продолжение успеха в процедуре графового сопоставления отличается от продолжения успеха в процедуре сопоставления выражений. В сопоставителе выражений продолжение успеха принимает словарь и количество съеденных сопоставителем элементов (это нужно для работы сегментных переменных). В графовом сопоставителе продолжение успеха принимает конечную вершину и словарь, полученный при сопоставлении подграфа.

```
(define (gmatch:compile-path path)
  (if (and (pair? path) (match:element-var? (car path)))
      (gmatch:finish-compile-path (cdr path)
        (gmatch:compile-var (car path)))
      (error "Неправильный путь:" path)))
```

Мы проверяем, что первый элемент списка `path` — элементная переменная; если да, компилируем ее в сопоставитель для переменной. Остаток пути, если он есть, компилируется через `finish-compile-path`[19]:

```
(define (gmatch:finish-compile-path rest-elts matcher)
  (if (null? rest-elts)
      matcher
      (gmatch:seq2 matcher
                   (gmatch:compile-path-elts rest-elts))))
```

где `seq2` порождает процедуру-сопоставитель, которая последовательно применяет свои сопоставители-аргументы:

```
(define (gmatch:seq2 match-first match-rest)
  (define (match-seq object dict succeed)
    (match-first object dict
                 (lambda (object* dict*)
                   (match-rest object* dict* succeed))))
  match-seq)
```

Сопоставитель переменной `match-first`, порожденный процедурой `compile-var`, сопоставится с первым элементом пути, и полученный словарь `dict*` затем будет передан результату процедуры `compile-path-elts` (переменная `match-rest`), чтобы с его помощью сопоставить оставшуюся часть пути, начиная с ребра `object*`.

При компиляции образцов для элементов пути есть всего несколько случаев. Либо путь начинается с метки ребра и вершины на его конце, либо с особой формы сопоставителя (`*`, `+`, `opt`, `or`, `and`):

```
(define (gmatch:compile-path-elts elts)
  (let ((elt (car elts))
        (rest (cdr elts)))
    (cond ((and (symbol? elt) (pair? rest))
           (gmatch:finish-compile-path (cdr rest)
             (gmatch:compile-edge elt (car rest))))
          ((pair? elt)
           (gmatch:finish-compile-path rest
             (gmatch:compile-path-elt elt)))
          (else
           (error "Неправильные элементы пути:" elts)))))
```

Меткой ребра может служить любой символ, кроме особых символов (`*`, `+`, `opt`, `or`, `and`), используемых в образцах графового сопоставителя. Сопоставитель простого ребра с меткой компилируется так:

[19] Настоящее имя процедуры `gmatch:finish-compile-path`; в тексте пояснений мы отбрасываем префикс `gmatch:` для краткости.

```
(define (gmatch:compile-edge label target)
  (let ((match-target (gmatch:compile-target target)))
    (define (match-edge object dict succeed)
      (and (graph-node? object)
           (object 'has-edge? label)
           (match-target (object 'edge-value label)
                         dict succeed)))
    match-edge))
```

Сопоставитель ребра `match-edge` проверяет, что переданный ему объект является вершиной графа, что из этой вершины исходит ребро с нужной меткой и что объект на конце ребра (`edge-value`) сопоставляется (при помощи процедуры `match-target`) с выражением для конца в графовом образце. Процедура-сопоставитель `match-target`, которую вызывает `match-edge`, создается компилятором `compile-target`.

При компиляции выражения для конца ребра есть только два варианта: переменная либо константа.

```
(define (gmatch:compile-target elt)
  (if (match:var? elt)
      (gmatch:compile-var elt)
      (let ()
        (define (match-constant object dict succeed)
          (and (eqv? elt object)
               (succeed object dict)))
        match-constant)))
```

Особые формы сопоставителя обрабатываются процедурой `compile-path-elt`:

```
(define (gmatch:compile-path-elt elt)
  (let ((keyword (car elt))
        (args (cdr elt)))
    (case keyword
      ((*) (gmatch:compile-* args))
      ((+) (gmatch:compile-+ args))
      ((opt) (gmatch:compile-opt args))
      ((or) (gmatch:compile-or args))
      ((and) (gmatch:compile-and args))
      (else (error "Неправильный элемент пути:" elt)))))
```

Компиляция образца с необязательными элементами пути работает так: делается рекурсивный вызов `compile-path-elts` с образцами для последовательности необязательных элементов; получается сопоставитель `matcher` для элементов, которые могут быть, а могут не быть на нужном нам пути. Когда процедура `match-opt` применяется к объекту-вершине, сначала применяется этот сопоставитель. Однако, если он терпит неудачу и возвращает `#f`, зовется продолжение успеха с исходным объектом и исходным словарем.

```
(define (gmatch:compile-opt elts)
  (let ((matcher (gmatch:compile-path-elts elts)))
    (define (match-opt object dict succeed)
```

```
      (or (matcher object dict succeed)
          (succeed object dict)))
    match-opt))
```

Образец с повторяющимися элементами, например `(* north (?* ,unoccupied))` из процедуры `basic-queen-move` со стр. 193, компилируется так:

```
(define (gmatch:compile-* elts)
  (gmatch:* (gmatch:compile-path-elts elts)))
```

Как и в случае образца с необязательной последовательностью элементов пути, компилятор рекурсивно вызывается и выдает сопоставитель для возможно повторяющейся последовательности, а затем этот сопоставитель скармливается процедуре `gmatch:*`:

```
(define (gmatch:* matcher)
  (define (match-* object dict succeed)
    (or (matcher object dict
                 (lambda (object* dict*)
                   (match-* object* dict* succeed)))
        (succeed object dict)))
  match-*)
```

Графовый сопоставитель `match-*` пытается применить свой аргумент `matcher` к переданной ему вершине графа. Если сопоставление удачно, `match-*` применит себя рекурсивно к той части графа, где остановилось последнее сопоставление. В конце концов он дальше пойти не сможет и вернет успех для того графового объекта, где `matcher` потерпел неудачу.

Компиляция образцов, требующих хотя бы одно, но разрешающих больше повторений последовательности элементов пути, что обозначается символом `+`, подобна обработке `*`. Здесь также зовется `gmatch:*`, но сначала мы требуем хотя бы одно удачное сопоставление:

```
(define (gmatch:compile-+ elts)
  (let ((matcher (gmatch:compile-path-elts elts)))
    (gmatch:seq2 matcher (gmatch:* matcher))))
```

Остаются особые образцы `and` и `or`. Каждый из них содержит набор подобразцов. Элемент `and` должен сопоставиться со всеми подобразцами, начиная с текущей вершины. Элемент `or` должен сопоставиться хотя бы с одним подобразцом, начиная с текущей вершины.

```
(define (gmatch:compile-and elt-lists)
  (gmatch:and (map gmatch:compile-path-elts elt-lists)))

(define (gmatch:compile-or elt-lists)
  (gmatch:or (map gmatch:compile-path-elts elt-lists)))

(define (gmatch:and matchers)
  (lambda (object dict succeed)
    (if (null? matchers)
```

```
        (succeed object dict)
        (let loop ((matchers matchers) (dict dict))
          ((car matchers) object dict
           (if (null? (cdr matchers))
               succeed
               (lambda (object* dict*)
                 (loop (cdr matchers) dict*))))))))

(define (gmatch:or matchers)
  (lambda (object dict succeed)
    (let loop ((matchers matchers))
      (if (pair? matchers)
          (or ((car matchers) object dict succeed)
              (loop (cdr matchers)))
          #f))))
```

Процедура `compile-var` компилирует переменную образца. Она вызывается из `compile-path` и `compile-target`; в ней четыре взаимоисключающих альтернативы для обработки переменных, где есть и где нет необязательные имя и предикат:

```
(define (gmatch:compile-var var)
  (cond ((match-list? var gmatch:var-type?)
         (gmatch:var-matcher (car var) #f #f))
        ((match-list? var gmatch:var-type? symbol?)
         (gmatch:var-matcher (car var) (cadr var) #f))
        ((match-list? var gmatch:var-type? symbol? procedure?)
         (gmatch:var-matcher (car var) (cadr var) (caddr var)))
        ((match-list? var gmatch:var-type? procedure?)
         (gmatch:var-matcher (car var) #f (cadr var)))
        (else
         (error "Неправильная переменная:" var))))
```

Процедура `var-type?` сопоставляется с символом типа переменной образца: `?` или `?*`. Для распознавания четырех разновидностей переменных `compile-var` использует вспомогательную процедуру `match-list?`, которая возвращает истину, если ее первый аргумент — список, и каждый его элемент удовлетворяет соответствующему аргументу-предикату.

```
(define (match-list? datum . preds)
  (let loop ((preds preds) (datum datum))
    (if (pair? preds)
        (and (pair? datum)
             ((car preds) (car datum))
             (loop (cdr preds) (cdr datum)))
    (null? datum))))
```

Теперь, когда мы разобрали синтаксис переменных, процедура `var-matcher` служит сопоставителем для них.

```
(define (gmatch:var-matcher var-type var-name restriction)
  (define (match-var object dict succeed)
```

```
    (and (or (not restriction)
             (restriction object dict))
         (if var-name
             (let ((dict*
                    (gmatch:bind var-type var-name object
                                 dict)))
               (and dict*
                    (succeed object dict*)))
             (succeed object dict))))
  match-var)
```

Вызов `bind` здесь добавляет связывание переменной `var-name` со значением `var-object` и возвращает новый словарь. Если в словаре уже есть связывание для переменной, а значение отличается от `object`, процедура сообщает о неудаче, возвращая `#f`.

На этом мы закончили писать сопоставитель для графов.

Упражнение 4.25. Сопоставление графов
Описанный здесь графовый сопоставитель весьма полезен, но для некоторых задач он приспособлен плохо. Какие интересные задачи требуют расширения сопоставителя? Найдите такую задачу, определите и реализуйте расширения и продемонстрируйте работу улучшенного сопоставителя на примерах.

4.6 Заключение

Образцы — вещь замечательная, но, кроме того, это очень полезный способ аддитивной организации частей системы. В этой главе мы видели, как строится система переписывания термов. Система переписывания термов, основанная на правилах, легко позволяет писать программы, которые последовательно заменяют фрагменты выражения на «эквивалентные» фрагменты и останавливаются, когда все возможные замены сделаны. Такие системы являются важными компонентами систем большего размера, предназначенных для манипуляций с символами. Одно из применений переписывания термов — упрощение алгебраических выражений, но еще больше подобной работы проделывают компиляторы, где таким образом построено вычисление оптимизаций, а иногда и порождение кода.

Мы показали также гибкий способ построения сопоставителя, а именно «компиляцию» образца в комбинацию элементарных сопоставителей, обладающих одинаковыми интерфейсами. При этом легко добавлять новые виды сопоставителей, и такую систему нетрудно сделать очень эффективной. При добавлении сегментных переменных, которые сопоставляются с неизвестным заранее числом элементов, нам приходится реализовать систему поиска с возвратами, поскольку с каждой конкретной структурой данных образец, если в нем больше одной сегментной переменной, может сопоставиться несколькими способами. Это значительно усложняет программу сопоставления. Помимо естественной сложности, присущей перебору с возвратами, нужно установить интерфейс между перебором внутри сопоставителя и системой перебора в программе обработки правил, которая вызывает сопоставитель. В разделе 5.4 мы рассмотрим более общие способы работы с перебором. Еще более мощные стратегии перебора с возвратами будут исследованы в разделе 7.5.2.

Если мы моделируем частично специфицированные данные в виде образцов с дырками (которым соответствуют переменные образца), то оказывается, что образцы нам надо сопоставлять друг с другом, чтобы собрать ограничения на данные и получить более полную их спецификацию. Мы рассмотрели *унификацию*: процесс слияния структур такого рода с частичной информацией. В сущности, это способ построения и решения символических уравнений для недостающих фрагментов данных. Унификация — мощный инструмент; мы показали, как на ее основе построить простой механизм вывода типов.

Мы обнаружили, что основные идеи сопоставления с образцом применимы не только к иерархическим выражениям, но и к графовым структурам общего вида. Это значительно упростило работу с такими сложными структурами, как шахматные доски, где с помощью образцов мы построили описание разрешенных ходов.

Образцы и сопоставление с образцами могут служить способом выражения математических мыслей, и в некоторых задачах они вносят больше ясности, чем любая другая программистская методика. Однако будьте осторожны: образцы не являются ответом на все вопросы мироздания, и не стоит питать к ним чрезмерного пристрастия.

Глава 5

Вычисление

Один из лучших способов работы над задачей — создать специализированный язык, на котором решение легко будет записать. Если придуманный вами язык будет достаточно мощным, то многие другие задачи, подобные той, которую вы решаете сейчас, тоже окажутся легко выражаемыми при помощи этого языка. Эта стратегия особенно эффективна, если начать с достаточно гибкого механизма. Мы исследовали эту идею в ограниченных контекстах в главах 2, 3 и 4. Пришло время рассмотреть ее в наиболее общем виде.

Создавая язык, мы должны придать ему значение. Если мы хотим, чтобы выражения языка описывали вычислительные процессы, требуется построить механизм, который, получив выражение нашего языка, выполнит нужный процесс. *Интерпретатор* — это как раз такой механизм. Мы исследуем эту область творчества, начиная с расширяемой версии интерпретатора `eval/apply` для языка Scheme с аппликативным порядком вычисления, подобного описанным в главе 4 книги SICP [1].

Процедуры языка Scheme строгие и требует, чтобы все их аргументы были вычислены до начала выполнения тела. Мы обобщим наш интерпретатор и добавим к списку параметров процедуры объявления. Эти объявления позволят процедуре откладывать вычисление соответствующего аргумента до тех пор, пока его значение реально не понадобится, и таким образом обеспечат ленивое вычисление, с мемоизацией значения или без оной. Механизм объявлений можно будет использовать и для другой информации, например для типов или единиц измерения.

Интерпретатор достаточно неэффективен, поскольку на каждом шаге требуется анализировать выражение, подлежащее интерпретации, чтобы понять, что нужно сделать. Этот шаг повторяется каждый раз, когда интерпретатор встречает одно и то же выражение. Поэтому нужно разделить интерпретацию на две фазы, анализ и исполнение. Исполнительная процедура работает, не имея доступа к выражению, из которого она скомпилирована. Все исполнительные процедуры имеют одинаковую форму и образуют систему комбинаторов.

Затем мы добавляем оператор `amb` Джона Маккарти, который позволяет производить недетерминистские вычисления и поиск. Интересно то, что при этом в анализирующей части интерпретатора не требуется никаких изменений. Нужно только изменить формат исполнительных процедур, переведя их в стиль с передачей

продолжений. Использование этого стиля подсказывает идею, что можно дать программисту доступ к нижележащему продолжению.

Процедура `call/cc`, которая дает доступ к нижележащему продолжению, является в языке Scheme стандартной. Оказывается, что ее достаточно, чтобы реализовать `amb` прямо в рамках Scheme. В завершение главы мы показываем, как это сделать.

5.1 Полиморфный интерпретатор `eval/apply`

Наш первый интерпретатор строится как расширяемый. Все его существенные части являются полиморфными процедурами, и мы тщательно избегаем лишних обязательств. Начнем!

Сущность интерпретатора составляют две процедуры: `eval` и `apply`. Процедура `eval` принимает на вход выражение и окружение. Выражение является комбинацией синтаксически склеенных между собой подвыражений. Окружение предоставляет значения для некоторых встречающихся в выражении символов. Существуют другие символы, значения которых закреплены в определении процедуры `eval`[1]. Однако большинство выражений интерпретируется как комбинации *оператора* и *операндов*. Вычисление оператора должно дать нам процедуру, а вычисление операндов — ее аргументы. Процедура и аргументы передаются процедуре `apply`. Эта процедура обычно дает аргументам имена через *формальные параметры*. Затем `apply` вычисляет *тело* процедуры (с помощью `eval`) в окружении, где формальные параметры процедуры *связаны* с аргументами. Так устроен центральный вычислительный цикл интерпретатора.

То, что мы сейчас описали, — это традиционный план строения интерпретатора с аппликативным порядком вычисления. В нашем интерпретаторе мы будем передавать в `apply` невычисленные операнды и окружение для их вычисления, чтобы сделать возможной реализацию различных стратегий вычисления, в том числе нормальный порядок наряду с аппликативным.

Мы будем реализовывать вариант языка Lisp[2]. Отсюда следует, что код выражается в виде списковых структур. В Lisp все составные выражения являются списками, и некоторые из них начинаются с выделенных ключевых слов. Составные выражения, начинающиеся с ключевых слов, называются *особые формы*. Составные выражения, не являющиеся особыми формами, интерпретируются как *применение* процедур к аргументам. Наша реализация будет организована в виде набора правил для каждого вида выражений, за исключением применений, которые опознаются по тому, что они не являются особыми формами. Каждое правило дает синтаксическое определение для некоторого типа выражений. С помощью такой стратегии можно реализовать почти любой язык; правда, потребуется программа синтаксического разбора. В случае языка Lisp процедура чтения преобразует последовательность символов в списковые структуры, которые естественным образом представляют аб-

[1] Однако, поскольку `eval` — полиморфная процедура, множество символов, в ней определенных, можно легко расширить, даже динамически.

[2] Реализация интерпретатора упрощается оттого, что базовым языком служит Scheme. От языка реализации мы наследуем процедуру чтения, что делает наш синтаксис крайне простым; наследуется хвостовая рекурсия, и поэтому нам не требуется соблюдать особую осторожность при реализации вызовов процедур; наконец, мы используем процедуры Scheme в качестве примитивов. Если бы мы выбрали для реализации другой язык, скажем C, пришлось бы заботиться о многих других вещах. Однако интерпретатор такого рода можно построить на любом языке.

страктное синтаксическое дерево (abstract syntax tree, AST) языка. В других языках структура этого дерева богаче, а программа синтаксического анализа намного сложнее.

5.1.1 eval

Мы определяем `g:eval` как полиморфную процедуру с двумя аргументами.

```
(define g:eval
  (simple-generic-procedure 'eval 2 default-eval))
```

Умолчательным случаем здесь является *применение* (иногда также называемое *комбинацией*).

```
(define (default-eval expression environment)
  (cond ((application? expression)
         (g:apply (g:advance
                   (g:eval (operator expression)
                           environment))
                  (operands expression)
                  environment))
        (else
         (error "Неизвестный тип выражения" expression))))
```

В языках на основе Lisp оператором в списке, представляющем применение, служит первый элемент, а остальные элементы являются операндами.

```
(define (application? exp) (pair? exp))
(define (operator app) (car app))
(define (operands app) (cdr app))
```

Заметим, как наш код следует образцу, описанному на стр. 210. Мы одновременно представляем и интерпретацию некоторой конструкции (применения), и определение ее синтаксиса. Кроме того, как там сказано, нужно рассматривать применение как умолчательный случай полиморфной процедуры, поскольку в Lisp нет специального ключевого слова для вызова процедуры — вместо этого применение распознается как список, *не* начинающийся ни с одного ключевого слова.

При применении мы сначала вычисляем операторную часть выражения, а затем передаем ее в процедуру `g:apply` наряду с операндами выражения и текущим окружением. Однако после того, как оператор вычислен, мы пропускаем его через полиморфную процедуру `g:advance`. Целью `g:advance` является завершить вычисления, которые были отложены. До раздела 5.2 мы никакие вычисления задерживать не будем, так что пока `g:advance` просто возвращает свой аргумент[3].

```
(define g:advance
  (simple-generic-procedure 'g:advance 1 (lambda (x) x)))
```

Это не традиционный способ определить `apply`. Передавая туда невычисленные операнды и окружение, мы оставляем себе возможность ввести нормальный порядок

[3]Префикс `g:` в имени `g:apply` и других показывает, что это часть «общего» (generic) интерпретатора. В последующих разделах мы введем другие версии интерпретатора, и у каждой будет свой собственный префикс.

вычисления наряду с аппликативным; кроме того, это позволит нам реализовать объявления формальных параметров и, возможно, другие расширения.

Для каждого типа выражений, кроме применения, мы создаем обработчик. Самовычисляющиеся выражения возвращают сами себя:

```
(define-generic-procedure-handler g:eval
  (match-args self-evaluating? environment?)
  (lambda (expression environment) expression))
```

В языках семейства Lisp самовычисляющиеся выражения включают в себя числа, булевы значения и строки. В языке Scheme предикат `number?` устроен довольно сложно. Ему удовлетворяют такие объекты, как целые числа различного размера, рациональные дроби, вещественные и комплексные числа[4].

```
(define (self-evaluating? exp)
  (or (number? exp)
      (boolean? exp)
      (string? exp)))
```

Могут быть и другие самовычисляющиеся выражения, так что, чтобы сделать этот обработчик совсем гибким, мы могли бы определить `self-evaluating?` как полиморфную процедуру. Однако здесь это делать необязательно, поскольку для любого нового типа самовычисляющихся объектов можно просто создать новый обработчик `g:eval`.

В языках, позволяющих работать с символьными выражениями, требуется конструкция *кавычки*[5]. Кавычка — это выражение, защищающее свое подвыражение от вычисления.

```
(define-generic-procedure-handler g:eval
  (match-args quoted? environment?)
  (lambda (expression environment)
    (text-of-quotation expression)))
```

В языках лисповской традиции представлением закавыченного выражения в виде списка служит список, начинающийся с ключевого слова `quote`. Процедура чтения (парсер) разворачивает всякое выражение, начинающееся с символа апострофа (например `'(a b c)`) в закавыченное выражение (здесь это `(quote (a b c))`).

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation quot) (cadr quot))
```

Помеченный список — это просто список, начинающийся с данного уникального символа:

```
(define (tagged-list? e t) (and (pair? e) (eq? (car e) t)))
```

[4]Вещественные числа обычно представляются в компьютере в виде чисел с плавающей точкой. Компонентами комплексного числа в Scheme могут быть, наряду с вещественными, целые и рациональные числа.

[5]Понимание закавычивания и его отношения к вычислению влечет глубокие последствия в аналитической философии языка. Одно из хороших обсуждений этой темы содержится в докторской диссертации Брайана Кантвелла Смита 1982 г. [112]

Переменные Scheme просто ищутся в окружении. В других языках правила, касающиеся переменных, могут быть более сложными. Например, в C есть l-значения и r-значения, которые обрабатываются по-разному.

```
(define-generic-procedure-handler g:eval
  (match-args variable? environment?)
  lookup-variable-value)
```

В языках семейства Lisp переменные представляются символами[6].

```
(define (variable? exp) (symbol? exp))
```

Процедура `lookup-variable-value` ищет свой аргумент в данном окружении. Если значения для переменной не найдено, она ищет значение в нижележащей системе Scheme[7]. Если и там значения нет, интерпретатор сообщает об ошибке `Несвязанная переменная`.

Обработчик для условных выражений с двумя ветвями (*if-then-else*) устроен просто. Если предикатная часть выражения при вычислении дает истинное значение, нужно вычислить часть-следствие, иначе — альтернативную часть выражения.

```
(define-generic-procedure-handler g:eval
  (match-args if? environment?)
  (lambda (expression environment)
    (if (g:advance
         (g:eval (if-predicate expression) environment))
        (g:eval (if-consequent expression) environment)
        (g:eval (if-alternative expression) environment))))
```

Для вычисленного предиката требуется позвать `g:advance`, поскольку, чтобы принять решение, нам нужно знать значение. Заметим, что вычислитель для `if` использует в своей работе конструкцию `if` языка реализации!

Синтаксис выражения `if` в Lisp также несложен. Если альтернатива не указана, значением выражения `if` с ложным предикатом будет значение глобальной переменной `the-unspecified-value`.

```
(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'the-unspecified-value))

(define (make-if pred conseq alternative)
  (list 'if pred conseq alternative))
```

[6]Символ — это атомарный объект, имеющий в качестве имени строку. Символ делает интересным то, что он уникален: любые два экземпляра символа с одинаковыми строковыми именами считаются идентичными (одинаковыми в смысле `eq?`).

[7]Многие элементарные процедуры, будучи найденными таким способом, работают правильно, например `car` или `+`. Однако элементарные процедуры, принимающие процедуры в качестве аргументов, например `map` или `filter`, не смогут обработать неэлементарные процедуры (т. е. созданные интерпретатором из лямбда-выражений). Эта проблема решается в упражнении 5.5 на стр. 222.

Первая действительно интересная особая форма — спецификация анонимной процедуры, представленной выражением `lambda`. Выражение `lambda` — это особая форма, конструктор процедур. При вычислении этой формы получается процедура на основе формальных параметров, тела и текущего окружения. Окружение должно храниться внутри процедуры, если переменные языка имеют лексическую область действия. В языке с лексической областью действия свободные переменные в теле лямбда-выражения (те, которые не являются формальными параметрами) получают значение из лексического контекста (где лямбда-выражение находится в тексте программы).

```
(define-generic-procedure-handler g:eval
  (match-args lambda? environment?)
  (lambda (expression environment)
    (make-compound-procedure
     (lambda-parameters expression)
     (lambda-body expression)
     environment)))
```

Синтаксис лямбда-выражений определяется так:

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters lambda-exp) (cadr lambda-exp))

(define (lambda-body lambda-exp)
  (let ((full-body (cddr lambda-exp)))
    (sequence->begin full-body)))

(define (make-lambda parameters body)
  (cons 'lambda
        (cons parameters
              (if (begin? body)
                  (begin-actions body)
                  (list body)))))
```

Заметим, что тело лямбды может содержать несколько выражений. Они должны выполняться последовательно, чтобы позволить нам работать с побочными эффектами вроде присваивания или вводом—выводом, например распечаткой. Для обработки последовательностей служит `sequence->begin`, создающая особую форму `begin`.

```
(define (sequence->begin seq)
  (cond ((null? seq) seq)
        ((null? (cdr seq)) (car seq))
        (else
         (make-begin
          (append-map (lambda (exp)
                        (if (begin? exp)
                            (begin-actions exp)
                            (list exp)))
                      seq)))))
```

Заметим, что процедура `sequence->begin` уплощает вложенные формы `begin` с сохранением порядка. Синтаксис и вычисление форм `begin` определяются и описываются на стр. 216.

Производные типы выражений

Уже описанные типы выражений достаточны для удобного написания большинства программ, однако часто бывает приятно добавить немного синтаксического сахара. Это можно реализовать, преобразуя выражения в комбинации более простых выражений. Способом обобщить такие трансформации могут быть макросы, однако мы решили в наш интерпретатор расширитель макросов не включать[8]. Здесь мы явным образом показываем, как условное выражение языка Lisp со многими ветвями можно преобразовать во вложенные выражения `if`.

```
(define-generic-procedure-handler g:eval
  (match-args cond? environment?)
  (lambda (expression environment)
    (g:eval (cond->if expression)
            environment)))
```

Процедура `cond->if` представляет собой достаточно простую манипуляцию с данными:

```
(define (cond->if cond-exp)
  (define (expand clauses)
    (cond ((null? clauses)
           (error "COND: ни одно значение не сработало"))
          ((else-clause? (car clauses))
           (if (null? (cdr clauses))
               (cond-clause-consequent (car clauses))
               (error "COND: ELSE не в конце"
                      cond-exp)))
          (else
           (make-if (cond-clause-predicate (car clauses))
                    (cond-clause-consequent (car clauses))
                    (expand (cdr clauses))))))
  (expand (cond-clauses cond-exp)))
```

Вот синтаксис особой формы `cond`:

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))
```

[8] Существенная проблема с макросами состоит в том, что они могут вводить новые связывания переменных, конфликтующие с существующими связываниями. Это делает макросы референциально непрозрачными. Существует несколько подходов к проблеме референциальной прозрачности, и они ведут к развитию *гигиенических* макросов языка Scheme (см. [73, 74, 8, 31]). Кроме того, резкая модификация языка при помощи новых особых форм усложняет понимание программы для читателя — приходится сначала выучить эти особые формы, а уже потом читать программу, которая их использует.

```
(define (cond-clause-predicate clause) (car clause))

(define (cond-clause-consequent clause)
  (sequence->begin (cdr clause)))

(define (else-clause? clause)
  (eq? (cond-clause-predicate clause) 'else))
```

Поскольку выражение `cond` позволяет в следствии ветви находиться нескольким выражениям, это определение также зависит от `sequence->begin`.

Локальные переменные могут вводиться с помощью выражения `let`. Эти выражения реализуются с помощью перевода в явные выражения `lambda`:

```
(define-generic-procedure-handler g:eval
  (match-args let? environment?)
  (lambda (expression environment)
    (g:eval (let->combination expression)
            environment)))
```

Синтаксис выражений `let` таков:

```
(define (let? exp) (tagged-list? exp 'let))

(define (let-bound-variables let-exp)
  (map car (cadr let-exp)))

(define (let-bound-values let-exp)
  (map cadr (cadr let-exp)))

(define (let-body let-exp)
  (sequence->begin (cddr let-exp)))

(define (let->combination let-exp)
  (let ((names (let-bound-variables let-exp))
        (values (let-bound-values let-exp))
        (body (let-body let-exp)))
    (cons (make-lambda names body)
          values)))
```

Эффекты

Если в языке есть выражения, которые производят эффекты, такие как присваивания или печать, они должны выполняться последовательно, поскольку имеет значение порядок. В синтаксисе языка Scheme мы представляем такие последовательности операций с помощью `begin`:

```
(define-generic-procedure-handler g:eval
  (match-args begin? environment?)
  (lambda (expression environment)
    (evaluate-sequence (begin-actions expression)
```

```
                        environment)))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))
(define (make-begin actions) (cons 'begin actions))
```

Настоящая работа происходит при вычислении последовательности:

```
(define (evaluate-sequence actions environment)
  (cond ((null? actions)
         (error "Пустая последовательность"))
        ((null? (cdr actions))
         (g:eval (car actions) environment))
        (else
         (g:eval (car actions) environment)
         (evaluate-sequence (cdr actions)
                            environment))))
```

Значение, возвращаемое при вычислении непустой последовательности выражений, — это значение последнего выражения в этой последовательности. Однако эффекты, вызываемые исполнением последовательности, происходят в указанном последовательностью порядке.

Большая часть эффектов реализуется путем присваиваний переменным. (В сущности, даже операции ввода—вывода на уровне аппаратуры обычно реализуются путем присваивания особо чувствительным позициям в пространстве адресов.) В языке Scheme мы позволяем программе делать присваивание переменной в лексическом окружении выражения присваивания:

```
(define-generic-procedure-handler g:eval
  (match-args assignment? environment?)
  (lambda (expression environment)
   (set-variable-value! (assignment-variable expression)
                        (g:eval (assignment-value expression)
                                environment)
                        environment)))
```

Синтаксис присваивания таков:

```
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable assn) (cadr assn))
(define (assignment-value assn) (caddr assn))
```

Кроме того, мы позволяем иметь определения, создающие новые переменные с заданным значением. Определение создает переменную в ближайшем лексическом кадре окружения определяющего выражения.

```
(define-generic-procedure-handler g:eval
  (match-args definition? environment?)
  (lambda (expression environment)
    (define-variable! (definition-variable expression)
                      (g:eval (definition-value expression)
```

```
                              environment)
                        environment)
    (definition-variable expression)))
```

Синтаксис определений сложнее, чем синтаксис присваиваний, поскольку мы разрешаем определять процедуры несколькими способами[9]:

```
(define (definition? exp) (tagged-list? exp 'define))

(define (definition-variable defn)
  (if (variable? (cadr defn))       ; (DEFINE foo       ...)
      (cadr defn)
      (caadr defn)))                ; (DEFINE (foo ...) ...)

(define (definition-value defn)
  (if (variable? (cadr defn))       ; (DEFINE foo       ...)
      (caddr defn)
      (cons 'lambda                 ; (DEFINE (foo p...) b...)
            (cons (cdadr defn)      ; =(DEFINE foo
                  (cddr defn)))))   ;    (LAMBDA (p...) b...))
```

На этом мы закончили обычный список особых форм, определяющих синтаксис языка. Разумеется, реализация на основе полиморфных процедур позволяет легко добавлять новые формы и расширять язык, чтобы удобно было выражать вычислительные идеи, не поддерживаемые базовым языком. Однако язык с большим числом синтаксических конструкций может оказаться трудно изучить, документировать и использовать; это классический инженерный компромисс (напомним максиму Алана Перлиса со стр. 146).

5.1.2 `apply`

Традиционная процедура `apply` языка Scheme принимает два аргумента: процедуру, которую требуется применить, и вычисленные аргументы, которые ей надо передать. Для Scheme этого достаточно, поскольку Scheme — язык со строгим аппликативным порядком вычислений, а все переменные в нем имеют лексическую область действия. Мы обобщаем `apply` и передаем ей три аргумента — процедуру, подлежащую применению, невычисленные операнды и окружение вызова. Таким способом мы делаем возможными процедуры, требующие нормального порядка вычислений для некоторых аргументов (например, вызов по необходимости), а также процедуры с объявлениями на параметрах, скажем, с типами или единицами измерения. Некоторые расширения такого рода мы создадим в разделе 5.2. Кроме того, аргумент-окружение позволяет работать с переменными, у которых область действия не лексическая, однако в эту сторону мы не пойдем — обычно это плохая идея. Начнем с аппликативного порядка, принятого в Scheme, но оставим крючки для расширения.

Итак, наша `apply` — это полиморфная процедура с тремя аргументами:

```
(define g:apply
  (simple-generic-procedure 'apply 3 default-apply))
```

[9] В MIT/GNU Scheme для определений разрешен еще более общий синтаксис, где `cadr` формы `define` раскрывается рекурсивно (см. стр. 334). Здесь мы так не делаем.

```
(define (default-apply procedure operands calling-environment)
  (error "Неизвестный тип процедуры" procedure))
```

Нам потребуются обработчики для различных видов процедур. Некоторые процедуры, скажем, арифметическое сложение (обычно обозначаемое знаком +), *строгие*: все их аргументы должны быть вычислены, прежде чем они смогут выдать значение. В языке Scheme строгими являются все процедуры, включая элементарные (реализуемые системой или аппаратурой компьютера на более низком уровне, чем язык). Поэтому нам нужен обработчик для строгих процедур.

```
(define-generic-procedure-handler g:apply
  (match-args strict-primitive-procedure?
              operands?
              environment?)
  (lambda (procedure operands calling-environment)
    (apply-primitive-procedure procedure
      (eval-operands operands calling-environment))))
```

На этом уровне детализации применение элементарной процедуры происходит «волшебным» образом. Вычислитель операндов, подобно `if` на стр. 213, должен применить к результату вычислений `g:advance`, чтобы наверняка получить значение.

```
(define (eval-operands operands calling-environment)
  (map (lambda (operand)
         (g:advance (g:eval operand calling-environment)))
       operands))
```

Заметим, что порядок вычисления операндов определяется поведением процедуры `map`.

Процедуры, создаваемые при вычислении выражений `lambda`, не являются элементарными. Здесь нужно обратиться к компонентам процедуры. Берем спецификацию параметров, а именно имена формальных параметров. Берем также тело процедуры и передаем его в `eval` вместе с окружением, включающим связывания для формальных параметров. В случае лексической области действия расширенное окружение строится на основе окружения, упакованного вместе с процедурой при вычислении лямбда-выражения, которое эту процедуру создало.

```
(define-generic-procedure-handler g:apply
  (match-args strict-compound-procedure?
              operands?
              environment?)
  (lambda (procedure operands calling-environment)
    (if (not (n:= (length (procedure-parameters procedure))
                  (length operands)))
        (error "Неправильное число операндов"))
        (g:eval (procedure-body procedure)
                (extend-environment
                 (procedure-parameters procedure)
                 (eval-operands operands calling-environment)
                 (procedure-environment procedure))))))
```

Предикат `strict-compound-procedure` истинен для всех составных процедур, у которых на параметрах нет никаких деклараций[10].

Управляющий цикл

Для взаимодействия с этим вычислителем требуется цикл чтения, вычисления и печати:

```
(define (repl)
  (check-repl-initialized)
  (let ((input (g:read)))
    (write-line (g:eval input the-global-environment))
    (repl)))
```

Процедура `g:read` выдает на терминал подсказку `eval>`. Затем она считывает символы и проводит синтаксический анализ, преобразуя их в s-выражение. Затем это s-выражение вычисляется процедурой `g:eval` относительно глобального окружения `the-global-environment`, а результат выводится обратно на терминал. Процедура `repl` вызывает сама себя хвостовым рекурсивным вызовом. Для работы этого хозяйства требуется инициализировать глобальное окружение:

```
(define the-global-environment
  'not-initialized)

(define (initialize-repl!)
  (set! the-global-environment (make-global-environment))
  'done)

(define (check-repl-initialized)
  (if (eq? the-global-environment 'not-initialized)
      (error
       "Интерпретатор не инициализирован. Запустите (init).")))
```

На этом наш элементарный вычислитель завершен.

Упражнение 5.1. Обработка несвязанных переменных

Во всех языках семейства Lisp, включая Scheme, попытка вычислить несвязанный символ приводит к ошибке. Однако в некоторых алгебраических процессах бывает разумно считать, что несвязанный символ является самовычисляющимся. Например, если мы полиморфным образом расширяем арифметику и строим алгебраические выражения с символьными значениями, как в главе 3, иногда бывает полезно разрешать такое:

```
(+ (* 2 3) (* 4 5))
26

(+ (* a 3) (* 4 5))
(+ (* a 3) 20)
```

[10] Здесь мы приняли решение, ограничивающее дальнейшие расширения. То, что мы требуем от списка параметров процедуры быть той же длины, что и список операндов, не дает нам расширить этот обработчик `g:apply` и разрешить процедуры с необязательными параметрами или с остаточными параметрами. То есть мы не сможем определить традиционный лисповский `+`, который принимает неизвестное число аргументов и все их складывает (см., однако, упражнение 5.2).

Полиморфная арифметика у нас позволяла символьные расширения: значения знаков операций + и * строили выражения, когда их аргументы не сводились к числам. Но они не позволяли использовать несвязанные переменные вместо чисел. Здесь же символ a несвязан. Можно захотеть, чтобы он был самовычисляющимся.

a. Напишите полиморфное расширение eval, которое позволяет такое поведение. Чтобы это работало с числовыми элементарными операциями (+, *, -, /), их поведение тоже придется расширить. Заметим, что значения операций придется изменить в окружении нижележащей системы Scheme. Подобно главе 3, можно с помощью механизма полиморфных процедур дать этим символам обработчики, действующие в нижележащей системе Scheme.

b. Расширьте также поведение apply, чтобы несвязанные символы в операторной позиции интерпретировались как литералы функций, для которых известно только имя: `(+ (f 3) (* 4 5)) ==> (+ (f 3) 20)`

В общем случае такое поведение eval и apply опасно, поскольку оно скрывает использование несвязанных переменных по ошибке. Сделайте так, чтобы оно включалось в зависимости от устанавливаемой пользователем переменной: allow-self-evaluating-symbols.

Упражнение 5.2. n-местные процедуры

В примечании 10 на стр. 220 мы говорим о неприятном ограничении, встроенном нами в обработчик g:apply, не позволяющем некоторые виды расширений. Если в языке Scheme на месте списка параметров стоит символ, то этот символ считается единственным параметром, который связывается со списком аргументов[11].

В этом упражнении мы учим интерпретатор принимать в качестве списка параметров одиночный символ, и таким образом определяются процедуры с произвольным списком аргументов. Процедура lambda-parameters в нашем интерпретаторе (стр. 214) спокойно может вернуть одинокий символ, а всюду, где она вызывается, результат передается в make-compound-procedure. Это значение затем извлекается процедурой procedure-parameters и передается в g:apply. Таким образом, изменить нам надо только g:apply.

Сделайте так, чтобы g:apply работала с новым видом составных процедур. Можно этого добиться, переписав существующий обработчик strict-compound-procedure? для g:apply (стр. 219). Однако проще и яснее будет оставить этот обработчик для случая, когда procedure-parameters являются правильным списком, а для случая, когда procedure-parameters — символ, написать новый обработчик.

Упражнение 5.3. Векторы процедур

В математических текстах часто встречается расширение нотации, когда кортеж функций считается функцией, возвращающей кортеж. Например, если `(cos 0.6)` дает результат `0.8253356149096783`, а `(sin 0.6)` дает результат `0.5646424733950354`, то мы ожидаем от вызова `((vector cos sin) 0.6)` возвращаемого значения `#(0.8253356149096783 0.5646424733950354)`.

[11] Параметр, принимающий список оставшихся аргументов после явно объявленных, называется в Scheme *остаточным параметром*. Если есть явно объявленные аргументы, можно использовать в качестве списка аргументов неканонический список (последовательность пар, где последний cdr не указывает на пустой список). Например, `(lambda (a b . c) ...)` — процедура, принимающая по крайней мере два аргумента, связываемые с переменными a и b; все дополнительные аргументы (после первых двух) собираются в список, и этот список становится значением переменной c. Если явно объявленных параметров нет, а есть только остаточный параметр, мы ставим его имя как символ на место списка параметров. Например, в Scheme можно написать `(lambda xs ...)`, определяя таким образом процедуру, принимающую произвольное число аргументов и связывающую со списком аргументов переменную xs.

Хотя в упражнении 3.2 мы уже расширяли арифметику на векторы, тогда мы не меняли нижележащий вычислитель языка. Нынешнее же расширение должно обогатить поведение `g:apply` и позволить ей обрабатывать векторы из функций как разновидность функций. Постройте это расширение, продемонстрируйте его работу и покажите, что оно нормально справляется с более обычным кодом.

Упражнение 5.4. Ваша очередь!

Изобретите сами новую интересную конструкцию, которую легко реализовать с помощью наших `eval/apply`, но было бы трудно воплотить без такой полиморфной поддержки.

Упражнение 5.5. Взаимодействие с нижележащей системой

Как указано на стр. 213, вычисление выражения

```
eval> (map (lambda (x) (* x x)) '(1 2 3))
```

в нашем интерпретаторе не работает, если `map` в выражении указывает на процедуру `map` нижележащей системы Scheme.

Однако, если мы переопределим `map` внутри системы, оно заработает:

```
eval> (define (map f l)
        (if (null? l)
            '()
            (cons (f (car l)) (map f (cdr l)))))
map

eval> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

Почему выражение не работает с нижележащими процедурами, принимающими процедурные аргументы вроде `map`? Объясните. Наметьте стратегию решения этой проблемы и реализуйте ее. Примечание: эту задачу трудно решить полностью, поэтому не нужно тратить бесконечное время, стараясь учесть все тонкости.

