The Pragmatic Programmers

Семь языков за семь недель

Практическое руководство по изучению языков программирования

Тейт Брюс

Pragmatic Bookshelf

ДМК Пресс Издательство

Отзывы читателей книги «Семь языков за семь недель»

Знание множества парадигм оказывает существенное влияние на наши подходы к проектированию, поэтому я всегда нахожусь в поиске хороших книг, которые помогали бы мне в изучении этих парадигм. Данная книга объединяет в себе описание основных парадигм программирования. Брюс имеет богатый опыт изучения и использования множества языков. Теперь и вы сможете воспользоваться его опытом благодаря этой книге. Я настоятельно рекомендую прочитать ее.

– Доктор *Венкат Субраманиам* (Dr. Venkat Subramaniam)
Титулованный автор и основатель Agile Developer, Inc.

Важность изучения новых языков, парадигм и технологий программирования для программиста невозможно преувеличить. Эта книга решает задачу знакомства с семью языками программирования поразительно кратким и необычным способом, показывая сильные их стороны и обосновывая необходимость их изучения. Эта книга послужит отличным введением для всех программистов, желающих расширить свои горизонты или оценить новые для них языки программирования, прежде чем приступать к детальному изучению.

– *Антонио Каньяно* (Antonio Cangiano)
Программист и технический популяризатор, IBM

Пристегните ремни, потому что поездка будет очень быстрой. Эта книга наполнена быстрыми переходами от одного языка программирования к другому. Брюсу удалось выстроить все в один ряд и в результате получить интереснейшую книгу, которая доставит немало удовольствий увлеченным программистам. Если вы любите осваивать новые языки или любите блеснуть своей эрудицией, если вы хотите подняться на новый уровень – эта книга для вас. Она не разочарует вас.

– *Фредерик Дауд* (Frederic Daoud)
Автор книг «Stripes ...and Java Web Development Is Fun Again»
и «Getting Started with Apache Click»

Хотите выбрать свой «язык года» из великолепной семерки? Хотите упорядочить свои представления о программировании в целом? Тогда вы уже нашли то, что вам нужно! Лично я на время чтения этой книги вернулся в свои студенческие годы, когда предпринимал первые попытки проложить курс через свои языки программирования и посто-

янно натыкался на мели. Разница лишь в том, что Брюс не даст вам сбиться с верного курса! Эта книга не предполагает неторопливого чтения – вы вынуждены будете активно работать с ней. Я уверен, что вы найдете ее чрезвычайно увлекательной и весьма практичной.

– *Мэтт Стайн* (Matt Stine)
Руководитель отдела разработки программного обеспечения
в детской больнице святого апостола Иуды Фаддея
(St. Jude Children's Research Hospital), США

На протяжении почти своей учебы в университете по направлению информатики я не хотел быть программистом, но так или иначе стал им. Книга «Семь языков за семь недель» изменила мои взгляды на многие проблемы и напомнила мне, что я все-таки люблю программировать.

– *Трэвис Каспар* (Travis Kaspar)
Программист в Northrop Grumman

Вот уже более 25 лет я занимаюсь созданием программ для разных аппаратных и программных платформ. После прочтения книги «Семь языков за семь недель» я начал понимать принципы оценки достоинств и недостатков языков программирования. Но самое важное, что я почувствовал, по каким критериям выбирать языки для решения разных задач.

– *Крис Каплер* (Chris Kappler)
Старший научный сотрудник, Raytheon BBN Technologies

Брюс Тейт

Семь языков за семь недель

Практическое руководство по изучению языков программирования

Bruce A. Tate

Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages

Edited by Jacquelyn Carter

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas

Брюс Тейт

Семь языков за семь недель

Практическое руководство по изучению языков программирования

Под редакцией Жаклин Картер

Москва, 2017

УДК 004.6
ББК 32.973.26
Т30

Тейт Брюс
Т30 Семь языков за семь недель. Практическое руководство по изучению языков программирования / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2017. – 384 с.: ил.