Упражнение 5.6. Другие правила закавычивания

Существовали интересные языки, где правила вычисления и закавычивания строились иначе. Например, в MDL (см. «Википедию» [91]) символ считается самовычисляющимся, а переменные обозначаются особым префиксным значком. Кроме того, в MDL комбинация является особой формой с неявным ключевым словом. Наш вычислитель нетрудно модифицировать так, чтобы он воспринимал MDL-подобный синтаксис, меняя при этом только синтаксические определения. Попробуйте!

Упражнение 5.7. Инфиксная нотация

Большинство компьютерных языков, в отличие от Lisp, использует инфиксную нотацию. Если бы мы включили в Scheme инфиксные выражения, можно было бы писать

```
(infix
  "fact := lambda n:
            if n == 0
               then 1
               else n*fact(n-1)")
(fact 6)           ; Лисповская процедура определена.
720
```

```
(infix "fact(5)")  ; Ее можно использовать в инфиксной форме.
120
```

Тут всего лишь небольшой вопрос синтаксиса (ха-ха!). Однако заставить это работать — интересный проект. Интерпретатор менять не надо. Задача состоит в том, чтобы разобрать символьную строку и перевести ее в соответствующее лисповское выражение, так же как работает `cond->if`. Программисты на Lisp много раз это проделывали, однако, похоже, люди, которые пишут на Lisp, предпочитают родную для него префиксную польскую запись[12]. Что ж...

5.2 Процедуры с нестрогими аргументами

В этом разделе мы рассмотрим, как добавить объявления к формальным параметрам процедуры и получить задержанное вычисление соответствующих операндов.

В языке Scheme процедуры *строгие*. Строгие процедуры требуют, чтобы все операнды вызывающего выражения были вычислены, а их значения связаны формальными параметрами до того, как начнется выполнение тела процедуры. Однако в выражении `if` требуется вычислить предикатную часть, чтобы выяснить, нужно ли выполнять подвыражение-следствие или подвыражение-альтернативу; оба они выполнены не будут. Именно поэтому `if` должен быть особой формой, а не процедурой.

Нестрогая процедура откладывает вычисление одного или более из своих операндов. Можем ли мы позволить программисту определять нестрогие процедуры по мере необходимости, вместо того чтобы использовать особые формы вроде `if`, чье значение указано в определении языка?

Например, допустим, что мы хотим написать процедуру `unless`, которая работает подобно `if`, не вычисляя ненужные альтернативы[13]. Используя `unless`, можно написать:

```
(define (fib n)
  (unless (< n 2)
          (+ (fib (- n 1)) (fib (- n 2)))
          n))
```

Чтобы это определение чисел Фибоначчи правильно работало, второй операнд выражения `unless` не должен вычисляться при $n < 2$, а третий не должен вычисляться при $n \geq 2$. Первый операнд выражения `unless` должен вычисляться всегда, чтобы определить выбор.

Нужен способ определить, какие операнды `unless` должны вычисляться, а какие задерживаться. Для этого мы вводим некую разновидность объявлений и пишем:

```
(define (unless condition (usual lazy) (exception lazy))
  (if condition exception usual))
```

Здесь мы выражаем процедуру `unless` в терминах особой формы `if`, но объявляем второй и третий аргументы как ленивые[14]. Первый аргумент по умолчанию строгий.

[12] На веб-сайте этой книги можно найти такой инфиксный парсер.

[13] Можно заметить, что это определение `unless` отличается от того, которое присутствует во многих языках семейства Lisp, включая Scheme [109] и Emacs Lisp.

[14] Согласно общей практике, термин «ленивое вычисление» означает, что вычисление аргумента задерживается, а результат мемоизируется. Здесь мы разделяем эти две идеи, и словом «ленивый» обозначаем только задержку.

Можно иметь много разновидностей объявлений при формальных параметрах, описывающих, как должны обрабатываться операнды и аргументы. Параметр может быть объявлен ленивым и мемоизированным, чтобы реализовать вызов по необходимости, как для всех аргументов процедур в языке Haskell. Можно объявить, что параметр должен удовлетворять каким-то предикатам; эти предикаты могут быть типами или единицами измерения и т. д.

Реализация обобщенных формальных параметров

Чтобы реализовать обобщенные формальные параметры, требуется еще один применитель, который будет разбирать новые случаи. Для этого мы добавляем новый обработчик в `g:apply`, подобно старому, который служит для строгих составных процедур (см. стр. 219):

```
(define-generic-procedure-handler g:apply
  (match-args general-compound-procedure?
              operands?
              environment?)
  (lambda (procedure operands calling-environment)
    (if (not (n:= (length (procedure-parameters procedure))
                  (length operands)))
        (error "Неправильное число операндов"))
        (let ((params (procedure-parameters procedure))
              (body (procedure-body procedure)))
          (let ((names (map procedure-parameter-name params))
                (arguments
                 (map (lambda (param operand)
                        (g:handle-operand param
                                          operand
                                          calling-environment))
                      params
                      operands)))
            (g:eval body
                    (extend-environment names arguments
                      (procedure-environment procedure)))))))
```

Здесь два отличия от строгого применителя: во-первых, нужно извлечь имена аргументов, поскольку они могут быть обернуты объявлениями; во-вторых, операнды нужно обрабатывать особым образом в зависимости от объявлений. За это отвечают полиморфные процедуры `procedure-parameter-name` и `g:handle-operand`.

Процедура `procedure-parameter-name` позволяет нам добавлять объявления к формальному параметру и по-прежнему иметь возможность добраться до его имени. Умолчательный обработчик просто возвращает аргумент, так что у параметра без украшений именем служит он сам.

```
(define procedure-parameter-name
  (simple-generic-procedure 'parameter-name 1 (lambda (x) x)))
```

Процедура `g:handle-operand` позволяет нам выбирать, как нужно обрабатывать операнд, в зависимости от соответствующего формального параметра:

```
(define g:handle-operand
  (simple-generic-procedure 'g:handle-operand 3
    (lambda (parameter operand environment)
      (g:advance (g:eval operand environment)))))
```

Умолчательный способ обработки операнда без объявлений — вычислить его, как раньше делала процедура `eval-operands` со стр. 219.

Нужен синтаксис, чтобы снабжать формальные параметры объявлениями. Мы используем список, начинающийся с имени формального параметра:

```
(define-generic-procedure-handler procedure-parameter-name
  (match-args pair?)
  car)
```

Для начала реализуем два вида объявлений. Во-первых, `lazy` (ленивый) будет означать, что операнд должен вычисляться только тогда, когда требуется его значение, например в предикате выражения `if`. Во-вторых, `lazy memo` (ленивый с мемоизацией) будет подобен `lazy`, но только в первый раз, как значение параметра вычисляется, оно запоминается, а последующие обращения перевычисления не требуют.

Если параметр объявлен как ленивый или ленивый с мемоизацией, вычисление операнда должно быть отложено. Отложенное выражение должно быть упаковано вместе с окружением, которое будет сообщать значения свободных переменных в выражении, когда потребуется его значение[15].

```
(define-generic-procedure-handler g:handle-operand
  (match-args lazy? operand? environment?)
  (lambda (parameter operand environment)
    (postpone operand environment)))

(define-generic-procedure-handler g:handle-operand
  (match-args lazy-memo? operand? environment?)
  (lambda (parameter operand environment)
    (postpone-memo operand environment)))
```

Разумеется, нужно расширить `g:advance`, у которой пока есть только обработчик по умолчанию (см. стр. 211), и научить ее производить задержанные вычисления. Заметим, что результат `g:advance` сам по себе может быть задержкой, так что нужно и его продвинуть.

```
(define-generic-procedure-handler g:advance
  (match-args postponed?)
  (lambda (object)
    (g:advance (g:eval (postponed-expression object)
                       (postponed-environment object)))))
```

Если выражение задержано с намерением мемоизировать результат, результат сохраняется процедурой `advance-memo!`:

[15] В исходной реализации вызова по имени в языке Algol-60 эти сочетания выражения с окружением назывались *санками*. Поскольку программы на Scheme часто используют процедуры без параметров для упаковки выражений, чтобы вычислить их впоследствии в каком-то другом окружении, такие нульместные процедуры мы тоже называем *санками*.

```
(define-generic-procedure-handler g:advance
  (match-args postponed-memo?)
  (lambda (object)
    (let ((value
           (g:advance
            (g:eval (postponed-expression object)
                    (postponed-environment object)))))
      (advance-memo! object value)
      value)))
```

Мемоизированное значение заново вычислять не требуется. Процедура `advance-memo!` изменяет тип задержанного объекта так, чтобы он удовлетворял предикату `advanced-memo?`, и сохраняет значение, до которого теперь можно добраться через `advanced-value`[16]:

```
(define-generic-procedure-handler g:advance
  (match-args advanced-memo?)
  advanced-value)
```

Пример: ленивые пары и списки

Процедуры с ленивыми параметрами дают нам новую выразительную мощность. Например, мы можем определить конструктор `kons` и селекторы `kar` и `kdr` и получить пары, не вычисляющие свое содержимое[17]. Мы реализуем `kons` как процедуру, принимающую аргументы с вызовом по необходимости (ленивые с мемоизацией). Она порождает обработчик сообщений `the-pair` для `kar` и `kdr`. Кроме того, она прикрепляет к `the-pair` «ярлык», чтобы пометить его как результат вызова `kons`.

```
(define (kons (x lazy memo) (y lazy memo))
  (define (the-pair m)
    (cond ((eq? m 'kar) x)
          ((eq? m 'kdr) y)
          (else (error "Неизвестное сообщение -- kons" m x y))))
  (hash-table-set! kons-registrations the-pair #t)
  the-pair)

(define (kar x)
  (x 'kar))

(define (kdr x)
  (x 'kdr))
```

Ярлык нам нужен для того, чтобы распознавать пару, созданную через `kons`:

[16] Помимо этого, процедура `advance-memo!` убирает указатель из задержанного объекта на окружение, где он должен вычисляться. Это позволяет окружению подвергнуться сборке мусора, если на него нет других ссылок.

[17] В древней, но важной статье Дэна Фридмана и Дэвида Уайза под названием «`Cons` не должен вычислять свои аргументы» [40] было показано, насколько богаты возможности ленивого функционального программирования. Ее можно смоделировать с нашим `kons` вместо `cons`.

```
(define (kons? object)
  (hash-table-exists? kons-registrations object))
```

При помощи механизма ленивых пар мы можем легко реализовать обработку потоков. Потоки подобны спискам, но они строятся по мере необходимости процессами, которые их потребляют[18]. Поэтому бесконечно длинный поток может обрабатываться шаг за шагом, так что в каждый момент существует только его часть.

Некоторые потоки конечны, поэтому полезно выбрать представление для пустого потока. Пусть оно будет такое же, как у пустого списка:

```
(define the-empty-stream '())

(define (empty-stream? thing)
  (null? thing))
```

Потоки можно складывать:

```
(define (add-streams s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else
         (kons (+ (kar s1) (kar s2))
               (add-streams (kdr s1) (kdr s2))))))
```

Можно обращаться к n-му элементу потока:

```
(define (ref-stream stream n)
  (if (= n 0)
      (kar stream)
      (ref-stream (kdr stream) (- n 1))))
```

Этих процедур достаточно, чтобы построить (потенциально бесконечный) поток чисел Фибоначчи, где есть два начальных элемента, а остальные получаются путем сложения самого потока с его `kdr`:

```
(define fibs
  (kons 0 (kons 1 (add-streams (kdr fibs) fibs))))
```

Можно взглянуть на несколько чисел Фибоначчи:

```
(ref-stream fibs 10)
55

(ref-stream fibs 100)
354224848179261915075
```

Обычная программа вычисления чисел Фибоначчи с двойной рекурсией имеет экспоненциальную сложность, так что таким методом нельзя надеяться получить 100-й элемент последовательности; но поскольку пары `kons` мемоизируются, задача сводится к линейной. Заметим, что к этой точке последовательности отношение двух соседних чисел Фибоначчи сошлось к золотому сечению со всей возможной точностью:

[18] В языке Scheme [109] для реализации потоков есть `delay` и `force`. Дополнительную информацию о потоках можно найти в SICP [1] и SRFI-41 [13].

```
(inexact
  (/ (ref-stream fibs 100)
     (ref-stream fibs 99)))
1.618033988749895
```

Упражнение 5.8. Интегрирование дифференциальных уравнений

К сожалению, использование `kons` само по себе не решает всех проблем с потоками. Например, проблема, которая в SICP [1] упоминается в разделе 4.2.3 (стр. 411) автоматически не исчезает. Допустим, нам надо проинтегрировать дифференциальное уравнение при некоторых начальных условиях. Строим такие определения:

```
(define (map-stream proc (items lazy memo))
  (if (empty-stream? items)
      items
      (kons (proc (kar items))
            (map-stream proc (kdr items)))))

(define (scale-stream items factor)
  (map-stream (lambda (x) (* x factor))
              items))

(define (integral integrand initial-value dt)
  (define int
    (kons initial-value
          (add-streams (scale-stream integrand dt)
                       int)))
  int)

(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map-stream f y))
  y)
```

Мы пытаемся найти приближение к числу e, проинтегрировав уравнение $x'(t) = x(t)$ при начальных условиях $x(0) = 1$. Известно, что $e = x(1)$, так что пишем

```
(ref-stream (solve (lambda (x) x) 1 0.001) 1000)
;Несвязанная переменная: dy
```

Ошибка!

Однако теперь у нас достаточно инструментов для исправления проблемы. Что нужно изменить,чтобы программа заработала как задумано? Исправьте ее так, чтобы получить:

```
(ref-stream (solve (lambda (x) x) 1 0.001) 1000)
2.716923932235896
```

(Да, мы знаем, что это очень грубое приближение числа e, но это упражнение показывает программистский прием, а не метод численного анализа!)

Упражнение 5.9. Почему `kons` не победил?

Особая форма `kons` эквивалентна `cons`, где оба аргумента ленивы и мемоизированы. Если бы мемоизации не было, вычисление `(ref-stream fibs 100)` занимало бы очень долгое время.

a. Есть ли преимущества у отказа от мемоизации? Когда это может иметь значение?

b. Почему нельзя было определить `kons` просто как

```
(define (kons (a lazy memo) (d lazy memo))
  (cons a d))
```

через встроенный `cons` из Scheme?

c. Вообще говоря, лисповское сообщество избегает менять поведение `cons` на `kons`, как рекомендуют Фридман и Уайз (см. примечание 17 на стр. 226). Каких потенциально серьезных проблем можно избежать, используя `cons`, а не `kons`? Предполагайте, что о малых константных множителях в производительности мы не заботимся.

Упражнение 5.10. Ограниченные параметры

Интересна идея встроить в объявление формального параметра ограничения. Можно потребовать, чтобы на параметре выполнялся произвольный предикат, подобно ограничениям, которые мы накладывали на переменные образца в разделе 4.3.2. К примеру, мы можем хотеть, чтобы процедура принимала три аргумента: первый должен быть целым числом, второй — простым, а третий может быть каким угодно. Это может выражаться так:

```
(define (my-proc (n integer?) (p prime?) g)
  ...)
```

К сожалению, такая приспособленная к конкретной задаче нотация плохо согласуется с другими объявлениями, например `lazy` и `memo`, если мы не установим строгий порядок объявлений или не сделаем их зарезервированными идентификаторами. Для удобства предположим, что мы объявляем `lazy` и `memo` особыми ключевыми словами, а остальные объявления тоже должны вводиться ключевым словом, например для предиката это будет `restrict-to`:

```
(define (my-proc (n restrict-to integer?)
                 (p restrict-to prime? lazy)
                 g)
  ...)
```

a. Проработайте соответствующий синтаксис. Убедитесь, что он расширяем, т. е. к нему можно добавлять новые виды объявлений. Выразите синтаксис через формы Бэкуса—Наура и измените синтаксические процедуры интерпретатора, поддержав его.

b. Реализуйте предикатные ограничения. Если во время выполнения ограничение нарушается, программа должна сигнализировать об ошибке. Может пригодиться процедура `guarantee`.

Упражнение 5.11. Опять n-местные процедуры!

a. В упражнении 5.2 на стр. 221 мы модифицировали `g:apply`, разрешив формальным параметрам процедуры быть выраженными одиночным символом, который связывается со списком аргументов. К сожалению, такой способ выразить остаточный аргумент плохо согласуется с системой, где формальные параметры снабжаются объявлениями. Однако можно придумать синтаксис для объявлений, который позволит нам объявлять процедуры с необязательными и остаточными аргументами.

Например, если мы позволим, чтобы последний формальный параметр в списке помечался словом `rest`, он будет связываться с необработанными аргументами. Объявление `rest` должно свободно сочетаться с другими объявлениями при том же аргументе. Должно быть возможно создавать процедуры вроде

```
(lambda (x
         (y restrict-to integer? lazy)
         (z rest restrict-to list-of-integers?))
  ...)
```

где предикат `list-of-integers?` истинен относительно списков целых чисел. Должно быть возможно использовать `rest` и с другими объявлениями, скажем, `lazy` и `restrict-to`.

Заставьте объявления `rest` работать!

b. Может быть полезно также давать процедурам необязательные аргументы, возможно, со значениями по умолчанию. Например, численная процедура может позволить пользователю задавать желаемую точность приближения, но указать умолчательное значение на случай, если пользователь ее пропустит:

```
(lambda (x (epsilon optional flo:ulp-of-one))
  ...)
```

Здесь `flo:ulp-of-one` — глобальный символ, указывающий наименьшую степень двойки, которая, будучи добавлена к 1.0, дает значение, не равное 1.0. В библиотеке языка C это значение называется `DBL_EPSILON`. (Для интересующихся — в случае плавающей точки с двойной точностью значение `flo:ulp-of-one` равно 2.220446049250313e-16.)

Заставьте объявления `optional` работать наряду с другими! Убедитесь, что ваше расширение свободно сочетается со всеми другими имеющими смысл объявлениями.

5.3 Компиляция в исполнительные процедуры

Вычислитель, с которым мы до сих пор работали, весьма гибок и легко расширяем, но слишком прост: программы в нем работают довольно медленно. Одна из причин в том, что вычислитель постоянно просматривает и разбирает синтаксис программы (как бы прост он ни был). В главе 4 мы избежали этой проблемы, переводя каждый образец сопоставителя в комбинацию сопоставляющих процедур с одинаковым интерфейсом — т. е. в комбинаторный язык, — в разделе 4.3. При интерпретации языка мы подобным же образом можем избежать повторного рассмотрения синтаксической структуры, скомпилировав ее в композицию исполнительных процедур. Поэтому давайте, прежде чем дальше углубляться в процесс исполнения, проведем это преобразование.

Главная идея состоит в том, чтобы разделить задачу вычисления выражения относительно окружения на две фазы. На первой фазе выражение анализируется и преобразуется в исполнительную процедуру. На второй фазе эта процедура применяется к окружению и получает ожидаемый результат вычисления. Мы реализуем эту идею напрямую как композицию двух фаз:

```
(define (x:eval expression environment)
  ((analyze expression) environment))
```

Анализ и преобразование выражения называется *компиляция*, и говорится, что работа, происходящая при этом, — задачи *времени компиляции*. Компилятор выделяет ту часть поведения программы, которая не зависит от значений свободных переменных выражения. В основном это синтаксический анализ, но, помимо него,

можно провести некоторые оптимизации, выразимые через синтаксические правила. Получающаяся исполнительная процедура зависит от отображения символов на значения, содержащегося в окружении; говорится, что она проделывает работу *времени выполнения*.

Анализ выражений

Поскольку мы хотим расширять синтаксис по необходимости, мы реализуем анализ в виде полиморфной процедуры, где вариант по умолчанию применяет оператор к операндам. Для Lisp/Scheme это обязано быть умолчательным значением, поскольку никакое синтаксическое ключевое слово не выделяет применение[19]:

```
(define x:analyze
  (simple-generic-procedure 'x:analyze 1 default-analyze))
```

Соглашение для процедуры `x:analyze` состоит в том, что она берет в качестве аргумента выражение и возвращает исполнительную процедуру, принимающую единственным аргументом окружение.

Процедура `analyze` выражает общую структуру использования:

```
(define (analyze expression)
  (make-executor (x:analyze expression)))
```

Исполнительная процедура оборачивается в процедуру `make-executor` ради удобства отладки. Получившийся *исполнитель* тоже является процедурой с теми же аргументами и возвращаемым значением, что и обернутая им процедура. Его смысл состоит, в частности, в том, чтобы поддерживать «след исполнения», что может оказаться полезным, если надо понять, как программа добралась до ошибочного состояния.

Как мы уже сказали, вариант анализа по умолчанию — применение.

```
(define (default-analyze expression)
  (cond ((application? expression)
         (analyze-application expression))
        (else (error "Неизвестный тип выражения" expression))))

(define (analyze-application expression)
  (let ((operator-exec (analyze (operator expression)))
        (operand-execs (map analyze (operands expression))))
    (lambda (environment)
      (x:apply (x:advance (operator-exec environment))
               operand-execs
               environment))))
```

Обратите внимание, как здесь разделяются задачи: оператор и операнды извлекаются из выражения и анализируются, превращаясь в исполнительные процедуры `operator-exec` и `operand-execs`. При этом может потребоваться значительный

[19] Этот вычислитель существенно отличается от предыдущих, поэтому, как упомянуто в примечании 3 на стр. 211, мы выбираем для него новый префикс (`x:` от англ. *eXecution procedure* — «исполнительная процедура») и отмечаем им процедуры, аналогичные процедурам других интерпретаторов.

анализ. В результате исполнительная процедура для применения — это процедура, принимающая окружение и производящая это применение. Процедура `x:apply` (стр. 235) аналогична `g:apply` в интерпретаторе, но `x:apply` принимает исполнительные процедуры операндов, а не выражения-операнды, как было в `g:apply`. Процедура `x:advance` аналогична `g:advance` и вводится по тем же причинам. Таким образом можно преобразовать все компоненты вычислителя.

Преобразование самовычисляющихся выражений (таких, как числа, булевы значения и строки) тривиально. Здесь единственная сложная часть работы — синтаксис выражений, но о нем заботится парсер Scheme. Текст преобразуется в элементы программы и s-выражения еще до того, как он попадет в наш вычислитель, так что здесь нам этими сложностями заниматься не нужно.

```
(define (analyze-self-evaluating expression)
  (lambda (environment) expression))

(define-generic-procedure-handler x:analyze
  (match-args self-evaluating?)
  analyze-self-evaluating)
```

Работа с кавычкой также проста, потому что о сложной части заботится парсер[20]:

```
(define (analyze-quoted expression)
  (let ((qval (text-of-quotation expression)))
    (lambda (environment) qval)))

(define-generic-procedure-handler x:analyze
  (match-args quoted?)
  analyze-quoted)
```

С переменными все тоже несложно. После того как мы определили переменную, вся работа содержится в исполнительной процедуре.

```
(define (analyze-variable expression)
  (lambda (environment)
    (lookup-variable-value expression environment)))

(define-generic-procedure-handler x:analyze
  (match-args variable?)
  analyze-variable)
```

Определения процедур, выражаемые в Lisp/Scheme через `lambda`, служат примером разделения работы. Прежде чем построить исполнительную процедуру, анализатор (компилятор) разбирает лямбда-выражение, выделяет описания формальных параметров и компилирует тело выражения. Поэтому исполнительная процедура для выражения `lambda` и код, исполняющий в конце концов тело, всем этим заниматься уже не должны.

[20] В языке Lisp (и в том числе в Scheme) получение текста под кавычкой элементарно — всего лишь `cadr`, — но мы хотим, чтобы интерпретатор обладал достаточной общностью для любого вида синтаксиса. В большинстве языков извлечение текста закавыченного выражения намного сложнее.

```
(define (analyze-lambda expression)
  (let ((vars (lambda-parameters expression))
        (body-exec (analyze (lambda-body expression))))
    (lambda (environment)
      (make-compound-procedure vars body-exec environment))))

(define-generic-procedure-handler x:analyze
  (match-args lambda?)
  analyze-lambda)
```

Особая форма `if` — еще один ясный пример преимуществ разделения между анализом и исполнением программы. Все три части выражения `if` анализируются во время компиляции, так что на долю исполнительной процедуры остается только извлечь булево значение предиката и решить, нужно ли вызвать следствие или альтернативу. Анализ подвыражений во время исполнения (когда используется исполнительная процедура) проводить не нужно.

```
(define (analyze-if expression)
  (let ((predicate-exec
         (analyze (if-predicate expression)))
        (consequent-exec
         (analyze (if-consequent expression)))
        (alternative-exec
         (analyze (if-alternative expression))))
    (lambda (environment)
      (if (x:advance (predicate-exec environment))
          (consequent-exec environment)
          (alternative-exec environment)))))

(define-generic-procedure-handler x:analyze
  (match-args if?)
  analyze-if)
```

Последовательности выражений служат особенно хорошим примером разделения анализа и исполнения. Не имеет смысла перекомпилировать последовательность каждый раз, как мы входим в тело процедуры; эту работу можно раз и навсегда сделать во время компиляции.

Процедура `analyze-begin` сначала анализирует все подвыражения выражения `begin` и получает список исполнительных процедур (сохраняя порядок подвыражений `begin`). Затем эти исполнительные процедуры склеиваются вместе через `reduce-right` и комбинатор для пары процедур, который берет их и порождает исполнительную процедуру, выполняющую два выражения одно за другим[21].

```
(define (analyze-begin expression)
  (reduce-right (lambda (exec1 exec2)
                  (lambda (environment)
                    (exec1 environment)
```

[21] В теле `analyze-begin` процедура `reduce-right` никогда не использует аргумент `#f`, поскольку обращение к `#f` происходит, только если список выражений пуст. Однако в таком случае сообщение об ошибке `Пустая последовательность` будет выдано еще до начала свертки.

```
                    (exec2 environment)))
                #f
                (map analyze
                     (let ((exps
                            (begin-actions expression)))
                       (if (null? exps)
                           (error "Пустая последовательность"))
                           exps))))

(define-generic-procedure-handler x:analyze
  (match-args begin?)
  analyze-begin)
```

В отсутствие оптимизаций присваивания сложности не представляют.

```
(define (analyze-assignment expression)
  (let ((var
         (assignment-variable expression))
        (value-exec
         (analyze (assignment-value expression))))
    (lambda (environment)
      (set-variable-value! var
                           (value-exec environment)
                           environment)
      'ok)))

(define-generic-procedure-handler x:analyze
  (match-args assignment?)
  analyze-assignment)
```

Однако, если оптимизации есть, присваивания превращаются в серьезную проблему. Действительно, присваивания вводят в программу понятие времени: что-то в программе происходит до присваивания, а что-то после него; присваивание может влиять на события, обращающиеся к присваиваемой переменной. Поэтому, скажем, общие подвыражения необязательно имеют одинаковые значения, если они обращаются к переменной, которой может быть присвоено новое значение!

Определения проблемы не представляют, если только мы не думаем о них (и некорректно их используем) как о присваиваниях; тогда они могут взаимодействовать с оптимизациями компилятора.

```
(define (analyze-definition expression)
  (let ((var
         (definition-variable expression))
        (value-exec
         (analyze (definition-value expression))))
    (lambda (environment)
      (define-variable! var
                        (value-exec environment)
                        environment)
      var)))
```

```
(define-generic-procedure-handler x:analyze
  (match-args definition?)
  analyze-definition)
```

Особые формы, реализуемые через преобразование выражений, такие как `cond` и `let`, в этой системе обрабатываются совсем просто: мы всего лишь компилируем преобразованное выражение. Это то место, где можно вставить очень общую подсистему макросов.

```
(define-generic-procedure-handler x:analyze
  (match-args cond?)
  (compose analyze cond->if))

(define-generic-procedure-handler x:analyze
  (match-args let?)
  (compose analyze let->combination))
```

Применение процедур

Исполнительная процедура комбинации вызывает исполнительную процедуру оператора и получает составную процедуру, подлежащую применению (см. `analyze-application` на стр. 231). Операнды также уже преобразованы в исполнительные процедуры.

Процедура `x:apply` аналогична процедуре `g:apply` из простейшего вычислителя (на стр. 218):

```
(define x:apply
  (simple-generic-procedure 'x:apply 3 default-apply))

(define (default-apply procedure operand-execs environment)
  (error "Неизвестный тип процедуры" procedure))
```

Заметим,что процедура `default-apply` здесь такая же, как и в `g:apply`, с точностью до имен двух неиспользуемых параметров.

Как и раньше, нам требуются обработчики для различных видов процедур с определенными видами параметров. Обработчик применения для строгих элементарных процедур должен вынудить все аргументы, а затем запустить элементарную процедуру.

```
(define-generic-procedure-handler x:apply
  (match-args strict-primitive-procedure?
              executors?
              environment?)
  (lambda (procedure operand-execs environment)
    (apply-primitive-procedure procedure
     (map (lambda (operand-exec)
            (x:advance (operand-exec environment)))
          operand-execs))))
```

Обработчик применения процедур общего вида лишь немного отличается от своего аналога из простейшего вычислителя, показанного ранее. Разница состоит в том, что теперь мы работаем не с выражениями-операндами, а с исполнительными процедурами.

```
(define-generic-procedure-handler x:apply
  (match-args compound-procedure? executors? environment?)
  (lambda (procedure operand-execs calling-environment)
    (if (not (n:= (length (procedure-parameters procedure))
                  (length operand-execs)))
        (error "Неправильное число операндов"))
    (let ((params (procedure-parameters procedure))
          (body-exec (procedure-body procedure)))
      (let ((names (map procedure-parameter-name params))
            (arguments
             (map (lambda (param operand-exec)
                    (x:handle-operand param
                                      operand-exec
                                      calling-environment))
                  params
                  operand-execs)))
        (body-exec (extend-environment names arguments
                     (procedure-environment procedure)))))))
```

Обработчик применения для составных процедур должен уметь работать с различными видами формальных параметров, которые могут присутствовать в составной процедуре. Это, как и обычно у нас, достигается тем, что процедура `x:handle-operand` полиморфная. Умолчательный вариант, когда операнд должен быть вычислен до входа в тело составной процедуры, состоит в том, чтобы немедленно запустить исполнительную процедуру операнда и получить значение. Однако для ленивых параметров и мемоизированных ленивых параметров мы должны иметь возможность задержать вычисление нужным образом.

```
(define x:handle-operand
  (simple-generic-procedure 'x:handle-operand 3
    (lambda (parameter operand-exec environment)
      (operand-exec environment))))

(define-generic-procedure-handler x:handle-operand
  (match-args lazy? executor? environment?)
  (lambda (parameter operand-exec environment)
    (postpone operand-exec environment)))

(define-generic-procedure-handler x:handle-operand
  (match-args lazy-memo? executor? environment?)
  (lambda (parameter operand-exec environment)
    (postpone-memo operand-exec environment)))
```

Задержка исполнительной процедуры для операнда работает так же, как задержка выражения операнда. Однако обработчики в полиморфной процедуре `x:advance`,

работающие с отложенными исполнительными процедурами операндов, отличаются от обработчиков в `g:advance`: отложенная исполнительная процедура должна вызываться в отложенном окружении, а не быть вычисленной по отношению к этому окружению (сравните с `g:advance` на стр. 211).

```
(define-generic-procedure-handler x:advance
  (match-args postponed?)
    (lambda (object)
      (x:advance ((postponed-expression object)
                  (postponed-environment object)))))

(define-generic-procedure-handler x:advance
  (match-args postponed-memo?)
  (lambda (object)
    (let ((value
           (x:advance ((postponed-expression object)
          (postponed-environment object)))))
      (advance-memo! object value)
      value)))
```

Обработка операндов в нашей версии `x:apply` устроена довольно незамысловато. Она в действительности проделывает довольно много «разбора» списка формальных параметров в исполнительной процедуре; таким образом, получается, что составные процедуры мы компилируем не полностью. Одним из шагов, улучшающим компиляцию, было бы выделить отдельный обработчик для строгих составных процедур, как мы делали раньше. Исправление этого недостатка будет вам поручено в упражнении 5.16.

Упражнение 5.12. Реализация n-местных процедур

В упражнениях 5.2 и 5.11 мы замечали, что часто бывает полезно иметь процедуры, принимающие неизвестное заранее количество аргументов. Примерами служат процедуры сложения и умножения в Scheme.

Чтобы определить такую процедуру в Scheme, мы в качестве формальных параметров лямбда-выражения указываем одиночный символ, а не список. Этот символ связывается со списком переданных в процедуру аргументов. Например, чтобы получить процедуру, принимающую несколько аргументов и возвращающую список их квадратов, мы пишем:

```
(lambda x (map square x))
```

либо:

```
(define (ss . x) (map square x))
```

и можем потом сказать

```
(ss 1 2 3 4) ==> (1 4 9 16)
```

Измените анализирующий интерпретатор, чтобы он поддерживал эту конструкцию.

Подсказка: Код, обрабатывающий `define` и `lambda` в определениях синтаксиса, трогать не надо! Все изменения содержатся в анализаторе.

Покажите, что ваше изменение позволяет писать процедуры нового вида и не вызывает других проблем.

Упражнение 5.13. Упрощаем отладку

Одна из проблем в нашем компиляторе состоит в том, что все исполнительные процедуры оказываются анонимными лямбда-выражениями. Поэтому при трассировке почти нечего

распечатать. Однако с этим легко справиться. Если переписать процедуру, создающую исполнительные процедуры для применений:

```
(define (analyze-application exp)
  (let ((operator-exec (analyze (operator exp)))
        (operand-execs (map analyze (operands exp))))
    (lambda (env)
      (x:apply (x:advance (operator-exec env))
               operand-execs
               env))))
```

таким образом:

```
(define (analyze-application exp)
  (let ((operator-exec (analyze (operator exp)))
        (operand-execs (map analyze (operands exp))))
    (define (execute-application env)
      (x:apply (x:advance (operator-exec env))
               operand-execs
               env))
    execute-application))
```

тогда (в реализации MIT/GNU Scheme) исполнительная процедура получит имя, и по этому имени будет видно, что это за процедура. Примените эту идею ко всем исполнительным процедурам.

Придумайте и, возможно, реализуйте другие способы улучшить отладку получаемого кода без потери производительности выполнения. Например, можно использовать выражение `exp` как «ярлык» при исполнительной процедуре.

Упражнение 5.14. Свертка констант

Допустим, у нас есть способ объявить анализатору, что некоторые символы имеют точное значение (скажем, объявить, что стандартные знаки арифметических операций (`+`, `-`, `*`, `/`, `sqrt`) ссылаются на константные процедуры). Тогда любую комбинацию этих операторов с константами, например `(/ (+ 1 (sqrt 5)) 2)`, можно будет вычислить во время компиляции и использовать результат, вместо того чтобы проделывать вычисление во время выполнения. Эта оптимизация при компиляции называется *свертка констант*.

Реализуйте в нашем анализаторе свертку констант. Для того чтобы она работала, анализатор должен знать, про какие символы в тексте программы он может с уверенностью сказать, что они привязаны к известным значениям. Например, он должен знать, связан ли символ `car` с селектором первого элемента пары. Можете предполагать, что анализатор может вызвать процедуру, сообщающую связывания известных символов. Процедура должна принимать символ и либо возвращать надежное значение, на которое анализатор может рассчитывать, либо `#f`, если символ не находится под контролем анализатора.

Упражнение 5.15. Другие оптимизации

Существует множество простых преобразований, способных ускорить исполнение программы. Например, с помощью нашей подсистемы сопоставления с образцом можно организовать систему переписывания термов, выполняющую локальные (peephole) оптимизации и выносящую инвариантный код за пределы циклов. Возможно, имеет смысл реализовать сокращение общих подвыражений; будьте, однако, осторожны с эффектами от присваиваний. Добавьте в анализ фазу оптимизации, реализуйте некоторые классические компиляторные оптимизации и покажите, насколько они помогают.

Упражнение 5.16. Компиляция объявлений формальных параметров
Несмотря на то что преобразование программы в композицию исполнительных процедур достаточно эффективно и порождает намного более быстрый код, чем простая интерпретация, представленная нами версия не очень хитроумна: исполнительная процедура для составных процедур в `x:apply` разбирает объявления в списке формальных параметров, определяя, что делать с операндами. Это следовало бы делать на стадии компиляции, а не на стадии выполнения: при анализе выражения `lambda`, создающего составную процедуру, должна получаться исполнительная процедура, которая уже знает, что делать с операндами и вызывающим окружением.

Разберитесь с этой задачей и решите ее. Убедитесь, что ссылки на окружение вызова сохраняются только там, где это строго необходимо.

Примечание: это большой проект.

5.4 Исследовательское поведение

При определении значения сегментных переменных в сопоставлении с образцом мы уже сталкивались с явным поиском методом перебора с возвратами. Но даже и в отсутствие сегментных переменных реализация системы переписывания термов опиралась на поиск с возвратами. Когда выражение-следствие правила решало, что сопоставление недостаточно избирательно, чтобы следствие заменяло выбранный фрагмент данных, даже несмотря на то, что образец в правиле сопоставляется с этим фрагментом, выражение-следствие возвращало `#f`, цикл сваливался обратно в режим сопоставления, и, возможно, применялось другое правило.

Механизм префиксного графа для оптимизации доступа к полиморфной процедуре также требует перебора с возвратами. Префиксный граф ищет последовательность предикатов, которой удовлетворяет последовательность аргументов. Однако начальный сегмент предикатов может соответствовать начальному сегменту аргументов несколькими способами, и поэтому в механизм префиксного графа неявным образом встроен поиск.

Обычно мы считаем перебор с возвратами и его активное использование при поиске методикой из области искусственного интеллекта. Однако можно рассматривать перебор с возвратами как способ построения модульных и способных к независимому развитию систем, чье поведение похоже на исследовательское поведение биологических систем. Рассмотрим простой, но практический пример: решение квадратного уравнения. Уравнение имеет два корня. Можно вернуть оба и считать, что пользователь решения знает, что делать с обоими. Можно вернуть один из них и надеяться, что все будет хорошо. (Каноническая процедура извлечения квадратного корня `sqrt` всегда возвращает положительный корень, несмотря на то, что их два!) Недостаток идеи вернуть сразу оба корня — в том, что получатель результата должен знать, что надо попробовать провести дальнейшее вычисление с обоими и либо отвергнуть один из них, либо вернуть результаты обоих, что в свою очередь ведет к дальнейшим альтернативам. Недостаток идеи вернуть один — в том, что это может для целей получателя быть неправильный корень. В имитационном моделировании физических систем этот вопрос может представлять реальную проблему.

Лингвистически неявный поиск

Процедуры поиска, которые мы явно построили, вполне хороши, однако, может быть, можно добиться большего, если встроить механизм перебора с возвратами

прямо в языковую инфраструктуру. Процедура квадратного корня должна вернуть один из корней, но быть готовой изменить решение и вернуть другой, если получатель решит, что первый вариант ему не подходит. Решение, являются ли данные для вычисления подходящими и приемлемыми, будет приниматься получателем, и так оно и должно быть. Это решение само по себе может потребовать дальнейших вычислений и требовать выбора, чьи последствия не сразу понятны без еще дополнительных вычислений; таким образом, процесс рекурсивен. Разумеется, это потенциально может привести нас к смертельному экспоненциальному поиску по всем возможным присваиваниям во всех местах, где нам в программе пришлось делать выбор.

Важно рассмотреть, до какой степени стратегия поиска отделяема от других частей программы, чтобы можно было менять эту стратегию без существенной модификации остальной программы. Здесь мы делаем дальнейший шаг, проталкивая поиск и управление поиском в инфраструктуру, поддерживаемую языком, совсем не встраивая поиск в нашу программу явным образом. Если поиск делается неявным, он может начать использоваться чрезмерно. Как обычно, модульная гибкость несет с собой опасности.

У этой мысли довольно давняя история. В 1961 г. Джон Маккарти [90] придумал недетерминистскую операцию `amb`, полезную при представлении недетерминистских автоматов. В 1967 г. Боб Флойд [35] придумал встроить перебор с возвратами в компьютерный язык как часть языковой инфраструктуры. В 1969 г. Карл Хьюитт [56] предложил язык PLANNER, где эти идеи были воплощены. В начале 1970-х гг. Кольмероэ, Ковальски, Руссель и Уоррен разработали Prolog, язык, основанный на ограниченной форме исчисления предикатов первого порядка, где поиск методом перебора с возвратами был встроен неявно[22].

5.4.1 `amb`

Операция `amb` Джона Маккарти принимает произвольное количество аргументов. Значением ее является значение одного из этих аргументов, но заранее неизвестно, какого. Например,

```
(amb 1 2 3)
```

выдает значение 1 либо 2, либо 3, в зависимости от будущего вычисления. Выражение `(amb)` без аргументов не может выдать ни одно значение; это вычислительная неудача, с ее помощью можно отвергнуть ранее принятые решения.

Выражение, где встречается `amb`, может иметь несколько значений. Чтобы их все распечатать, мы печатаем одно, а затем вызываем неудачу и таким образом вынуждаем программу породить следующее значение, пока они все не закончатся.

```
(begin
  (newline)
  (write-line (list (amb 1 2 3) (amb 'a 'b)))
  (amb))
;;; Начинается работа над новой задачей
(1 a)
```

[22] В обзоре Эрика Сандеволла [107], посвященном системам, обладающим инструментами «немонотонного вывода», приводится больше контекста, чем мы можем себе позволить здесь.

```
(2 a)
(3 a)
(1 b)
(2 b)
(3 b)
;;; Больше значений нет
```

При помощи `amb` довольно легко можно породить пифагоровы тройки. Через `amb` мы порождаем тройки целых чисел и отвергаем те, которые не являются пифагоровыми.

Для удобства программирования с `amb` мы вводим вспомогательную процедуру. `Require` используется как фильтр, который порождает неудачу и откат, если предикатное выражение-аргумент окажется ложным.

```
(define (require p)
  (if (not p) (amb) 'ok))
```

Для получения числа в интервале пишем

```
(define (an-integer-between low high)
  (require (<= low high))
  (amb low (an-integer-between (+ low 1) high)))
```

При наличии этих вспомогательных процедур поиск пифагоровых троек записывается очень естественно:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j))
                    (* k k)))
        (list i j k)))))

(begin
  (newline)
  (write-line (a-pythagorean-triple-between 1 20))
  (amb))
;;; Начинается работа над новой задачей
(3 4 5)
(5 12 13)
(6 8 10)
(8 15 17)
(9 12 15)
(12 16 20)
;;; Больше значений нет
```

Кажется, этот инструмент будет полезным во многих задачах. Давайте рассмотрим, как можно сделать его частью нашего языка.

5.4.2 Реализация amb

Хорошо, что мы отделили анализ выражений языка от выполнения, поскольку теперь мы можем менять процедуры выполнения, не трогая ничего в синтаксическом анализе. Поэтому встраивание недетерминистского поиска в язык сводится к изменению только исполнительных процедур. Здесь основной шаг состоит в том, что исполнительные процедуры преобразуются в стиль с передачей продолжений, когда кроме аргумента-окружения каждая исполнительная процедура принимает два аргумента-продолжения. Одно, обычно называемое `succeed`, вызывается, если вычисление окажется успешным, а другое, по имени `fail`, зовется, когда вычисление терпит неудачу.

Исполнительная процедура возвращает предлагаемое значение, вызывая продолжение успеха и передавая ему значение и продолжение неудачи. Продолжение неудачи — «отдел жалоб»: если некоторому вычислению в будущем не понравится предоставленное ему значение, оно может позвать продолжение неудачи без аргументов, ожидая получить другой результат. (В разделе 4.3 мы использовали возвращаемое значение `#f` в качестве сигнала о неудаче. В разделе 4.2.2 для этой цели служили продолжения успеха и неудачи. Продолжения успеха и неудачи — более гибкий инструмент; его также можно расширить и передавать информацию, почему значение было отвергнуто.)

Итак, общая форма исполнительной процедуры у нас будет такая:

```
(lambda (environment succeed fail)
  ;; succeed = (lambda (value fail)
  ;; Попробуй значение value.
  ;; Если оно не нравится, позови (fail).
  ;; ...)
  ;; fail = (lambda () ...)
  ...
  ;; Попробуй породить результат, Если не получится, зови (fail).
  ...)
```

Преобразование в стиль с передачей продолжения не очень приятное, поскольку при этом код заметно раздувается, но в основном оно чисто механическое. Например, процедура `analyze-application` со стр. 231 выглядит как:

```
(define (analyze-application expression)
  (let ((operator-exec (analyze (operator expression)))
        (operand-execs (map analyze (operands expression))))
    (lambda (environment)
      (x:apply (x:advance (operator-exec environment))
               operand-execs
               environment))))
```

При переводе в стиль с передачей продолжений получаем[23]:

```
(define (analyze-application exp)
  (let ((operator-exec (analyze (operator exp)))
        (operand-execs (map analyze (operands exp))))
```

[23] В этом вычислителе для обозначения аналогичных процедур мы используем префикс `a:` (по имени `amb`) (см. примечание 3 на стр. 211).

```
    (lambda (env succeed fail)
      (operator-exec env
                     (lambda (operator-value fail-1)
                       (a:advance operator-value
                                  (lambda (procedure fail-2)
                                    (a:apply procedure
                                             operand-execs
                                             env
                                             succeed
                                             fail-2))
                                  fail-1))
                     fail))))
```

Эта исполнительная процедура сложнее той, из которой она получена. В теле исходной исполнительной процедуры выражение `(operator-exec environment)` возвращает значение в выражение `(x:advance ...)`, которое в свою очередь возвращает значение в выражение `(x:apply ... ... ...)`. В новой исполнительной процедуре больше нет этих вложенных выражений. Каждая процедура «возвращается» путем вызова процедуры, принимающей результаты вычисления.

Поскольку часто нам будет нужно вынудить значение, полезно создать абстракцию, выражающую это вынуждение. Процедура `execute-strict` прячет неинтересные подробности, связанные с процессом, когда вынуждается вычисление отложенного выражения, чтобы получить его незадержанное значение. В приведенной выше процедуре `analyze-application` с ее помощью вынуждается значение оператора, чтобы получить применимую процедуру.

```
(define (execute-strict executor env succeed fail)
  (executor env
            (lambda (value fail-1)
              (a:advance value succeed fail-1))
            fail))
```

Процедура `execute-strict` вызывает данную ей исполнительную процедуру `executor`. Ее результат передается в `a:advance` для вынуждения. Затем вынужденное значение передается дальше в продолжение успеха `succeed` процедуры `execute-strict`, и таким образом, по сути, результат вынуждения возвращается тому, кто вызвал `execute-strict`.

С помощью `execute-strict` можно переписать `analyze-application`:

```
(define (analyze-application exp)
  (let ((operator-exec (analyze (operator exp)))
        (operand-execs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (execute-strict operator-exec
                      env
                      (lambda (procedure fail-2)
                        (a:apply procedure
                                 operand-execs
                                 env
                                 succeed
```

```
                              fail-2))
                    fail))))
```

Таким же образом требуется переписать все исполнительные процедуры. В `analyze-if` мы используем `execute-strict`, чтобы вынудить значение предиката условного выражения, без которого не знаем, что делать дальше:

```
(define (analyze-if exp)
  (let ((predicate-exec (analyze (if-predicate exp)))
        (consequent-exec (analyze (if-consequent exp)))
        (alternative-exec (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (execute-strict predicate-exec
                      env
                      (lambda (pred-value pred-fail)
                        ((if pred-value
                             consequent-exec
                             alternative-exec)
                         env succeed pred-fail))
                      fail))))
```

Большинство преобразований не представляет сложности, и в этом тексте мы не будем нагружать вас деталями. Есть, однако, интересный случай: присваивания. В системах с перебором часто требуется два вида присваиваний. Обычное постоянное присваивание `set!` служит для накопления информации в процессе поиска, например сколько раз исследуется определенная ветвь. Но есть и временное присваивание `maybe-set!`, которое должно отменяться, если откатывается ветвь, на которой оно расположено. Обычное постоянное присваивание реализуется как:

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (value-exec (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (value-exec env
                  (lambda (new-val val-fail)
                    (set-variable-value! var new-val env)
                    (succeed 'ok val-fail))
                  fail))))
```

Временное присваивание устроено сложнее. Продолжение неудачи, передаваемое в ветвь удачного присваивания, восстанавливает предыдущее значение присваиваемой переменной:

```
(define (analyze-undoable-assignment exp)
  (let ((var (assignment-variable exp))
        (value-exec (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (value-exec env
                  (lambda (new-val val-fail)
                    (let ((old-val
                           (lookup-variable-value var env)))
```

```
                    (set-variable-value! var new-val env)
                    (succeed 'ok
                             (lambda ()
                               (set-variable-value! var
                                                    old-val
                                                    env)
                               (val-fail)))))
                fail))))
```

Единственный оставшийся интересный случай — реализация самого оператора `amb`. Именно здесь, если предыдущее выданное значение отвергнуто, нам нужно выбрать следующую альтернативу. Это делается в продолжении неудачи текущей альтернативы:

```
(define (analyze-amb exp)
  (let ((alternative-execs
         (map analyze (amb-alternatives exp))))
    (lambda (env succeed fail)
      (let loop ((alts alternative-execs))
        (if (pair? alts)
            ((car alts) env
                        succeed
                        (lambda ()
                          (loop (cdr alts))))
            (fail))))))
```

Если альтернатив не осталось, исполнительная процедура для `amb` зовет продолжение неудачи, с которым была вызвана. Таким образом, программа производит перебор дерева альтернатив в глубину. Альтернативы просматриваются через проход по списку процедурой `cdr`; таким образом, поиск проходит слева направо[24].

С точностью до поправок в управляющем цикле (стр. 220), чтобы заставить его работать со структурой продолжений, наш интерпретатор закончен!

Упражнение 5.17. Головоломка

Формализуйте и решите с помощью `amb` следующую головоломку.

> Две женщины (Лиза и Ева) и четверо мужчин (Бен, Хьюго, Пабло и Дайко) сидят за круглым столом и играют в карты. Ни у каких двух игроков карты не равны по силе.
>
> - Бен сидит напротив Евы.
> - Карты мужчины справа от Лизы сильнее, чем у Дайко.
> - Карты мужчины справа от Евы сильнее, чем у Бена.
> - Карты мужчины справа от Бена сильнее, чем у Пабло.
> - Карты мужчины справа от Бена сильнее, чем у Евы.

[24] Наша реализация `amb` не совсем такая, как имел в виду Маккарти. Его `amb` умеет «предвидеть» и сходится к значению, даже если расходится одна из альтернатив. Поскольку наш `amb` проводит поиск в глубину слева направо, если выражение e расходится (зацикливается или заканчивается с ошибкой), расходится и `(amb` e `5)`. У Маккарти такой `amb` вернул бы 5. Это изящно разъясняет Уильям Клингер [21].

- Карты женщины справа от Дайко сильнее, чем у Пабло.
- Карты женщины справа от Пабло сильнее, чем у Хьюго.

Каково расположение игроков за столом? Единственно ли оно возможное с точностью до поворотов?

Выбирайте альтернативы при каждом возможном выборе с помощью `amb`. Посчитайте также, сколько будет решений, если вместо «карты мужчины справа от Бена сильнее, чем у Пабло» нам говорится только «мужчина справа от Бена не Пабло». Объясните этот результат.

Примечание: напрямую закодированное решение работает медленно; на нашем ноутбуке оно считается несколько часов (в 2017 г.). Однако, если переформулировать задачу более умно, программа завершается примерно за 2 мин[25].

Упражнение 5.18. Обнаружение неудач

Реализуйте новую языковую конструкцию `if-fail`, которая позволяет программе обнаружить неудачное исполнение выражения. Выражение `if-fail` принимает два подвыражения. Первое вычисляется как обычно, и в случае успеха возвращается его результат. Однако, если вычисление оказывается неудачным, возвращается значение второго выражения, как в примере:

```
(if-fail (let ((x (amb 1 3 5)))
           (require (even? x))
           x)
         'all-odd)
all-odd

(if-fail (let ((x (amb 1 3 4 5)))
           (require (even? x))
           x)
         'all-odd)
4
```

Подсказка: решение тривиально!

Упражнение 5.19. Присваивание

Каковы будут результаты следующих вычислений, если `if-fail` определено как в упражнении 5.18?

```
(let ((pairs '()))
  (if-fail (let ((p (prime-sum-pair '(1 3 5 8) '(20 35 110))))
             (set! pairs (cons p pairs))
             (amb))
           pairs))

(let ((pairs '()))
  (if-fail (let ((p (prime-sum-pair '(1 3 5 8) '(20 35 110))))
             (maybe-set! pairs (cons p pairs))
             (amb))
           pairs))
```

Можно использовать определения:

[25] Измерения проводились на встроенном интерпретаторе с исследовательским поведением, который сам интерпретируется нижележащей реализацией Scheme. Если скомпилировать встроенный интерпретатор компилятором, скорость увеличивается примерно в 30 раз.

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))

(define (an-element-of lst)
  (if (null? lst)
      (amb)
      (amb (car lst)
           (an-element-of (cdr lst)))))

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

(define (smallest-divisor n)
  (define (find-divisor test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divides? test-divisor n) test-divisor)
          (else (find-divisor (+ test-divisor 1)))))
  (define (divides? a b)
    (= (remainder b a) 0))
  (find-divisor 2))
```

Упражнение 5.20. Порядок вариантов

Механизм `amb`, как мы его написали, всегда перебирает варианты в том порядке, как они указаны в выражении `amb`. Однако иногда контекстуальная информация позволяет нам улучшить этот порядок. Например, в настольной игре выбор хода из списка разрешенных ходов должен зависеть от позиции на доске. Давайте изобретем разновидность `amb`, которая даст нам достаточную для этого гибкость. Пусть каждое выражение-вариант будет снабжено числовым выражением-весом:

```
(choose (<вариант-1> <вес-1>) ... (<вариант-n> <вес-n>))
```

Можно вычислить все веса, и с их помощью выбрать следующее выражение-вариант для вычисления и возврата значения. Разумеется, после того как выбор сделан, он исчерпан, и, если неуспех снова приведет к тому же экземпляру выбора, оставшиеся выражения-веса нужно перевычислить, прежде чем выбрать новый вариант.

a. Реализуйте `choose` так, чтобы выбирался вариант с наибольшим весом.

b. В реальных ситуациях обычно веса — недостаточно сильный механизм, чтобы остановиться ровно на одном варианте. Иногда хорошей стратегией будет выбирать случайно с вероятностями, пропорциональными вычисленным весам. Реализуйте новую конструкцию выбора `pchoose`, которая работает так же, как `choose`, но выбирает вариант таким образом.

5.5 Явная работа с нижележащими продолжениями

Теперь мы готовы столкнуться с настоящим волшебством!

Большинство языков, включая Scheme, строится вокруг понятия выражения. У выражения есть значение, которое оно «возвращает». Выражение составлено из подвыражений, и у каждого из них есть значение, возвращаемое в большее выражение, частью которого оно является. В чем состоит основная идея выражения?

Рассмотрим составное выражение

```
(+ 1 (* 2 3) (/ 8 2))
```

Разумеется, его значение равно 11. При его вычислении сначала определяются значения оператора и операндов, а затем значение оператора (процедура) применяется к значениям операндов (аргументов).

Этот процесс можно прояснить, если переформулировать вычисление в *стиле с передачей продолжений*. Мы изобретаем новые знаки операций `**`, `//` и `++` как имена для процедур умножения, деления и сложения в стиле с передачей продолжений:

```
(define (** m1 m2 continue)
  (continue (* m1 m2)))

(define (// n d continue)
  (continue (/ n d)))

(define (++ a1 a2 continue)
  (continue (+ a1 a2)))
```

Эти процедуры отличаются от обычных `*`, `/` и `+` тем, что они не возвращают управление в вызвавшую их точку кода; вместо этого они вызывают свой последний аргумент и передают ему вычисленное значение. Аргумент, который получает это значение, называется процедурой-*продолжением*. Мы пользовались стилем с передачей продолжений в разделах 4.2.2 и 5.4.2, а также в унификаторе на стр. 171.

Вычисление `(+1 (* 2 3) (/ 8 2))` в стиле с передачей продолжений выглядит так:

```
(** 2 3
    (lambda (the-product)            ; A
      (// 8 2
          (lambda (the-quotient)     ; B
            (++ 1 the-product the-quotient
                k)))))
```

где `k` — окончательная процедура-продолжение, принимающая один аргумент, а именно значение всего вычисления 11[26].

В этом примере процедура `**` вычисляет произведение 2 и 3 и вызывает свое продолжение (выражение `lambda`, отмеченное комментарием `A`) со значением 6. Таким образом, в теле `A` переменная `the-product` равна 6. В теле `A` процедура `//` вычисляет результат деления 8 на 2 и передает получившееся значение 4 в процедуру с меткой `B`, где переменная `the-quotient` равна 4. В теле `B` процедура `++` считает сумму 1, 6 и 4 и передает получившееся значение 11 в процедуру-продолжение `k`.

В стиле с передачей продолжений нет никаких вложенных выражений, возвращающих значения. Все результаты передаются в процедуры-продолжения. Отсюда нет никакой нужды в стеке, поскольку возвращаемого значения никто не ждет! Мы на самом деле перевели дерево выражения в линейный вид точно так же, как это

[26] Идею стиля с передачей продолжений ввели теоретики компьютерных языков, чтобы прояснить их семантику. Полную историю этой идеи можно найти в [103]. В языке Scheme продолжения, используемые в вычислении подвыражений, явно представляются как полноправные процедуры [120, 61, 109].

должен делать компилятор, чтобы вычислить выражение на машине, работающей последовательно.

Нижележащие продолжения

Идея состоит в том, что место в выражении, где содержится подвыражение, — это всего лишь синтаксический сахар для процедуры, принимающей значение подвыражения, чтобы затем использовать его, когда вычисление выражения продолжится. Это важная и мощная идея, поскольку продолжение представляет все будущее вычисления. Такое более глубокое понимание значения выражения позволяет нам выйти из тисков стиля программирования, когда у всякого выражения есть только одно значение. При этом мы платим существенную цену, увеличивая сложность и степень синтаксической вложенности.

Каждый раз, когда вычисляется какое-то выражение, имеется продолжение, готовое принять результат вычисления. Например, если вычисление происходит на верхнем уровне, это продолжение берет результат, печатает его, выводит подсказку, считывает следующее входное выражение, вычисляет его, и т. д. без конца. По большей части продолжения включают действия, определяемые пользовательским кодом, как, например, продолжение, которое примет результат, умножит его на значение, хранимое в локальной переменной, добавит 7 и передаст то, что получилось, верхнеуровневому продолжению для распечатки. Как правило, эти вездесущие продолжения спрятаны за кулисами, и программисты о них особенно не думают. Язык Scheme дает программисту возможность получить это нижележащее продолжение у любого выражения. Нижележащее продолжение является полноправным объектом, который можно передать как аргумент, вернуть как значение и сохранить внутри структуры данных. Большинство других языков не поддерживает использование полноправных продолжений. (Языки, где такая поддержка есть, включают SML, Ruby и Smalltalk.)

Явное представление нижележащих продолжений — один из самых мощных (и самых опасных) инструментов, доступных программисту. Продолжения дают программисту власть над временем. Вычисление можно захватить и приостановить в какой-то один момент, а затем восстановить и продолжить когда-нибудь в будущем. Это позволяет организовать сопрограммы (кооперативную многозадачность), а при добавлении механизма прерываний от таймера — и разделение времени (вытесняющую многозадачность). В тех случаях, когда требуется работать с продолжениями явным образом, Scheme позволяет создать процедуру, представляющую собой текущее продолжение. Однако, прежде чем мы овладеем этой властью, нужно лучше понять, что такое продолжения.

Продолжение представляет собой захваченное состояние структуры управления вычислением[27]. Когда продолжение вызывается, вычисление продолжается с того места, которое оно представляет. Продолжение может представлять такое действие, как возврат значения подвыражения в вычисление объемлющего выражения. В таком случае это процедура, которая при вызове вернет свой аргумент в вычисление этого объемлющего выражения как значение подвыражения. Продолжение можно

[27] Состояние структуры управления не стоит смешивать с полным состоянием системы. Полное состояние — это вся информация, которая, наряду с самой программой, определяет будущее вычисление. Она включает все текущие значения изменяемых переменных и структур данных. Продолжение не захватывает текущие значения переменных и структур данных.

вызывать много раз, и вычисление продолжится с определенной точки с различными значениями, которые это продолжение возвращает. Вскоре мы увидим пример.

Scheme содержит элементарную процедуру `call-with-current-continuation` (сокращается как `call/cc`), дающую доступ к нижележащему продолжению под структурой текущего выражения. Аргументом `call/cc` служит процедура, которая примет в качестве аргумента продолжение выражения `call/cc`. Повторим еще раз: продолжение является полноправной процедурой, принимающей один аргумент — значение, которое будет возвращено при вызове этого продолжения[28]. Вот простой пример:

```
(define foo)

(set! foo
  (+ 1
     (call/cc
        (lambda (k)
          ;; k --- продолжение
          ;; выражения call/cc.
          ;; Если мы позовем k с аргументом 6,
          ;; то foo получит значение 11
          (k (* 2 3))))
     (/ 8 2)))

foo
11
```

Итак, `call/cc` зовет свой аргумент, передавая ему продолжение выражения `call/cc`. Пока что ничего особенно интересного! Это ничем не отличается от обычного вычисления `(+ 1 (* 2 3) (/ 8 2))`.

Однако время жизни процедур в Scheme не ограничено. Это совершенно меняет дело. Давайте сохраним продолжение и позовем его еще раз.

```
(define bar)
(define foo)

(set! foo
  (+ 1
     (call/cc
        (lambda (k)
          (set! bar k)
          (k (* 2 3))))
     (/ 8 2)))

foo
11

(bar -2)
```

[28]Заметим, что стандарт Scheme [109] позволяет продолжению принимать произвольное число аргументов.

```
foo
3
```

Смотрите, что произошло! Мы сохранили в переменной `bar` будущее вычисление, которое в конце концов присваивает значение переменной `foo`. Когда мы вызвали это продолжение с новым значением, присваивание переменной `foo` случилось опять, и значение `foo` изменилось.

5.5.1 Продолжения как нелокальные возвраты

Рассмотрим следующий простой пример продолжения нелокального возврата (заимствовано из «Сообщения о языке Scheme» [109]):

```
(call/cc
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x) (exit x)))
              '(54 0 37 -3 245 -19))     ; **
    (exit #t)))
-3
```

Поскольку процедура `for-each` в Scheme просматривает список слева направо, первым встреченным отрицательным элементом бужет -3, и он немедленно возвращается. Если бы в списке не было отрицательных элементов, было бы возвращено значение `#t` (поскольку телом внешнего выражения `lambda` является последовательность двух выражений: сначала выражения `for-each`, затем возврата значения `#t`).

Выражение `call/cc` может использоваться внутри других, как в следующем определении. (По традиции, символ, связываемый с нижележащим продолжением, начинается с буквы `k`.)

```
(define (first-negative list-of-numbers)
  (call/cc
   (lambda (k_exit)
     (or (call/cc (lambda (k_shortcut)
                    (for-each (lambda (n)
                                (cond ((not (number? n))
                                       (pp `(not-a-number: ,n))
                                       (k_exit #f))
                                      ((negative? n)
                                       (k_shortcut n))
                                      (else
                                       'keep-looking)))
                              list-of-numbers)
                    #f))
         'no-negatives-found))))
```

Этот код ведет себя так:

```
(first-negative '(54 0 37 -3 245 -19))
-3
```

```
(first-negative '(54 0 37  3 245  19))
no-negatives-found

(first-negative '(54 0 37 no 245 -19))
(not-a-number: no)
#f
```

Этот пример показывает вложенные продолжения: внешнее продолжение `k_exit` выходит из всего вызова `first-negative`, а внутреннее продолжение `k_shortcut` выходит только из объемлющей дизъюнкции, а затем продолжает с этого места.

Короче говоря, если продолжение, захваченное конструкцией `call/cc`, вызывается с некоторым значением, то следующим шагом вычисления будет возврат этого значения из вызова `call/cc`, который это продолжение захватил, и затем вычисление продолжится с этой точки обычным образом.

Упражнение 5.21. Нелокальные возвраты
Это упражнение лучше делать в родной реализации Scheme; там его будет проще отлаживать и модифицировать, чем в нашем встроенном интерпретаторе. В родной системе Scheme присутствует эффективная реализация `call/cc`.

a. Определите простую процедуру `snark-hunt`[29], которая принимает на вход дерево и рекурсивно по нему спускается в поисках символа `snark` в любом листе. Если она его находит, нужно немедленно остановиться и вернуть `#t`. Если нет — вернуть `#f`. Используйте `call/cc`. Например:

   ```
   (snark-hunt '(((a b c) d (e f)) g (((snark . "oops") h) (i . j))))
   #t
   ```

   Заметим, что входной аргумент процедуры `snark-hunt` не обязан быть каноническим списком.

b. Как проверить, что `snark-hunt` выпрыгивает немедленно, а не возвращается молчаливо через множество уровней дерева? Для демонстрации этого определите новую процедуру `snark-hunt/instrumented`.

Примечание: при тщательном использовании может сработать установка флага и сигнализирование об ошибке на излишних возвратных путях, но проще может оказаться простая трассировка через `pp`. Нашей целью здесь является наработать интуицию, касающуюся продолжений, а не написать код промышленного качества. Кратко объясните свою стратегию.

5.5.2 Нелокальная передача управления

Предыдущие примеры были довольно простыми, поскольку захваченные продолжения использовались только для нелокального возврата. Однако продолжения — более мощная вещь; после вызова в них можно войти еще раз. Эту идею иллюстрирует следующий пример[30]:

```
(define the-continuation #f)
```

[29] См. «Охота на Снарка», Льюис Кэрролл, 1876.
[30] Пример сделан на основе «Википедии» [25].

```
(define (test)
  (let ((i 0))
    ;; Аргумент call/cc присваивает продолжение,
    ;; созданное call/cc, глобальной
    ;; переменной the-continuation.
    (call/cc (lambda (k) (set! the-continuation k)))
    ;; Когда вызывается the-continuation,
    ;; вычисление продолжается отсюда.
    (set! i (+ i 1))
    i))
```

Поведение этого кода может показаться удивительным. Процедура `test` создает локальную переменную `i` с начальным значением 0. Кроме того, она создает продолжение, представляющее состояние управления при возврате из вызова `call/cc` в теле выражения `let`, и сохраняет это продолжение в глобальной переменной `the-continuation`. Затем она увеличивает переменную `i` на единицу и возвращает ее новое значение: 1.

```
(test)
1
```

Когда вызывается `the-continuation`, `call/cc` возвращается в тело выражения `let`. Вычисление продолжается, переменная `i` опять увеличивается, и возвращается новое значение. Можно позвать продолжение снова, увеличить `i` еще раз и опять вернуть значение.

```
(the-continuation 'OK)
2

(the-continuation 'OK)
3
```

(Аргумент `OK` возвращается как значение `call/cc`; в теле `let` это значение игнорируется.)

Сохраним продолжение в переменной `another-continuation`, чтобы можно было создать еще одно в `the-continuation` при новом вызове `test`. Вызов `test` создает еще один экземпляр переменной `i` с начальным значением 0.

```
(define another-continuation the-continuation)

(test)
1
```

Новое продолжение независимо от уже сохраненного в `another-continuation`.

```
(the-continuation 'OK)
2

(another-continuation 'OK) ; сохраненное продолжение
4
```

Рассмотрим теперь чуть более интересный сценарий.

```
(define the-continuation #f)
(define sum #f)

(begin
  (set! sum
        (+ 2 (call/cc
              (lambda (k)
                (set! the-continuation k)
                (k 3)))))
  'ok)
ok

sum
5

(the-continuation 4)
ok

sum
6

(the-continuation 5)
ok

sum
7
```

Внимательно проследите, как захваченное продолжение при вызове `the-continuation` возвращает управление в точку перед сложением и, следовательно, перед присваиванием переменной `sum` и перед возвратом значения `ok`. Вот почему вызов продолжения всегда приводит к возврату `ok`. Однако в переменной `sum` сохраняется сумма 2 и аргумента, переданного в `the-continuation`. Поэтому при запросе значения `sum` мы получаем эту новую сумму. Это показывает, как можно с помощью захваченного продолжения возвращаться к промежуточным точкам ввычисления. В разделе 5.5.3 мы увидим, как этот механизм использовать для перебора с возвратами.

5.5.3 От продолжений к `amb`

Выясняется, что почти все, что нам хочется, включая реализацию `amb`, можно сделать при помощи родного `call/cc`, встроенного в Scheme. Давайте рассмотрим, как это выходит.

Продолжения — естественный механизм для поддержки перебора с возвратами. Происходит выбор альтернативы, а если в будущем эта альтернатива окажется неподходящей, можно выбрать другую и рассмотреть ее последствия. (Вот бы такая возможность существовала и в реальной жизни!) В нашем примере с квадратным корнем программа должна вернуть `amb` от обоих корней, где `amb` — оператор, выбирающий и возвращающий один из них, но оставляющий возможность получить и другой, если первый будет отвергнут. Получатель может просто начать использо-

вать данное ему решение; а если в какой-то момент получатель обнаружит, что его вычисление нарушает какие-то важные условия, он просто позовет `fail`, поэтому оператор `amb` пересмотрит свой выбор и вернет через свое продолжение другую альтернативу. В сущности, продолжение позволяет писать генератор альтернатив как сопрограмму, взаимодействующую с получателем/программой проверки вариантов.

Сердцем процедуры поиска с возвратами служит процедура `amb-list`, принимающая последовательность санков, каждый из который отвечает за возможное значение выражения `amb`. Санки порождаются макросом `amb`, который синтаксически преобразует выражение `amb` в выражение `amb-list` следующим образом:

```
(amb e1 ... en) ==>
  (amb-list (list (lambda () e1) ... (lambda () en)))
```

Макрос `amb`, записанный в переносимой форме `syntax-rules`, выглядит так:

```
(define-syntax amb
  (syntax-rules ()
    ((amb exp ...)
     (amb-list (list (lambda () exp) ...)))))
```

Например,

```
(pp (syntax '(amb a b c) user-initial-environment))
(amb-list (list (lambda () a) (lambda () b) (lambda () c)))
```

Программа поиска поддерживает расписание, коллекцию санков, которые можно позвать, когда потребуется вернуть новое альтернативное значение из выражения `amb`. Процедура `amb-list` сначала добавляет санки для своих альтернатив в это расписание, а затем передает управление первому санку в нем. Если альтернатив нет (выражение было просто `(amb)`), то `amb-list` отдает управление, ничего не добавив в расписание, и увеличивает глобальный счетчик для проверки процесса поиска.

```
(define (amb-list alternatives)
  (if (null? alternatives)
      (set! *number-of-calls-to-fail*
            (+ *number-of-calls-to-fail* 1)))
  (call/cc
   (lambda (k)
     ((add-to-search-schedule)
      (map (lambda (alternative)
             (lambda ()
               (within-continuation k alternative)))
           alternatives))
      (yield))))
```

Для каждого конкретного выражения `amb` санки альтернатив построены так, чтобы возвращаться из этого `amb` с помощью продолжения `k`, которое захватывается на входе в объемлющий их `amb-list`[31].

[31] Мы используем расширение MIT/GNU Scheme в виде процедуры `within-continuation`, которая здесь примерно равносильна вызову `(k (alternative))`, но избегает захватывать фрагменты стека управления, не нужные для правильного продолжения вычислений.

Способ отдать управление — извлечь из расписания санк, описывающий альтернативу, если такой есть, и выполнить его. Расписание поиска одновременно является стеком и очередью; вскоре мы увидим, почему.

```
(define (yield)
  (if (deque-empty? (*search-schedule*))
      ((*top-level*) 'no-more-alternatives)
      ((pop! (*search-schedule*)))))
```

Вас может удивить, что для получения процедур, выполняющих работу, мы вызываем `*top-level*` и `add-to-search-schedule`. Кроме того, для получения объекта-расписания мы вызываем `*search-schedule*`. Причина такого непрямого обращения в том, что это *объекты-параметры* языка Scheme (см. стр. 344). Мы их таким образом определили потому, что, как вскоре увидим, будем динамически их связывать с различными значениями. В начале расписание `*search-schedule*` должно быть пустым:

```
(define *search-schedule*
  (make-parameter (empty-search-schedule)))
```

Волшебство заключается в вызове `call/cc` внутри `amb-list`. `Amb-list` (и таким образом, `amb`) выполняет `yield`. Продолжение этого `call/cc`, оно же продолжение `amb`, помещается в расписание для каждой альтернативы внутри `amb`. Когда альтернатива извлекается из расписания и выполняется, ее значение возвращается из того выражения `amb`, которое поместило ее в расписание.

Поскольку наше расписание одновременно является стеком и очередью, мы можем организовать поиск как в глубину, так и в ширину. Эти два вида поиска отличаются только порядком альтернатив в расписании. Управление происходит через динамическое связывание параметра `add-to-search-schedule` тем или другим порядком, как показано ниже.

По умолчанию перебор происходит в глубину:

```
(define add-to-search-schedule
  (make-parameter add-to-depth-first-search-schedule))
```

Следующие две процедуры используются для управления порядком поиска при выполнении санка, охватывающего подлежащую решению задачу, через динамическое связывание `add-to-search-schedule`. Пример их использования можно видеть в упражнении 5.22 на стр. 259.

```
(define (with-depth-first-schedule problem-thunk)
  (call/cc
   (lambda (k)
     (parameterize ((add-to-search-schedule
                     add-to-depth-first-search-schedule)
                    (*search-schedule*
                     (empty-search-schedule))
                    (*top-level* k))
       (problem-thunk)))))

(define (with-breadth-first-schedule problem-thunk)
```

```
  (call/cc
   (lambda (k)
     (parameterize ((add-to-search-schedule
                     add-to-breadth-first-search-schedule)
                    (*search-schedule*
                     (empty-search-schedule))
                    (*top-level* k))
       (problem-thunk)))))
```

Также эти процедуры локально переинициализируют `*search-schedule*` и `*top-level*` через динамическое связывание, и, таким образом, областью управления является промежуток времени, а не синтаксическая область действия. Если у выражения `yield` не осталось альтернатив, оно может прекратить поиск и вернуть процедуре, вызвавшей `with-...-first-schedule`, значение `no-more-alternatives`. Например:

```
(define search-order-demo
  (lambda ()
    (let ((x (amb 1 2)))
      (pp (list x))
      (let ((y (amb 'a 'b)))
        (pp (list x y))))
    (amb)))

(with-depth-first-schedule search-order-demo)
(1)
(1 a)
(1 b)
(2)
(2 a)
(2 b)
no-more-alternatives

(with-breadth-first-schedule search-order-demo)
(1)
(2)
(1 a)
(1 b)
(2 a)
(2 b)
no-more-alternatives
```

Процедуры, реализующие поиск в том или ином порядке, реализованы через низкоуровневую модификацию стека или очереди. При поиске в глубину альтернативы помещаются в начало расписания, а при поиске в ширину — в конец. В обоих случаях варианты появляются в расписании в том же порядке, как и при вызове `amb`.

```
(define (add-to-depth-first-search-schedule alternatives)
  (for-each (lambda (alternative)
```

```
            (push! (*search-schedule*) alternative))
          (reverse alternatives)))

(define (add-to-breadth-first-search-schedule alternatives)
  (for-each (lambda (alternative)
              (add-to-end! (*search-schedule*) alternative))
            alternatives))
```

Параметр `*top-level*` инициализируется таким образом, что, когда никаких альтернатив не найдено, система возвращается в управляющий цикл с данным результатом `result`. (В приведенном выше коде процедура `yield` передает на верхний уровень символ `no-more-alternatives`.) Заметим, что `with-...-first-schedule` заново связывает `*top-level*`.

```
(define *top-level*
  (make-parameter
   (lambda (result)
     (abort->nearest
      (cmdl-message/active
       (lambda (port)
         (fresh-line port)
         (display "; " port)
         (write result port)))))))
```

Для того, чтобы запустить всю систему, требуется еще:

```
(define (init-amb)
  (reset-deque! (*search-schedule*))
  (set! *number-of-calls-to-fail* 0)
  'done)
```

И наконец, почти любая программа, работающая с `amb`, захочет использовать конструкцию `require`:

```
(define (require p)
  (if (not p) (amb) 'ok))
```

На этом все! Удивительно, как много можно достичь с помощью `call/cc`. Таким образом, если у нас в родном окружении есть `call/cc`, то для реализации `amb` не нужно писать встроенную систему (как мы делали в разделе 5.4.2).

Другие способы перебора альтернатив

Если есть разные варианты решения некоторой подзадачи, и только некоторые из них подходят для решения задачи большего размера, попытки применить их один за другим по схеме порождения и фильтрации — это только один из возможных подходов. Например, если какие-то варианты ведут к очень длинным (возможно, бесконечным) вычислениям при проверке, а другие могут быстро привести к успеху или неудаче, имеет смысл выделить для каждой альтернативы по потоку и исполнять эти потоки параллельно. Потребуется какой-то механизм, позволяющий потокам взаимодействовать и, возможно, позволяющий успешному потоку убивать своих

братьев. Все это можно реализовать на основе продолжений, организуя взаимодействие одного потока с другим через транзакции.

Упражнение 5.22. Ширина и глубина
Напомним наш неэффективный метод поиска пифагоровых троек (стр. 241). Добавим в процедуру поиска счетчик проверенных троек:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (set! triples-tested (+ triples-tested 1))
        (require (= (+ (* i i) (* j j))
                    (* k k)))
        (list i j k)))))

(define triples-tested 0)
```

Рассмотрим теперь следующий эксперимент. Попробуем сначала поиск в ширину:

```
(begin (init-amb)  ; сбрасываем счетчик неудач
       (set! triples-tested 0)
       (with-breadth-first-schedule
        (lambda ()
          (pp (a-pythagorean-triple-between 10 20)))))
(12 16 20)

triples-tested
246

*number-of-calls-to-fail*
282
```

затем поиск в глубину:

```
(begin (init-amb)
       (set! triples-tested 0)
       (with-depth-first-schedule
        (lambda ()
          (pp (a-pythagorean-triple-between 10 20)))))
(12 16 20)

triples-tested
156

*number-of-calls-to-fail*
182
```

a. Объясните разницу в значениях `triples-tested` для поиска в глубину и в ширину (достаточно грубых оценок; точные значения высчитывать необязательно).

b. Объясните разницу между числом опробованных троек и значением `*number-of-calls-to-fail*`. Откуда берутся дополнительные неудачи?

c. При том, что перебор в ширину проделывает больше работы, почему следующую процедуру невозможно использовать со стратегией поиска в глубину, хотя при поиске в ширину она отлично работает?

```
(define (a-pythagorean-triple-from low)
  (let ((i (an-integer-from low)))
    (let ((j (an-integer-from i)))
      (let ((k (an-integer-from j)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))

(define (an-integer-from low)
  (amb low (an-integer-from (+ low 1))))

(with-depth-first-schedule
  (lambda ()
    (pp (a-pythagorean-triple-from 10))))
```

Упражнение 5.23. Еще менее детерминистский недетерминизм

Ева Лу Атор замечает, что наша реализация `amb` не настолько недетерминистична, как иногда бы хотелось. А именно, когда в форме `amb` есть список альтернатив, сначала всегда выбирается самая левая, затем вторая и т. д., в порядке слева направо.

Она предлагает добавить возможность изменить этот выбор, например проходя альтернативы справа налево или даже в случайном порядке. А именно, ей бы хотелось иметь что-то вроде

`(with-left-to-right-ordering` *задача*`)`
`(with-right-to-left-ordering` *задача*`)`
`(with-random-ordering` *задача*`)`

Она замечает также, что выбор порядка перебора альтернатив независим от стратегии поиска (в ширину, в глубину или как-то иначе).

a. При каких условиях может понадобиться неупорядоченный `amb` (со случайным порядком перебора)? Придумайте конкретный короткий пример и используйте его, чтобы протестировать пункт b.

b. Реализуйте эти три порядка перебора альтернатив и приведите пример для каждого. Ради простоты и единообразия постройте свой код так же, как `with-depth-first-schedule`, `add-to-depth-first-search-schedule` и т. д. Подсказка: можно использовать встроенную в Scheme процедуру `random`.

Упражнение 5.24. Стратегии вложения

Хотелось бы, чтобы стратегии поиска в ширину и в глубину могли произвольным образом комбинироваться при вложенных поисках. Можно ли организовать вложение разных стратегий работы с расписанием в нашей нынешней реализации? А именно придумайте пример, демонстрирующий ошибку (если она есть) либо на примере показывающий, что все работает как надо. Объясните свою логику.

Здесь требуется построить эксперименты, различающие работу поиска в ширину и в глубину, а также интересным образом скомбинировать их, чтобы показать локальное управление вложенными экземплярами поиска.

Найдите естественный класс задач, где эта гибкость окажется полезной, а не просто программистским трюком ради формальной правильности.

Упражнение 5.25. Отменяемое присваивание

В версии `amb` для встроенного интерпретатора из раздела 5.4.2 мы показали, как можно использовать присваивание двух видов: обычное постоянное, обозначаемое `set!`, и отменяемое присваивание, обозначаемое `maybe-set!`, которое при откате перебора возвращает

старое значение. В реализации этого раздела, встроенной в родную реализацию Scheme, можно реализовать общую обертку для отменяемых эффектов:

```
(define (effect-wrapper doer undoer)
  (force-next
   (lambda () (undoer) (yield)))
  (doer))

(define (force-next thunk)
  (push! (*search-schedule*) thunk))
```

И тогда `maybe-set!` реализуется в виде макроса:

```
(define-syntax maybe-set!
  (syntax-rules ()
    ((maybe-set! var val)
     (let ((old-val var))
       (effect-wrapper
        (lambda ()
          (set! var val))
        (lambda ()
          (set! var old-val)))))))
```

К сожалению, это работает только при переборе в глубину; при переборе в ширину эта реализация не имеет смысла. Объясните, почему так. Можно ли это исправить?