ISBN 978-5-97060-555-4

Вместе с семью языками программирования вы исследуете наиболее важные из современных моделей программирования. Вы познакомитесь с динамической типизацией, которая делает языки Ruby, Python и Perl такими гибкими. Постигнете систему прототипов, лежащую в основе языка JavaScript. Увидите, как сопоставление с образцом в языке Prolog сказалось на формировании языков Scala и Erlang. Узнаете, чем функциональное программирование на языке Haskell отличается от программирования на языках семейства Lisp, включая Clojure.

Издание предназначено для программистов разной квалификации, в том числе выбирающих для изучения новый язык программирования.

УДК 004.6
ББК 32.973.26

ISBN 978-1-934356-59-3 (анг.)
ISBN 978-5-97060-555-4 (рус.)

Содержание

Посвящение

Пять месяцев, с декабря 2009-го по апрель 2010-го, выдались самыми тяжелыми в моей жизни. Сначала моему брату (которому не было и 47 лет) потребовалась срочная операция на сердце. При этом совершенно ничто не предвещало беды. (Операция прошла успешно, и он чувствует себя хорошо.) В конце марта у моей сестры был обнаружен рак молочной железы. Но самый большой шок я испытал в начале марта, когда моей матери был поставлен страшный диагноз: рак в последней стадии. После этого она прожила всего несколько недель.

Как вы уже поняли, мне пришлось пережить горечь утраты. Но, как ни странно, этот тяжелый опыт нельзя назвать полностью отрицательным. Дело в том, что моя матушка жила в ладу с собой и окружающими, ее отношения с семьей были крепкими и наполненными, и они строились именно так, как она того хотела, в соответствии с ее убеждениями и верованиями.

Линда Лейла Тейт (Lynda Lyle Tate) воплощала свою творческую энергию в рисунках акварелью. Она делилась своим искусством с окружающим миром с помощью галереи на Медисон-авеню и через своих учеников. До того как я покинул отчий дом, у меня была возможность взять у нее несколько уроков. Для меня, обладающего техническим складом ума, этот опыт давался с большим трудом. Я начинал воспроизводить свой очередной шедевр на белом листе. Но по мере того, как рисунок приобретал более отчетливые очертания, он становился все дальше от первоначального замысла. Когда я приходил в отчаяние от своей неспособности исправить картину, ко мне подходила мама, заглядывала через плечо и говорила – что она видит. Затем легкими движениями кисти добавляла несколько штрихов, подчеркивающих глубину и некоторые детали, и я с удивлением обнаруживал, что был совсем недалеко от желаемого! Достаточно было лишь нескольких талантливых прикосновений, чтобы уберечь мое творение от катастрофы. Тогда я вздымал руки в победном жесте и возвещал всему классу о своем творении, не замечая, что каждый ученик прошел через свой собственный триумф маленькой победы.

Спустя некоторое время я узнал, что мама работала и над другими холстами. Через ее церковь и ее профессию она находила людей, сломавшихся духом. Сталкиваясь с такими людьми то тут, то там, моя мама приводила их в класс, где с помощью красок и бумаги помогала им вновь обрести себя и найти свой путь. В последнюю неделю к ней приходило много людей, удрученных скорой потерей Учителя, и для

каждого мама находила удачную шутку или доброе слово, успокаивая тех, кто приехал успокаивать ее. Я встретился с холстами человеческих душ, подправленных рукой Мастера и вдохновленных на большие дела. Я был потрясен.

Когда я сообщил матери о своем желании посвятить ей эту книгу, она сказала, что ей будет приятно, но она ничего не смыслит в компьютерах. Это – правда. Даже сама мысль о Windows делала ее беспомощной. Но, мама, ты сделала для меня все возможное. Твои своевременные слова поддержки вдохновляли меня, твоя любовь к творчеству помогла мне сформироваться, а твой энтузиазм и жизнелюбие до сих пор сопровождают меня. Вспоминая о пережитых событиях, я понимаю, что был не в силах помочь, но мне намного легче от того, что я тоже ощущаю себя холстом, побывавшим под кистью Мастера.

Эта книга посвящается Линде Лейле Тейт (Lynda Lyle Tate), 1936–2010.

Благодарности

Это – самая сложная книга из всех, написанных мной. Она же оказалась самой полезной. Очень много людей помогали мне в ее создании. Прежде всего я хочу поблагодарить свою семью. Кайла (Kayla) и Джулия (Julia), ваша манера изложения мыслей удивляет меня. Вы пока не представляете, каких высот можете достигнуть! Мэгги (Maggie), ты – мой источник радости и вдохновения.

В сообществе пользователей Ruby спасибо Дейву Томасу (Dave Thomas), познакомившему меня с новым языком, который перевернул всю мою карьеру и помог опять познать радость работы. Спасибо также Мацу (Matz) за его дружбу и совет поделиться своими знаниями с читателями. Вы пригласили меня посетить Японию, где родился Ruby, и эта поездка вдохновила меня намного сильнее, чем вы могли себе представить. Спасибо Чарльзу Наттеру (Charles Nutter), Эвану Фениксу (Evan Phoenix) и Тиму Брею (Tim Bray) за помощь в подготовке этой темы, вошедшей в книгу, – возможно, это было утомительно для вас, но вы помогли мне сформировать этот раздел книги.

В сообществе пользователей Io спасибо Джереми Трегану (Jeremy Tregunna) за помощь в подготовке нескольких примеров на Io. Ваши отзывы были одними из самых ценных – они всегда были весьма кстати и очень помогли в создании главы об этом языке. Стив Декорт (Steve Dekorte), вы создали особый язык, и я надеюсь, что он когда-нибудь займет подобающее ему положение в мире программирования. Поддержка параллелизма в Io просто превосходна, и сам язык наполнен удивительной внутренней красотой. Я точно могу сказать, насколько хорош этот язык. Спасибо вам за помощь в его установке. Спасибо также за ваши вдумчивые отзывы, которые помогли мне ухватить суть Io. Вам удалось овладеть умами первых читателей и побудить их к использованию вашего языка.

В сообществе пользователей Prolog спасибо Брайану Тарбоксу (Brian Tarbox), что поделился своим богатым опытом с моими читателями. Проекты исследования дельфинов определенно придали красок главе, посвященной языку Prolog. Отдельное спасибо Джо Армстронгу (Joe Armstrong), чьи отзывы оказали существенное влияние на формирование главы и всей книги в целом. Спасибо вам также за пример раскрашивания карты и за ваши идеи – они оказались как нельзя к месту.

В сообществе пользователей Scala спасибо моему доброму другу Венкату Субраманиаму (Venkat Subramaniam). Ваша книга о языке

Scala написана понятным языком и богата отличными примерами. Я в значительной степени опирался на эту книгу. Я высоко ценю вашу помощь, которую вы оказывали мне. Она помогла мне избежать значительных трудностей и сосредоточиться на обучении. Спасибо также Мартину Одерски (Martin Odersky), что поделился своими мыслями с читателями. Язык Scala являет собой удивительное сочетание функциональной и объектно-ориентированной парадигм программирования. Ваши знания в освещении этого языка очень пригодились мне.

В сообществе пользователей Erlang я еще раз хочу поблагодарить Джо Армстронга (Joe Armstrong). Ваши доброжелательность и энергия помогли мне облечь идеи в правильные словесные формы. Ваши предложения по построению отказоустойчивых распределенных систем действительно работают. Философия языка Erlang «позвольте приложению потерпеть неудачу» («Let it crash») пришлась мне по душе больше, чем философии в других языках, представленных в этой книге. Я надеюсь, что эти идеи получат более широкое распространение.