Упражнение 5.26. Управление поиском во встроенной системе
Как можно изменить встроенную систему из раздела 5.4.2 с анализом, который порождает комбинаторы, имеющие продолжения успеха и неудачи, так, чтобы она поддерживала перебор и в глубину (как сейчас), и в ширину? Объясните свой план. Создайте новую реализацию, которая обладает способностью управлять стратегией перебора. Примечание: потребуется довольно большая трансформация.

5.6 Власть и ответственность

В этой главе мы увидели, что универсальность вычисления по Чёрчу—Тьюрингу дает нам громадную власть над поведением программ. Никогда нельзя жаловаться: «В языке, который я вынужден использовать, нужную мне идею невозможно выразить». Если мы умеем строить интерпретаторы и компиляторы, то всегда сможем избежать ограничений любого конкретного языка, потому что для любой задачи всегда можно построить подходящий специализированный язык. В нашем изложении мы используем в качестве базового языка Scheme и на его основе строим мощные лиспоподобные языки. Синтаксис Lisp мы выбрали потому, что он значительно упрощает изложение этих идей. (Смотрите упражнение 5.7 об инфиксной нотации. Если бы нам пришлось работать в языке со сложным синтаксисом, наш рассказ был бы во много раз длиннее и скучнее.) Однако сила интерпретации доступна программисту в любом Тьюринг-полном языке.

Для сохранения гибкости в будущем важно, чтобы создаваемые нами языки были простыми и обладали достаточной общностью. Число механизмов должно быть ограничено: элементарные конструкции, средства комбинирования и средства абстракции. Нужно иметь возможность расширять их при необходимости, а также сочетать и смешивать компоненты программ. И самое важное, когда у нас есть

несколько языков, каждый из которых приспособлен для решения части большой задачи, эти языки должны иметь возможность взаимодействовать.

Большая власть приводит к еще большей ответственности. Каждый раз, когда мы создаем новый язык, этот язык должен быть документирован, чтобы других можно было ему научить. Программы, которые мы пишем сегодня, завтра будут читать и модифицировать другие. (Даже когда мы сами в следующем году будем читать то, что написали в прошлом году, мы сами будем другими и можем не вспомнить детали своей работы.) Поэтому важно использовать свою власть умеренно, и при этом тщательно документировать свои действия. Иначе следующему программисту после нас (или даже нам самим!) достанется нечитаемая гора обрывков, которую можно только выбросить и переписать. Не участвуйте в строительстве новой Вавилонской башни!

Системы, происходящие от UNIX, дают нам множество и хороших, и плохих примеров. У каждой команды свой собственный язык. Если вы знакомы с `awk`, `grep` и `sed`, то знаете, что все они различаются языками, включая уродливый и плохо определенный язык регулярных выражений, проанализированный нами в разделе 2.2. Каждый из этих языков облегчал решение какой-то конкретной задачи. Однако за ними не стоит никакой общей стройной идеи, которая помогла бы выучить и понять их. Подумайте только, как соглашения о кавычках работают в программной оболочке и в `grep`, и вы оцените то, что мы говорим. Чтобы стать специалистом в UNIX, нужно выучить множество уродливых особых случаев. С другой стороны, в UNIX есть замечательно простой и изящный способ сочетания программ между собой: *потоки*. Всякая из базовых программ UNIX читает поток на входе и выдает потоки на выходе. Их можно соединять, направляя выходной поток одной программы на вход другой. Этот урок достоин внимательного изучения.

Глава 6

Слои

В разделе 1.1 мы высказали мысль, что программирование как дисциплина могло бы воспринять некоторые практики из области архитектуры. Программист может начинать работу с исполнимого скелетного плана (*эскиза*), который помогает опробовать и оценить основные идеи. Когда эскиз начинает выглядеть и вести себя разумно, программист может наполнить его новой информацией.

Например, объявления типов в реализации программы могут обеспечить эффективную компиляцию кода и предотвратить появление ошибок типа. Объявления размерностей и единиц измерения позволяют избежать некоторых ошибок и улучшают документирование кода. Контрольные утверждения, проверяющие предикаты в различных точках программы, могут помочь в локализации ошибок, возникающих во время выполнения; кроме того, с их помощью можно строить автоматически либо вручную доказательства «правильности» программ. Указания, какая степень точности требуется для числовых величин и операций, могут прояснить некоторые проблемы численного анализа. Отметки об альтернативных реализациях подзадач могут помочь получить плавную деградацию программы. Можно отследить родословную данных, снабдив их записями о зависимостях.

Однако обычный способ добавления этой важной и полезной информации в текст программы превращает текст в запутанную кашу. Если продолжить архитектурную аналогию, при этом не отделяется обслуживаемое пространство от обслуживающих пространств. Отделение «сущностных» свойств программы (кода, определяющего ее поведение) от «вспомогательных» (например, информации о типах для компилятора или кода для протоколирования) было темой многих исследований. Одной из попыток решить часть этой задачи стало аспектно-ориентированное программирование [67], где явно выделялись «сквозные проблемы» вроде ведения протокола. Многослойная организация — еще один способ обеспечить это разделение.

При построении гибких систем возможность аннотировать любой фрагмент данных или кода другими данными или кодом является важнейшим механизмом. Орнаментация значений — это обобщение меток, используемых для поддержки расширяемых полиморфных операций. Здесь мы вводим понятие *многослойного программирования*. Как данные, так и обрабатывающие их процедуры будут строиться в различных слоях, и таким образом, обеспечивается аддитивное аннотирование, не вносящее беспорядка.

6.1 Использование слоевой структуры

Слои дают нам возможность набросать эскиз вычисления, а затем обогатить это вычисление метаданными, которые будут обрабатываться совместно с основным вычислением. Рассмотрим некоторые аннотации, которые, по нашему мнению, пригодятся во многих ситуациях. Предположим, например, что мы хотим использовать закон Ньютона о гравитационной силе:

```
(define (F m1 m2 r)
  (/ (* G m1 m2) (square r)))
```

Это простое численное вычисление, но можно его дополнить, снабдив вспомогательной информацией и единицами измерения.

Ньютонову константу G мы берем из данных недавнего измерения, опубликованного NIST (Национальным институтом стандартов и технологий США):

```
(define G
  (layered-datum 6.67408e-11
    unit-layer (unit 'meter 3 'kilogram -1 'second -2)
    support-layer (support-set 'CODATA-2018)))
```

Здесь записаны значение в виде числа, единицы измерения ($м^3/(кг \cdot с^2)$) и источник данных. Можно дополнить эти сведения интервалом неопределенности как еще одним слоем, но здесь мы этого делать не будем.

Из других источников можем получить массу Земли, массу Луны и расстояние до Луны (половину большой полуоси):

```
(define M-Earth
  (layered-datum 5.9722e24
    unit-layer (unit 'kilogram 1)
    support-layer
    (support-set 'Astronomical-Almanac-2016)))

(define M-Moon
  (layered-datum 7.342e22
    unit-layer (unit 'kilogram 1)
    support-layer
    (support-set 'NASA-2006)))

(define a-Moon
  (layered-datum 384399e3
    unit-layer (unit 'meter 1)
    support-layer
    (support-set 'Wieczorek-2006)))
```

Если теперь мы зададим вопрос: «Какова сила гравитационного притяжения между Землей и Луной на этом расстоянии?» — то получим ответ:

```
(pp (F M-earth M-Moon a-moon))
#[layered-datum 1.9805035857209e20]
(base-layer 1.9805035857209e20)
```

```
(unit-layer (unit kilogram 1 meter 1 second -2))
(support-layer
 (support-set Wieczorek-2006
              NASA-2006
              Astronomical-Almanac-2016
              CODATA-2018))
```

В результате указаны числовое значение, единицы измерения и источники, от которых этот результат зависит.

6.2 Реализация слоев

Реализация многослойной системы состоит из двух частей. Во-первых, должно быть возможно создать элемент данных, содержащий несколько слоев информации. В нашем примере для этой цели использовалась конструкция `layered-datum`. Во-вторых, нужно уметь расширить процедуру так, чтобы она каждый слой обрабатывала (до некоторой степени) независимо. Расширенная таким образом процедура называется *многослойной* или *слоеной*.

Требуется также способ давать слоям имена. У каждого из них должно быть имя, чтобы всякий слой во всяком элементе данных был идентифицирован. Кроме того, с помощью этого имени внутри многослойной процедуры обработка слоя связывается с соответствующими слоями входных данных. В нашем примере на имена слоев ссылаются переменные; так, переменная `unit-layer` имеет значением имя слоя для обработки единиц измерения. Таким образом, интерфейс пользователя независим от подробностей определения имени слоя; это окажется впоследствии полезным.

Еще одна деталь именования слоев заключается в том, что должен иметься выделенный *базовый слой*, представляющий нижележащее вычисление, которое мы производим. В нашем примере с использованием `layered-datum` значение базового слоя отличается тем, что это первый аргумент, и с ним не связано никакого имени.

Многослойные данные можно строить из простых структур данных. Можно использовать любую структуру, способную связать имя слоя со значением и позволяющую иметь несколько таких связываний. Для базового слоя можно использовать какое-нибудь особое имя, и тогда структура данных будет проста и единообразна.

Построение многослойных процедур организовано сложнее, потому что обработка большинства слоев потребует какую-то информацию от основного слоя вычислений. Пусть, например, мы перемножаем между собой два числа, имеющих при себе вспомогательную информацию, поддерживающую их. В обычном случае поддерживающая информация о результате будет объединением поддержки аргументов. Однако предположим, что у одного из аргументов базовое значение равно нулю; тогда поддержка результата будет состоять в поддержке этого нуля, а поддержка другого аргумента не имеет значения.

Базовый слой не должен зависеть ни от какого небазового, поскольку иначе теряет смысл сама идея базового слоя, а именно что это независимое вычисление, которое остальные слои расширяют. Небазовый слой тоже не должен зависеть ни от какого другого небазового слоя. В общем случае такой слой не должен разделять информацию ни с каким другим небазовым слоем, поскольку иначе его поведение будет различаться в зависимости от присутствия этого другого слоя. Это не сочетается с нашим общим подходом к построению аддитивных программ.

Таким образом, построение многослойной процедуры требует баланса между предоставлением информации из базового слоя в небазовые и изоляцией между слоями во всех остальных случаях. Мы обратим на это внимание при рассмотрении деталей реализации многослойной системы.

6.2.1 Многослойные данные

Элемент многослойных данных состоит из базового значения, снабженного дополнительной информацией. Эта информация состоит из ассоциации имен слоев со значениями. Например, число 2 может служить базовым значением для множества элементов данных: если мы работаем с данными о картофеле, у нас может быть цена в 2 долл. за мешок весом 2 фунта. Каждый экземпляр числа 2 должен быть отдельным элементом данных, с различными значениями (доллары или фунты) в слое единиц измерения. Могут быть и другие слои: цена в 2 долл. может нести при себе информацию, как именно она была получена из цены, уплаченной фермеру, стоимости перевозки и обработки.

Для решения этой задачи мы вводим понятие *многослойного элемента данных*. Такой элемент представляется в виде пакета, содержащего ассоциацию между слоями и их значениями. Так что количество в 2 фунта и цена в 2 долл. будут двумя различными многослойными элементами:

```
(define (make-layered-datum base-value alist)
  (if (null? alist)
      base-value
      (let ((alist
             (cons (cons base-layer base-value)
                   alist)))
        (define (has-layer? layer)
          (and (assv layer alist) #t))
        (define (get-layer-value layer)
          (cdr (assv layer alist)))
        (define (annotation-layers)
          (map car (cdr alist)))
        (bundle layered-datum?
                has-layer? get-layer-value
                annotation-layers))))
```

Связывания между слоями и их значениями представляются в виде *ассоциативного списка*, или *а-списка*, — списка пар вида ключ—значение.

Для удобства мы предоставляем пользователю процедуру `layered-datum`, которая принимает аргументы в виде *списка свойств* (имя слоя и значение по очереди, как в примерах на стр. 264) и вызывает `make-layered-datum` с соответствующим а-списком.

```
(define (layered-datum base-value . plist)
  (make-layered-datum base-value (plist->alist plist)))
```

Такая организация обладает большой гибкостью. Может существовать множество разновидностей многослойных данных, и для каждой из них в принципе нет априорного обязательства иметь какой-то определенный слой или заранее заданное

число слоев. Единственное общее свойство состоит в том, что у любого многослойного элемента есть выделенный базовый слой `base-layer`, и он содержит объект, к которому значения остальных слоев служат аннотациями.

Каждый слой также представляется пакетом, содержащим данные об особенностях этого слоя. Самый простой слой базовый:

```
(define base-layer
  (let ()
    (define (get-name) 'base)
    (define (has-value? object) #t)
    (define (get-value object)
      (if (layered-datum? object)
          (object 'get-layer-value base-layer)
          object))
    (bundle layer? get-name has-value? get-value)))
```

Здесь показана основная операция слоя: `get-value`, которая извлекает значение слоя, если оно присутствует, или умолчательное значение. В случае базового слоя умолчанием является объект сам по себе.

Слои аннотации устроены несколько сложнее. В дополнение к тому, что описано выше, они содержат еще набор именованных процедур, которые мы исследуем, когда будем рассматривать многослойные процедуры. Процедура `make-annotation-layer` задает общую инфраструктуру, используемую всеми слоями аннотации; те части, которые относятся к конкретному слою, порождаются при вызове ее аргумента `constructor`.

```
(define (make-annotation-layer name constructor)
  (define (get-name) name)
  (define (has-value? object)
    (and (layered-datum? object)
         (object 'has-layer? layer)))
  (define (get-value object)
    (if (has-value? object)
        (object 'get-layer-value layer)
        (layer 'get-default-value)))
  (define layer
    (constructor get-name has-value? get-value))
  layer)
```

С помощью `make-annotation-layer` мы порождаем слой единиц измерения:

```
(define unit-layer
  (make-annotation-layer 'unit
    (lambda (get-name has-value? get-value)
      (define (get-default-value)
        unit:none)
      (define (get-procedure name arity)
        См. определение на стр. 270.)
      (bundle layer?
              get-name has-value? get-value
              get-default-value get-procedure))))
```

В этой процедуре мы видим оставшуюся часть структуры слоя: процедуру `get-default-value`, выдающую умолчательное значение, и процедуру `get-procedure`, реализующую в данном слое поддержку многослойных процедур; мы ее рассмотрим в следующем разделе (стр. 270).

Ради удобства при часто встречающемся варианте использования `layer-accessor` порождает процедуру доступа, эквивалентную вызову делегата `get-value`:

```
(define (layer-accessor layer)
  (lambda (object)
    (layer 'get-value object)))

(define base-layer-value
  (layer-accessor base-layer))
```

6.2.2 Многослойные процедуры

Процедуры тоже являются данными, которые могут содержать слои. Многослойная процедура похожа на полиморфную с обработчиками для аргументов различного типа. Вместо этого многослойная процедура содержит реализации для различных слоев входных данных, обрабатывает все эти слои и порождает многослойный результат[1]. Например, если мы сочетаем числовой слой со слоем единиц измерения, процедура может обрабатывать числовые части своих аргументов с помощью числового слоя, а части, относящиеся к единицам измерения, — с помощью слоя единиц измерения.

В числовом примере, показанном нами в разделе 6.1, код процедуры `F`, вычисляющей Ньютонову силу, представляет собой эскиз, базовый план вычисления, которое необходимо произвести. Вычисление работает с числами, а единицы вычисления служат аннотациями для этих чисел. Многослойные полиморфные процедуры, реализующие арифметические операции вроде умножения, имеют базовую компоненту, работающую с числами на базовом слое, а также имеют и другие компоненты, по одной для каждого слоя данных, которыми базовый слой может быть проаннотирован. Слой единиц измерения — это слой аннотации, сообщающий дополнительную информацию о данных и о вычислении, но для сущности вычисления он не является необходимым.

В многослойной системе базовый слой должен быть способен работать, не обращаясь к другим слоям. Однако слоям аннотации может требоваться доступ к значениям из базового слоя. Если у аргумента функции отсутствует аннотация некоторого слоя, процедурная аннотация для этого слоя может использовать значение по умолчанию либо просто отказаться работать. Базовый слой работает в любом случае.

Для того чтобы построить многослойную процедуру, нужно уникальное имя `name`, арность `arity` и базовая процедура `base-procedure`, которая реализует основное вычисление.

```
(define (make-layered-procedure name arity base-procedure)
  (let* ((metadata
```

[1]Заметим, что реализация слоя для многослойной процедуры сама по себе может являться полиморфной процедурой. Точно так же обработчик полиморфной процедуры может быть многослойным.

```
        (make-layered-metadata name arity base-procedure))
       (procedure
        (layered-procedure-dispatcher metadata)))
  (set-layered-procedure-metadata! procedure metadata)
  procedure))
```

Информация о многослойной процедуре содержится в ее метаданных. Кроме того, метаданные управляют обработчиками для базового слоя и слоев аннотации.

Метаданные многослойной процедуры реализованы в виде пакета. При создании им сообщаются имя процедуры, арность и базовая процедура (обработчик базового слоя). К каждому из этих параметров метаданных можно обратиться. Кроме того, метаданные имеют операцию `set-handler!`, устанавливающую обработчик для некоторого слоя аннотации, и операцию `get-handler`, выдающую обработчик, соответствующий слою.

Каждый слой аннотации, скажем, слой единиц измерения, имеет операцию `get-procedure`, которая, имея имя процедуры и арность, выдает соответствующий обработчик для этой процедуры с этой арностью в этом слое. Операция `get-handler` у метаданных многослойной процедуры сначала проверяет, имеется ли у нее обработчик этого слоя. Если да, она возвращает этот обработчик; если нет, она обращается к `get-procedure` этого слоя.

```
(define (make-layered-metadata name arity base-procedure)
  (let ((handlers (make-weak-alist-store eqv?)))
    (define (get-name) name)
    (define (get-arity) arity)
    (define (get-base-procedure) base-procedure)
    (define has? (handlers 'has?))
    (define get (handlers 'get))
    (define set-handler! (handlers 'put!))
    (define (get-handler layer)
      (if (has? layer)
          (get layer)
          (layer 'get-procedure name arity)))
    (bundle layered-metadata?
            get-name get-arity get-base-procedure
            get-handler set-handler!)))
```

Сама работа по применению многослойной процедуры проделывается внутри `layered-procedure-dispatcher`.Диспетчер должен уметь найти и применить базовую процедуру и привязанные к ней процедуры слоев аннотации. Вся эта информация содержится в метаданных.

```
(define (layered-procedure-dispatcher metadata)
  (let ((base-procedure (metadata 'get-base-procedure)))
    (define (the-layered-procedure . args)
      (let ((base-value
             (apply base-procedure
                    (map base-layer-value args)))
            (annotation-layers
             (apply lset-union eqv?
```

```
                    (map (lambda (arg)
                           (if (layered-datum? arg)
                               (arg 'annotation-layers)
                               '()))
                         args))))
      (make-layered-datum base-value
        (filter-map        ; выкидывает все #f
         (lambda (layer)
           (let ((handler (metadata 'get-handler layer)))
             (and handler
                  (cons layer
                        (apply handler base-value args)))))
         annotation-layers))))
  the-layered-procedure))
```

При вызове многослойная процедура сначала вызывает базовую процедуру на значениях базового слоя аргументов и получает базовое значение результата. Затем, просматривая аргументы, она определяет, какие слои аннотации применимы; если нет ни одного слоя аннотации с обработчиком, процедура просто вернет базовое значение, поскольку таково будет значение `make-layered-datum` (стр. 266) в этом случае. Обработчику для конкретного слоя сообщаются уже вычисленное значение базового слоя и аргументы многослойной процедуры; ему не нужны значения никаких слоев, кроме собственных и базового. В общем случае результатом процедуры будет многослойное значение, содержащее базовое значение и результаты применимых обработчиков слоев аннотации.

Чтобы посмотреть, как все это работает на практике, давайте рассмотрим реализацию слоя единиц измерения (стр. 267). Обработчик `get-procedure` этого слоя (см. ниже) находит по имени процедуру, относящуюся к этому слою, если это имя — арифметическая операция; затем он вызывает эту процедуру с единицами измерения для каждого аргумента. (Особым случаем является `expt`, где второй аргумент не должен содержать единиц — это просто число.) Для остальных процедур обработчик единиц не определен, и `get-procedure` сообщает об этом, возвращая `#f`.

```
(define (get-procedure name arity)
  (if (operator? name)
      (let ((procedure (unit-procedure name)))
        (case name
          ((expt)
           (lambda (base-value base power)
             (procedure (get-value base)
                        (base-layer-value power))))
          (else
           (lambda (base-value . args)
             (apply procedure (map get-value args))))))
      #f))
```

Заметим, что, поскольку `get-procedure` — внутренняя процедура слоя единиц измерения, ей доступна `get-value`, унаследованная из процедуры `make-annotation-layer` (стр. 267). Процедуру `unit-procedure` мы увидим, когда будем обсуждать реализацию единиц измерения в разделе 6.3.1.

Рассмотрим пример. Вот простая процедура `square`, возводящая аргумент в квадрат:

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

Создадим многослойную версию этой процедуры, передав числовую версию базовому слою:

```
(define layered-square
  (make-layered-procedure 'square 1 square))
```

Эта многослойная процедура возведения в квадрат работает так же, как базовая версия:

```
(layered-square 4)
16

(layered-square 'm)
(* m m)
```

Однако, если в аргументе будет слой единиц, базовый слой и слой единиц обработаются по отдельности и будут содержаться в результате:

```
(pp (layered-square
     (layered-datum 'm
                    unit-layer (unit 'kilogram 1))))
#[layered-datum (* m m)]
(base-layer (* m m))
(unit-layer (unit kilogram 2))
```

6.3 Слоеная арифметика

Теперь мы знаем, как создавать многослойные процедуры, и можем добавить слои к арифметике. Нужно только построить ее, задав в базовой арифметике для каждой операции многослойную процедуру. Начнем с достаточно богатой арифметики:

```
(define (generic-symbolic)
  (let ((g (make-generic-arithmetic
            make-simple-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (extend-generic-arithmetic! g symbolic-extender)
    g))
```

и добавим к этой основе расширитель для обработки слоев:

```
(define generic-with-layers
  (let ((g (generic-symbolic)))
    (extend-generic-arithmetic! g layered-extender)
    g))
```

У этого расширителя довольно большая задача. Он создает многослойную арифметику, работающую с многослойными данными. Предикат области определения для многослойного расширения арифметики — `layered-datum?`. Базовым предикатом многослойных операций является предикат области определения нижележащей арифметики с дополнительным условием, что многослойные данные должны отвергаться[2]. Константы новой арифметики — константы базы, а для каждой арифметической операции запускается многослойная процедура, применимая, если хотя бы один из аргументов имеет слои, а ее базовая процедура наследуется из нижележащей арифметики.

```
(define (layered-extender base-arith)
  (let ((base-pred
         (conjoin (arithmetic-domain-predicate base-arith)
                  (complement layered-datum?))))
    (make-arithmetic (list 'layered
                           (arithmetic-name base-arith))
                     layered-datum?
                     (list base-arith)
      (lambda (name base-value)
        base-value)
      (lambda (operator base-operation)
        (make-operation operator
          (any-arg (operator-arity operator)
                   layered-datum?
                   base-pred)
          (make-layered-procedure operator
            (operator-arity operator)
            (operation-procedure base-operation)))))))
```

Здесь почти весь код устроен стандартно; в том числе то, что констант мы не трогаем, а для операций требуем, чтобы хотя бы один из аргументов имел слои. Единственный интересный момент содержится в последних трех строках, где процедура для операции базовой арифметики завертывается в многослойную процедуру. Как имя этой многослойной процедуры используется знак операции, так что если какой-то операции это потребуется, всякий слой может выделить ее в особый случай.

6.3.1 Арифметика единиц измерения

Нам нужна арифметика единиц измерения для слоя аннотаций единиц нашей арифметики. Спецификация единиц содержит именованные базовые единицы и при каждой базовой единице степень[3]. В арифметике единиц произведение двух спецификаций — это новая спецификация единиц, где степень при каждой базовой единице равна сумме степеней при соответствующих базовых степенях аргументов.

[2]Процедуры `conjoin` и `complement` — это комбинаторы для предикатов: `conjoin` создает новый предикат, применяющий к аргументам булеву операцию `and`, а `complement` создает предикат, являющийся отрицанием аргумента.

[3]Внимание! «Базовые единицы» не имеют отношения к базовому слою в нашей системе многослойных данных. Система единиц строится на нескольких базовых единицах, например килограммах, метрах и секундах. Существуют также производные единицы, например ньютон, который является комбинацией базовых единиц: $1\ \text{Н} = 1\ \text{кг} \cdot \text{м} \cdot \text{с}^{-2}$.

```
(unit:* (unit 'kilogram 1 'meter 1 'second -1)
        (unit 'second -1))
(unit kilogram 1 meter 1 second -2)
```

Мы предполагаем, что базовые единицы называются просто символами, например `kilogram`.

Представление спецификаций единиц

Чтобы спецификацию единиц было легче создавать, ее внешнее представление мы делаем списком свойств (с поочередно идущими ключами и значениями) из имен единиц и степеней.

Однако внутри удобнее представлять спецификацию единиц как a-список; таким образом, нужно преобразовать исходный список свойств во внутреннее представление с помощью `plist->alist`. A-списки мы держим отсортированными по имени базовой единицы. При преобразовании проводится некоторая проверка на ошибки. Список аргументов вызова `unit` должен иметь форму списка свойств. Степень, связанная с именем каждой базовой единицы, должна быть точным рациональным числом (обычно целым). Если имя базовой единицы повторяется, это ошибка. Сортировка по именам базовых единиц пожалуется, если какое-то из их имен будет не символом.

```
(define (unit . plist)
  (guarantee plist? plist 'unit)
  (let ((alist
         (sort (plist->alist plist)
               (lambda (p1 p2)
                 (symbol<? (car p1) (car p2))))))
    (if (sorted-alist-repeated-key? alist)
        (error "Базовая единица повторяется" plist))
    (for-each (lambda (p)
                (guarantee exact-rational? (cdr p)))
              alist)
    (alist->unit alist)))

(define (sorted-alist-repeated-key? alist)
  (and (pair? alist)
       (pair? (cdr alist))
       (or (eq? (caar alist) (caadr alist))
           (sorted-alist-repeated-key? (cdr alist)))))
```

Процедура `alist->unit` всего лишь прикрепляет к a-списку уникальную метку, а `unit->alist` извлекает a-список из спецификации единиц:

```
(define (alist->unit alist)
  (cons %unit-tag alist))

(define (unit->alist unit)
  (guarantee unit? unit 'unit->alist)
  (cdr unit))
```

Значением `%unit-tag` является уникальный символ, с помощью которого мы помечаем а-список спецификации единиц. Чтобы распечатка спецификации единиц выглядела как список свойств, который мы подаем процедуре `unit` при ее создании, мы настраиваем процедуру печати Scheme так, чтобы она распечатывала спецификации единиц в форме списков свойств. Эта волшебная настройка (мы ее здесь не приводим) срабатывает, когда в начале списка стоит метка спецификации единиц.

Предикат `unit?` истинен, если его аргумент является правильно построенной спецификацией единиц:

```
(define (unit? object)
  (and (pair? object)
       (eq? (car object) %unit-tag)
       (list? (cdr object))
       (every (lambda (elt)
                (and (pair? elt)
                     (symbol? (car elt))
                     (exact-rational? (cdr elt))))
              (cdr object))))
```

Арифметические операции над единицами

Арифметику над единицами мы строим как отображение имен операций на процедуры, реализующие желаемое поведение. Обыкновенные числа, например π, единиц не имеют. При умножении величины с единицами на число без единиц получаются те же единицы, что и у исходной величины. Поэтому в арифметике единиц требуется нейтральный элемент для умножения на числа без единиц; это `unit:none`. Процедура `simple-operation` объединяет символ операции, проверку на применимость и процедуру, реализующую эту операцию:

```
(define (unit-arithmetic)
  (make-arithmetic 'unit unit? '()
    (lambda (name)
      (if (eq? name 'multiplicative-identity)
          unit:none
          (default-object)))
    (lambda (operator)
      (simple-operation operator
                        unit?
                        (unit-procedure operator)))))
```

Для получения соответствующей процедуры по символу операции зовется `unit-procedure`:

```
(define (unit-procedure operator)
  (case operator
    ((*) unit:*)
     ((/) unit:/)
     ((remainder) unit:remainder)
     ((expt) unit:expt)
     ((invert) unit:invert)
```

```
    ((square) unit:square)
    ((sqrt) unit:sqrt)
    ((atan) unit:atan)
    ((abs ceiling floor negate round truncate)
     unit:simple-unary-operation)
    ((+ - max min)
     unit:simple-binary-operation)
    ((acos asin cos exp log sin tan)
     unit:unitless-operation)
    ((angle imag-part magnitude make-polar make-rectangular
            real-part)
     ;; первое приближение:
     unit:unitless-operation)
    (else
     (if (eq? 'boolean (operator-codomain operator))
         (if (n:= 1 (operator-arity operator))
             unit:unary-comparison
             unit:binary-comparison)
         unit:unitless-operation))))
```

Для каждого из перечисленных выше случаев нужно предоставить соответствующую операцию. Например, чтобы умножить две спецификации единиц, нужно сложить одноименные степени и убрать базовые единицы с нулевой степенью:

```
(define (unit:* u1 u2)
  (alist->unit
   (let loop ((u1 (unit->alist u1)) (u2 (unit->alist u2)))
     (if (and (pair? u1) (pair? u2))
         (let ((factor1 (car u1)) (factor2 (car u2)))
           (if (eq? (car factor1) (car factor2)) ; same unit
               (let ((n (n:+ (cdr factor1) (cdr factor2))))
                 (if (n:= 0 n)
                     (loop (cdr u1) (cdr u2))
                     (cons (cons (car factor1) n)
                           (loop (cdr u1) (cdr u2)))))
               (if (symbol<? (car factor1) (car factor2))
                   (cons factor1 (loop (cdr u1) u2))
                   (cons factor2 (loop u1 (cdr u2))))))
         (if (pair? u1) u1 u2)))))
```

Некоторые операции, такие как `remainder`, `expt`, `invert`, `square`, `sqrt` и `atan`, требуют отдельной обработки. Остальные разбиваются на несколько простых классов. Простые одноместные операции, например `negate`, просто переносят единицы своего аргумента на результат:

```
(define (unit:simple-unary-operation u)
  u)
```

Некоторые операции, например сложение, проверяют, что мы не «складываем яблоки с апельсинами»:

```
(define (unit:simple-binary-operation u1 u2)
  (if (not (unit=? u1 u2))
      (error "несовместимые единицы:" u1 u2))
      u1)
```

Упражнение 6.1. Производные единицы

Хотя процедуры вычисления единиц, приведенные выше, правильны и достаточно полны, использовать их неудобно. Например, спецификация единиц для кинетической энергии (показанная на стр. 273) такова:

```
(unit kilogram 1 meter 2 second -2)
```

В терминах базовых единиц (килограмм, метр, секунда) Международной Системы (СИ) все верно, но было бы намного приятнее выразить это через джоули, производную единицу энергии СИ:

```
(unit joule 1)
```

Полная система базовых единиц СИ такова: килограмм, метр, секунда, ампер, кельвин, моль, кандела, и существует также одобренный список производных единиц. Например:

- ньютон = килограмм · метр · секунда$^{-2}$;
- джоуль = ньютон · метр;
- кулон = ампер · секунда;
- ватт = джоуль · секунда$^{-1}$;
- вольт = ватт · ампер$^{-1}$;
- ом = вольт · ампер$^{-1}$;
- сименс = ом$^{-1}$;
- фарад = кулон · вольт$^{-1}$;
- вебер = вольт · секунда;
- генри = вебер · ампер$^{-1}$;
- герц = секунда$^{-1}$;
- тесла = вебер · метр$^{-2}$;
- паскаль = ньютон · метр$^{-2}$.

a. Напишите процедуру, принимающую описание единицы в терминах базовых единиц СИ и выдающую, если возможно, более простое описание через производные единицы.

b. Выражение через производные единицы не единственно — может существовать множество эквивалентных описаний. Это похоже на задачу упрощения алгебраических выражений, но критерий «простоты» тут не очевиден. Постройте нравящуюся вам версию такого критерия и объясните, за что она вам нравится.

c. Полезно также уметь использовать стандартные сокращения и приставки для кратных и дольных единиц. Например, 1 мА — удобный способ записи для 0.001 А, или 1/1000 ампера. Спроектируйте и реализуйте простую расширяемую систему, которая позволит использовать эти варианты записи как при вводе, так и при выводе. Однако помните, что «синтаксический сахар вызывает рак точки с запятой».

6.4 Отслеживание зависимостей между значениями

Один из видов аннотаций, которые программист может захотеть ввести в некоторых частях своей программы, — отслеживание зависимостей. Каждый элемент данных (или процедура) откуда-то появились. Либо они попали в вычисление в качестве предпосылки, для которой можно пометить ее внешнее происхождение, либо это результат обработки других данных. Можно добавить к элементарным операциям системы слои аннотации, которые при обработке данных, имеющих обоснования, пометят соответствующими обоснованиями результат вычислений.

Обоснования могут иметь различную степень подробности. Простейшая их разновидность — просто множество посылок, участвовавших в вычислении нового элемента данных. Процедура вроде сложения может породить сумму и обосновать ее объединением множеств, обосновывающих поданные на вход слагаемые. Умножение работает похоже, но нулевой множитель тут достаточен, чтобы заключить, что произведение равно нулю, поэтому обоснования других множителей незачем включать в обоснование произведения.

Такие простые обоснования можно вычислять и хранить вместе с данными всего лишь с приблизительно линейным удорожанием процесса, но они могут оказаться бесценными при отладке сложных процессов и при присвоении награды или штрафа за результат вычисления. Этого уже достаточно для поддержки перебора с возвратами, управляемого зависимостями (см. раздел 7.5).

Данные, полученные извне, можно аннотировать *посылкой*, указывающей на их происхождение. В более общем случае всякое значение данных можно аннотировать множеством посылок, которое называется его *множеством поддержки*. Часто множество поддержки, аннотирующее элемент данных, называют просто его *поддержкой*. Когда в процедуру, знающую о поддержке данных, передается несколько аргументов, она должна скомбинировать множества поддержки аргументов и аннотировать свой результат.

Управление множествами поддержки — очевидное приложение для нашего механизма многослойных данных. Мы добавляем слой поддержки к нашей полиморфной арифметике для обработки множеств поддержки. Этот слой может работать одновременно с другими, например со слоем единиц измерения. Таким образом, это аддитивная функциональность.

На стр. 271 мы построили арифметику, поддерживающую многослойные данные и процедуры:

```
(define generic-with-layers
  (let ((g (generic-symbolic)))
    (extend-generic-arithmetic! g layered-extender)
    g))

(install-arithmetic! generic-with-layers)
```

Указывать, какие слои поддерживаются в `layered-extender`, не нужно, потому что эта процедура автоматически использует слои, упомянутые в аргументах многослойной процедуры. Скажем, если процедура `+` вызывается с аргументами, имеющими единицы измерения, то и результат тоже будет иметь единицы измерения. Но если ни у одного из аргументов единиц нет, не будет их и в результате, а процедура сложения единиц не будет вызвана. Подобным же образом, если у аргументов есть поддержка, результат тоже будет иметь поддержку. Но если у аргументов нет под-

держки, результат тоже будет без поддержки, а процедура сложения для множеств поддержки вызвана не будет.

Например, можно определить кинетическую энергию частицы с массой `m` и скоростью `v`:

```
(define (KE m v)
  (* 1/2 m (square v)))
```

Посмотрим на результат вычисления кинетической энергии для некоторых аргументов:

```
(pp (KE (layered-datum 'm
                       unit-layer (unit 'kilogram 1)
                       support-layer (support-set 'cph))
        (layered-datum 'v
                       unit-layer (unit 'meter 1 'second -1)
                       support-layer (support-set 'gjs))))
#[layered-datum (* (* 1/2 m) (square v))]
(base-layer (* (* 1/2 m) (square v)))
(unit-layer (unit kilogram 1 meter 2 second -2))
(support-layer (support-set gjs cph))
```

Каждому аргументу мы даем аннотации слоя единиц измерения и аннотации слоя поддержки. Для слоя поддержки это набор посылок (множество поддержки). Здесь каждый аргумент поддерживается единственной посылкой, соответственно, `cph` и `gjs`. В результате получаем многослойный объект с тремя слоями: базовый слой полиморфной арифметики, где содержится нужное алгебраическое выражение; правильное описание единиц измерения; множество поддержки, содержащее имена посылок, повлиявших на значение.

Мы получили такое поведение `KE`, не указывая для нее никакой явной поддержки. В более общем случае мы можем такую поддержку добавить. Например, можно указать, что `KE` поддерживается посылкой `KineticEnergy-classical`. В таком случае, если какая-то сложная процедура выдаст нам результат, который кажется неправильным, можно будет посмотреть, какие процедуры повлияли на вычисление неверного результата, а не только какие числовые либо символические входные данные были использованы. Эту задачу мы рассмотрим в упражнении 6.2.

Необязательно все посылки, упомянутые в аргументах вычисления, появятся в результате. Например, если один из множителей равен нулю, этого достаточно, чтобы нулю было равно произведение независимо от остальных конечных множителей. В этом можно убедиться, задав нулевую массу:

```
(pp (KE (layered-datum 0
         unit-layer (unit 'kilogram 1)
         support-layer (support-set 'jems))
        (layered-datum 'v
         unit-layer (unit 'meter 1 'second -1)
         support-layer (support-set 'gjs))))
#[layered-datum 0]
(base-layer 0)
(unit-layer (unit kilogram 1 meter 2 second -2))
(support-layer (support-set jems))
```

Здесь поддержка того, что числовой результат равен нулю, — это только поддержка, заданная для нулевого значения массы.

6.4.1 Слой поддержки

Рассмотрим теперь, как реализуется слой поддержки. Он несколько отличается от слоя единиц измерения, поскольку единицы можно сочетать между собой независимо от ссылок на базовый слой, а слой поддержки в некоторых операциях должен смотреть на базовый слой.

Слой поддержки кое в чем проще слоя единиц измерения, поскольку все арифметические операции, кроме трех, используют поведение по умолчанию: множество поддержки результата равно объединению множеств поддержки аргументов.

```
(define support-layer
  (make-annotation-layer 'support
    (lambda (get-name has-value? get-value)
      (define (get-default-value)
        (support-set))
      (define (get-procedure name arity)
        (case name
          ((*) support:*)
          ((/) support:/)
          ((atan2) support:atan2)
          (else support:default-procedure)))
      (bundle layer?
              get-name has-value? get-value
              get-default-value get-procedure))))

(define support-layer-value
  (layer-accessor support-layer))

(define (support:default-procedure base-value . args)
  (apply support-set-union (map support-layer-value args)))
```

Первый интересный случай — умножение. Слой поддержки должен посмотреть на значения базовых арифметических аргументов при вычислении поддержки. Если какой-то аргумент равен нулю, то поддержка результата равна поддержке только этого аргумента.

```
(define (support:* base-value arg1 arg2)
  (let ((v1 (base-layer-value arg1))
        (v2 (base-layer-value arg2))
        (s1 (support-layer-value arg1))
        (s2 (support-layer-value arg2)))
    (if (exact-zero? v1)
        (if (exact-zero? v2)
            (if (< (length (support-set-elements s1))
                   (length (support-set-elements s2)))
                s1
                s2)   ;произвольный выбор
```

```
          s1)
       (if (exact-zero? v2)
           s2
           (support-set-union s1 s2)))))
```

Деление (и арктангенс, который здесь не показан) также должно проверить базовый слой и особо обработать нулевые аргументы. Если делимое равно нулю, этого достаточно, чтобы поддержать нулевое частное в результате. Делитель никогда нулем не будет, поскольку в этом случае вычисление базового слоя просигнализирует об ошибке, и этот код не будет запущен.

```
(define (support:/ base-value arg1 arg2)
  (let ((v1 (base-layer-value arg1))
        (s1 (support-layer-value arg1))
        (s2 (support-layer-value arg2)))
    (if (exact-zero? v1)
        s1
        (support-set-union s1 s2))))
```

Эти оптимизации для `*` и `/` имеют смысл только тогда, когда мы можем доказать, что аргумент действительно равен нулю, а не некоторому неупрощенному символическому выражению. (Но если выражение упрощается до точного нуля, это мы можем использовать!)

```
(define (exact-zero? x)
  (and (n:number? x) (exact? x) (n:zero? x)))
```

Абстракция множества поддержки реализуется как список, начинающийся с символа `support-set`:

```
(define (%make-support-set elements)
  (cons 'support-set elements))

(define (support-set? object)
  (and (pair? object)
       (eq? 'support-set (car object))
       (list? (cdr object))))

(define (support-set-elements support-set)
  (cdr support-set))
```

Для завершения абстракции требуется еще несколько вспомогательных процедур:

```
(define (make-support-set elements)
  (if (null? elements)
      %empty-support-set
      (%make-support-set (delete-duplicates elements))))

(define (support-set . elements)
  (if (null? elements)
      %empty-support-set
```

```
      (%make-support-set (delete-duplicates elements))))

(define %empty-support-set
  (%make-support-set '()))

(define (support-set-empty? s)
  (null? (support-set-elements s)))
```

Нужно вычислять объединение множеств поддержки и добавлять к ним новые элементы. Поскольку мы решили хранить элементы в виде списка, можно использовать библиотеку `lset` языка Scheme[4]:

```
(define (support-set-union . sets)
  (make-support-set
   (apply lset-union eqv?
          (map support-set-elements sets))))

(define (support-set-adjoin set . elts)
  (make-support-set
   (apply lset-adjoin eqv? (support-set-elements set) elts)))
```

Упражнение 6.2. Процедурная ответственность
Слой поддержки, основанный на арифметике, — это очень низкоуровневый инструмент. Каждая элементарная арифметическая операция обращает внимание на поддержку, и способа отключить эту работу нет. Требуется ввести средства абстракции. Допустим, например, что у нас есть процедура, численно вычисляющая определенный интеграл функции. Единицы измерения для значения интеграла — это произведение единиц измерения значений интегрируемой функции и единиц измерения пределов интегрирования. (Единицы измерения верхнего и нижнего пределов должны быть одни и те же!) Однако протаскивать вычисление единиц сквозь все арифметические детали вычислительного процесса интегрирования не самая хорошая идея. Нужна возможность аннотировать процедуру интегрирования так, чтобы у результата были правильные единицы без того, чтобы требовать от каждого внутреннего сложения и умножения работы с многослойными данными.

a. Добавьте возможность разрешать составным процедурам, построенным из элементарных арифметических процедур (или чего-то другого), расширять множество поддержки своих результатов добавочной посылкой (вроде «сделано Джорджем»).

b. Позвольте составным процедурам выполняться так, чтобы их тела были спрятаны от слоя поддержки. Так, например, доверенная библиотечная процедура может помечать свой результат соответствующим множеством поддержки, но операции в ее теле при этом не будут нести бремя вычисления поддержки промежуточных результатов.

c. Слой поддержки организован вокруг операций арифметической подсистемы. Однако иногда бывает полезно различать различные вхождения одной и той же операции. Например, при работе с точностью численных вычислений будет не слишком полезно сказать, что потеря точности произошла из-за вычитания двух почти одинаковых величин. Правильнее было бы указать, какое именно вычитание привело к потере. Возможно ли добавить к слою поддержки способность различать экземпляры операций?

[4]Если множества поддержки станут большими, их можно представить намного более эффективно, но здесь мы работаем только с маленькими множествами.

Упражнение 6.3. Параноидальное программирование
Иногда мы бываем не полностью уверены, что библиотечная процедура делает именно то, что нам нужно. В таких случаях разумным шагом будет «обернуть» эту процедуру тестом, проверяющим результат. Например, пусть у нас есть программа `solve`, принимающая на вход множество уравнений и множество неизвестных, встречающихся в этих уравнениях, и порождающая множество подстановок для неизвестных, которые удовлетворяют уравнениям. Можно обернуть программу `solve` процедурой, проверяющей, что в результате подстановки в исходные уравнения действительно получаются тавтологии. Однако эту обертку мы не хотим делать частью эскизного проекта. Как такую возможность реализовать в виде слоя? Объясните свой проект и реализуйте его.

Упражнение 6.4. IDE для многослойных программ

Это упражнение представляет собой большой проект: изобретение и реализация IDE (интегрированной среды разработки) для многослойных систем.

Идея многослойных программ, которые используют многослойные данные и процедуры, весьма перспективна. Цель состоит в том, чтобы была возможность аннотировать программы полезными и исполнимыми метаданными — такими, как объявления типов, контрольные утверждения, единицы измерения и множества поддержки, — без замусоривания текста основной программы. Однако текст программы должен быть связан с текстом аннотаций, и при редактировании текста основной программы должны редактироваться и аннотации. Предположим, например, что нужно отредактировать текст базового слоя в некоторой многослойной процедуре. В слоях может содержаться информация вроде объявления типов или то, как процедура работает с единицами измерения и множествами поддержки. Было бы здорово, если бы редактор, когда это необходимо, показывал эти слои и как они связаны с текстом основной программы. Возможно, редактирование текста основной программы должно вести к изменениям в тексте слоев аннотации. Иногда это можно проделывать автоматически, но часто добавочные слои должен редактировать программист.

a. Представьте, как по-вашему должна работать IDE, поддерживающая разработку многослойных систем. Что должно отображаться на экране? Как поддерживать редактируемые фрагменты программ в согласованном состоянии?

b. Мощную структуру, на основе которой можно построить такую IDE, предоставляет Emacs. В этом редакторе поддерживаются множественные окна и отдельные режимы редактирования для каждого окна. Имеется синтаксическая поддержка для множества компьютерных языков, включая Scheme. Есть подсистемы вроде режима `org`, где работа идет с уровневой структурой документа. Можно ли расширить эту среду для поддержки многослойного программирования? Сделайте предварительный набросок, как построить искомую IDE при помощи Emacs.

c. Постройте небольшой, но способный к расширению прототип на основе Emacs. Попробуйте с ним поработать. С какими проблемами вы столкнулись? Был ли Emacs хорошей отправной точкой? Если нет, почему? Опишите свой опыт.

d. Если прототип оказался удачным, расширьте его до полноценной системы и сделайте эту систему загружаемой библиотекой в Emacs, чтобы мы все могли ее использовать.

6.4.2 Хранение обоснований

В более сложных обоснованиях можно записывать конкретные операции, примененные для получения данных. Обоснования такого рода могут содержать полные

объяснения (доказательства); однако они неизбежно занимают много места — потенциально рост здесь линейный по количеству выполненных операций. Однако в некоторых случаях имеет смысл присоединять к данным полную историю, описывающую их происхождение, чтобы какой-то последующий процесс использовал ее для каких-то целей или оценивал правильность вывода во время отладки[5].

Во многих случаях, например в юридических аргументах, требуется знать родословную данных: где они получены, как получены, кто их получил, кто разрешил сбор данных и т. д. Подробный вывод фрагмента данных, включающий родословную каждого исходного факта, может иметь решающее значение при определении, принимается ли свидетельство в суде.

Один из способов сделать это — символьная арифметика, которую мы построили в разделе 3.1. В сущности, если символьная арифметика используется как слой поверх числовой арифметики, каждое числовое значение будет снабжено своим полным выводом. Аннотации символьной арифметики могут быть очень дорогими, поскольку символическое выражение для каждого применения арифметической операции включает в себя символические выражения для всех входов. Однако, поскольку для каждого входа нам нужен только указатель, часто стоимость такой аннотации по времени и занимаемой памяти оказывается приемлемой[6]. Поэтому такого рода обоснования можно добавить, когда требуется полное объяснение работы или даже временно — для поиска трудноуловимой ошибки.

Упражнение 6.5. Обоснования
Кратко перечислите вопросы, возникающие при построении обоснований данных. Заметим, что обоснование каждого значения зависит от значений, из которых оно получено, а также от способа, которым эти исходные значения были скомбинированы. Что делать, если обоснованием служит взвешенная комбинация многих множителей, как в глубокой нейронной сети? Это исследовательская задача, на которую следует обратить внимание, если мы хотим сделать системы, от которых зависим, ответственными за решения.

6.5 Что обещает многослойность

Мы лишь поверхностно коснулись возможностей, которые предоставляет простой и удобный механизм построения слоев данных и программ. Это открытая область для исследований. Развитие систем, поддерживающих многослойную работу, может иметь громадные последствия в будущем.

Анализ чувствительности — важная задача, которую можно решить с помощью аннотированных данных и многослойных процедур. Например, в механике, когда мы строим историю развития системы дифференциальных уравнений из некоторого начального состояния, часто желательно бывает понимать, как изменяется семейство

[5] В замечательной книге Патрика Саппса «Введение в логику» [118] доказательства записываются в четыре колонки. Это метка строки, утверждение, производимое в этой строке, правило, по которому это утверждение выведено из предыдущих строк, и набор посылок, поддерживающих его. Эта структура доказательства послужила вдохновением для нашего представления обоснований и множеств поддержки.

[6] Это не совсем так. Проблема в том, что вычисление численной операции может не влечь за собой существенной цены по памяти, а создание символьного выражения, сколь угодно малого, требует обращения к памяти. Время доступа к памяти громадно по сравнению со временем арифметической операции, производимой над регистрами процессора. Увы...

траекторий, окружающих опорную. Обычно это проделывается путем интегрирования вариационной системы наряду с опорной траекторией. Подобным образом в некоторых видах анализа можно хранить распределение вероятностей вокруг номинального значения наряду с самим номинальным значением. Этого можно достичь, если снабдить значения аннотациями в виде распределений и добавить к операциям слой процедур, комбинирующих эти распределения на основе номинальных значений, возможно, производя некоторый байесовский анализ. Разумеется, качественно реализовать эту идею — нелегкая задача.

Связанная с этим, но еще более захватывающая идея — пертурбационное программирование. По аналогии с примером про дифференциальные уравнения, может быть, мы можем запрограммировать символическую систему так, чтобы наряду с опорной траекторией хранилась «трубка» вариаций, что позволит нам рассматривать небольшие вариации запроса. Рассмотрим, например, задачу поиска. Дан набор ключевых слов, и система производит некоторые волшебные действия, дающие в результате список документов, относящихся к этим ключевым словам. Допустим теперь, что мы изменяем одно из слов запроса. Насколько чувствителен поиск к этому слову? Или, что более важно, можно ли заново использовать часть работы, проделанной ранее по сходному запросу? Ответа на эти вопросы мы не знаем, но если это возможно, нам хотелось бы выразить методы решения через пертурбационную программу некоторого рода, построенную как дополнительный слой в основной программе.

Зависимости смягчают несогласованность

Аннотации о зависимостях в данных могут служить мощным механизмом для организации человекоподобных вычислений. Например, у всякого человека набор верований внутренне противоречив: разумный человек может быть привержен научному методу, но при этом сохранять прочные привязанности к некоторым суевериям и ритуалам; можно верить в святость человеческой жизни и в то же время считать, что в некоторых случаях оправдана высшая мера наказания. Если бы мы на самом деле были логиками, такая несогласованность была бы смертельна: если кто-то одновременно верит в высказывания P и НЕ-P, он должен верить в любое высказывание! Однако каким-то образом нам удается избегать того, чтобы противоречивые мнения отменяли всякую полезную мысль. Наши персональные системы верований кажутся локально согласованными, т. е. никаких видимых противоречий в них нет. Если же мы замечаем противоречие, наше мышление не терпит немедленную катастрофу; мы всего лишь чувствуем некоторую неловкость или даже смеемся над собой.

К каждому высказыванию можно приписать множество поддерживающих его предположений и позволить логическим выводам зависеть от этих предположений. Если встречается противоречие, процесс может определить конкретный «нехороший набор» противоречивых предположений. Система может «посмеяться» и решить, что никакой вывод, основанный на надмножестве этих предположений, не достоин веры. Этот процесс, перебор с возвратами, управляемый зависимостями, может использоваться для оптимизации сложного поиска и позволить ему использовать собственные ошибки наилучшим образом. Однако сама идея позволить процессу одновременно хранить мнения, основанные на взаимно противоречивом наборе посылок, избегая при этом логической ямы, вполне революционна.

Ограничения на использование данных

С данными часто бывают связаны ограничения, указывающие, как эти данные разрешается использовать. Ограничения могут диктоваться законом, контрактными обязательствами, обычаем или соображениями приличия. Некоторые ограничения служат для управления распространением данных, другие продиктованы стремлением ограничить последствия действий, зависящих от обладания данными.

Разрешение использовать данные может быть дополнительно сужено источником сообщения: «Я говорю, что эта информация конфиденциальна. Вам не разрешается ее использовать в целях конкуренции со мной и не разрешается разглашать ее моим конкурентам.» Получатель сообщения также может вводить ограничения: «Я не хочу знать об этом ничего такого, что я не смогу сказать своей супруге».

Хотя в процессе передачи данных от одного человека либо организации к другому подробности могут быть достаточно хитроумными, ограничения на способы использования данных часто изменяются способами, которые можно выразить в виде алгебраических выражений. Эти выражения описывают, как ограничения на использование некоторого элемента данных можно вывести исходя из истории его передачи между агентами: условия, которые добавляются и убираются на каждом шаге. Когда часть одного набора данных сливается с частью другого набора, ограничения на то, как можно использовать эти поднаборы, и на то, как их можно сочетать между собой, определяют ограничения на результат сочетания. Формализацией этого процесса служит описание в *алгебре назначений данных*.

Слои алгебры назначений данных могут помочь в создании систем, отслеживающих распространение и использование чувствительных данных, чтобы позволить проводить ревизию и предотвратить неверное использование данных. Однако эта разновидность приложений намного шире, чем всего лишь простой вопрос послойной организации. Их можно сделать эффективными, лишь обеспечив безопасность процесса, предотвратив утечку через неконтролируемые каналы и компрометацию отслеживающих слоев. Здесь еще требуется немало исследовательской работы.

Глава 7

Распространение информации

Десятилетия программистского опыта сказались на нашем коллективном воображении. Мы происходим из культуры, привычной к скудности, где вычисления и память были дороги, а параллельные вычисления было трудно организовать и трудно управлять ими. Теперь все уже не так. Но наши языки, алгоритмы и архитектурные идеи основаны на этих предположениях. Наши языки основаны на последовательном направленном исполнении — даже функциональные языки предполагают, что вычисление строится вокруг значений, передающихся по деревьям выражений. Многонаправленные ограничения в функциональных языках выражаются с трудом.

Побег из смирительной рубашки фон Неймана

Модель вычисления на основе распространителей [99] предлагает один из способов избавиться от этих ограничений. Эта модель строится на идее, что основные вычислительные элементы представляют собой распространители, автономные независимые машины, связанные разделяемыми ячейками, через которые они общаются. Каждая машина-распространитель непрерывно рассматривает ячейки, с которыми она связана, и добавляет в некоторые ячейки информацию, основанную на вычислениях, которые она производит, исходя из информации, полученной из других ячеек. Ячейки собирают в себе информацию, а распространители порождают ее.

Поскольку инфраструктура распространителей основана на передаче информации между связанными между собой независимыми машинами, структуры распространителей лучше выражаются диаграммами связи, чем деревьями выражений. В такой системе частичные результаты полезны, несмотря на свою неполноту. Например, обычный способ вычисления квадратного корня — последовательные приближения методом Герона. В традиционном программировании результат вычисления квадратного корня не становится доступен для последующих действий, пока он не получен с достаточной точностью. Напротив, в аналоговой цепи, выполняющей ту же самую функцию, частичные результаты могут использоваться следующими стадиями как первые приближения в их собственных вычислениях. Это не вопрос аналогового или цифрового представления, а вопрос организации. В механизме распространителей частичными результатами цифрового процесса можно пользоваться, не ожидая окончательного результата.

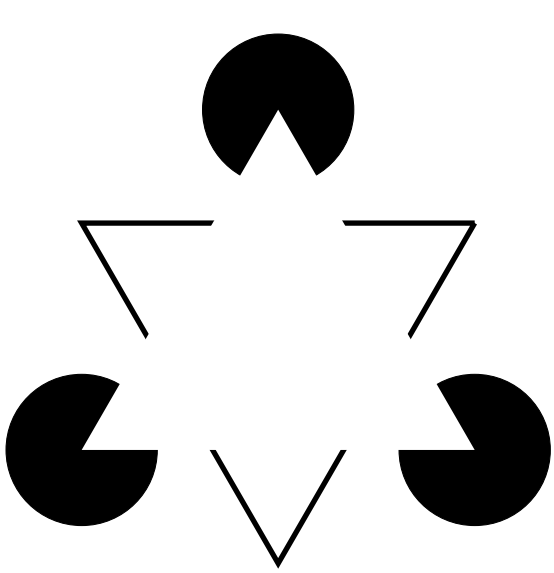


Рис. 7.1. Треугольник Канижи — классический пример дополняющей иллюзии. Белого треугольника тут нет!

Заполнение деталей

Такую вычислительную структуру естественно использовать для построения систем, заполняющих детали картины. Эта структура аддитивна: новые способы получения информации добавляются просто как новые фрагменты сети: либо простые распространители, либо целые подсети. К примеру, если неизвестная величина выражается как интервал, новый способ вычисления его верхней границы можно добавить, не трогая остальные части системы.

Заполнение деталей играет важную роль в том, как мы используем информацию. Рассмотрим, например, треугольник Канижи (7.1). На основе нескольких фрагментарных деталей мы видим белый треугольник (на белом фоне!), хотя его там нет (как правило, его описывают более ярким, чем фон). Мы дополнили отсутствующие детали предполагаемой фигуры. При восприятии речи на слух мы также дополняем детали, исходя из наблюдаемого контекста, используя закономерности фонологии, морфологии, синтаксиса и семантики. Опытный инженер-электронщик при виде частичной схематической диаграммы дополняет ее деталями, так что получается осмысленный механизм. Это дополнение деталей не является последовательным процессом; оно продвигается каждый раз, когда из окружающих признаков можно сделать локальный вывод. Выводы могут подталкивать друг друга, так что, если оказывается заполнен некоторый фрагмент, он может послужить новым признаком для продолжающегося процесса заполнения.

Зависимости и перебор с возвратами

При помощи слоев мы естественным и эффективным образом можем ввести в структуру распространителей зависимости. Это позволяет системе отслеживать и хранить информацию о родословной каждого значения. Через эту родословную можно построить непротиворечивое объяснение, откуда получилось значение, указав источ-

ники и правила сочетания исходного материала. Это особенно важно, когда имеется несколько источников, каждый из которых сообщает о значении частичную информацию. Отслеживание зависимостей служит также средой для отладки (а может быть, и для интроспективной самоотладки).

Помимо базовых верований, можно, используя машины на основе `amb`, вводить гипотетические мнения. Они дают нам альтернативные значения, основанные на посылках, которые можно при желании безболезненно отбросить. В отличие от систем, моделируемых в языках на основе выражений вроде Lisp, здесь не возникает никакого излишнего потока управления, который основан на структуре выражений, загрязняет зависимости и требует при откате дорогого перепостроения уже однажды вычисленных результатов.

Дублирование, избыточность и параллелизм

Модель распространителей содержит механизмы, поддерживающие включение избыточных (в сущности, дублирующих) подсистем, так что одна задача может решаться многими различными способами. Многократно дублированные системы могут эффективно противостоять атакам: если нет единого потока исполнения, который можно было бы подавить, атака, ломающая или задерживающая один из путей, не помешает вычислению, поскольку продолжает работать альтернативный путь. Дублированные и избыточные параллельные вычисления поддерживают целостность и отказоустойчивость: вычисления, проведенные многими альтернативными путями, можно проверить на непротиворечивость. Задача разрушения параллельных вычислений усложняется из-за инвариантов, проверяемых между различными потоками.

Модель распространителей в основе своей параллельна, распределена и масштабируема. В ней присутствует строгая изоляция и встроено предположение о параллельности вычислений. Существует множество независимых распространителей, они вычисляют и дополняют информацию в разделяемых между собой ячейках, где информация сливается, а противоречия обнаруживаются и обрабатываются.

7.1 Пример: расстояния до звезд

Рассмотрим астрономическую задачу, определение расстояний до звезд. Это сложная задача, поскольку расстояния чрезвычайно велики. Даже в случае ближайших звезд, где можно измерить параллакс, используя радиус орбиты Земли как базис, разница угловой позиции звезды составляет малые доли угловой секунды. В самом деле, единицей длины для звездных расстояний служит *парсек*, высота треугольника с орбитой Земли в качестве основания, угол вершины которого равен 2 секундам. Параллакс измеряют, наблюдая изменение позиции звезды относительно фона по мере вращения Земли вокруг Солнца (см. рис. 7.2).

Определим распространитель, строящий отношение между параллаксом звезды в радианах и расстоянием до этой звезды в парсеках:

```
(define-c:prop (c:parallax<->distance parallax distance)
  (let-cells (t (AU AU-in-parsecs))
    (c:tan parallax t)
    (c:* t distance AU)))
```

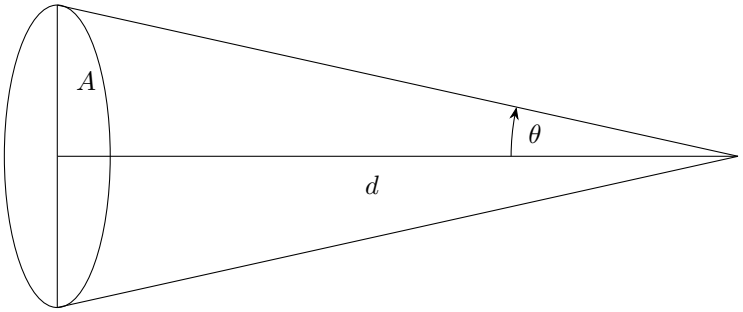


Рис. 7.2. Угол θ треугольника с вершиной в далекой звезде, основанного на половине большой полуоси орбиты Земли вокруг Солнца, называется параллаксом звезды. Заметим, что $A/d = \mathrm{tg}(\theta)$. Если θ равен 1 угловой секунде, расстояние d определяется как 1 парсек. Длина половины большой оси A равна 1 астрономической единице (AE) = 149 597 870 700 м

Здесь особая форма `define-c:prop` определяет процедуру особого рода, конструктор с именем `c:parallax<->distance`. Когда `c:parallax<->distance` получает две ячейки, которые внутри нее называются `parallax` и `distance`, в качестве аргументов, эта процедура создает распространитель ограничений, связывающий эти две ячейки. С помощью особой формы `let-cells` она создает две новые ячейки, одну с локальным именем `t`, а вторую с локальным именем `AU`. Ячейка с именем `t` не получает начального значения; ячейка с именем `AU` инициализируется числовым значением астрономической единицы, большой полуоси земной орбиты, в парсеках. Ячейки `parallax` и `t` связываются через элементарный распространитель ограничений, порождаемый вызовом `c:tan`, который задает условие, что значение в ячейке `t` всегда должно равняться тангенсу значения, содержащегося в ячейке `parallax`. Подобным образом ячейки с именами `t`, `distance` и `AU` связаны элементарным распространителем ограничений, создаваемым через вызов `c:*`, который задает условие, что произведение значения в ячейке `t` и значения в ячейке `distance` равно значению в ячейке `AU`.

Давайте рассмотрим расстояние до звезды Вега, измеренное через параллакс. Создаем две ячейки — `Vega-parallax-distance` для хранения расстояния и `Vega-parallax-angle` для хранения угла параллакса:

```
(define-cell Vega-parallax-distance)
(define-cell Vega-parallax)
```

Теперь можно связать наши две ячейки через конструктор распространителя, только что написанный нами:

```
(c:parallax<->distance Vega-parallax Vega-parallax-distance)
```

Система полученных таким образом ячеек и распространителей изображена на рис. 7.3.

Распространители ограничений сами по себе строятся на основе направленных распространителей, как показано на рис. 7.4. Направленный распространитель, например умножитель, который строит процедура `p:*`, изменяет значение в ячейке произведения так, чтобы оно согласовалось со значениями в ячейках множителей.

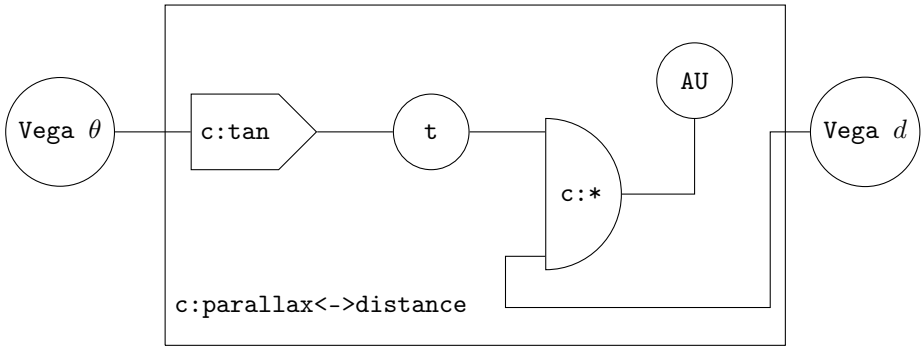


Рис. 7.3. «Диаграмма связей» системы распространителей, построенной вызовом `c:parallax<->distance` по отношению к ячейкам `Vega-parallax-distance` (на диаграмме d Веги) и `Vega-parallax-angle` (на диаграмме θ Веги). Кружки обозначают ячейки, а другие формы — распространители, связывающие эти ячейки. Эти распространители не имеют направления; они устанавливают алгебраические ограничения. По нашему соглашению имена конструкторов распространителей ограничений начинаются с префикса `c:`. Например, распространитель с именем `c:*` обеспечивает условие, что произведение содержимого ячейки `t` и ячейки `Vega-parallax-distance` равно содержимому ячейки `AU`

Смешивать в системе распространителей направленные распространители и распространители ограничений — совершенно нормальная практика[1].

Давайте проведем вычисление с помощью этой небольшой системы. В 1837 г. Фридрих Г. В. фон Струве опубликовал оценку параллакса Веги: $0.125'' \pm 0.05''$[2]. Это была первая правдоподобная публикация параллакса звезды, однако, поскольку измерения основывались на скудных данных, а Струве впоследствии противоречил своим собственным данным, заслуга первого действительного измерения принадлежит Фридриху Вильгельму Бесселю, который в 1838 г. тщательно измерил параллакс звезды 61 Лебедя. Однако оценка Струве весьма близка к лучшему современному измерению параллакса Веги. Сообщим нашей системе распространителей оценку Струве в 125 угловых миллисекунд плюс-минус 50 миллисекунд:

```
(tell! Vega-parallax
       (+->interval (mas->radians 125) (mas->radians 50))
       'FGWvonStruve1837)
```

Процедура `tell!` принимает три аргумента: ячейку распространителя, значение этой ячейки и символ-посылку, которая описывает родословную данных. Процедура `mas->radians` переводит угловые миллисекунды в радианы. Процедура `+->interval` создает интервал с центром в своем первом аргументе:

[1] Идея распространения ограничений была предложена Дэвидом Уолтцем в докторской диссертации об интерпретации рисунков [125]. Джеральд Джей Сассман и Ричард Столлмен разработали на основе распространения ограничений инструменты для анализа электрических цепей [119, 114]. Юджин Фройдер [39] превратил идею программирования, основанного на ограничениях, в целое интеллектуальное направление с собственным журналом [24]. В докторской диссертации Гая Стила [116] показано, как построить язык программирования, основанный на ограничениях.

[2] См. [127], стр. 71.

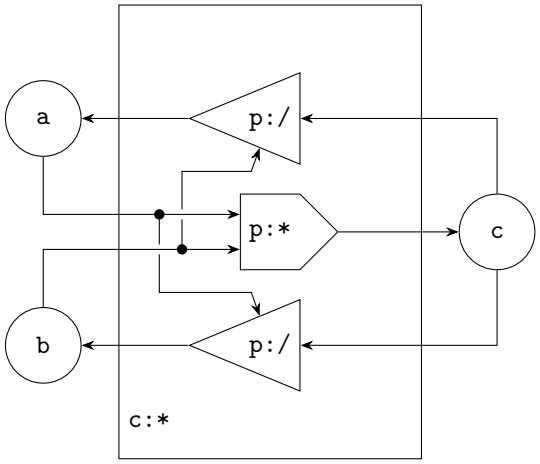


Рис. 7.4. Распространитель ограничений, создаваемый вызовом `c:*`, состоит из трех направленных распространителей. По нашему соглашению конструкторы направленных распространителей имеют имена, начинающиеся с префикса `p:`. Направленный распространитель-умножитель, создаваемый вызовом `p:*`, заставляет значение в ячейке `c` быть произведением ячеек `a` и `b`. Распространители-делители, создаваемые через `p:/`, заставляют значения своих ячеек-частных (`a` и `b`) равняться результату деления ячейки-делимого (`c`) на значение делителя (`b` и `a`)

```
(define (+->interval value delta)
  (make-interval (n:- value delta) (n:+ value delta)))
```

Таким образом, ячейка `Vega-parallax` получает в качестве значения интервал

```
(+->interval (mas->radians 125) (mas->radians 50))
(interval 3.6361026083215196e-7 8.48423941941688e-7)
```

Оценка Струве ошибки его результата была достаточно большой долей оценки самого параллакса. Поэтому оценка расстояния до Веги также получается очень широкой (примерно от 5.7 до 13.3 или 9.5 ± 3.8 парсек):

```
(get-value-in Vega-parallax-distance)
(interval 5.7142857143291135 13.333333333343721)

(interval>+- (get-value-in Vega-parallax-distance))
(+- 9.523809523883163 3.8095238095540473)
```

Это интервальное значение поддерживается посылкой `FGWvonStruve1837`.

```
(get-premises Vega-parallax-distance)
(support-set fgwvonstruve1837)
```

Воспользуемся процедурой `inquire`, которая печатает значение ячейки и поддержку этого значения в удобном виде[3]:

[3]Печатается также «причина» значения в виде списка с заголовком `because`. В нашем случае значение в ячейке `Vega-parallax-distance` было получено делением содержимого ячейки `AU` на

```
(inquire Vega-parallax-distance)
((vega-parallax-distance)
 (has-value (interval 5.7143e0 1.3333e1))
 (depends-on fgwvonstruve1837)
 (because
  ((p:/ c:* c:parallax<->distance)
   (au 4.8481e-6)
   (t (interval 3.6361e-7 8.4842e-7)))))
```

Более точная оценка, полученная в 1982 г. Рассел и др. [106], равна:

```
(tell! Vega-parallax
       (+->interval (mas->radians 124.3) (mas->radians 4.9))
       'JRussell-etal1982)
```

что довольно близко к середине интервала оценки Струве. С этим измерением интервал оценки расстояния сужается до

```
(inquire Vega-parallax-distance)
((vega-parallax-distance)
 (has-value (interval 7.7399 8.3752))
 (depends-on jrussell-etal1982))
```

Заметим, что наша оценка расстояния до Веги теперь зависит только от измерения Рассел. Поскольку интервал измерения Рассел полностью содержится в интервале измерения Струве, измерение Струве не сообщает никакой новой информации. Однако ячейка запоминает измерение Струве и его родословную, так что при необходимости его можно вспомнить.

В 1995 г. появились еще более качественные измерения[4]:

```
(tell! Vega-parallax
       (+->interval (mas->radians 131) (mas->radians 0.77))
       'Gatewood-deJonge1995)
((vega-parallax)
  (has-value (the-contradiction))
  (depends-on jrussell-etal1982 gatewood-dejonge1995)
  (because
   ((has-value (interval 5.7887e-7 6.2638e-7))
    (depends-on jrussell-etal1982))
   ((has-value (interval 6.3137e-7 6.3884e-7))
    (depends-on gatewood-dejonge1995))))
```

содержимое ячейки `t` в распространителе, построенном при вызове `c:parallax<->distance`. Направленный распространитель деления `p:/` являлся частью распространителя ограничений `c:*`, который сам был частью распространителя ограничений `c:parallax<->distance`. Эти «причины» быстро становятся очень длинными. В случаях, когда в них нет особого смысла, мы будем убирать подсписки `because` из результатов вызова `inquire`.

Рекурсивно связывая «причины», можно получить очень подробное объяснение, откуда взялось некоторое значение. «Причины» являются обоснованиями в смысле раздела 6.4.2.

[4]Здесь мы солгали! На самом деле измерение Гейтвуда и де Йонге [43] немного отличается. Центр их интервала равен 130 угловых миллисекунд, а не 131, как мы даем здесь. Мы сместили оценку, чтобы чуть позже в тексте проиллюстрировать некоторую вычислительную деталь.

Как видим, противоречие зависит от двух источников информации. Каждый из них задает некоторый интервал, и эти интервалы не пересекаются. Допустим, мы решили, что измерение Гейтвуда и де Йонге выглядит подозрительно. Давайте отзовем его:

```
(retract! 'Gatewood-deJonge1995)
```

Все значения, зависящие от отозванной посылки, отменяются, и видимое значение для расстояния снова равно:

```
(inquire Vega-parallax-distance)
((vega-parallax-distance)
 (has-value (interval 7.7399 8.3752))
 (depends-on jrussell-etal1982))
```

Это то, что мы получили от Рассел и др. В самом деле, именно эта посылка служит поддержкой значения.

Однако интрига продолжает закручиваться, поскольку спутник «Гиппарх» (как сообщает ван Леувен [83]) проделал весьма впечатляющие измерения параллакса Веги:

```
(tell! Vega-parallax
       (+->interval (mas->radians 130.23) (mas->radians 0.36))
       'FvanLeeuwen2007Nov)
((vega-parallax)
 (has-value (the-contradiction))
 (depends-on jrussell-etal1982 fvanleeuwen2007nov)
 (because
  ((has-value (interval 5.7887e-7 6.2638e-7))
    (depends-on jrussell-etal1982))
  ((has-value (interval 6.2963e-7 6.3312e-7))
   (depends-on fvanleeuwen2007nov))))
```

Кому верить[5]? Давайте отвергнем результат Рассел:

```
(retract! 'JRussell-etal1982)

(inquire Vega-parallax-distance)
((vega-parallax-distance)
 (has-value (interval 7.6576 7.7))
 (depends-on fvanleeuwen2007nov))
```

Это спутниковый результат в отдельности.

Вернем теперь результат Гейтвуда и посмотрим, что произойдет.

```
(assert! 'Gatewood-deJonge1995)

(inquire Vega-parallax-distance)
```

[5]На самом деле, с данными «Гиппарха» есть определенные проблемы. А именно, измеренные им расстояния до некоторых очень ярких скоплений, например до Плеяд, не соотносятся с более точными измерениями методом базовой радиоинтерферометрии. Однако это несоответствие не влияет на другие измерения, сделанные «Гиппархом».

```
((vega-parallax-distance)
 (has-value (interval 7.6576 7.6787))
 (depends-on gatewood-dejonge1995 fvanleeuwen2007nov))
```

Мы получили более сильную оценку, поскольку пересечение интервалов ван Леувена и Гейтвуда меньше, чем каждый интервал по отдельности[6]. (Результат Гейтвуда `(interval 7.589 7.6787)` отдельно не показан.)

Звездная величина

Существуют другие способы оценки расстояния до звезд. Известно, что видимая яркость звезды (блеск) убывает по квадрату расстояния от нее, так что, если нам известна ее абсолютная яркость (светимость), можно получить расстояние путем измерения видимой яркости.

К настоящему времени теория может дать очень точные и надежные оценки светимости некоторых типов звезд. Для этих звезд спектроскопический анализ получаемого от них света дает информацию, например об их состоянии, химическом составе и массе; а исходя из этих величин можно оценить светимость. Вега — очень хороший пример звезды, о которой мы много знаем.

Астрономы описывают яркость звезд в *величинах*. Разница в 5 величин определяется как различие в яркости в 100 раз[7]. Светимость звезды задается как величина, которую она имела бы, находясь в 10 парсеках от наблюдателя. Это называется *абсолютной звездной величиной* звезды. Связь между яркостью и расстоянием выражается простой формулой, которая связывает закон обратного квадрата с определением звездной величины. Если абсолютная величина звезды равна M, видимая величина m, а расстояние до звезды в парсеках d, то $m - M = 5(\log_{10}(d) - 1)$. Эту формулу можно воспроизвести в виде конструктора распространителя ограничений[8]:

```
(define-c:prop
  (c:magnitudes<->distance apparent-magnitude
                           absolute-magnitude
                           magnitude-distance)
  (let-cells (dmod dmod/5 ld10 ld
              (ln10 (log 10)) (one 1) (five 5))
    (c:+ absolute-magnitude dmod apparent-magnitude)
    (c:* five dmod/5 dmod)
```

[6] Именно поэтому мы исказили измерение Гейтвуда и де Йонге. Если бы мы его привели правильно, их результат не пересекался бы с результатом «Гиппарха» по краям. Измерение «Гиппарха» полностью содержалось бы в границах интервала Гейтвуда и де Йонге.

[7] Эта немного странная система происходит из работ древнегреческого астронома Гиппарха (около 190 до н. э. — около 120 до н. э.). Он присвоил числовую величину каждой звезде в своем каталоге. Самые яркие звезды назывались звездами *первой величины*, менее яркие — звездами *второй величины*, а самые тусклые звездами *шестой величины*. Проект космической астрометрии «Гиппарх» Европейского космического агентства (см. стр. 294) назван в честь астронома Гиппарха.

[8] Этот язык некрасив и неудобен, поскольку для всех промежуточных величин в выражении приходится создавать ячейки и давать им имена. Есть множество способов исправить неудобство, однако базовые идеи будут яснее, если начать с грубого, но очень конкретного языка диаграмм связи. Нетрудно написать небольшой компилятор, преобразующий ограничения, записанные в виде алгебраических выражений, в фрагменты диаграмм распространителей (см. упражнение 7.1 на стр. 298 и упражнение 7.6 на стр. 320).

```
    (c:+ one dmod/5 ld10)
    (c:* ln10 ld10 ld)
    (c:exp ld magnitude-distance)))
```

Давайте теперь внесем информацию о Веге. Определяем несколько ячеек и связываем их с распространителями:

```
(define-cell Vega-apparent-magnitude)
(define-cell Vega-absolute-magnitude)
(define-cell Vega-magnitude-distance)

(c:magnitudes<->distance Vega-apparent-magnitude
                         Vega-absolute-magnitude
                         Vega-magnitude-distance)
```

Добавляем результаты измерений. Вега — чрезвычайно яркая звезда: ее видимая звездная величина почти равна нулю. (Это очень точное измерение сделано с помощью телескопа Хаббл. См. Болин и Гиллиланд [14].)

```
(tell! Vega-apparent-magnitude
       (+->interval 0.026 0.008)
       'Bohlin-Gilliland2004)
```

Абсолютная звездная величина Веги также известна с довольно высокой точностью [44]:

```
(tell! Vega-absolute-magnitude
       (+->interval 0.582 0.014)
       'Gatewood2008)
```

В результате получаем неплохую оценку расстояния до Веги, которая зависит только от этих измерений:

```
(inquire Vega-magnitude-distance)
((vega-magnitude-distance)
 (has-value (interval 7.663 7.8199))
 (depends-on gatewood2008 bohlingilliland2004))
```

К сожалению, расстояние содержится у нас в двух различных ячейках, так что давайте свяжем их распространителем:

```
(c:same Vega-magnitude-distance Vega-parallax-distance)
```

К этому моменту у нас появляется еще более хорошее значение расстояния до Веги — интервал, чье верхнее значение такое же, как раньше (на стр. 295), но нижнее значение немного выше:

```
(inquire Vega-parallax-distance)
((vega-parallax-distance)
 (has-value (interval 7.663 7.6787))
 (depends-on fvanleeuwen2007nov gatewood-dejonge1995
             gatewood2008 bohlingilliland2004))
```

Имеет ли по-прежнему значение измерение Гейтвуда и де Йонге 1995 г.? Давайте проверим:

```
(retract! 'Gatewood-deJonge1995)

(inquire Vega-parallax-distance)
((vega-parallax-distance)
 (has-value (interval 7.663 7.7))
 (depends-on fvanleeuwen2007nov
             gatewood2008
             bohlingilliland2004))
```

Да, имеет. Измерение 1995 г. повлияло на верхний конец интервала.

Улучшенные измерения!

Итак, у нас есть два способа измерить расстояние до Веги — через параллакс и через звездную величину. Вот нечто замечательное: интервалы измерения параллакса и звездной величины улучшились, используя информацию друг о друге. Это потребовалось, чтобы система оставалась согласованной.

Посмотрим на видимую звездную величину Веги. Исходное значение, данное в работе Болина и Гиллиланда, было $m = 0.026 \pm 0.008$. Это переводится в интервал

```
(+->interval 0.026 0.008)
(interval .018 .034)
```

Однако теперь значение немного лучше — $[0.018, 0.028456]$:

```
(inquire Vega-apparent-magnitude)
((vega-apparent-magnitude)
 (has-value (interval 1.8e-2 2.8456e-2))
 (depends-on gatewood2008
             fvanleeuwen2007nov
             bohlin-gilliland2004))
```

Верхний конец пришлось сместить немного вниз, чтобы согласовать с измерениями параллакса. Подобное произошло со всеми измеряемыми величинами. Абсолютная звездная величина, данная в работе Гейтвуд 2008 (стр. 296), была:

```
(+->interval 0.582 0.014)
(interval .568 .596)
```

Однако теперь нижний конец сместился вверх:

```
(inquire Vega-absolute-magnitude)
((vega-absolute-magnitude)
 (has-value (interval 5.8554e-1 5.96e-1))
 (depends-on gatewood2008
             fvanleeuwen2007nov
             bohlin-gilliland2004))
```

Параллакс также улучшился, используя измерения звездной величины:

```
(inquire Vega-parallax)
((vega-parallax)
 (has-value (interval 6.2963e-7 6.3267e-7))
 (depends-on fvanleeuwen2007nov
             gatewood2008
             bohlin-gilliland2004))
```

То, что вычисление распространяется во всех направлениях, дает нам мощный инструмент для понимания следствий любой новой информации.

Упражнение 7.1. Облегчаем написание сетей распространителей

В нашей системе распространителей даже для простых сетей писать код сложно, поскольку требуется давать имена всем внутренним ячейкам. Например, распространитель ограничений, преобразующий температуры по Цельсию в температуры по Фаренгейту, выглядит так:

```
(define-c:prop (celsius fahrenheit)
  (let-cells (u v (nine 9) (five 5) (thirty-two 32))
    (c:* celsius nine u)
    (c:* v five u)
    (c:+ v thirty-two fahrenheit)))
```

Было бы приятнее использовать для некоторых распространителей синтаксис выражений и писать:

```
(define-c:prop (celsius fahrenheit)
  (c:+ (ce:* (ce:/ (constant 9) (constant 5))
             celsius)
       (constant thirty-two)
       fahrenheit))
```

Здесь `ce:*` и `ce:+` — конструкторы распространителей, создающие для значения ячейку и возвращающие ее вызывающей процедуре. Процедуру `ce:+` можно было бы написать так:

```
(define (ce:+ x y)
  (let-cells (sum)
    (c:+ x y sum)
    sum))
```

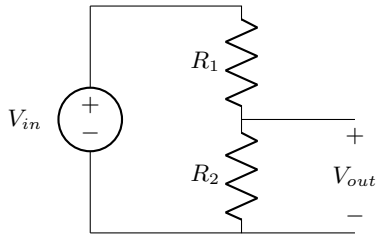
Помимо распространителей ограничений, бывают еще направленные распространители вроде `p:+`. Для его варианта в виде выражения можно взять имя `pe:+`.