В сообществе пользователей Clojure спасибо Стюарту Халлоуэю (Stuart Halloway) за его отзывы и идеи, заставившие меня работать над книгой с еще большим усердием, чтобы сделать ее более полезной для моих читателей. Ваши глубокие знания Clojure и интуиция помогли мне понять, что является наиболее важным. Ваша книга также оказала существенное влияние на главу о языке Clojure и фактически изменила мои подходы к решению некоторых задач в других главах. Ваши подходы к обучению, наработанные за многолетнюю практику, пользуются заслуженной славой. Спасибо также Ричу Хикки (Rich Hickey) за освещение идей, легших в основу его языка, и за лекцию по теме: «Что значит быть диалектом языка Lisp». Некоторые идеи языка Clojure кажутся революционными и вместе с тем являются абсолютно практичными. Поздравляю – вы нашли способ провести революцию в Lisp. Уже в который раз.

В сообществе пользователей Haskell спасибо Филиппу Вадлеру (Phillip Wadler) за возможность заглянуть внутрь процесса создания языка Haskell. Мы оба питаем страсть к обучению других людей, и у вас это здорово получается. Спасибо также Саймону Пейтону-Джонсу (Simon Peyton-Jones). Мне было очень приятно общаться с вами, я постарался донести до читателей ваш опыт и ваши знания.

Существенный вклад в создание этой книги внесли рецензенты. Спасибо Владимиру Г. Ивановичу (Vladimir G. Ivanovic), Крей-

гу Рикки (Craig Riecke), Полу Бучеру (Paul Butcher), Фреду Дауду (Fred Daoud), Аарону Бедрею (Aaron Bedra), Давиду Эйзингеру (David Eisinger), Антонио Каньяно (Antonio Cangiano) и Брайану Тарбоксу (Brian Tarbox). Вы сформировали самую эффективную команду рецензентов, с которыми я когда-либо работал. Книги – намного более сложный продукт, чем думают многие. Я знаю, что рецензирование на таком уровне детализации – это неблагодарный и тяжелый труд. Спасибо всем вам, кто способствует выходу технических книг. Без вашего участия издательский бизнес просто прекратил бы свое существование.

Я также хочу поблагодарить тех, кто делился со мной идеями в выборе языков и философий программирования. Не раз важными мыслями со мной делились Нил Форд (Neal Ford), Джон Хайнц (John Heintz), Майк Перхам (Mike Perham) и Ян Воршак (Ian Warshak). Общение с вами помогло мне выглядеть более умным, чем я есть на самом деле.

Хочу также поблагодарить первых читателей рукописи книги и подталкивавших меня к продолжению работы над ней. Ваши комментарии показали, что многие из вас проявили немалое внимание при чтении. Я внес изменения в книгу, опираясь на сотни ваших комментариев, и могу обещать, что продолжу учитывать ваши пожелания в следующих ее изданиях.

Наконец, выражаю самую искреннюю благодарность коллективу издательства Pragmatic Bookshelf. Дейв Томас (Dave Thomas) и Энди Хант (Andy Hunt), вы неоднократно оказывали важное влияние на мою карьеру, сначала как программиста, а потом как автора. Это издательство сделало мой труд как писателя ненапрасным. Благодаря вам книги, не предназначенные для широкой публики, такие как эта, обретают вполне определенную материальную ценность. Спасибо всему коллективу издательства. Джеки Картер, ваши нежные руки и твердое руководство были тем, что необходимо для этой книги, и я надеюсь, что наше общение было для вас таким же приятным, как для меня. Спасибо всем, кто трудился над этой книгой, стараясь сделать ее еще лучше. Особую благодарность я хочу выразить редакторам и корректорам, улучшавшим оформление книги и исправлявшим мои ошибки, и в их числе: литературному редактору Киму Вимпсетту (Kim Wimpsett); составителю алфавитного указателя Сету Мейслину (Seth Maislin); наборщику Стиву Питеру (Steve Peter) и продюсеру Джанет Фарлоу (Janet Furlow). Без вас эта книга стала бы такой, какая она есть сейчас.