У нас есть доступ к именам всех элементарных арифметических операций. Напишите программу, которая возьмет эти имена и для каждого символа операций построит распространители, как направленные, так и распространители ограничений.

Упражнение 7.2. Проектирование электрических схем

Замечание: чтобы решить эту задачу, электронику знать не обязательно.

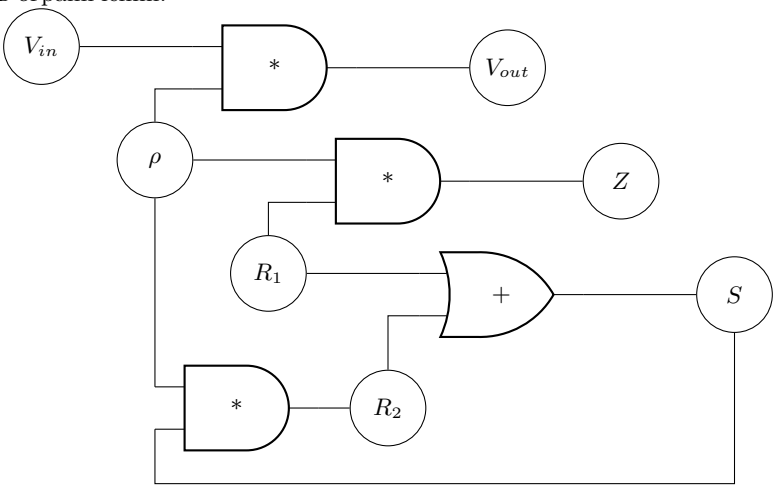
Анна Лог проектирует транзисторный усилитель. В качестве одной из подзадач требуется построить делитель напряжения, чтобы сместить транзистор. Делитель напряжения содержит два резистора с сопротивлениями R_1 и R_2. Отношение выходного напряжения V_{out} к напряжению блока питания V_{in} равно ρ. Имеется также величина Z, выходное сопротивление делителя.



Вот соответствующие уравнения:

$$\begin{aligned} \rho &= \frac{V_{out}}{V_{in}} \\ \rho &= \frac{R_2}{R_1 + R_2} \\ Z &= R_1\rho. \end{aligned}$$

Поскольку Анне нужно решать множество подобных задач, она создает себе в помощь сеть ограничений.



a. Постройте сеть распространителей, реализующую эту диаграмму.

b. У Анны есть источник питания с напряжением от 14.5 до 15.5 В, и ей нужно, чтобы выход делителя был от 3.5 до 4.0 В: $V_{in} \in [14.5, 15.5]$ и $V_{out} \in [3.5, 4.0]$.

   Для R_2 у нее имеется резистор на 47 000 Ом. Из какого диапазона значений ей нужно выбрать R_1? Возможно ли выбрать R_1, удовлетворяющий требованиям?

c. Кроме того, Анне нужно, чтобы сопротивление на выходе делителя составляло от 20 000 до 30 000 Ом: $Z \in [20\,000, 30\,000]$.

   Таким образом, ее настоящая задача состоит в том, чтобы найти подходящие значения величин для резисторов делителя сопротивления R_1 и R_2 при требуемом коэффициенте деления ρ и заданных требованиях к Z.

   Если вместо того, чтобы выбирать R_2 (не забудьте удалить поддержку этого значения!), она задаст интервал для Z, это должно определить R_1 и R_2; однако данная сеть не найдет значение R_2! Почему? Объясните, в чем проблема.

d. Если теперь мы ограничим R_2 интервалом от 1000 Ом до 500 000 Ом, сеть распространителей сойдется к осмысленному ответу для реального значения R_2. Почему? Объясните!

Упражнение 7.3. Проект: локальная согласованность

Распространение информации — способ работы с задачами локальной согласованности. Например, алгоритм Уолтца [125] — метод распространения информации для интерпретации чертежей многогранников. С помощью распространения можно также решать задачи раскраски карт и другие, подобные им.

Основная идея в том, что имеется граф с вершинами, каждой из которых можно присвоить одну метку из некоторого набора, и что вершины связываются ограничениями, говорящими, какие метки разрешены при определенном сочетании меток на соседних узлах. Например, в алгоритме Уолтца у каждой линии есть одна метка из набора. Линия соединяет две вершины. Вершина требует, чтобы линии, заканчивающиеся в ней, были согласованы с одной из нескольких возможных интерпретаций вершины. Однако интерпретация линии должна быть одинаковой на двух ее концах.

a. В этих экспериментах вам потребуется «арифметика» дискретных множеств. Нужны будут объединение, пересечение и разность множеств. Постройте эту арифметику.

b. Множество возможностей для вершины — частичная информация о ее статусе: чем это множество возможностей уже, тем больше у нас информации о вершине. Если мы представляем наши знания о статусе вершины как ячейку распространителя, слияние двух множеств — это их пересечение. Это соответствует пересечению интервалов для вещественных значений. Сделайте пересечение дискретных множеств обработчиком полиморфной процедуры `merge`.

c. Постройте и продемонстрируйте решение задачи локальной согласованности на основе такой организации кода.

d. Обратите внимание, что во многих графах присвоение значения вершине зависит только от нескольких ограничений. Покажите, как с помощью отслеживания поддержки строить объяснения для решений о статусе вершины.

7.2 Механизм распространения

Основной механизм системы распространения информации состоит из *ячеек*, *распространителей* и *планировщика*. *Ячейка* собирает в себе информацию о значении. Она должна быть способна сказать, какая информация в ней содержится, а также должна принимать обновления этой информации. Наконец, она должна уметь оповещать об изменениях распространители, заинтересованные в ее содержимом. Каждая ячейка хранит множество распространителей, которым может быть интересно ее содержимое; они называются ее *соседями*.

Распространитель представляет собой процедуру без состояния (функциональную), запускаемую при любых изменениях ячеек, в которых он заинтересован. Ячейки, которые могут активировать распространитель, называются его *входными* ячейками. Активированный распространитель собирает информацию из входных ячеек и может вычислить обновление для одной или более *выходной* ячейки. Одна и та же ячейка может служить для распространителя и входной, и выходной.

Содержимое ячейки — это информация, накопленная о ее значении. При запросе значения, например исходящем от распространителя, она отвечает *наиболее сильным* утверждением, на которое способна. Мы видели, как это работает при использовании интервалов — ячейка выдает самый узкий интервал, известный для значения. Когда ячейка получает информацию на вход, она решает, изменяет ли эта информация наиболее сильное значение. Если наиболее сильное значение изменяется, ячейка оповещает соседей. Это дает планировщику сигнал, что соседи должны быть активированы. *Планировщик* отвечает за выделение вычислительных ресурсов активированным распространителям. Желательно, чтобы вычислительный результат процесса распространения не зависел от подробностей порядка планировки.

Ячейки и распространители организованы в иерархию. У каждой ячейки и распространителя есть *имя*, *родитель* и, возможно, множество *детей*. С помощью этой

иерархии можно получить уникальное полное имя для каждой ячейки и каждого распространителя в иерархии. Через это полное имя к элементу можно обратиться и можно его идентифицировать в печатном выводе системы. Всякая ячейка и распространитель создается либо пользователем, либо составным распространителем. Параметр `*my-parent*` динамически связан с родителем элемента. Это позволяет новой ячейке или распространителю встроить себя в иерархию.

7.2.1 Ячейки

Ячейка реализована как процедура, принимающая сообщения, через макрос `bundle`. Она хранит свою информацию в переменной `content` с начальным значением `the-nothing` (которое можно распознать предикатом `nothing?`), представляющим отсутствие информации о значении. Значение, которое ячейка выдает по запросу — самое сильное (`strongest`) значение, известное ей в данный момент. Кроме того, ячейка поддерживает множество соседей `neighbors` — это распространители, которым нужно слать сигнал, когда наиболее сильное значение ячейки меняется. Дополнительная структура данных `relations` сохраняет семейные отношения ячейки.

Вот базовая структура конструктора ячеек. Интерес представляют процедуры `add-content!` и `test-content!`, которые объясняются далее.

```
(define (make-cell name)
  (let ((relations (make-relations name (*my-parent*)))
        (neighbors '())
        (content the-nothing)
        (strongest the-nothing))
    (define (get-relations) relations)
    (define (get-neighbors) neighbors)
    (define (get-content) content)
    (define (get-strongest) strongest)
    (define (add-neighbor! neighbor)
      (set! neighbors (lset-adjoin eq? neighbors neighbor)))
    (define (add-content! increment)
      (set! content (cell-merge content increment))
      (test-content!))
    (define (test-content!)
      См. определение на стр. 302.)
    (define me
      (bundle cell? get-relations get-neighbors
              get-content get-strongest add-neighbor!
              add-content! test-content!))
    (add-child! me (*my-parent*))
    (set! *all-cells* (cons me *all-cells*))
    me))
```

Ячейка получает новую информацию через вызов процедуры `add-content!`. Новую информацию `increment` нужно слить со старой информацией, содержащейся в переменной `content`. В общем случае процесс слияния зависит от типа сливаемой информации, так что требуется указать для ячейки механизм слияния. Однако значение `the-nothing`, представляющее отсутствие всякой информации, ведет себя

особым образом. Любая информация при слиянии с `the-nothing` остается неизменной.

Причина, по которой мы используем не замену, а слияние, в том, что мы хотим с помощью частичной информации уточнять наши знания о значении[9]. Например, при описанном нами выше вычислении расстояний до звезд интервалы сливаются через пересечение, и таким образом получаются более точные оценки. В примере с выводом типов (см. раздел 4.4.2) мы сливали различные описания через унификацию и получали более точные описания. В разделе 7.4 мы рассмотрим общую задачу слияния значений.

В некоторых случаях слияние двух фрагментов информации оказывается невозможным. Например, значение неизвестного числа не может одновременно быть равным нулю и единице. В этом случае процедура `cell-merge` возвращает объект-*противоречие*, в котором может содержаться информация о подробностях конфликта. Если дополнительной информации нет, объект-противоречие представляет собой символ `the-contradiction`, удовлетворяющий элементарному предикату `contradiction?`. Более сложные противоречия распознаются полиморфной процедурой `general-contradiction?`. Если это возможно, противоречия разрешаются процедурой `handle-cell-contradiction`; она описывается в разделе 7.5.

Если наиболее сильное значение ячейки меняется, оповещаются ее соседи. Однако, если новые данные не влияют на наиболее сильное значение, они не сообщают никакой дополнительной информации. В этом случае оповещать соседей не следует во избежание бесполезных циклов. Все это реализуется в процедуре `test-content!`, определяемой как внутренняя процедура `make-cell`.

```
(define (test-content!)
  (let ((strongest* (strongest-value content)))
    (cond ((equivalent? strongest strongest*)
           (set! strongest strongest*)
           'content-unchanged)
          ((general-contradiction? strongest*)
           (set! strongest strongest*)
           (handle-cell-contradiction me)
           'contradiction)
          (else
           (set! strongest strongest*)
           (alert-propagators! neighbors)
           'content-changed))))
```

Кроме того, с помощью процедуры `test-content!` ячейки оповещаются, когда меняется статус той или иной посылки. Каждая ячейка проверяет, изменилось ли ее наиболее сильное значение и требуется ли при этом некоторое действие, например сигнал о противоречии либо оповещение соседей-распространителей (см. раздел 7.3).

Чтобы скрыть подробности реализации, мы вводим процедуры для удобного доступа к ячейкам:

```
(define (add-cell-neighbor! cell neighbor)
  (cell 'add-neighbor! neighbor))
```

[9]Это главная идея докторской диссертации А. Радула [99, 100].

```
(define (add-cell-content! cell increment)
  (parameterize ((current-reason-source cell))
    (cell 'add-content! increment)))

(define (cell-strongest cell)
  (cell 'get-strongest))
```

Параметр `current-reason-source` в процедуре `add-cell-content!` принадлежит слою, хранящему «причины» для каждого значения, как описано в примечании 3 на стр. 292. Это полезное свойство нашей программы мы далее здесь обсуждать не будем.

7.2.2 Распространители

При создании распространителя мы должны задать список ячеек на входе, список ячеек на выходе и процедуру `activate!`, которая запускается при оповещениях. Конструктор сообщает о распространителе входным ячейкам через вызов `add-cell-neighbor!`. Кроме того, он возбуждает созданный распространитель, так что он при необходимости запустится.

```
(define (propagator inputs outputs activate! name)
  (let ((relations (make-relations name (*my-parent*))))
    (define (get-inputs) inputs)
    (define (get-outputs) outputs)
    (define (get-relations) relations)

    (define me
      (bundle propagator? activate!
              get-inputs get-outputs get-relations))
    (add-child! me (*my-parent*))
    (for-each (lambda (cell)
                (add-cell-neighbor! cell me))
              inputs)
    (alert-propagator! me)
    me))
```

Элементарные распространители направленные, т. е. множества их выходных ячеек не пересекаются со входными. Мы строим элементарные распространители из процедур языка Scheme, у которых всегда одно значение на выходе. По нашему соглашению элементарному распространителю передаются вместе входные и выходная ячейка, причем выходная стоит последней. Можно было бы создать элементарный распространитель, порождающий несколько выходов, например целочисленное деление с остатком; однако здесь нам это не требуется.

```
(define (primitive-propagator f name)
  (lambda cells
    (let ((output (car (last-pair cells)))
          (inputs (except-last-pair cells)))
      (propagator inputs (list output)
        (lambda ()
```

```
      (let ((input-values (map cell-strongest inputs)))
        (if (any unusable-value? input-values)
            'do-nothing
            (add-cell-content! output
                               (apply f input-values)))))
    name))))
```

Будучи активированным, распространитель может решить вычислить результат с помощью `f`. Результат вызова `f` со входными значениями добавляется к выходной ячейке. Этот процесс выбора мы называем *политикой активации*. Мы требуем здесь, чтобы все входы были значениями, *пригодными к использованию*. По умолчанию объекты-противоречия и `the-nothing` использовать нельзя, но в будущем мы можем добавить и другие объекты. Возможны и другие политики активации.

Можно конструировать распространители путем сочетания других распространителей. Для создания составного распространителя нужна процедура `to-build`, создающая требуемую подсеть из частей. Составной распространитель не строится до тех пор, пока он не понадобится в вычислении. Однако эта нужда возникает только тогда, когда в одной ячейке или более на входе появляются данные, и вычисление активируется. Однако нам не хочется каждый раз при появлении новых значений во входных ячейках перестраивать распространитель заново, так что конструктор должен позаботиться, чтобы он вызывался только один раз. Для этого служит булев флаг `built?`, который выставляется в истину, когда построение закончено.

```
(define (compound-propagator inputs outputs to-build name)
  (let ((built? #f))
    (define (maybe-build)
      (if (or built?
              (and (not (null? inputs))
                   (every unusable-value?
                          (map cell-strongest inputs))))
          'do-nothing
          (begin (parameterize ((*my-parent* me))
                   (to-build))
                 (set! built? #t)
                 'built)))
    (define me
      (propagator inputs outputs maybe-build name))
    me))
```

Политика активации составного распространителя отличается от политики активации элементарного распространителя. Здесь мы строим сеть, если можно использовать значение хотя бы на одном из входов. Это правильное решение, поскольку часть сети может произвести полезное вычисление, даже если получены не все входы.

Возня с `parameterize` нужна для того, чтобы поддержать иерархическую организацию элементов распространителя. Составной распространитель оказывается отцом всех ячеек и всех распространителей, создаваемых при построении сети.

Как описано на рис. 7.4 на стр. 292, распространители ограничений строятся путем сочетания направленных распространителей. Например, можно создать распространитель, воплощающий условие, что произведение значений в двух ячейках равно значению третьей:

```
(define-c:prop (c:* x y product)
  (p:* x y product)
  (p:/ product x y)
  (p:/ product y x))
```

Как видим, для создания ограничения сочетаются три направленных распространителя. Это работоспособная структура, потому что мы сливаем значения, а не заменяем их. Кроме того, эквивалентные значения не распространяются. Если бы они распространялись, всякий распространитель, похожий на `c:*`, приводил бы к бесконечному циклу[10].

Макрос `define-c:prop` служит всего лишь синтаксическим сахаром. Настоящий код, который он порождает, выглядит так:

```
(define (c:* x y product)
  (constraint-propagator
   (list x y product)
   (lambda ()
     (p:* x y product)
     (p:/ product x y)
     (p:/ product y x))
   'c:*))
```

где `constraint-propagator` — всего лишь:

```
(define (constraint-propagator cells to-build name)
  (compound-propagator cells cells to-build name))
```

Все ячейки, связанные с распространителем ограничений, одновременно являются входными и выходными.

7.3 Альтернативные точки зрения

В примере с вычислением расстояния до звезд мы показали, что каждое значение несет при себе множество поддержки, состоящее из посылок, использованных в его вычислении, а также «причину» (распространитель, построивший это значение, и значения, поданные на вход этому распространителю). Эта информация получена с помощью механизма многослойных данных, введенного нами в разделе 6.4. Однако некоторые «факты» противоречат друг другу. В нашем примере мы изменяли статус нашего доверия разным посылкам и получали локально согласованные точки зрения в зависимости от того, каким посылкам доверяем.

Посылка находится либо в состоянии `in` (система ей доверяет), либо `out` (система не доверяет). В нашем примере пользователь мог либо сказать посылке `assert!`, переводя ее в состояние `in`, либо сказать `retract!` и перевести в состояние `out`. «Волшебство» системы состоит в том, что наблюдаемые значения всегда *полностью*

[10]Мы не говорим здесь о серьезной проблеме определения эквивалентности для неточных величин. Никакое глобальное понятие не может представлять критерии эквивалентности без дополнительной локальной информации. Чтобы решить эту проблему, можно было бы задавать локальное понятие эквивалентности в каждой ячейке со значением по умолчанию для точных величин.

поддержаны — все поддерживающие их посылки в состоянии доверия, даже притом, что состояние посылок может меняться со временем[11].

Было бы глупо при каждом изменении состояния посылок пересчитывать все значения. Лучше запоминать значения, не полностью поддержанные в данный момент. Это позволяет нам восстановить посылку и возобновить статус значений, поддерживаемых ей, без перевычисления. Когда состояние доверия посылки меняется, ячейки должны проверить, изменилось ли их самое сильное значение. Это происходит путем вызова процедуры `test-content!` для каждой ячейки; ячейка, у которой самое сильное значение поменялось, оповещает распространители, зависящие от ее значения. Каждый из этих распространителей получает наиболее сильные значения своих входных ячеек и вычисляет (или перевычисляет!) свое значение на выходе. Если это значение совпадает с тем, которое уже хранится в выходной ячейке, больше ничего не происходит. Если же состояние доверия наиболее сильного значения выходной ячейки изменяется, это приводит к перевычислению соседних распространителей. Однако наиболее сильное значение выходной ячейки может опираться на независимую поддержку, и тогда распространение на этом остановится.

Чтобы все это работало, поле `content` в каждой ячейке должно содержать *множество значений*, и при каждом значении хранить посылки, от которых оно зависит. Ячейка извлекает наиболее сильное значение из этого множества и хранит его в поле `strongest`. Это значение можно получить через вызов процедуры `cell-strongest`. Наиболее сильное значение — это лучший выбор из полностью поддержанных значений в множестве[12] либо, если нет ни одного полностью поддержанного значения, `the-nothing`.

Нам остается разъяснить процедуру `strongest-value`, которая должна уметь работать с простыми входными данными, с многослойными данными и с множествами значений. Поэтому естественно сделать ее полиморфной. Самое сильное значение для неаннотированного элемента данных — сам этот элемент, так что это вариант по умолчанию.

```
(define strongest-value
        (simple-generic-procedure 'strongest-value 1
                                  (lambda (object) object)))
```

Если многослойный элемент данных полностью поддержан, он сам является своим наиболее сильным значением, в противном случае наиболее сильное значение не содержит информации.

```
(define-generic-procedure-handler strongest-value
  (match-args layered-datum?)
  (lambda (elt)
```

[11] В нашей реализации есть очень плохая идея. Изменение статуса доверия для посылки реализуется как глобальная операция — при моделировании параллельного процесса это никогда не бывает удачным решением! Более удачная реализация распространяла бы изменение статуса доверия посылок при помощи локальных процессов, подобно тому, как распространяются значения, поддерживаемые этими посылками. Но мы так не сделали — извините!

[12] На самом деле выбор наилучшего значения — достаточно сложная задача. Если одно полностью поддержанное значение более точное, чем другое, например более узкий интервал, то оно лучше. Кроме того, если у значения меньше посылок в множестве поддержки, чем у другого «эквивалентного» значения, то оно тоже лучше, поскольку для веры в него нужно веры в меньшее количество посылок. Это поддерживается механизмом слияния множеств значений, который мы опишем в разделе 7.4.3.

```
    (if (all-premises-in? (support-layer-value elt))
        elt
        the-nothing)))
```

Наиболее сильное значение множества значений — это его наиболее сильное следствие.

```
(define-generic-procedure-handler strongest-value
  (match-args value-set?)
  (lambda (set) (strongest-consequence set)))
```

Процедура `strongest-consequence` сливает полностью поддержанные элементы множества значений. Для определения «наилучшего» из полностью поддержанных элементов множества значений она использует `merge-layered` (см. раздел 7.4.2). Если полностью поддержанных значений нет, у нас нет никакой информации, и результатом будет `the-nothing`.

```
(define (strongest-consequence set)
  (fold (lambda (increment content)
          (merge-layered content increment))
        the-nothing
        (filter (lambda (elt)
                  (all-premises-in?
                    (support-layer-value elt)))
                (value-set-elements set))))
```

7.4 Слияние значений

Мы пока еще не обсудили, что значит сливать значения. Это сложный процесс, состоящий из трех частей: слияния базовых значений, включая числа и интервалы, слияния значений с поддержкой и слияния множеств значений. Процедура `cell-merge` внутри `add-content!` должна поддерживать алгоритм слияния, соответствующий данным, с которыми работают распространители. На стр. 319 процедура `setup-propagator-system` инициализирует `cell-merge` значением `merge-value-sets`.

7.4.1 Слияние базовых значений

В нашей системе распространителей имеется лишь несколько типов базовых значений: `the-nothing`, `the-contradiction`, числа, булевы значения и интервалы. Числа и булевы значения обрабатываются просто — сливать можно только эквивалентные значения. Если слить не получается, выходит противоречие. При слиянии с `the-nothing` все что угодно остается неизменным. Все что угодно при слиянии с `the-contradiction` дает `the-contradiction`. Процедура `merge` для базовых значений полиморфна, и обработчик по умолчанию отвечает за все простые случаи — все, кроме интервалов.

```
(define merge
  (simple-generic-procedure 'merge 2
```

```
    (lambda (content increment)
      (cond ((nothing? content) increment)
            ((nothing? increment) content)
            ((contradiction? content) content)
            ((contradiction? increment) increment)
            ((equivalent? content increment) content)
            (else the-contradiction)))))
```

В астрономическом примере мы работаем также с арифметикой интервалов, так что требуется сливать интервалы:

```
(define (merge-intervals content increment)
  (let ((new-range (intersect-intervals content increment)))
    (cond ((interval=? new-range content) content)
          ((interval=? new-range increment) increment)
          ((empty-interval? new-range) the-contradiction)
          (else new-range))))
```

Можно слить число с интервалом. Если число лежит внутри интервала, оно и будет результатом, иначе получается противоречие:

```
(define (merge-interval-real int x)
  (if (within-interval? x int)
      x
      the-contradiction))
```

Все это вместе склеивается в обработчик полиморфной процедуры:

```
(define-generic-procedure-handler merge
  (any-arg 2 interval? real?)
  (lambda (x y)
    (cond ((not (interval? x)) (merge-interval-real y x))
          ((not (interval? y)) (merge-interval-real x y))
          (else (merge-intervals x y)))))
```

Других случаев при слиянии базовых значений не бывает.

7.4.2 Слияние значений с поддержкой

Значение с поддержкой реализовано как многослойный объект данных, содержащий уровень поддержки и базовое значение, подлежащее распространению. Поэтому слияние значений с поддержкой должно быть многослойной процедурой:

```
(define merge-layered
  (make-layered-procedure 'merge 2 merge))
```

Уровень поддержки реализует слияние через процедуру `support:merge`, которой дается три аргумента: слитое значение, полученное базовым уровнем, текущее значение и новые данные. Задача `support:merge` — определить множество поддержки слитого значения. Если слитое значение совпадает с текущим или со значением новых данных, можно использовать поддержку соответствующего аргумента. Однако, если слитое значение отличается, нужно скомбинировать множества поддержки.

```
(define (support:merge merged-value content increment)
  (cond ((equivalent? merged-value
                      (base-layer-value content))
         (support-layer-value content))
        ((equivalent? merged-value
                      (base-layer-value increment))
         (support-layer-value increment))
        (else
         (support-set-union
           (support-layer-value content)
           (support-layer-value increment)))))

(define-layered-procedure-handler merge-layered support-layer
  support:merge)
```

Процедура `define-layered-procedure-handler` здесь присоединяет процедуру `support:merge` в качестве обработчика для слоя `support-layer` многослойной процедуры `merge-layered`.

7.4.3 Слияние множеств значений

Чтобы слить множества значений, нужно присоединить новое значение к содержимому, получая новое множество. Процедура `->value-set` преобразует свой аргумент в множество значений.

```
(define (merge-value-sets content increment)
  (if (nothing? increment)
      (->value-set content)
      (value-set-adjoin (->value-set content) increment)))
```

Когда к содержимому множества присоединяется новый элемент, если этот новый элемент покрывается одним из существующих элементов, его добавлять не надо.

```
(define (value-set-adjoin set elt)
  (if (any (lambda (old-elt)
             (element-subsumes? old-elt elt))
           (value-set-elements set))
      set
      (make-value-set
       (lset-adjoin equivalent?
                    (value-set-elements set)
                    elt))))
```

Критерии покрытия довольно сложные. Один элемент покрывает другой, если его базовое значение по крайней мере столь же информативно, как базовое значение другого, а множество поддержки включается в множество поддержки другого. (Заметим, что, если множество поддержки меньше, оно сильнее, поскольку зависит от меньшего количества посылок.)

```
(define (element-subsumes? elt1 elt2)
  (and (value-implies? (base-layer-value elt1)
```

```
                    (base-layer-value elt2))
    (support-set<= (support-layer-value elt1)
                   (support-layer-value elt2))))
```

Процедура `value-implies?` полиморфна, поскольку она должна работать со многими видами базовых данных, в том числе с интервалами.

Упражнение 7.4. Слияние с унификацией
Мы видели, как интервалы, частично характеризующие некоторое значение, можно сливать и получать более точную информацию об этом значении. Еще одна разновидность частичной информации — символьные образцы с пропусками в местах, где информация отсутствует. Как описано в разделе 4.4, такие данные можно сливать с помощью унификации. Через унификацию мы реализовали простой вариант вывода типов, но ее можно использовать и в более общих контекстах для сочетания символических выражений, заданных частично. Подходящим примером могут служить записи о Бене Франклине из раздела 4.4. Один из способов рассуждений о системе распространителей — считать каждую ячейку маленькой базой данных, хранящей данные о каком-то конкретном объекте. Распространители, связывающие ячейки, позволяют делать логические выводы. Одна из многообещающих областей — классификация топологических пространств в точечной топологии. Еще одна — организация места, где вы живете, где, например, хранятся отношения соседства между комнатами и социальные отношения между их обитателями. Выберите область, которая вас интересует. Используйте воображение!

a. Спроектируйте сеть распространителей, где каждая ячейка хранит какую-то разновидность символической информации. Например, ячейка может хранить все, что известно о каком-то студенте MIT. Этой информацией может быть имя, адрес, номер телефона, год поступления, основная специальность, день рождения, лучшие друзья... Потребуется спроектировать расширяемую структуру данных, где можно сохранить всю эту и другую информацию. Потребуются также распространители, которые связывают информацию о людях между собой. Можно через одного человека или нескольких получить информацию о другом. Может быть, получится неплохая модель распространения слухов. Создайте элементарные распространители, работающие с этими символьными величинами, и постройте интересную сеть.

b. Добавьте унификацию в качестве обработчика полиморфной процедуры в `merge` и покажите, как с ее помощью можно сливать частичную символическую информацию из разных источников.

c. Найдите какие-нибудь интересные составные распространители, через которые можно представить часто встречающиеся комбинации связанных персонажей вашей сети.

7.5 Поиск в возможных мирах

Было бы здорово обойтись без поиска. К сожалению, во многих разновидностях реальных задач бывает полезно «предположить в интересах рассуждения» нечто, что потом может оказаться ложным. Затем мы прорабатываем последствия такого предположения. Если оно ведет к противоречию, мы отзываем предположение и пробуем что-то еще. Но в любом случае предположение может оказаться полезным в других выводах, помогающих решить задачу.

Мы начали исследовать эту идею в разделе 5.4, где ввели в язык оператор `amb` и использовали его в задачах поиска. В этих упражнениях с `amb` мы работали с языком, основанным на выражениях, и порядок вычислений был ограничен тем,

как вычисляются выражения. Отчасти нам удалось избежать этого ограничения через болезненную работу с продолжениями, будь то структурирование вычисления с явной передачей процедур-продолжений (как в разделе 5.4.2) или использование продолжений, встроенных в Scheme, с помощью `call/cc` (как в разделе 5.5.3). Но даже `call/cc` не дает нам достаточной степени управления процессом поиска.

В разделе 6.4 мы показали, как привязать к каждому значению его множество поддержки, т. е. множество посылок, от которых это значение зависит. Если каждое предположение связано с именем новой посылки, мы будем в точности знать, какое сочетание предположений привело к противоречию. Если мы будем достаточно умны, то сможем избежать попыток утверждения того же самого набора предположений при поиске в будущем. Однако при вычислении выражений нелегко отделить утверждение предположений от потока управления.

Проблема в том, что в языке, основанном на выражениях, решения принимаются в процессе вычисления выражений, и получается ветвящееся дерево решений. Дерево решений рассматривается в каком-то порядке, например перебором в глубину или в ширину. Следствия каждой последовательности решений исследуются после того, как эти решения приняты. Если случилась неудача (мы пришли к противоречию), виноваты в этом какие-то из решений на текущей ветке. Однако, если виноваты только некоторые из сделанных решений, могут присутствовать также невинные предположения, сделанные после последнего виновного. При откате к последнему неверному решению теряются вычисления, зависящие только от невинных решений. Таким образом, откат текущей ветки до некоторого давнего решения может вести к потере многих полезных рассуждений.

В отличие от этой картины в реальных задачах обычно последствия каждого решения локализованы и ограничены. Например, при решении кроссворда мы часто оказываемся в тупике — не можем заполнить ни одну клетку с уверенностью. Однако, если мы предположим, что в некоторой клетке стоит какая-то определенная буква, не имея для этого предположения достаточных оснований, можно продвинуться в решении на несколько шагов. Такое предположение позволяет продолжить работу, но в конце концов мы можем обнаружить, что догадка была неверной, и ее следует отозвать. Однако многие шаги, сделанные после предположения, были верными, поскольку от предположения не зависели. Когда уничтожаются последствия неудачной догадки, эти верные выводы мы не отменяем. Нам нужно отозвать последствия неправильных предположений, а последствия других предположений сохранить. В языковой системе, основанной на выражениях, этого довольно трудно добиться.

В системе с распространителями мы избегаем структуры управления, основанной на вычислении выражений; ценой этого является картина, где каждый распространитель выглядит как независимое устройство, работающее параллельно с другими. Поскольку ячейка такой системы может содержать множество значений, чьи элементы имеют различные слои, с каждым значением можно связать множество поддержки. В системе распространителей значение пользуется доверием только тогда, когда мы верим во все элементы его множества поддержки, и только те значения, в которые мы верим, подвергаются распространению. Таким образом, мы можем переключать точки зрения, управляя статусом доверия каждой посылки по отдельности.

Некоторые сочетания посылок противоречивы. Противоречие обнаруживается, когда система пытается слить два несовместимых значения с полной поддержкой

и получает объект-противоречие. Множество поддержки объекта-противоречия содержит предположения, которые привели к противоречию.

Чтобы это заработало, мы вводим распространитель с выбором, подобный `amb`. Этот распространитель строит предположения о значении ячейки, которой он управляет. Каждое предположение поддержано *гипотетической посылкой*; распространитель с выбором создает это предположение, может заявить его как утверждение или отозвать. Сеть распространителей вычисляет последствия альтернативных присваиваний значений предположениям, сделанным распространителями с выбором, в сети, пока не будет найдено непротиворечивое присваивание.

Пифагоровы тройки

Рассмотрим задачу поиска пифагоровых троек для натуральных чисел до десяти. (Мы имели дело с подобной задачей на стр. 241. Здесь мы построим еще более глупый алгоритм!) Можно ее сформулировать в виде программы на распространителях:

```
(define (pythagorean)
  (let ((possibilities '(1 2 3 4 5 6 7 8 9 10)))
    (let-cells (x y z x2 y2 z2)
      (p:amb x possibilities)
      (p:amb y possibilities)
      (p:amb z possibilities)
      (p:* x x x2)
      (p:* y y y2)
      (p:* z z z2)
      (p:+ x2 y2 z2)
      (list x y z))))
```

Этот код строит сеть распространителей с тремя распространителями-умножителями и распространителем для сложения. Эта сеть будет удовлетворена, если значения в ячейках `x`, `y` и `z` составляют пифагорову тройку. Каждая из этих ячеек связана с распространителем выбора, который создается через `p:amb` и выбирает элемент из списка `possibilities`.

Для запуска программы нужно запустить систему распространителей:

```
(initialize-scheduler)
```

Теперь можно построить сеть распространителей и извлечь из нее все тройки. Процедура `pythagorean` строит эту сеть и возвращает список из трех интересующих нас ячеек. Процедура `run` включает планировщик и таким образом запускает сеть. В процессе работы распространители выбора предлагают значения для `x`, `y` и `z`, пока либо не будет обнаружено неразрешимое противоречие, либо процесс распространения не успокоится. Если противоречия не найдено, процедура `run` возвращает значение `done`, и печатаются наиболее сильные значения каждой из интересующих нас ячеек. Затем эта комбинация значений отвергается, и цикл повторяется новым вызовом `run`.

```
(let ((answers (pythagorean)))
  (let try-again ((result (run)))
    (if (eq? result 'done)
        (begin
```

```
          (pp (map (lambda (cell)
                     (get-base-value
                      (cell-strongest cell)))
                   answers))
          (force-failure! answers)
          (try-again (run)))
        result)))
(3 4 5)
(4 3 5)
(6 8 10)
(8 6 10)
(contradiction #[cell x])
```

7.5.1 Перебор с возвратами, управляемый зависимостями

Перебор с возвратами, управляемый зависимостями, — мощная методика, оптимизирующая поиск с возвратами, избегая заново пробовать наборы посылок, которые поддерживают ранее обнаруженные противоречия[13]. Мы используем стратегию перебора, управляемого зависимостями, на основе понятия *нехорошего множества* — множества посылок, которые не могут все приниматься одновременно, поскольку известно, что из их конъюнкции выводится противоречие. Если ячейка содержит два или более противоречащих друг другу значения, объединение множеств поддержки этих значений образует нехорошее множество.

Когда обнаруживается противоречие, нехорошее множество для этого противоречия запоминается, чтобы программа перебора впоследствии не пыталась заново использовать эту комбинацию. Чтобы упростить работу механизма перебора, множество не хранится напрямую: оно распределяется по всем посылкам, которые в него входят. Каждая посылка получает копию нехорошего множества, из которой сама она исключается. Например, если нехорошее множество равно $ABC\ldots$, посылка A получит множество $BC\ldots$, посылка B получит множество $AC\ldots$ и т. д. Для каждой посылки список частичных нехороших множеств, собранных на основе противоречий, где эта посылка участвовала, можно получить вызовом процедуры `premise-nogoods`.

После того как нехорошее множество сохранено, программа перебора выбирает из этого множества гипотетическую посылку (если такие есть) и отзывает ее. Отзыв посылки активирует распространители-соседи ячеек, чьи значения были поддержаны этой гипотезой, включая распространитель, который эту гипотезу изначально утверждал. Этот распространитель делает, если это возможно, альтернативное ги-

[13]Перебор с возвратами, управляемый зависимостями, впервые описали Ричард Столлмен и Джеральд Джей Сассман в контексте анализа электрических схем [114]. Очень похожий метод «дизъюнктного обучения» был разработан Карлом Либерхерром [84] в контексте логики. Сейчас дизъюнктное обучение используется в самых успешных SAT-решателях. Рамин Забих, Дэвид Макаллестер и Дэвид Чэпмен показали один из способов встроить эту методику в Lisp-код [132]. Гай Стил обнаружил изящный способ включить перебор с возвратами, управляемый зависимостями, в язык ограничений [116]. Основываясь на работах Джона Дойла [30] и Дэвида Макаллестера [88], Кен Форбус и Йохан де Клеер развили теорию и практику зависимостей в «системах поддержания истины» [36]; это общий способ мышления о зависимостях и переборе. Способ, которым мы пользуемся в этой книге для работы с перебором, управляемым зависимостями, разработали Алексей Радул и Джеральд Джей Сассман [99, 100].

потетическое утверждение. Если в нехорошем множестве нет гипотетических посылок, программа перебора ничего не может сделать и возвращает неудачу.

Разумеется, чтобы все это заработало, нужно проделать множество вспомогательной работы. Давайте посмотрим, как она может быть реализована.

Гипотезы порождаются и управляются процедурой `binary-amb`

Простейший распространитель выбора создается процедурой `binary-amb`. В результате применения `binary-amb` к ячейке получается распространитель двоичного выбора с этой ячейкой и на входе, и на выходе. Этот распространитель управляет значением ячейки, присваивая ей истину или ложь, пока не будет найдено согласованное присваивание для всех таких ячеек.

Процедура `binary-amb` вводит две новые посылки, которые помечаются как гипотетические. *Гипотетическая посылка* — это такая посылка, статус которой может при необходимости автоматически изменяться. Процедура `binary-amb` присваивает ячейке противоречивое начальное значение: процедура `make-hypotheticals` создает и истинное значение, и ложное; каждое из них поддерживается одной из новых гипотетических посылок, и `make-hypotheticals` оба этих значения добавляет в содержимое ячейки. Добавление этих значений активирует ячейку; у нее вызывается процедура `test-content!`; эта процедура запускает механизм обработки противоречий, и в конце концов распространитель двоичного выбора получает сообщение о неудовлетворенной ячейке. Противоречие будет разрешено процедурой `activate!` двоичного распространителя, а именно `amb-choose`:

```
(define (binary-amb cell)
  (let ((premises (make-hypotheticals cell '(#t #f))))
    (let ((true-premise (car premises))
          (false-premise (cadr premises)))
      (define (amb-choose)
        (let ((reasons-against-true
               (filter all-premises-in?
                       (premise-nogoods true-premise)))
              (reasons-against-false
               (filter all-premises-in?
                       (premise-nogoods false-premise))))
          (cond ((null? reasons-against-true)
                 (mark-premise-in! true-premise)
                 (mark-premise-out! false-premise))
                ((null? reasons-against-false)
                 (mark-premise-out! true-premise)
                 (mark-premise-in! false-premise))
                (else
                 (mark-premise-out! true-premise)
                 (mark-premise-out! false-premise)
                 (process-contradictions
                  (pairwise-union reasons-against-true
                                  reasons-against-false)
                  cell)))))
      (let ((me (propagator (list cell) (list cell)
```

```
                        amb-choose 'binary-amb)))
      (set! all-amb-propagators
            (cons me all-amb-propagators))
      me))))
```

Процедура `amb-choose` с помощью нехороших множеств посылок определяет, можно ли верить в посылку, поддерживающую истинное значение, или в посылку, поддерживающую ложное значение. Каждый элемент списка `premise-nogoods` посылки — это множество посылок, такое что, если признать их все, текущую посылку признавать нельзя. Таким образом, если `amb-choose` найдет для посылки полностью поддержанное нехорошее множество, в эту посылку верить нельзя.

Если посылка, поддерживающая истинное значение, либо посылка, поддерживающая ложное значение, доступны для признания, `amb-choose` делает, соответственно, утверждение об истинности или ложности. Если нельзя верить ни в одну из двух посылок, она переходит к обработке противоречий более высокого уровня (`process-contradictions`) в надежде, что после перестройки статуса доверия других посылок, когда текущий распространитель снова будет запущен, можно будет сделать утверждение об истинности или ложности.

Аргументом процедуры `process-contradictions` служит множество нехороших множеств, порождаемое через `pairwise-union`. Каждое из этих нехороших множеств является объединением набора посылок, исключающих выбор истины, и набора посылок, исключающих выбор лжи. Таким образом, любое из этих нехороших множеств предотвращает выбор любой из альтернатив[14]:

```
(define (pairwise-union nogoods1 nogoods2)
  (append-map (lambda (nogood1)
                (map (lambda (nogood2)
                       (support-set-union nogood1 nogood2))
                     nogoods2))
              nogoods1))
```

Обучение на основе противоречий

Процедура `process-contradictions` сохраняет все полученные ей нехорошие множества и распространяет информацию в этих множествах по нехорошим множествам отдельных посылок. Затем она выбирает, в какое из нехороших множеств следует перестать верить, отозвав одну из его гипотетических посылок, если таковая имеется.

```
(define (process-contradictions nogoods complaining-cell)
  (update-failure-count!)
  (for-each save-nogood! nogoods)
  (let-values (((to-disbelieve nogood)
                (choose-premise-to-disbelieve nogoods)))
    (maybe-kick-out to-disbelieve nogood complaining-cell)))
```

[14] Такое использование процедуры `pairwise-union` реализует логическое правило *сечения*, обобщение *модус поненс*. В исчислении высказываний правило сечения записывается как $(A \vee B) \wedge (\neg B \vee C) \vdash (A \vee C)$. В сочетании с унификацией (раздел 4.4) это правило служит основой знаменитого алгоритма резолюции для доказательства теорем, который изобрел Робинсон [104].

Процедура `save-nogood!` дополняет поле `premise-nogoods` каждой посылки в данном ей множестве `nogood` множеством остальных посылок, с которыми она несовместима. Таким образом система учится на своих прошлых неудачах. Обновляемая посылка не включается в свои собственные нехорошие множества, поскольку сама с собой она не может быть несовместима.

```
(define (save-nogood! nogood)
  (for-each (lambda (premise)
              (set-premise-nogoods! premise
                (adjoin-support-with-subsumption
                 (support-set-remove nogood premise)
                 (premise-nogoods premise))))
            (support-set-elements nogood)))
```

Новое нехорошее множество для посылки может покрывать либо покрываться одним из старых нехороших множеств той же посылки; наиболее полезны минимальные нехорошие множества.

Разрешение противоречий

Чтобы разрешить противоречие, требуется отозвать одну из посылок нехорошего множества, поддерживающего это противоречие. Отозваны могут быть только гипотетические посылки, которые были приняты «для поддержки рассуждения». Если противоречие поддерживает более одного нехорошего множества, мы выбираем то, где меньше гипотетических посылок, поскольку отвержение маленького нехорошего множества уничтожает больше возможностей, чем отвержение нехорошего множества с большим количеством гипотез.

```
(define (choose-premise-to-disbelieve nogoods)
  (choose-first-hypothetical
   (car (sort-by nogoods
          (lambda (nogood)
            (count hypothetical?
                   (support-set-elements nogood)))))))
```

Однако выбор гипотезы для отзыва из конкретного нехорошего множества неочевиден. Здесь мы произвольным образом берем первую гипотетическую посылку.

```
(define (choose-first-hypothetical nogood)
  (let ((hyps (support-set-filter hypothetical? nogood)))
    (values (and (not (support-set-empty? hyps))
                 (car (support-set-elements hyps)))
            nogood)))
```

Процедура `maybe-kick-out` завершает работу по разрешению противоречия. Если процедура выбора нашла подходящую гипотезу для отвержения, эта гипотеза отзывается, и распространение продолжается обычным образом. Если нет, процесс распространения останавливается и оповещает пользователя о противоречии.

```
(define (maybe-kick-out to-disbelieve nogood cell)
  (if to-disbelieve
      (mark-premise-out! to-disbelieve)
      (abort-process (list 'contradiction cell))))
```

Обнаружение противоречия в ячейке

Если при добавлении в ячейку нового содержимого обнаружено противоречие, недовольная ячейка вызывает процедуру `handle-cell-contradiction` и передает ей себя в качестве аргумента. К этому моменту наиболее сильным значением в ячейке будет объект-противоречие, а множеством поддержки этого объекта будет вызвавшее его нехорошее множество. Его следует направить в `process-contradictions` для обработки.

```
(define (handle-cell-contradiction cell)
  (let ((nogood (support-layer-value (cell-strongest cell))))
    (process-contradictions (list nogood) cell)))
```

Вышеописанного достаточно для поддержки перебора с возвратами, управляемого зависимостями.

Недвоичный `amb`

Конструкцию `binary-amb` можно использовать для формулировки многих задач. Но большинство альтернатив не являются бинарными. Можно построить механизм n-местного выбора на основе `binary-amb` через цепь условных распространителей, управляемых ячейками, а истинные или ложные значения этих ячеек связать с двоичными распространителями выбора. Однако такая конструкция неэффективна и использует слишком много излишних деталей. Поэтому мы вводим элементарный механизм n-местного выбора процедурой `p:amb`. Процедура `p:amb` аналогична `binary-amb`. В случае `binary-amb` у нас есть ровно две альтернативы, `#t` и `#f`, и каждую из них поддерживает гипотетическая посылка. Когда к ячейке применяется `p:amb` со списком возможных значений, процедура `make-hypotheticals` добавляет эти значения к ячейке, и каждое из них поддерживается новой гипотетической посылкой.

Когда активируется распространитель, построенный внутри `p:amb`, вызывается процедура `amb-choose`. Сначала она пытается найти среди своих гипотез такую, которая не исключается своими нехорошими множествами `premise-nogoods`. Если таковая имеется, она переводится в состояние `in`, а остальные — в состояние `out`; таким способом мы выбираем значение, связанное с этой гипотезой, в качестве значения ячейки. Если невозможно поверить ни в одну из гипотетических посылок, все посылки переводятся в состояние `out`, строится новый набор нехороших множеств и передается в `process-contradictions`, которая, если возможно, отзовет одну из гипотетических посылок в одном из этих множеств. Обобщением процедуры `pairwise-union` на более чем два множества служит процедура `cross-product-union`. Как и раньше, она является шагом резолюции.

```
(define (p:amb cell values)
  (let ((premises (make-hypotheticals cell values)))
    (define (amb-choose)
      (let ((to-choose
             (find (lambda (premise)
                     (not (any all-premises-in?
                               (premise-nogoods premise))))
                   premises)))
```

```
      (if to-choose
          (for-each (lambda (premise)
                      (if (eq? premise to-choose)
                          (mark-premise-in! premise)
                          (mark-premise-out! premise)))
                    premises)
          (let ((nogoods
                 (cross-product-union
                  (map (lambda (premise)
                         (filter all-premises-in?
                                 (premise-nogoods premise)))
                       premises))))
            (for-each mark-premise-out! premises)
            (process-contradictions nogoods cell)))))
  (let ((me (propagator (list cell) (list cell)
                        amb-choose 'amb)))
    (set! all-amb-propagators
          (cons me all-amb-propagators))
    me)))
```

Распространители выбора, построенные через `p:amb`, вводят ровно столько гипотетических посылок, сколько вариантов передано. Конструкции для выбора из $n > 2$ альтернатив на основе `binary-amb` создают примерно вдвое больше посылок.

7.5.2 Решение комбинаторных задач

Чтобы показать, как поиск, управляемый зависимостями, используется для эффективного решения комбинаторных задач, рассмотрим знаменитую «загадку о расселении» [29]:

> Бейкер, Купер, Флетчер, Миллер и Смит живут на разных этажах пятиэтажного дома. Бейкер живет не на верхнем этаже. Купер живет не на нижнем этаже. Флетчер живет не на верхнем и не на нижнем этаже. Миллер живет выше, чем Купер. Смит не живет на соседнем этаже с Флетчером. Флетчер не живет на соседнем этаже с Купером. Кто где живет?

Можно построить на основе этой загадки задачу для распространителей. Вот весьма несложная формулировка:

```
(define (multiple-dwelling)
  (let-cells (baker cooper fletcher miller smith)
    (let ((floors '(1 2 3 4 5)))
      (p:amb baker floors)    (p:amb cooper floors)
      (p:amb fletcher floors) (p:amb miller floors)
      (p:amb smith floors)
      (require-distinct
       (list baker cooper fletcher miller smith))
      (let-cells ((b=5 #f) (c=1 #f) (f=5 #f)
                  (f=1 #f) (m>c #t) (sf #f)
```

```
            (fc #f)  (one 1)  (five 5)
            s-f  as-f  f-c  af-c)
      (p:= five baker b=5)      ;Бейкер не на пятом.
      (p:= one cooper c=1)      ;Купер не на первом.
      (p:= five fletcher f=5)   ;Флетчер не на пятом.
      (p:= one fletcher f=1)    ;Флетчер не на первом.
      (p:> miller cooper m>c)   ;Миллер выше Купера.
      (c:+ fletcher s-f smith)  ;Флетчер и Смит
      (c:abs s-f as-f)          ; не живут на
      (p:= one as-f sf)         ; соседних этажах.
      (c:+ cooper f-c fletcher) ;Купер и Флетчер
      (c:abs f-c af-c)          ; не живут на
      (p:= one af-c fc)         ; соседних этажах.
      (list baker cooper fletcher miller smith)))))
```

Этот код говорит, что Бейкер, Купер, Флетчер, Миллер и Смит живут каждый на одном из пяти этажей, и все эти этажи различны. Затем идут ограничения их выбора, выраженные в виде графа распространителей. Некоторые ячейки, например `b=5`, инициализируются булевым значением. Таким образом, строка `(p:= five baker b=5)` представляет условие, что Бейкер не живет на пятом этаже. Условие, что Купер и Флетчер не живут на соседних этажах, реализуется присваиванием ячейке `fc` значения `#f` и последними тремя ограничениями.

Для использования системы распространителей нам нужно определить элементарные распространители и соответствующую структуру слоев данных:

```
(define (setup-propagator-system arithmetic)
  (define layered-arith
    (extend-arithmetic layered-extender arithmetic))
  (install-arithmetic! layered-arith)
  (install-core-propagators! merge-value-sets
                             layered-arith
                             layered-propagator-projector))
```

Эта довольно сложная процедура инициализации задает информацию, позволяющую построить и установить распространители с арифметикой, снабженной слоями, которые можно отслеживать, и причинами, облегчающими отладку. По умолчанию в момент загрузки системы распространителей мы используем арифметику для числовых данных:

```
(setup-propagator-system numeric-arithmetic)
```

Теперь можем запустить пример с загадкой:

```
(initialize-scheduler)

(define answers (multiple-dwelling))
(run)
(map (lambda (cell)
       (get-base-value (cell-strongest cell)))
     answers)
;Value: (3 2 4 5 1)
```

```
*number-of-calls-to-fail*
;Value: 106
```

Мы видим (правильный) ответ: на каком этаже живет каждый персонаж. Кроме того, мы видим, что для поиска правильного распределения этажей требуется около 100 неудачных попыток[15]. Оказывается также, что это решение единственное: ни одно другое присваивание не удовлетворяет перечисленным условиям.

Заметим, что полное число неограниченных присваиваний равно $5^5 = 3125$, однако мы решаем задачу всего за примерно 100 попыток. Это нам удается потому, что система обучается на своих ошибках: для каждой неудачи она сохраняет информацию, какие наборы посылок не могут одновременно выполняться. При правильном использовании эта информация предотвращает рассмотрение вариантов, которые, согласно прошлым экспериментам, оказались безнадежными.

Упражнение 7.5. Загадка об имени яхты

Сформулируйте и решите с помощью распространителей следующую загадку[16]:

> У отца Мэри Энн Мур есть яхта, и яхты есть также у четырех его друзей: полковника Даунинга, мистера Холла, сэра Барнакла Худа и доктора Паркера. У каждого из них есть также по дочери, и каждый назвал свою яхту в честь дочери одного из своих друзей. Яхта сэра Барнакла называется «Габриэлла», у мистера Мура «Лорна», а у мистера Холла «Розалинда». «Мелисса», яхта полковника Даунинга, названа в честь дочери сэра Барнакла. Отец Габриэллы владеет яхтой, названной в честь дочери доктора Паркера. Кто отец Лорны?

Упражнение 7.6. Загадка о расселении

Задачу о расселении нетрудно сформулировать для вычислителя с `amb` из раздела 5.4. Это даже проще, чем в системе распространителей, поскольку можно думать и писать в терминах выражений. Так, условие, что Флетчер и Купер не живут на соседних этажах, можно записать как:

```
(require (not (= (abs (- fletcher cooper)) 1)))
```

а не

```
(c:+ cooper f-c fletcher)
(c:abs f-c af-c)
(p:= one af-c fc)
```

где нужно еще объявить ячейки `f-c`, `af-c` и `fc`, а также проинициализировать `one` и `fc`. Это потому, что система распространителей — общая система описания графов, а не выражений.

a. Сформулируйте и решите загадку о расселении с помощью вычислителя `amb` из раздела 5.4. Сделайте так, чтобы система подсчитывала число неудач. Сколько неудачных попыток потребовалось для решения?

[15] Точное число неудачных попыток присваивания зависит от подробностей вычисления. В этой задаче число неудачных попыток варьируется примерно от 60 до 200, в зависимости от порядка активации распространителей. Однако для этой формулировки задачи в среднем число неудач близко к 100.

[16] Эта загадка позаимствована из книжки под названием «Загадочные развлечения», напечатанной в 1960-х гг. издательством «Литтон индастриз». В книжке эта загадка приписывается газете «Кэнзас стейт энджинер».

b. Напишите небольшой компилятор, преобразующий ограничения, записанные в виде выражений, во фрагменты диаграмм распространителей. Вы увидите, что это очень простая задача. Первую попытку это сделать мы предпринимали в упражнении 7.1 на стр. 298. Однако теперь нам нужен настоящий транслятор из кода раздела 5.4 в язык распространителей. Покажите, что ваш компилятор порождает правильный результат.

c. Сколько неудач происходит при решении задачи с диаграммой распространителей, порожденной вашим компилятором? Если это число больше 200, у вас получается очень плохой код!

Упражнение 7.7. Снова головоломка про карты

Переделайте упражнение 5.17 на основе распространителей.

Упражнение 7.8. Вывод типов

В разделе 4.4.2 мы в качестве примера приложения для унификации построили механизм вывода типов. В этом упражнении (на самом деле это серьезный проект) мы реализуем вывод типов с использованием преимуществ распространителей.

a. Пусть дана программа на Scheme. Постройте сеть распространителей, в которой каждой позиции, где имеет смысл присвоить тип, соответствует по ячейке. Каждая такая ячейка будет хранилищем информации об этой точке программы. Постройте распространители, связывающие эти ячейки и задающие типовые ограничения, диктуемые структурой программы. Используйте унификацию в качестве операции `cell-merge`. Унификация может выдавать противоречие, если присвоить типы в программе невозможно.

b. В программе могут присутствовать ячейки, где тип недостаточно ограничен типами соседних ячеек. Однако можно стимулировать распространение информации, если в такую ячейку поместить типовую переменную и позволить этой переменной собирать ограничения через распространение. Это называется «плюхнуть» (plunk) переменную. Попробуйте.

c. В сложных случаях вывод типов может потребовать угадывания (гипотетических построений) и перебора с возвратами при противоречиях. Приведите примеры, где это необходимо.

d. Отслеживание посылок и причин позволяет строить информативные сообщения об ошибках. Однако для этого каждое место в программе нужно связать с соответствующей ячейкой, чтобы информацию, полученную при распространении, можно было соотнести с аннотируемой программой. Для построения двунаправленной связи между точкой в программе и ячейкой можно использовать любую разновидность «ярлыков». В любом случае постарайтесь порождать понятные объяснения, почему та или иная точка программы имеет тот или иной тип, а также почему типы в программе построить не удалось.

e. Можно ли вашу реализацию вывода типов использовать на практике? Почему? Если нет, как можно ее улучшить?

Мораль этой истории

Решать комбинаторные загадки интересно и забавно, но настоящее значение того, что мы сделали, состоит не в этом. «SAT-решатели» играют большую роль в работе с задачами такого рода. Однако для проектирования вычислительных систем

можно сделать более глубокие выводы. Обобщив программирование от работы со структурами выражений на графовые диаграммы (которые могут быть неудобны — но это неудобство можно смягчить компиляцией), мы сумели ввести в программы недетерминистский выбор естественным и эффективным образом. Можно вводить гипотетические предположения, позволяющие получить альтернативные значения, поддерживаемые утверждениями, которые можно затем безболезненно отбросить. Это позволяет нам правильно работать с задачами вроде квадратных уравнений. Они действительно имеют по два решения, и всякое вычисление, основанное на выборе одного из них, имеет право его отвергнуть, в то время как другое решение может после долгих вычислений привести к приемлемому результату. Например, если `p:sqrt` традиционным образом вычисляет положительный квадратный корень действительного числа, можно написать конструктор направленного распространителя `p:honest-sqrt` со входной ячейкой `x^2` и выходной ячейкой `x`, который дает своим пользователям (скрытый) выбор квадратных корней:

```
(define-p:prop (p:honest-sqrt (x^2) (x))
  (let-cells (mul +x)
    (p:amb mul '(-1 +1))
    (p:sqrt x^2 +x)
    (p:* mul +x x)))
```

Здесь важно то, что такой выбор может вводиться без того, чтобы окружающий код знал и умел обрабатывать неоднозначность. Например, распространитель ограничений, связывающий числа с их квадратами, может просто ссылаться на `p:honest-sqrt`:

```
(define-c:prop (c:square x x^2)
  (p:square x x^2)
  (p:honest-sqrt x^2 x)))
```

7.6 Распространители поддерживают дублирование

При проектировании любой системы заметного размера для каждой компоненты на каждом уровне детализации предлагается по несколько возможных вариантов реализации. Однако, когда система наконец создана, это разнообразие планов теряется; обычно принятым и реализованным оказывается только один объединенный план. Как и в экологической системе, эта потеря разнообразия в традиционном инженерном процессе приводит к серьезным последствиям.

Дублирование редко встраивается в программы. Отчасти это происходит потому, что оно дорого, а отчасти из-за того, что не существует общепринятых формальных механизмов, помогающих его использовать. Однако идея распространителей естественным образом ведет к механизму, поддерживающему дублирование. Использование в ячейках структур с частичной информацией (введенных в работе Радула и Сассмана [99]) позволяет сливать различные и, возможно, перекрывающиеся источники информации. Мы продемонстрировали это на интервалах в примере с расстоянием до звезд в разделе 7.1. Однако способов комбинировать частичную информацию существует много: частично заданные символьные выражения можно сливать через унификацию, как показано в разделе 4.4.2. Идея частичной информации не ограничивается системами, построенными на распространителях, но в системе с распространителями, как предлагается делать в упражнении 7.8 на стр. 321,

у нас есть парадигма, позволяющая легко сочетать вклад различных независимых механизмов.

Подобным образом мы рассматриваем еще одну идею дублированных систем из мира задач искусственного интеллекта: запуск вычислений, управляемый целью. Тут мысль состоит в том, чтобы вместо указания, «как» достигнуть цели через имя процедуры, достигающей ее, мы говорим, «что» требуется достигнуть, и связываем с целью процедуры, потенциально способные помочь в ее достижении. Часто для этой связи используется сопоставление с образцом, однако это лишь одна из возможных методик, а не суть подхода[17]. Если существует несколько способов достижения цели, то место, где из этих способов выбирается одна подходящая процедура, можно зарегистрировать как точку выбора в переборе с возвратами. Однако хронологический перебор, ограниченный потоком управления в языке, ориентированном на выражения, чрезвычайно неэффективен. Для того чтобы перебор, управляемый зависимостями, хорошо работал, требуется выйти за рамки структуры вычислений, основанной на выражениях, и в этом направлении распространители — один из возможных шагов. Перебор по-прежнему имеет экспоненциальную сложность, однако комбинаторика поиска существенно урезается, поскольку многие неудачные ходы исключаются через нехорошие множества, полученные на прошлом опыте.

Разумеется, помимо выбора конкретного способа достижения цели через поиск с возвратами, существуют другие способы запуска дублированных методов. Можно, например, запускать в параллель несколько способов решения задачи и выбирать тот, который финиширует первым.

Допустим, у нас есть несколько независимо написанных процедур, предназначенных для решения одного и того же (неточно специфицированного) общего класса задач. Допустим, что каждый из методов более или менее разумен и правильно работает для большинства конкретных задач, встречающихся на практике. Тогда мы знаем, что можно получить более устойчивое целое, если собрать все эти процедуры в более крупную систему, которая независимо запускает каждую из них, сравнивает результаты и выбирает наилучший. Если для комбинации у нас есть независимый способ выбрать самый приемлемый ответ, все просто замечательно. Но, даже если решать приходится голосованием, получается система, способная надежно покрыть более широкое пространство решений. Более того, если такая система автоматически регистрирует в журнале все случаи, когда один из планов терпит неудачу, эту обратную связь можно будет использовать для улучшения характеристик неудачной процедуры.

Эту стратегию дублирования можно использовать на всех уровнях организации. Каждая компонента каждой подсистемы может также быть продублирована. Если компоненты разделяются между подсистемами, получается мощная управляемая избыточность. Но можно сделать и еще лучше. Можно собрать механизм проверки на согласованность промежуточных результатов в независимо разработанных подсистемах даже в том случае, когда никакое отдельное значение одной подсистемы напрямую не соответствует никакому отдельному значению другой подсистемы.

В качестве простого примера представим себе, что у нас есть две подсистемы и они должны породить один и тот же результат, вычисленный двумя совершенно различными способами. Допустим, их проектировщики согласны, что на некоторой

[17] *Запуск процедур, управляемый образцом*, был изобретен Карлом Хьюиттом в системе PLANNER [56] и Аленом Кольмероэ в языке Prolog [78]. Затем эта идея распространилась на многие другие системы и языки.

стадии в одном проекте произведение каких-то двух переменных должно совпадать с суммой двух других переменных в другом проекте[18]. Нет никакой причины не вычислить этот предикат, как только станут известны четыре величины, от которых он зависит; таким образом, мы получим проверку разумности поведения во время выполнения и важную отладочную информацию для разработчиков. Проверку можно организовать как встроенную локальную сеть ограничений.

[18]Это совершенно реальный пример. В вариационной механике сумма лагранжиана системы и гамильтониана, связанного с ним преобразованием Лежандра, равна скалярному произведению 1-формы обобщенного момента и вектора обобщенной скорости.

Глава 8

Эпилог

Техника всерьез существует всего лишь несколько тысяч лет. Все наши попытки обдуманно строить очень сложные надежные системы пока в лучшем случае являются незрелыми. Нам еще предстоит извлечь уроки, усвоенные биологической эволюцией на протяжении нескольких миллионов лет.

До сих пор нас больше заботила эффективность и правильность построений, чем та надежность биологических систем, которая происходит из оптимизации способности к развитию, гибкости и устойчивости к атакам. Такой подход разумен при производстве критически важных систем, у которых ресурсов едва хватает на выполнение своих функций. Однако быстрое развитие микроэлектроники снизило для многих приложений остроту ресурсной проблемы. В то же время наша растущая зависимость от вычислительной и коммуникационной инфраструктуры и появление все более изощренных атак на эту инфраструктуру заставляют нас уделять больше внимания гибкости.

Мы не призываем использовать биомиметику, однако наблюдение за биологическими системами дает примеры того, как можно включить в инженерную практику мощные принципы, обеспечивающие устойчивость. Многие из этих принципов прямо конфликтуют с установившимися обычаями оптимизировать эффективность и с желанием доказать правильность построений. В этой книге мы намеренно нарушаем эти обычаи и исследуем, насколько возможно оптимизировать гибкость. Наш подход мотивирован наблюдением, что большинство систем, выдержавших испытание временем, имеют в основе набор специализированных языков, каждый из которых позволяет облегчить построение некоторых фрагментов системы.

В процессе попыток построить обладающие разумом символические системы сообщество исследователей искусственного интеллекта в качестве побочного эффекта разработало технические инструменты, которые могут служить основой проектирования гибких и устойчивых систем. Например, перебор с возвратами можно рассматривать не как метод организации поиска, а как способ увеличить применимость компонент сложной системы, организованной для выполнения заданных извне ограничений. Мы считаем, что при таком новом синтетическом подходе наши аппаратные и программные системы станут лучше.

В главе 2 для начала обсуждения мы привели несколько практически не вызывающих возражения универсальных приемов. Мы продемонстрировали стратегию построения систем комбинаторов — библиотек, состоящих из параметрических ком-

понент со стандартизованными интерфейсами. Такие компоненты можно сочетать множеством способов и удовлетворять множество различных требований. Мы показали, как с помощью этой идеи упростить построение системы сопоставителей в языке регулярных выражений. Мы ввели понятие оберток, которые помогают приспосабливать компоненты приложений к иным стандартам, чем те, на которые эти компоненты были исходно рассчитаны, и с помощью этого понятия построили язык для преобразования единиц измерения. Затем мы создали интерпретатор языка, позволяющего выражать правила настольных игр вроде шашек.

В главе 3 мы пустились в захватывающее и опасное приключение: исследование, чего можно добиться, если позволить настройку значений элементарных процедур языка. Мы расширили арифметику и позволили ей работать с символическими выражениями и функциями, а также с помощью механизма расширений включили в арифметику автоматическое дифференцирование в прямом режиме. Расширения такого рода опасны, однако при должной осторожности с их помощью можно ввести в программы новые возможности, не затрагивая старую функциональность. Для того, чтобы эта стратегия работала эффективно и имела еще большую мощность, мы исследовали типы, определяемые пользователем, с объявляемыми отношениями подтипирования, и с их помощью построили простую, но легко расширяемую игру-бродилку.

Введенные в главе 4 сопоставление с образцом и вызов процедур, управляемый образцом, являются чрезвычайно важными приемами при построении специализированных языков. Вначале мы создали правила переписывания термов для упрощения алгебраических выражений. Затем мы продемонстрировали изящную стратегию компиляции образцов в структуры, составленные из элементарных сопоставителей, в системе комбинаторов сопоставления с образцом. Затем набор инструментов для сопоставления был расширен, и мы разрешили использовать переменные на обеих сторонах сопоставления; таким образом, была реализована унификация. С ее помощью мы создали элементарную систему вывода типов. Наконец, мы построили сопоставители, работающие с произвольными графами, а не только с деревьями выражений, и выразили в терминах графов и графового сопоставления правила ходов в шахматах.

Поскольку все разумные компьютерные языки обладают свойством универсальности, программисты не имеют права жаловаться, что решение некоторой задачи в некотором языке невозможно выразить. Если очень нужно, хороший программист способен написать компилятор или интерпретатор любого языка на том, в котором он вынужден работать. Это не очень сложная задача, и при этом, вероятно, самый сильный ход из доступных программисту. В главе 5 мы показали, как с помощью интерпретации и компиляции строить все более мощные языки. Начали мы с простого интерпретатора для Scheme-подобного языка с аппликативным порядком вычислений. Ради легкости расширения этот интерпретатор строился на полиморфных процедурах. Затем мы добавили в язык возможность объявлять в заголовках процедур ленивые формальные параметры. Затем скомпилировали язык в комбинацию исполнительных процедур — в систему комбинаторов. Затем добавили модель недетерминистского вычисления с оператором `amb`. Наконец, мы показали, как, имея доступ к нижележащим продолжениям, можно получить возможности `amb` на основе встроенной реализации языка Scheme.

В главе 6 мы начали исследование многослойных вычислений, основанных на новом механизме, тесно связанном с понятием полиморфной процедуры. Например,

мы изменили арифметику языка так, чтобы программу, вычисляющую численный результат на основе численных аргументов, можно было без модификации расширить и получить те же самые результаты, дополненные единицами измерения. Единицы измерения в результате автоматически выводятся из единиц измерения входных данных, а вычисления проверяются на правильность использования единиц: добавление 5 килограмм к 2 метрам приведет к сообщению об ошибке. Тот же самый механизм слоев мы использовали, чтобы снабдить программы отслеживанием зависимостей, так, что результат автоматически ссылается на источники данных, на основании которых он получен.

Модель распространителей из главы 7, в сущности, является способом мышления об устройстве больших систем. Несмотря на то, что в примерах, используемых нами в этой главе, все распространители являются простыми арифметическими функциями и отношениями, идея обладает значительно большей общностью. Распространитель может быть реализован программно или аппаратно. Он может быть простой функцией или громадным компьютером, перемалывающим тонны данных. Если это программа, он может быть написан на любом языке. В сущности, система распространителей не обязана быть гомогенной. Различные распространители могут строиться по-разному. Ячейки могут быть приспособлены к хранению различных видов информации, и слияние этой информации они могут организовывать каждая своим предпочтительным способом. Связь между распространителями и ячейками может быть устроена как сигналы внутри микросхемы или как пересылка данных по глобальной сети. Важен только протокол, позволяющий распространителю запрашивать информацию у ячейки и добавлять к ней новую информацию.

Мы изложили в этой книге множество программистских идей. Теперь ваше право оценивать эти идеи и, возможно, применять.

Приложение A

Программное обеспечение

Весь код, встречающийся в этой книге, и всю программную инфраструктуру, поддерживающую его, можно скачать в виде архива по адресу
`http://groups.csail.mit.edu/mac/users/gjs/sdf.tgz`
Архив устроен как дерево каталогов, и каждый подкаталог примерно соответствует разделу этой книги. Программы работают в среде MIT/GNU Scheme версии 10.1.10 или более поздней; скачать ее можно по адресу
`http://www.gnu.org/software/mit-scheme`
В программах используются некоторые характерные особенности реализации MIT/GNU, так что в других дистибутивах Scheme они работать не будут. Перенос на другие реализации должен быть возможен, но мы не пытались это сделать; наверняка потребуется определенная работа. Поскольку код является свободным программным обеспечением (под лицензией GPL), вы можете его модифицировать и распространять.

Архив представляет собой `tar`-файл с именем `sdf.tgz`. Его можно распаковать командой

```
tar xf .../sdf.tgz
```

Эта команда создает подкаталог `sdf` внутри того каталога, где она была выполнена.

Основным интерфейсом к архиву кода служит управляющая программа, поставляемая вместе с архивом. Чтобы использовать ее, запустите MIT/GNU Scheme и выполните команду

```
(load ".../sdf/manager/load")
```

где `.../` — имя каталога, где распакован архив. Управляющая программа создает в глобальном окружении одно определение по имени `manage`. После того, как она загружена, управляющую программу перечитывать с диска не требуется, если только не запущен новый экземпляр Scheme.

Допустим, вы работаете с разделом 4.2 «Переписывание термов» и желаете поэкспериментировать с программой или поработать над упражнением. Загрузчик кода для этого раздела расположен в подкаталоге `.../sdf/term-rewriting`; там же находятся и другие относящиеся к этому разделу файлы. Однако знать, как именно работает загрузчик, не обязательно. (Разумеется, никто не мешает вам читать код управляющей программы. Он достаточно интересный.)

Команда `manage`

```
(manage 'new-environment 'term-rewriting)
```

создаст новое окружение верхнего уровня, загрузит все необходимые файлы из этого раздела и запустит управляющий цикл в новом окружении. После того, как вы закончите работу с этим разделом, можно будет загрузить другой раздел командой `manage`, заменив `term-rewriting` на имя, соответствующее этому новому разделу.

Обычно в качестве аргумента команды `(manage 'new-environment ...)` можно использовать имя подкаталога. В таком контексте имя подкаталога называется *разновидностью* (flavor). Однако некоторые подкаталоги содержат по несколько разновидностей, и в таких случаях имена разновидностей отличаются от имен подкаталогов.

Соответствия между разделами книги и именами подкаталогов/разновидностей в архиве можно найти в файле

```
.../manager/sections.scm
```

Кроме того, есть еще два особых подкаталога: `common` содержит общие файлы, используемые отовсюду, а в `manager` лежит реализация команды `manage`.

Команда управления программным обеспечением `manage` имеет множество других полезных возможностей. Среди них управление рабочими окружениями по имени, поиск файлов, где определяется и используется какое-либо имя, и запуск юнит-тестов. Информацию о них можно найти в документации внутри подкаталога `manager`.

При использовании программ могут потребоваться дополнительные шаги, не описанные в тексте книги, например инициализация. В каждом подкаталоге имеются тесты: всякий файл с именем `test-`*`FOO`*`.scm` представляет собой «стандартный» тест, использующий инфраструктуру тестирования, подобную имеющейся в других языках программирования. Кроме того, файлы `load-spec` в каждом подкаталоге могут содержать ссылки на тесты, отмеченные символом `inline-test?`. Эти тесты пользуются другой инфраструктурой, похожей на записи протоколов управляющего цикла. В этих тестах можно найти примеры, как следует запускать программы.

Приложение B

Scheme

> Языки программирования следует проектировать не нагромождая конструкции друг на друга, а устраняя слабости и ограничения, которые делают дополнительные конструкции необходимыми. Scheme показывает, что достаточно весьма малого числа правил построения выражений без всяких ограничений на сочетание этих выражений, чтобы построить практичный и эффективный язык программирования, гибкости которого хватает для поддержки большинства основных используемых в настоящее время парадигм программирования.
>
> Стандарт языка Scheme IEEE [61], стр. 3

Здесь мы поместили элементарное введение в диалект Scheme языка программирования Lisp. Более полное введение можно найти в книге *Структура и интерпретация компьютерных программ* [1].

Более точное описание языка содержится в стандарте IEEE [61] и «Семикратно пересмотренном сообщении об алгоритмическом языке Scheme» (R7RS) [109].

Некоторые программы в этой книге зависят от нестандартных конструкций MIT/GNU Scheme; документацию по этой системе можно найти в «Справочном руководстве по MIT/GNU Scheme» [51]. Кроме того, в случае конструкций Scheme, для которых есть отдельные обсуждения, индекс в Справочном Руководстве дает ссылки на соответствующие документы.

B.1 Базовые конструкции Scheme

Scheme — простой язык программирования, основанный на выражениях. Выражение служит именем значения. Например, числовая константа `3.14` служит именем приближения известного числа, а числовая константа `22/7` — имя другого приближения к тому же числу. Существуют элементарные выражения, скажем числовые константы, которые мы распознаем непосредственно, а также составные выражения нескольких видов.

Составные выражения ограничиваются круглыми скобками. Выражения, начинающиеся с выделенных ключевых слов, вроде `if`, называются *особыми формами*. Выражения, не являющиеся особыми формами, называются *комбинациями* и обозначают применение процедур к аргументам.

Комбинации

Комбинация, называемая также *применение процедуры*, представляет собой последовательность выражений, ограниченных скобками:

(*оператор операнд-1 ... операнд-n*)

Первое подвыражение в комбинации, называемое *оператор*, рассматривается как именование процедуры, а остальные подвыражения, называемые *операнды*, рассматриваются как именования аргументов этой процедуры. Значение, возвращаемое процедурой при применении к данным аргументам, является значением комбинации. Например,

```
(+ 1 2.14)
3.14

(+ 1 (* 2 1.07))
3.14
```

— это две комбинации, именующие то же самое число, что и числовая константа 3.14[1]. В этих случаях символы `+` и `*` служат именами процедур которые, производят, соответственно, сложение и умножение чисел. Если в любом выражении мы заменим любое подвыражение на другое выражение, именующее тот же самый объект, то объект, именуемый объемлющим выражением, не меняется.

Заметим, что в языке Scheme все скобки имеют значение: нельзя ни добавлять новые скобки, ни убирать имеющиеся.

Лямбда-выражения

Так же, как мы с помощью числовых констант именуем числа, с помощью выражений `lambda` мы именуем процедуры[2]. Например, функция, возводящая свой аргумент в квадрат, может быть записана как

```
(lambda (x) (* x x))
```

[1] В примерах мы приводим значение, которое должна напечатать система Scheme, показывая его вслед за вводимым выражением и используя *курсивный* шрифт.

[2] Логик Алонсо Чёрч [16] придумал λ-нотацию для задания анонимных функций с именованным параметром: λx[выражение, зависящее от x]. Это читается так: «Функция одного аргумента, получаемая подстановкой этого аргумента вместо x в указанном выражении.»

Это выражение можно прочитать как «процедура одного аргумента x, умножающая x на x». Разумеется, мы можем использовать это выражение в любом контексте, где требуется процедура. Например,

```
((lambda (x) (* x x)) 4)
16
```

Общая форма выражения `lambda` имеет вид:

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

где *формальные-параметры* (обычно) представляют собой заключенный в скобки список символов, которые будут именами формальных параметров процедуры. Когда процедура применяется к аргументам, формальные параметры получают в качестве значений аргументы. *Тело* — это выражение, которое может ссылаться на формальные параметры. Значением применения процедуры будет значение ее тела, где на место формальных параметров подставлены аргументы[3].

В приведенных выше примерах символ `x` — единственный формальный параметр процедуры, именованной выражением `(lambda (x) (* x x))`. Эта процедура применяется к значению числовой константы 4, так что в теле `(* x x)` символ `x` имеет значение 4, а значение комбинации `((lambda (x) (* x x)) 4)` равно 16.

Выше мы сказали «обычно», потому что существуют исключения. Некоторые процедуры, скажем та, которая перемножает числа и именуется символом `*`, могут принимать произвольное число аргументов. Мы объясним, как это делается, позже (на стр. 339).

Определения

С помощью особой формы `define` можно дать имя любому объекту. Говорится, что имя задает *переменную*, чьим значением является объект. Например, если мы создадим определения:

```
(define pi 3.141592653589793)

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

то затем сможем использовать символы `pi` и `square` всюду, где могли бы стоять эти числовая константа и лямбда-выражение. Например, площадь поверхности сферы радиусом 5 равна

```
(* 4 pi (square 5))
314.1592653589793
```

Определения процедур могут выражаться более удобным образом с использованием «синтаксического сахара». Процедуру возведения в квадрат можно определить через:

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

[3] Мы говорим, что формальные параметры *связаны* с аргументами, а *сферой действия* этого связывания является тело процедуры.

что читается как «Для возведения x в квадрат умножьте x на x».

В языке Scheme процедуры являются *полноправными объектами*: их можно передавать как аргументы, возвращать как значения и включать в структуры данных. Например, можно создать процедуру, реализующую математическое понятие композиции двух функций[4]:

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

((compose square sin) 2)
.826821810431806

(square (sin 2))
.826821810431806
```

Следует заметить, что значения `f` и `g` в возвращаемой процедуре `(lambda (x) (f (g x)))` равны значениям формальных параметров объемлющей процедуры `(lambda (f g) ...)`. В этом сущность дисциплины лексической области действия языка Scheme. Значение каждой переменной получается путем поиска связывания в лексически видимом контексте. Кроме того, существует подразумеваемый контекст для всех глобально определенных в системе переменных. (Например, символ `+` в глобальном контексте связан с процедурой сложения чисел.)

Используя синтаксический сахар, показанный выше на примере `square`, можно более удобно записать определение `compose`:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

В системе MIT/GNU Scheme этот сахар можно употребить рекурсивно и написать:

```
(define ((compose f g) x)
  (f (g x)))
```

Иногда бывает полезно создать определение, локальное по отношению к другому определению. Например, `compose` можно записать так:

```
(define (compose f g)
  (define (fog x)
    (f (g x)))
  fog)
```

Имя `fog` не видно за пределами определения `compose`, поэтому в этом случае оно не особенно полезно, однако более обширные куски кода часто становятся проще для чтения, когда их внутренние фрагменты получают имена. Внутренние определения всегда должны стоять раньше, чем выражения в теле процедуры, не являющиеся определениями.

[4]Отступы в примерах добавлены, чтобы улучшить читаемость. Scheme не интересуется лишними пробелами, так что можно вставлять их сколько угодно, чтобы было удобнее читать текст программы.

Условные выражения

Условные выражения используют, когда из нескольких выражений нужно выбрать одно, которое получит значение. Например, можно написать процедуру, вычисляющую модуль числа:

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

Условное выражение `cond` содержит несколько *ветвей*. Каждая ветвь состоит из *выражения-предиката*, которое может быть истинным или ложным, и *выражения-следствия*. Значение выражения `cond` равно значению выражения-следствия первой ветви, для которой выражение-предикат истинно. Общая форма условного выражения такова:

```
(cond (предикат-1 следствие-1)
      ...
      (предикат-n следствие-n))
```

Для удобства существует особое ключевое слово `else`, которое можно использовать как предикат последней ветви `cond`.

Особая форма `if` представляет другой способ построить условное выражение, когда выбор производится только из двух альтернатив. Например, поскольку нечто особое нужно сделать только при отрицательном аргументе, можно определить `abs` таким образом:

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

В общем виде выражение `if` строится так:

```
(if предикат следствие альтернатива)
```

Если *предикат* истинен, значением выражения `if` будет значение *следствия*, иначе оно равно значению *альтернативы*.