Как обычно, в книге все еще могут оставаться ошибки, не замеченные этой великолепной командой. Всем, кому они встретятся, я заранее приношу свои самые искренние извинения. Любые оплошности я допускал непреднамеренно.

Наконец, спасибо всем вам, мои читатели. Я считаю, что бумажные книги имеют определенную ценность, и я могу следовать своему увлечению и писать для вас.

Брюс Тейт (Bruce Tate)

Предисловие

Из еще не написанной книги «How Proust Can Make You a Better Programmer»

Джо Армстронг (Joe Armstrong), создатель языка Erlang

– Редактор Gmail неправильно воспринимает типографские кавычки.

– Печально, – сказала Марджери, – это признак безграмотности программиста и упадка культуры.

– Что будем делать с этим?

– Мы должны настоять, чтобы следующий программист, которого наймем, обязательно прочитал роман «A la recherche du temps perdu»[1].

– Все семь томов?

– Все семь томов.

– Это поможет ему лучше узнать правила пунктуации и правильно интерпретировать кавычки?

– Необязательно, но этот роман сделает его лучшим программистом, потому что в нем он найдет многие секреты мастерства...

Обучение программированию сродни обучению плаванию. Никакая теория не заменит практических занятий, когда обучаемый молотит руками и ногами по воде, судорожно хватая ртом воздух. Впервые погрузившись в воду с головой, вы паникуете, но когда вы качаетесь на поверхности, неторопливо вдыхая воздух, вы чувствуете легкий восторг. Вы думаете: «Я могу плавать!» По крайней мере, именно эта мысль пришла мне в голову, когда я научился плавать.

То же самое происходит, когда вы обучаетесь программированию. Первые шаги даются с большим трудом, и вам нужен хороший тренер, который поможет вам прыгнуть в воду.

Брюс Тейт (Bruce Tate) – именно такой тренер. Эта книга поможет вам сделать самый первый и трудный шаг в обучении программированию.

Предположим, что вы миновали первый сложный этап загрузки и установки интерпретатора или компилятора языка, интересного вам. Что делать дальше? Какую программу написать первой?

---

[1] Пруст М. В поисках утраченного времени. – М.: АСТ, 2011. ISBN: 978-5-17-067438-1. – *Прим. перев.*

Брюс четко и обстоятельно отвечает на этот вопрос. Просто вводите программы и их фрагменты, предлагаемые в книге, и наблюдайте получаемые результаты. Не думайте пока о создании собственной программы – просто пробуйте воспроизводить примеры из книги. Набравшись опыта, вы сможете приступить к созданию собственного проекта.

Первый шаг в обретении новых знаний заключается не в попытке создать что-то свое, а в повторении того, что было уже создано другими. Это самый быстрый способ овладеть навыками программирования.

Первые шаги в изучении нового языка программирования обычно заключаются не в глубоком проникновении в его философию, а в знакомстве с особенностями расстановки точек с запятой и запятых и исследовании сообщений, которые выдает система, когда вы допускаете ошибки. И только когда вы преодолеете первый этап, научившись вводить программы без ошибок и компилировать их, можно начинать задумываться о назначении различных языковых конструкций.

Пройдя первый этап механистического ввода программы и ее запуска, можно откинуться на спинку кресла и расслабиться. Все остальное за вас сделает ваше подсознание. В то время как сознание будет запоминать правила расстановки точек с запятой, подсознание будет стараться проникнуть в суть программных конструкций. И однажды вы обнаружите, что понимаете логику программы и роль данных конструкций в данном языке.

Поверхностное знакомство со многими языками весьма полезно. Я часто обнаруживаю, что знакомство с Python или Ruby помогает мне решить конкретную задачу. Программы, которые я загружаю из Интернета, часто написаны на разных языках, и мне нередко требуется внести в них небольшие изменения перед использованием.