Рекурсивные процедуры

При помощи условных выражений и определений можно записывать рекурсивные процедуры. Например, чтобы вычислить факториал n, пишем:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

(factorial 6)
720

(factorial 40)
815915283247897734345611269596115894272000000000
```

Локальные имена

Выражение `let` используется, чтобы давать имена объектам в локальном контексте. Например,

```
(define (f radius)
  (let ((area (* 4 pi (square radius)))
        (volume (* 4/3 pi (cube radius))))
    (/ volume area)))

(f 3)
1
```

Общая форма выражения `let` такова:

```
(let ((переменная-1 выражение-1)
      ...
      (переменная-n выражение-n))
  тело)
```

Значение выражения `let` равно значению выражения *тело* в контексте, где каждая *переменная-i* имеет значение *выражения-i*. *Выражения-i* не имеют права ссылаться на *переменные-j*, которым дается значение в том же выражении `let`.

Выражение `let*` подобно `let`, но только *выражение-i* может ссылаться на *переменные-j*, значение которым дано раньше в том же выражении `let*`.

Небольшое изменение в выражении `let` позволяет удобным способом записывать циклы. Процедуру, которая реализует альтернативный алгоритм вычисления факториала, можно выразить так:

```
(define (factorial n)
  (let factlp ((count 1) (answer 1))
    (if (> count n)
        answer
        (factlp (+ count 1) (* count answer)))))

(factorial 6)
720
```

Здесь символ `factlp`, стоящий после слова `let`, локально определен как процедура, принимающая в качестве формальных параметров переменные `count` и `answer`. Вначале, при инициализации цикла, она вызывается со значениями 1 и 1 в качестве аргументов. Каждый раз, когда впоследствии зовется процедура по имени `factlp`, эти переменные получают новые значения — значения выражений-операндов `(+ count 1)` и `(* count answer)`.

Эквивалентный способ определить эту процедуру — с помощью явно заданной внутренней процедуры:

```
(define (factorial n)
  (define (factlp count answer)
    (if (> count n)
        answer
        (factlp (+ count 1) (* count answer))))
  (factlp 1 1))
```

Процедура `factlp` определена локально, она существует только внутри тела `factorial`. Поскольку `factlp` лексически включается в определение `factorial`, значение переменной `n` в ее теле равно значению формального параметра процедуры `factorial`.

Составные данные — списки, векторы и записи

Элементы данных можно склеивать вместе и получать составные структуры данных. *Список* представляет собой структуру, элементы которой связаны последовательно. *Вектор* является структурой, элементы которой упакованы в линейный массив. К спискам можно добавлять новые элементы, но доступ к n-му элементу списка занимает время, пропорциональное n. В отличие от списка, вектор имеет фиксированную длину, а к его элементам можно обратиться за константное время. *Запись* подобна вектору, но к ее полям обращаются по имени, а не по номерам индексов. Кроме того, записи создают новые типы данных, которые можно проверить типовыми предикатами, и гарантируется, что они отличаются от всех других типов.

Составные структуры данных создаются из компонент процедурами, которые называются *конструкторы*, а доступ к их компонентам можно получить через *селекторы*.

Процедура `list` — конструктор для списков. Предикат `list?` истинен для любого списка и ложен для всех других типов данных[5]:

```
(define a-list (list 6 946 8 356 12 620))

a-list
(6 946 8 356 12 620)

(list? a-list)
#t

(list? 3)
#f
```

Здесь `#t` и `#f` — печатные представления булевых значений истина и ложь[6].

Списки строятся из пар. *Пара* создается конструктором `cons`. Селекторы двух компонентов пары называются `car` и `cdr` (произносится «куддер»)[7].

```
(define a-pair (cons 1 2))

a-pair
(1 . 2)
```

[5] *Предикат* — это процедура, возвращающая истину или ложь. Культурная традиция языка Scheme говорит, что предикатам обычно дается имя с вопросительным знаком (`?`) на конце, кроме элементарных предикатов арифметического сравнения `=`, `<`, `>`, `<=` и `>=`. Это всего лишь стилистическое соглашение. Вопросительный знак в Scheme — обыкновенный символ.

[6] Удобно, но некоторых программистов раздражает, что условные выражения (`if` и `cond`) считают любое значение, не равное в точности `#f`, истинным.

[7] Эти имена — историческая случайность, сокращения выражений Contents of the Address part of Register «содержимое адресной части регистра» и Contents of the Decrement part of Register «содержимое декрементной части регистра» для компьютера IBM 704, где в 1950-е гг. работала первая реализация языка Lisp. Scheme является диалектом Lisp.

```
(car a-pair)
1

(cdr a-pair)
2
```

Список представляет собой цепочку пар, так что `car` каждой пары содержит элемент списка, а `cdr` каждой пары содержит следующую пару, кроме последнего `cdr`, где лежит отдельное значение, называемое пустым списком и записываемое как `()`. Таким образом,

```
(car a-list)
6

(cdr a-list)
(946 8 356 12 620)

(car (cdr a-list))
946

(define another-list
  (cons 32 (cdr a-list)))

another-list
(32 946 8 356 12 620)

(car (cdr another-list))
946
```

У списков `a-list` и `another-list` общий хвост (`cdr`).

Предикат `pair?` истинен по отношению к парам и ложен для всех остальных типов данных. Предикат `null?` истинен только для пустого списка.

Векторы проще списков. Есть конструктор `vector`, при помощи которого векторы можно создавать, и селектор `vector-ref` для доступа к элементам вектора. В языке Scheme все селекторы, использующие числовой индекс, считают начиная с нуля:

```
(define a-vector
  (vector 37 63 49 21 88 56))

a-vector
#(37 63 49 21 88 56)

(vector-ref a-vector 3)
21

(vector-ref a-vector 0)
37
```

Печатное представление вектора отличается от печатного представления списка символом #, который ставится перед начальной скобкой.

Имеется предикат `vector?`, истинный для векторов и ложный для всех остальных типов данных.

Scheme также содержит числовой селектор для элементов списка `list-ref`, аналогичный селектору для векторов:

```
(list-ref a-list 3)
356

(list-ref a-list 0)
6
```

Записи устроены сложнее; прежде, чем запись можно использовать, ее следует объявить. Простое объявление типа записи может выглядеть так:

```
(define-record-type point
    (make-point x y)
    point?
  (x point-x)
  (y point-y))
```

После этого объявления становится можно создавать и использовать точки:

```
(define p (make-point 1 2))

(point? p)
#t

(point-x p)
1

(point-y p)
2
```

Элементами списков, векторов и записей могут быть любые виды данных, включая числа, процедуры, списки, векторы и записи. Множество других процедур для работы со списками, векторами и записями можно найти в онлайн-документации по Scheme.

Процедуры с произвольным количеством аргументов

В процедурах, которые мы видели до сих пор, указан список формальных параметров, связываемых с аргументами при вызове функции. Однако многие процедуры принимают произвольное количество аргументов. Например, сколько угодно аргументов принимает арифметическая процедура умножения чисел. Для того, чтобы определить такую процедуру, мы вместо списка символов указываем один символ. Этот единственный символ связывается со списком аргументов при вызове процедуры. Например, если у нас есть двухместная процедура умножения `*:binary`, можно записать

```
(define * (lambda args (accumulate *:binary 1 args)))
```

где процедура `accumulate` выглядит так:

```
(define (accumulate proc initial lst)
  (if (null? lst)
      initial
      (proc (car lst)
            (accumulate proc initial (cdr lst)))))
```

Иногда нам хочется создать процедуру, принимающую несколько именованных аргументов и произвольное число других. Если в определении процедуры список параметров содержит точку перед последним именем (это называется *запись с хвостом через точку*), такая запись означает, что параметры перед точкой будут связаны с начальными аргументами, а последний параметр свяжется со списком оставшихся аргументов. В примере с процедурой `*` выше начальных параметров нет, так что значением `args` будет список всех аргументов. Другим способом записать `*` будет такой:

```
(define (* . args) (accumulate *:binary 1 args))
```

Процедура, называемая `-`, более интересна, поскольку она требует хотя бы один аргумент: если аргумент один, он берется с обратным знаком; если их больше, происходит вычитание остальных аргументов из первого:

```
(define (- x . ys)
  (if (null? ys)       ; Только один аргумент?
      (-:unary x)
      (-:binary x (accumulate +:binary 0 ys))))
```

Можно записать и так:

```
(define -
  (lambda (x . ys)
     (if (null? ys)
         (-:unary x)
         (-:binary x (accumulate +:binary 0 ys)))))
```

Параметры вроде `args` и `ys` в вышеприведенных примерах называются *остаточными параметрами*, поскольку они связываются с остатком аргументов.

Символы

Символы — очень важный элементарный тип данных; из них строятся программы и алгебраические выражения. Наверное, вы заметили, что программы на Scheme выглядят просто как списки. На самом деле, они *и есть* списки. Некоторые элементы списков, из которых состоят программы — это символы вроде `+` или `vector`[8].

Поскольку нам нужно писать программы, обрабатывающие другие программы, нужно уметь записать выражение, называющее такой символ. Для этого служит механизм *кавычек*. Именем символа `+` служит выражение `'+`. В общем случае именем

[8]Символ состоит из произвольного количества знаков. В нормальном случае символ не может содержать пробельные знаки и разделители вроде круглых и квадратных скобок, кавычек, запятой и #; однако есть специальные соглашения о нотации, позволяющие включать в имя символа любые знаки.

любого выражения может служить само это выражение с кавычкой спереди. Таким образом, именем выражения `(+ 3 a)` служит `'(+ 3 a)`.

Проверить, равны ли два символа между собой, можно через предикат `eq?`. Например, можно написать программу, проверяющую, является ли выражение суммой:

```
(define (sum? expression)
  (and (pair? expression)
       (eq? (car expression) '+)))

(sum? '(+ 3 a))
#t

(sum? '(* 3 a))
#f
```

Рассмотрим, что произошло бы, если бы мы не поставили кавычку в выражении `(sum? '(+ 3 a))`. Если переменная `a` имеет значение 4, мы задавали бы вопрос, является ли 7 суммой. Но хотели-то мы узнать, является ли суммой выражение `(+ 3 a)`. Поэтому кавычка нужна.

Обратная кавычка

Для работы с образцами и другими формами синтаксиса на основе списков часто оказывается полезным сочетать внутри одного выражения закавыченные и вычисляемые фрагменты. В Lisp-системах есть механизм *квазикавычки*, облегчающий такую работу.

Для обыкновенного закавычивания используется знак прямой кавычки, и точно также для квазизакавычивания используется знак обратной кавычки[9]. Такое частично закавыченное выражение задается как список, где перед фрагментами, подлежащими вычислению, стоит запятая. Например,

```
`(a b ,(+ 20 3) d)
(a b 23 d)
```

Кроме того, механизм обратной кавычки содержит конструкцию для «вклеивания»: вычисленное выражение порождает список, и он вклеивается в объемлющий список. Например,

```
`(a b ,@(list (+ 20 3) (- 20 3)) d)
(a b 23 17 d)
```

Более подробное объяснение квазизакавычивания содержится в «Сообщении о языке Scheme»[109].

Эффекты

Иногда в процессе вычислений требуется выполнить какое-то действие, скажем вывести на экран точку или распечатать значение. Такое действие называется *эф-*

[9]На американской клавиатуре знак обратной кавычки «`» — это знак нижнего регистра на той же клавише, где знаком верхнего регистра является тильда «~».

фект[10]. Например, чтобы более подробно увидеть, как программа вычисления факториала получает свой результат, можно вставить в тело внутренней процедуры `factlp` предложение `write-line` и распечатать на каждом проходе цикла список из счетчика и текущего результата:

```
(define (factorial n)
  (let factlp ((count 1) (answer 1))
    (write-line (list count answer))
    (if (> count n)
        answer
        (factlp (+ count 1) (* count answer)))))
```

При вызове такой измененной процедуры `factorial` можно видеть, как счетчик увеличивается и постепенно строится результат:

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

Тело всякой процедуры или выражения `let`, а также следствие всякой ветви `cond` может содержать предложения, производящие эффекты. Как правило, предложение с эффектом не имеет никакого полезного значения. Возвращаемое значение тела или ветви условной конструкции производится последним выражением. В нашем примере значение процедуры `factorial` получается в выражении `if`.

Присваивание

Эффекты в виде точки на экране или распечатки значения довольно безвредны, однако существуют намного более мощный (и опасный) класс эффектов, называемый *присваивание*. Присваивание изменяет значение переменной или поля в структуре данных. Почти все, что мы вычисляем, является математической функцией: из определенных входных данных всегда получается одинаковый результат. Однако с помощью присваивания мы можем создавать объекты, изменяющие свое поведение в процессе использования. Например, через `set!` можно создать устройство, увеличивающее счетчик при каждом вызове[11]:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      count)))
```

[10] Это жаргон, принятый в информатике. Эффект представляет собой изменение в чем-то. Например, процедура `write-line` изменяет состояние дисплея, выводя на него текст.

[11] Еще одно культурное соглашение состоит в том, что имя процедуры с «побочными эффектами» заканчивается восклицательным знаком (`!`). Этот знак предупреждает читателя, что изменение порядка эффектов может повлиять на результат программы.

Давайте создадим два счетчика:

```
(define c1 (make-counter))
(define c2 (make-counter))
```

Каждый из счетчиков обладает независимым внутренним состоянием. При вызове счетчика он увеличивает локальную переменную состояния `count` и возвращает ее значение:

```
(c1)
1

(c1)
2

(c2)
1

(c1)
3

(c2)
2
```

Для присваиваний элементам структуры данных, скажем пары, списка или вектора, в языке Scheme имеются:

```
(set-car! пара новое-значение)
(set-cdr! пара новое-значение)

(list-set! список индекс новое-значение)
(vector-set! вектор индекс новое-значение)
```

Запись можно определить так, чтобы разрешить присваивание полям (сравните со стр. 339):

```
(define-record-type point
    (make-point x y)
    point?
  (x point-x set-x!)
  (y point-y set-y!))

(define p (make-point 1 2))

(point-x p)
1

(point-y p)
2

(set-x! p 3)
```

```
(point-x p)
3

(point-y p)
2
```

В общем случае полезно по возможности избегать присваиваний, но когда они действительно нужны, в языке они есть[12].

B.2 Более сложные конструкции

В языке Scheme есть много мощных конструкций, но здесь мы не будем пытаться их описывать. К примеру, вы, скорее всего, захотите узнать о хеш-таблицах. Вообще говоря, лучшие источники информации — «Семикратно пересмотренное сообщение об алгоритмическом языке Scheme» [109] и «Справочное руководстве по MIT/GNU Scheme» [51]. Однако вот две достаточно сложные конструкции, которые могут вам понадобиться при чтении этой книги.

Динамическое связывание

Иногда требуется указать способ, которым будет произведено некоторое вычисление или действие — скажем, задать основание системы счисления при распечатке числа. Для этого используется объект, называемый *параметр*.

Например, процедура языка Scheme `number->string` порождает строку, представляющую число с заданным основанием:

```
(number->string 100 2)
"1100100"

(number->string 100 16)
"64"
```

Допустим, нам нужно использовать `number->string` во многих точках сложной программы, запускаемой вызовом `myprog`, но при этом мы хотим управлять, какое основание использовать при запуске. Можно этого добиться, создав параметр `radix` со значением по умолчанию, равным 10:

```
(define radix (make-parameter 10))
```

Значение параметра можно получить, вызвав этот параметр без аргументов:

```
(radix)
10
```

Определяем специализированную версию `number->string` для использования вместо обыкновенной:

[12] Дисциплина программирования без присваиваний называется *функциональное программирование*. Как правило, функциональные программы проще для понимания и содержат меньше ошибок, чем *императивные*.

```
(define (number->string-radix number)
  (number->string number (radix)))
```

При выполнении `myprog` все вызовы `number->string-radix` породят десятичные строки, поскольку умолчательное значение `(radix)` равно 10. Однако мы можем обернуть программу вызовом `parameterize`, чтобы при выполнении использовалось другое основание:

```
(parameterize ((radix 2))
  (myprog))
```

Синтаксис конструкции `parameterize` такой же, как у `let`, но она может использоваться только с параметрами, созданными через `make-parameter`.

Пакеты

В MIT/GNU Scheme есть простой механизм для построения наборов связанных процедур с общим разделяемым состоянием: *пакеты*. Пакет — это процедура, делегирующая работу набору именованных процедур: первым аргументом пакета служит имя делегата, к которому надо обратиться, а остальные аргументы передаются указанному делегату. Это похоже на то, как работают некоторые объектно ориентированные языки, но намного проще, без классов и наследования.

Иногда пакет называют *процедурой, принимающей сообщения*, где тип сообщения — это имя делегата, а тело сообщения — аргументы[13]. Это подчеркивает, что пакет поддерживает протокол передачи сообщений и может рассматриваться как узел в сети взаимодействий.

Вот простой пример:

```
(define (make-point x y)
  (define (get-x) x)
  (define (get-y) y)
  (define (set-x! new-x) (set! x new-x))
  (define (set-y! new-y) (set! y new-y))
  (bundle point? get-x get-y set-x! set-y!))
```

Процедура `make-point` определяет четыре внутренних процедуры, у которых есть общие переменные `x` и `y`. Макрос `bundle` создает пакетную процедуру, где эти внутренние процедуры служат делегатами.

Первым аргументом макроса `bundle` является предикат, создаваемый процедурой `make-bundle-predicate`. Создаваемый пакет будет удовлетворять этому предикату:

```
(define point? (make-bundle-predicate 'point))

(define p1 (make-point 3 4))
(define p2 (make-point -1 1))

(point? p1)
#t
```

[13] Эта терминология восходит к системе ACTOR [58] и языку программирования Smalltalk [46].

```
(point? p2)
#t
(point? (lambda (x) x))
#f
```

Аргументом процедуры `make-bundle-predicate` служит символ, с помощью которого можно распознать предикат при отладке.

Если предикат не требуется, макрос `bundle` может принять в качестве первого аргумента `#f`. В этом случае создаваемую пакетную процедуру никак нельзя отличить от других процедур.

Оставшиеся аргументы макроса `bundle` — это имена процедур-делегатов: `get-x`, `get-y`, `set-x!` и `set-y!`. Эти имена макрос ищет в своем лексическом окружении и получает соответствующие процедуры-делегаты. Затем создается пакетная процедура, связывающая каждое из этих имен со своим делегатом.

При вызове получившейся пакетной процедуры первым аргументом должен быть символ, совпадающий с именем одной из процедур-делегатов. С помощью этой связи выбирается нужная процедура, она вызывается, и ей передаются оставшиеся аргументы пакетной процедуры.

Пакет проще использовать, чем описать:

```
(p1 'get-x)
3
(p1 'get-y)
4
(p2 'get-x)
-1
(p2 'get-y)
1

(p1 'set-x! 5)

(p1 'get-x)
5
(p2 'get-x)
-1
```

Литература

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs* (2nd ed.). Cambridge, MA: MIT Press, 1996. Русский перевод: Харольд Абельсон, Джеральд Джей Сассман при участии Джули Сассман. *Структура и интерпретация компьютерных программ*. М.: КДУ; Добросвет. 2006.

[2] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss; «Amorphous Computing», in *Communications of the ACM*, 43(5) (May 2000): 74-82.

[3] Lee Altenberg; «The Evolution of Evolvability in Genetic Programming», in *Advances in Genertic Programming*, ed. Kenneth E. Kinnear Jr., 47-74. Cambridge, MA: MIT Press, 1994.

[4] *The ARRL Handbook for Radio Amateurs*, American Radio Relay League, Newington, CT (annual).

[5] Jean-Paul Arcangeli and Christian Pomian; «Principles of Plasma Pattern and Alternative Structure Compilation», in *Theoretcal Computer Science*, 71 (1990): 177-191.

[6] Franz Baader and Wayne Snyder; «Unification Theory», in *Handbook of Automated Reasoning*, ed. Alan Robinson and Andrei Voronkov. Elsevier Science Publishers B. V., 2001.

[7] Jonathan B.L. Bard; *Morphogenesis*, Cambridge: Cambridge University Press, 1990.

[8] Alan Bawden and Jonathan Rees; «Syntactic Closures», in *Proc. Lisp and Functional Programming* (1988).

[9] Jacob Beal; *Generating Communication Systems Through Shared Context*, S. M. thesis, MIT, также Artificial Intelligence Laboratory Technical Report 2002-002. January 2002.

[10] Jacob Beal; «Programming an Amorphous Computational Medium», in *Unconventional Programming Paradigms International Workshop* (September 2004). Обновленная версия Lecture Notes on Computer Science, 3566 (August 2005).

[11] M.R. Bernfield, S.D. Banerjee, J.E. Koda, and A.C. Rapraeger; «Remodeling of the basement membrane as a mechanism of morphogenic tissue interaction», in *The role of extracellular matrix in development*, ed. R.L. Trelstad, 542-572. New York: Alan R. Liss, 1984.

[12] Martin Berz; «Automatic differentiation as nonarchimedean analysis», in *Computer Arithmetic and Enclosure Methods*, ed. L.Atanassova and J. Herzberger. Elsevier Science Publishers B. V. (North-Holland), 1992.

[13] Philip L. Bewig; *Scheme Requests for Implementattion 41: Streams* (2008). `https://srfi.schemers.org/srfi-41`

[14] R.C. Bihlin and R.L. Gilliland; «Hubble Space Telescope. Absolute Spectrophotometry of Vega from the Far-Ultraviolet to the Infrared», in *The Astronomical Journal*, 127(6) (June 2004): 3508-3515.

[15] J.P. Brocks; «Amphibian limb regeneration: rebuilding a complex structure», in *Science*, 276 (1997): 81-87.

[16] Alonzo Church; *The Calculi of Lambda-Conversion.* Princeton, NJ: Princeton University Press, 1941.

[17] Alonzo Church; «An Unsolvable Problem of Elementary Number Theory», *American Journal of Mathematics*, 58 (1936): 345-363.

[18] Alonzo Church; «A Note on the Entscheidungsproblem», in *Journal of Symbolic Logic*, 1 (1936): 40-41.

[19] Lauren Clement and Radhika Nagpal; «Self-Assembly and Self-Repairing Topologies», in *Workshop on Adaptability in Multi-Agent Systems*, RoboCup Australian Open, January 2003.

[20] William Kingdon Clifford; «Preliminary sketch of bi-quaternions», in *Proceedings of the London Mathematical Society*, 4 (1873): 381-395.

[21] William Clinger; «Nondeterministic Call by Need is Neither Lazy Nor by Name», in *Proceedings of the 1982 ACM symposium on LISP and functional programming*, 226-234 (August 1982).

[22] William Clinger and Jonathan Rees; «Macros that work», in *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, 155-162 (1991).

[23] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel; *Un systeme de communication homme-machine en français*, Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy, 1973.

[24] *Constraints, An International Journal* ISSN: 1383-7133 (Print) 1572-9354 (Online).

[25] Статья о продолжениях в «Википедии». `https://wikipedia.org/wiki/Continuation`

[26] Haskell Brooks Curry; «Grundlagen der Kombinatorischen Logik», in *American Journal of Mathematics*, Baltimore: Johns Hopkins University Press, 1930.

[27] Johan deKleer, Jon Doyle, Guy Steele, and Gerald J. Sussman; «AMORD: Explicit control of reasoning», in *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, 116-125 (1977).

[28] E.M. del Pino and R.P. Elinson; «A novel developmental pattern for frogs: gastrulation produces an embryonic disk», in *Nature*, 306 (1983): 589-591.

[29] Howard P. Dinesman; *Superior Mathematical Puzzles.* New York: Simon and Schuster, 1968.

[30] Jon Doyle; «A truth maintenance system», in *Artificial Intelligence*, 12 (1979): 231-272.

[31] K. Dybvig, R. Hieb, and C. Bruggerman; «Syntactic abstraction in Scheme», in *Proc. Lisp and Symbolic Computation* (1993).

[32] G.M. Edelman and J.A. Gally; «Degeneracy and complexity in biological systems», *Proceedings of the National Academy of Sciences*, 98 (2001): 13763-13768.

[33] M. D. Ernst, C. Kaplan, and C. Chambers; «Predicate Dispatching: A Unified Theory of Dispatch», in *ECOOP'98—Object-Oriented Programming: 12th European Conference, Proceedings»,* ed. Eric Jul, 186-211, Lecture Notes in Computer Science, 250. Berlin: Springer, 1998.

[34] Zsuzsa Farkas; «LISTLOG — A PROLOG extension for list processing», in *TAPSOFT 1987*, ed. Ehrig H., Kowalski R., Levi G., Montanari U., Lecture Notes in Computer Science, 250. Berlin: Springer, 1987.

[35] Robert Floyd; «Nondeterministic algorithms», in *Journal of the ACM*, 14(4) (1967): 636-644.

[36] Kenneth D. Forbus and Johan de Kleer; *Building Problem Solvers.* Cambridge, MA: MIT Press, 1993.

[37] Stephanie Forrest, Anil Somayaji, David H. Ackley; «Building Diverse Computer Systems», in *Proceedings of the 6th workshop on Hot Topics in Operating Systems*, 67-72. Los Angeles, CA: IEEE Computer Society Press, 1997.

[38] Joseph Frankel; *Pattern Formation, Ciliate Studies and Models.* New York: Oxford University Press, 1989.

[39] Eugene C. Freuder; *Synthesizing Constraint Expressions.* AI Memo 2709, MIT Artificial Intelligence Laboratory, July 1976.

[40] Daniel P. Friedman and David S. Wise; «Cons should not evaluate its arguments», in *Automata Languages and Programming*; Proc. Third International Colloquium at the University of Edinburgh, ed. S. Michaelson and R. Milner, 257-284 (July 1976).

[41] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes; *Essentials of Programming Languages.* Cambridge, MA: MIT Press/McGraw-Hill, 1992.

[42] Richard P. Gabriel and Linda DeMichiel; «The Common Lisp Object System: An Overview», in *Proceedings of ECOOP'87 European Conference on Object-Oriented Programming*, ed. Jean Bezivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, 151-170. Paris: Springer, 1987.

[43] George Gatewood and Joost Kiewiet de Jonge; «Map-based Trigonometric Parallaxes of Altair and Vega», in *The Astronomical Journal*, 450 (September 1995): 364-368.

[44] George Gatewood; «Astrometric Studies of Aldebaran, Arcturus, Vega, The Hyades, and Other Regions», in *The Astronomical Journal*, 136 (1) (2008): 452-460.

[45] Kurt Gödel; «On Undecidable Propositions of Formal Mathematical Systems», *Lecture notes taken by Kleene and Rosser at the Institute for Advanced Study* (1934), переиздано в Martin Davis *The Undecidable: Basic papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, 39-74. New York: Raven, 1965.

[46] Adele Goldberg and David Robson; *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley, 1983.

[47] Michael Gordon, Robin Milner, and Christopher Wadsworth; *Edinburgh LCF*, Lecture Notes in Computer Science, 78. New York: Springer-Verlag, 1979.

[48] Cordell Green; «Application of theorem proving to problem solving», in *Proceedings of the International Joint Conference on Artificial Intelligence*, 219-240 (1969).

[49] Cordell Green and Bertram Raphael; «The use of theorem-proving techniques in question-answering systems», in *Proceedings of the ACM National Conference*, 169-181 (1968).

[50] John V. Guttag; «Abstract data types and the development of data structures», *Communications of the ACM*, 20(6) (1977): 397-404.

[51] Chris Hanson; *MIT/GNU Scheme Reference Manual*. `https://www.gnu.org/software/mit-scheme/`

[52] Chris Hanson; SOS Software: Scheme Object System, 1993.

[53] Chris Hanson, Tim Berners-Lee, Lalana Kagal, Gerald Jay Sussman, and Daniel Weitzner; «Data-Purpose Algebra: Modeling Data Usage Policies», in *Eighth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07)*, (June 2007).

[54] Hyman Hartman and Temple F. Smith; «The Evolution of the Ribosome and the Genetic Code», in *Life*, 4 (2014): 227-249.

[55] Jacques Herbrand; «Sur la non-contradiction de l'arithmetique», *Journal fur die reine und angewandte Mathematik*, 166 (1932): 1-8.

[56] Carl E. Hewitt; «PLANNER: A language for proving theorems in robots», in *Proceedings of ths International Joint Conference on Artificial Intelligence*, 295-301 (1969).

[57] Carl E. Hewitt; «Viewing control structures as patterns of passing messages», in *Journal of Artificial Intelligence*, 8(3) (1977): 323-364.

[58] Carl E. Hewitt, Peter Bishop, Richard Steiger; «A Universal Modular ACTOR Formalism for Artificial Intelligence», in *IJCAI-73: Proceedings of the Third International Joint Conference on Artificial Intelligence*, 235-245 (1973).

[59] Edwin Hewitt; «Rings of real-valued continuous functions. I», in *Transactions of the American Mathematical Society*, 64 (1948): 45-99.

[60] Paul Horowitz and Winfield Hill; *The Art of Electronics.* Cambridge: Cambridge University Press, 1980.

[61] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*, Institute of Electrical and Electronic Engineers, Inc., 1991.

[62] Paul-Alan Johnson; *The Theory of Acrhitecture: Concepts, Themes, & Practices.* New York: Van Nostrand Reinhold, 1994.

[63] Jerome H. Keisler; «The hyperreal line. Real numbers, generalizations of the reals, and theories of continua», in *Synthese Library*, 242, 207-237. Dordrecht: Kluwer Academic, 1994.

[64] Richard Kelsey, William Clinger, and Jonathan Rees (editors); *Revised*^{5} *Report on the Algorithmic Language Scheme* (1998).

[65] Richard Kelsey; *Scheme Requests for Implementation 9: Defining Record Types* (1999). `https://srfi.schemers.org/srfi-9/`

[66] Gregor Kiczales; Tiny CLOS software: Kernelized CLOS, with a metaobject protocol, 1992.

[67] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin; «Aspect-Oriented Programming», in *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, 220-242 (1997).

[68] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow; *The Art of the Metaobject Protocol.* Cambridge, MA: MIT Press, 1991.

[69] Simon Kirby; *Language evolution without natural selection: From vocabulary to syntax in a population of learners.*, Edinburgh Occasional Paper in Linguistics EOPL-98-1, University of Edinburgh Department of Linguistics (1998).

[70] Marc W. Kirschner and John C. Gerhart; *The Plausibility of Life: Resolving Darwin's Dilemma.* New Haven: Yale University Press, 2005.

[71] Marc W. Kirschner, Tim Mitchison; «Beyond self-assembly: from microtubules to morphogenesis», in *Cell*, 45(3) (May 1986): 329-342.

[72] D. Knuth, P. Bendix; «Simple word problems in universal algebras», in *Computational problems in Abstract Algebra*, ed. John Leech, 263-297. London: Pergamon Press, 1970.

[73] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba; «Hygienic Macro Expansion», in *ACM Conference on LISP and Functional Programming* (1986).

[74] E. Kohlbecker and Mitchell Wand; «Macro by Example: Deriving syntactic transformations from their specifications», in *Proc. Symposium on Principles of Programming Languages* (1987).

[75] Milos Konopasek and Sundaresan Jayaraman; *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business and Education.* Berkeley, CA: Osborne/McGraw Hill, 1984.

[76] Robert Kowalski; *Predicate logic as a programming language,* Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, 1973.

[77] Robert Kowalski; *Logic for Problem Solving.* New York: North-Holland, 1979.

[78] Robert M. Kowalski; «The Early Years of Logic Programming», in *Communications of the ACM*, 31(1) (January 1988): 38-43.

[79] Temur Kutsia; «Pattern Unification with Sequence Variables and Flexible Arity Symbols», in *Electronic Notes in Theoretical Computer Science*, 66(5) (2002): 52-69.

[80] Butler Lampson, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek; *Report on the programming language Euclid*, Technical report, Computer Systems Research Group, University of Toronto, 1981.

[81] Peter Landin, «A correspondence between Algol 60 and Church's lambda notation: Part I», *Communications of the ACM*, 8(2) (1965): 89-101.

[82] Henrietta S. Leavitt; «1777 variables in Magellanic Clouds», in *Annals of Harvard College Observatory*, 60 (1908): 87-108.

[83] Floor van Leeuwen; «Validation of the new Hipparcos reduction», in *Astronomy & Astrophysics*, 474(2) (2007): 653-664.

[84] Karl Lieberherr; *Informationsverdichtung von Modellen in der Aussagenlogik und das P=NP Problem*, ETH Dissertation, 1977.

[85] Barbara H. Liskov and Stephen N. Zilles; «Specification techniques for data abstractions», in *IEEE Transactions on Software Engineering*, 1(1) (1975): 7-19.

[86] Harvey Lodish, Arnold Berk, S Lawrence Zipursky, Paul Matsudaira, David Baltimore, and James E Darnell; *Molecular Cell Biology* (4th ed.). New York: W. H. Freeman & Co., 1999.

[87] Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreevich Radul, David R. Rush, and Jeffrey Mark Siskind; «Confusion of Tagged Perturbations in Forward Automatic Differentiation of Higher-Order Functions», arxiv:1211.4892 (2012).

[88] David Allen McAllester; *A three-valued truth maintenance system*, AI Memo 473, MIT Artificial Intelligence Laboratory, 1978.

[89] David Allen McAllester; «An outlook on truth maintenance», AI Memo 551, MIT Artificial Intelligence Laboratory, 1980.

[90] John McCarthy; «A basis for a mathematical theory of computation», in *Computer Programming and Formal Systems*, ed. P. Braffort and D. Hirshberg, 33-70. Amsterdam: North-Holland, 1963.

[91] Статья в «Википедии» об MDL.
`https://en.wikipedia.org/wiki/MDL_(programming_language)`

[92] Piotr Mitros; *Constraint-Satisfaction Modules: A Methodology for Analog Circuit Design*, PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 2007.

[93] Paul Penfield Jr.; *MARTHA User's Manual*, MIT Research Laboratory of Electronics, Electrodynamics Memorandum No. 6 (1970).

[94] Barak A. Perlmutter and Jeffrey Mark Siskind; «Lazy Multivariate Higher-Order Forward-Mode AD», in *Proc. POPL'07*, 155-160. New York: ACM, 2007.

[95] Tim Peters; *PEP 20 — The Zen of Python.* `http://www.python.org/dev/peps/pep-0020/`

[96] *POSIX.1-2017*, «Base Definitions», Chapter 9, «Regular Expressions.» `http://pubs.opengroup.org/onlinepubs/9699919799/`

[97] Jonathan Bruce Postel; *RFC 760: DoD standard Internet Protocol* (January 1980). `http://www.rfc-editor.org/rfc/rfc-760.txt`

[98] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling; «Richardson Extrapolation and the Bulirsch-Stoer Method», in *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.), 718-725. Cambridge: Cambridge University Press, 1992.

[99] Alexey Andreevich Radul and Gerald Jay Sussman; «The Art of the Propagator», MIT CSAIL Technical Report MIT-CSAIL-TR-2009-002; Сокращенная версия в *Proc. 2009 International Lisp Conference* (March 2009). `http://hdl.handle.net/1721.1/44215`

[100] Alexey Andreevich Radul; *Propagation networks: a flexible and expressive substrate for computation*, PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 2009. `http://hdl.handle.net/1721.1/54635`

[101] Eric Raymond; *The New Hacker's Dictionary* (2nd ed.). Cambridge, MA: MIT Press, 1993. Русский перевод: Эрик С. Рэймонд. *Новый словарь хакера*. М.: ЦентрКом, 1996.

[102] Jonathan Rees and Norman I. Adams IV; «T: A dialect of Lisp or, lambda: The ultimate software tool», in *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, 114-122 (1982).

[103] John C. Reynodls; «The discoveries of continuations», in *Proc. Lisp and Symbolic Computation*, 233-248 (1993).

[104] J.A. Robinson; «A Machine-Oriented Logic Based on the Resolution Principle», in *Journal of the ACM*, 12(1) (January 1965): 23-41.

[105] Guido van Rossum; *The Python Language Reference Manual*, ed. Fred L. Drake Jr., Network Theory Ltd, 2003.

[106] Jane L. Russell, George D. Gatewood, and Thaddeus F. Worek; «Parallax Studies of Four Selected Fields», in *The Astronomical Journal*, 87(2) (February 1982): 428-432.

[107] Erik Sandewall; «From systems to logic in the early development of nonmonotonic reasoning», in *Artificial Intelligence*, 175 (2011): 416-427.

[108] Moses Schönfinkel; «Uber die Bausteine der mathematischen Logik», in *Mathematische Annalen*, 92 (1924): 305-316.

[109] Alex Shinn, John Cowan, and Arthur Glecker (editors); *Revised^7 Report on Algorithmic Language Scheme* (2013). `http://www.r7rs.org/`

[110] Alex Shinn; *Scheme Requests for Implementation 115: Scheme Regular Expressions* (2014). `https://srfi.schemers.org/srfi-115/`

[111] Jeffrey Mark Siskind and Barak A. Perlmutter; «Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD», in *Implementation and application of functional languages — 17th international workshop*, ed. Andrew Butterfield, Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60, 2005.

[112] Brian Cantwell Smith; *Procedural Reflection in Programming Languages*, PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 1982.

[113] Richard Matthew Stallman; *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, AI Memo 519A, MIT Artificial Intelligence Laboratory, March 1981.

[114] Richard Matthew Stallman and Gerald Jay Sussman; «Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis», in *Artificial Intelligence*, 9 (1977): 135-196.

[115] Guy Lewis Steele Jr.; *Common Lisp the language.* Maynard, MA: Digital Equipment Corporation, 1990.

[116] Guy L. Steele Jr.; *The Definition and Implementation of a Computer Programming Language Based on Constraints*, PhD thesis, MIT, also Artificial Intelligence Laboratory Technical Report 595, August 1980.

[117] Guy Lewis Steele Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow; *The Hacker's Dictionary.* New York: Harper & Row, 1983.

[118] Patrick Suppes; *Introduction to Logic.* New York: D. Van Nostrand, 1957.

[119] Gerald Jay Sussman and Richard Matthew Stallman; «Heuristic Techniques in Computer-Aided Circuit Analysis», in *IEEE Transactions on Circuits and Systems*, 22(11) (November 1975): 857-865.

[120] Gerald Jay Sussman and Guy L. Steele Jr.; «The First Report on Scheme Revisited», in *Higher Order and Symbolic Computation*, 11(4) (December 1998):399-404.

[121] Gerald Jay Sussman and Jack Wisdom; *Structure and Interpretation of Classical Mechanics.* Cambridge, MA: MIT Press, 2001/2014.

[122] Gerald Jay Sussman and Jack Wisdom with Will Farr; *Functional Differential Geometry.* Cambridge, MA: MIT Press, 2013.

[123] *The TTL Data Book for Design Engineers*, by the Engineering Staff of Texas Instruments Incorporated, Semiconductor Group.

[124] Alan M. Turing; «On Computable Numbers, with an Application to the Entscheidungsproblem», in *Proceedings of the London Mathematical Society (Series 2)*, 42 (1936): 230-265.

[125] David L. Waltz; *Generating Semantic Descriptions From Drawings of Scenes With Shadows*, PhD thesis, MIT, также Artificial Intelligence Laboratory Technical Report 271, November 1972. `http://hdl.handle.net/1721.1/6911`

[126] Stephen A. Ward and Robert H. Halstead Jr.; *Computation Structures.* Cambridge, MA: MIT Press, 1990.

[127] Stephen Webb; *Measuring the Universe: The Cosmological Distance Ladder.* Springer-Praxis Series in Astronomy and Astrophysics, Berlin: Springer, 1999.

[128] Daniel J. Weitzner, Hal Abelson, Tim Berners-Lee, Chris Hanson, Jim Hendler, Lalana Kagal, Debora McGuinness, Gerald Jay Sussman, and K. Krasnow Waterman; *Transparent Accountable Data Mining: New Strategies for Privacy Protection*, MIT CSAIL Technical Report MIT-CSAIL-TR-2006-007, January 2006.

[129] Robert Edwin Wengert; «A simple automatic derivative evaluation program», in *Communications of the ACM*, 7(8) (1964): 463-464.

[130] Carter Wiseman; *Louis L. Kahn: Beyond Time and Style: A Life in Architecture.* New York: W. W. Norton, 2007.

[131] Lewis Wolpert, Rosa Beddington, Thomas Jessell, Peter Lawrence, Elliot Meyerowitz, and Jim Smith; *Principles of Development* (2nd ed.). Oxford: Oxford University Press, 2001.

[132] Ramin Zabih, David McAllester, and David Chapman; «Non-deterministic Lisp with dependency-directed backtracking», in *AAAI-87.* (1987): 59-64.

Предметный указатель

> Все неточности в этом указателе объясняются тем, что его готовили при помощи вычислительной машины.
>
> Дональд Э. Кнут, *Основные алгоритмы* (*Исскусство программирования для ЭВМ*, том 1)

Номера страниц для определений процедур даны курсивом.
Буква n после номера страницы отсылает к примечанию.

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу

(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89** **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com

Хансон К., Сассман Д. Дж.

Проектирование гибких программ

Как не загнать себя в угол

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Бр Ю. Н.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 30,71. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

«Идеи этой книги богаты и глубоки; внимание как к словам, так и к программам будет вознаграждено».

Гай Стил, Лексингтон, Массачусетс

Бывает так, что при написании программы вы попадаете в тупик. Возможно, это потому, что вы, как оказалось, не учли некоторые особенности исходной задачи. Однако до обидного часто дело в том, что на начальной стадии проектирования вы приняли какое-то решение, выбрали какую-то структуру данных или способ организации кода, который затем оказался слишком ограниченным, а теперь его трудно заменить.

Эта книга служит мастер-классом по стратегиям организации программ, которые позволяют сохранить гибкость. В каждой главе можно видеть, как два эксперта демонстрируют тот или иной передовой метод, шаг за шагом разрабатывая работающую подсистему, объясняют на ходу стратегию своей работы и время от времени указывают на подводный камень или способ обойти то или иное ограничение.

Исследуются разнообразные способы повышения гибкости программ:

- организация систем комбинаторов с взаимозаменяемыми компонентами — от маленьких функций до полных систем арифметики со стандартизованными интерфейсами;
- расширение данных независимыми слоями аннотации, например единицами измерения или родословной вычисления;
- сочетание независимых фрагментов частичной информации через унификацию или распространение ограничений;
- отделение структуры управления от представления данных через модели предметной области, системы правил и сопоставление с образцом, распространение ограничений и перебор, управляемый зависимостями;
- расширение языка программирования с использованием динамически дополняемых вычислителей.

Крис Хансон — инженер компании Datera.
Джеральд Джей Сассман — профессор по электротехнике в MIT. Соавтор вышедшей на русском языке книги «Структура и интерпретация компьютерных программ».

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

The MIT Press

ДМК издательство
www.дмк.рф

ISBN 978-5-97060-955-2