Каждый язык имеет свой набор идиом, свои достоинства и недостатки. Изучив несколько разных языков программирования, вы сможете выбирать тот язык, который лучше подходит для решения текущей задачи.

Мне приятно видеть, что Брюс обладает разнообразием вкусов в отношении языков программирования. Он охватывает не только широко известные языки, такие как Ruby, но также менее распространенные и недооцененные, такие как Io. В конечном счете программирование – это понимание, а понимание – это познание идей. Соответственно, открытость новым идеям дает более глубокое понимание программирования вообще.

Гуру может сказать, что для более глубокого познания математики необходимо изучить латынь. Так же и с программированием. Чтобы полнее понимать объектно-ориентированное программирование, следует изучить логическое или функциональное программирование. Чтобы полнее понимать функциональное программирование, следует изучить программирование на языке Ассемблера.

Когда я еще только учился программированию, большой популярностью пользовались книги, проводящие сравнительный анализ разных языков программирования, но большинство из них были чересчур академичны и почти не несли практических рекомендаций по использованию сравниваемых языков. Такой подход был характерен для того времени. Вы могли многое узнать об идеях, положенных в основу языка, но не имели возможности опробовать их на практике.

В настоящее время мы можем не только знакомиться с идеями, но и пытаться применять их на практике. Это большая разница, как между стоянием на краю бассейна с мыслью «а хорошо ли мне будет в воде» и получением удовольствия от плавания.

Я настоятельно рекомендую прочитать эту книгу и надеюсь, что вам она понравится так же, как и мне.

Джо Армстронг (Joe Armstrong),
создатель языка Erlang
2 марта 2010
Стокгольм

Глава 1

Введение

Люди изучают иностранные языки по самым разным причинам. Первый язык изучается, чтобы жить. Знание родного языка помогает общаться с окружающими. Побудительные мотивы к изучению второго языка у разных людей могут быть разными. Кому-то знание иностранного языка может пригодиться для построения карьеры или для адаптации в новом окружении при смене места жительства. Но иногда решение изучить новый язык принимается не потому, что он необходим, а просто ради желания учиться. Знакомство со вторым языком может расширить кругозор. Кому-то знание каждого нового языка помогает сформировать новый способ мышления.

То же относится и к языкам программирования. В этой книге я представлю вам семь разных языков. Моя цель вовсе не в том, чтобы заставить вас, как это делают многие мамы, заставляя своих чад выпить утром ложку рыбьего жира. Я хочу пригласить вас в увлекательное путешествие, которое изменит ваши взгляды на программирование. Я не буду делать из вас экспертов, но я расскажу вам чуть больше, чем требуется, чтобы написать программу «Hello, World».

1.1. Логика описания

Часто, приступая к изучению нового языка или фреймворка, я стараюсь найти в Интернете интерактивное руководство, чтобы опробовать возможности языка в управляемом окружении. Я могу написать свой сценарий, чтобы заняться дальнейшими исследованиями, но обычно я ищу информацию, которая быстро разбудит во мне интерес, пример синтаксического сахара и описание базовых концепций.

Однако обычно для меня этого недостаточно. Если я встретил новый язык, являющийся более чем тонкой оберткой вокруг уже известного мне языка, мне потребуется более глубокое его исследование. Эта книга даст вам такую возможность целых семь раз. Здесь вы найдете ответы на следующие вопросы:

- *Поддерживаемая модель типов данных.* Строгая (как в Java) или слабая (как в C), статическая (как в Java) или динамическая (как в Ruby). Во всех языках, описываемых в этой книге, используются модели со строгим контролем типов, но среди них вы встретите и статические, и динамические разновидности. Вы увидите, какое влияние на разработчика оказывает модель типов и как она сказывается на способах решения задач. Все языки, представленные в этой книге, имеют модели типов со своими неповторимыми особенностями.
- *Модель программирования.* Объектно-ориентированная (OO), функциональная, процедурная или смешанная? Языки программирования, описываемые в этой книге, поддерживают четыре разные модели программирования, а некоторые – даже комбинации сразу нескольких из них. Здесь вы познакомитесь с языком логического программирования (Prolog), двумя языками, полностью поддерживающими объектно-ориентированные концепции (Ruby, Scala), четырьмя языками, имеющими функциональную природу (Scala, Erlang, Clojure, Haskell), и одним языком, основанном на использовании прототипов (Io). Некоторые языки, такие как Scala, поддерживают сразу несколько парадигм. Мультиметоды в Clojure позволят вам даже реализовать собственную парадигму. Знакомство с новыми парадигмами программирования является одной из важнейших особенностей этой книги.
- *Тип языка.* Компилирующий или интерпретирующий, наличие виртуальной машины. При исследовании языков в данной книге я буду использовать интерактивные оболочки, если таковые имеются. Программный код больших проектов я всегда сохраняю в файлах. Но мы не будем заниматься созданием проектов, достаточно больших, чтобы полностью раскрыть возможности модели пакетов.
- *Конструкции принятия решений и базовые структуры данных.* Возможно, вас удивит, что существует множество языков программирования, в которых поддерживается возможность принятия решений без операторов `if` и `while` и их разновидностей. Вы познакомитесь с приемом сопоставления с образцом в языке Erlang и унификации в языке Prolog. Коллекции играют важную роль практически в любом языке. В одних языках, таких как Smalltalk и Lisp, коллекции являются определяющими характеристиками языка. В других, таких как C++ и Java,

напротив, коллекции организованы нелогично, непоследовательно. Но в любом случае мы обязательно будем знакомиться с поддержкой коллекций.

- *Основные особенности языка, придающие ему уникальность.* Некоторые из представленных языков поддерживают дополнительные возможности, упрощающие программирование параллельных вычислений. Другие предоставляют уникальные высокоуровневые конструкции, такие как макросы в Clojure или средства интерпретации сообщений в Io. Третьи основаны на суперзаряженной виртуальной машине, такой как BEAM в Erlang, благодаря которой Erlang позволяет создавать распределенные отказоустойчивые системы намного быстрее, чем любые другие языки. Некоторые языки поддерживают модели программирования, сфокусированные на решении узкого круга задач, например использование логики для поиска решения в рамках имеющихся ограничений.

Прочитав эту книгу, вы не станете экспертом ни в одном из рассматриваемых здесь языков, но вы *будете* знать об их уникальных особенностях и возможностях. А теперь пройдемся по списку языков.

1.2. Языки

Выбор языков для этой книги был сделан намного проще, чем вам могло бы показаться. Я просто прислушался к пожеланиям потенциальных читателей. К концу голосования имелось восемь кандидатов. Я выбросил из списка JavaScript, потому что он и так пользуется большой популярностью, и заменил его следующим по популярности языком на основе прототипов – Io. Я также убрал Python, потому что я хотел включить в книгу не более одного объектно-ориентированного языка, а выше в списке уже стоял язык Ruby. Благодаря этому в списке освободилось место для неожиданного кандидата, языка Prolog, который вошел в первую десятку. Ниже перечислены языки, которые я выбрал, и перечислены причины, почему я это сделал:

- *Ruby.* Этот объектно-ориентированный язык высоко ценится за простоту использования и удобочитаемость синтаксиса. Первоначально я вообще не хотел включать в книгу объектно-ориентированные языки, но мне хотелось сравнить другие парадигмы программирования с объектно-ориентированной, поэтому было решено включить хотя бы один объектно-ориен-

тированный язык. Кроме того, я хотел бы погрузиться в Ruby чуть глубже, чем многие программисты, и познакомить читателей с основными решениями, повлиявшими на дизайн языка. Также я посчитал нужным окунуться в метапрограммирование на Ruby, позволяющее расширять синтаксис языка. В целом я остался доволен результатом.

- *Io*. Как и Prolog, язык Io – один из наиболее противоречивых языков, включенных в книгу. Он не добился коммерческого успеха, но его конструкции параллельного программирования и однородный синтаксис заслуживают большего внимания. Минимальный синтаксис обладает широкими возможностями, а сходства с языком Lisp иногда прямо-таки бросаются в глаза. Io не имеет обширной родословной, основан на прототипах, подобно JavaScript, и поддерживает уникальный механизм управления сообщениями, познакомиться с которым вам будет очень интересно.
- *Prolog*. Да, я знаю, это очень старый язык, но я также знаю, что он чрезвычайно мощный. Реализация игры судоку на Prolog открыла мне глаза на этот язык. То, что на Prolog получалось почти играючи, на Java или C требовало приложения серьезных усилий. Джо Армстронг (Joe Armstrong), создатель языка Erlang, помог мне более взвешенно оценить этот язык, оказавший немалое влияние на Erlang. Если прежде вам не приходилось сталкиваться с языком Prolog, я думаю, что вы будете приятно удивлены.
- *Scala*. Один из языков нового поколения, основанных на виртуальной машине Java. Язык Scala привнес мощные функциональные концепции в экосистему Java. Он также поддерживает парадигму объектно-ориентированного программирования. Оглядываясь назад, я вижу поразительное сходство с C++, который способствовал сближению процедурного и объектно-ориентированного программирования. По мере знакомства с сообществом Scala вы поймете, почему программистами, исповедующими исключительно функциональный стиль, Scala воспринимается как ересь, а разработчиками на Java – как благословение.
- *Erlang*. Один из старейших языков в этом списке, Erlang позиционируется как функциональный язык, обладающий средствами поддержки параллельных вычислений, а также создания распределенных и отказоустойчивых систем. Создатели

CouchDB, одной из новейших облачных баз данных, выбрали язык Erlang и никогда не жалели о своем выборе. Потратив совсем немного времени на этот язык, вы поймете – почему. Erlang значительно упрощает разработку параллельных, распределенных и отказоустойчивых приложений.

- *Clojure*. Еще один язык на основе JVM. Это – диалект Lisp, реализующий механизмы параллельных вычислений, в корне отличающиеся от тех, что мы привыкли использовать в JVM. Это единственный язык в книге, использующий ту же стратегию версионирования, что применяется в системах управления базами данных для поддержки многозадачного доступа. Будучи диалектом Lisp, язык Clojure обладает огромной мощностью и поддерживает, возможно, самую гибкую модель программирования из всех языков, рассматриваемых в этой книге. Но, в отличие от других диалектов Lisp, он существенно уменьшает количество круглых скобок в исходном коде и опирается на огромную экосистему, включая гигантскую библиотеку Java и повсеместно доступные платформы развертывания.
- *Haskell*. Этот язык является единственным исключительно функциональным языком программирования из числа рассматриваемых в книге. Это означает, что в нем полностью отсутствуют изменяемые переменные. Любая функция на этом языке, вызванная с одними и теми же параметрами, всегда будет возвращать одно и то же значение. Из всех строго типизированных языков Haskell поддерживает самую обширную модель типов. Как и в случае с языком Prolog, потребуется некоторое время, чтобы научиться писать на этом языке, но результат стоит того.

Прошу прощения, если ваш любимый язык не попал в этот список. Я уже получил массу гневных писем по этому поводу. Мы выдвинули на голосование, упоминавшееся выше, несколько десятков языков. Я выбрал для освещения в книге не самые популярные языки, но каждый из них обладает уникальными особенностями, знакомство с которыми пойдет вам на пользу.

1.3. Купите эту книгу

...если вы опытный программист и желаете расширить свой кругозор. Это утверждение может показаться туманным, но не судите меня строго.

Учитесь учиться

Дейв Томас (Dave Thomas) – один из основателей этого издательства – каждый год обучал новым языкам программирования тысячи студентов. В самом худшем случае, изучая новый язык программирования, вы научитесь вкладывать новые концепции в свой код на своем языке.

Работа над этой книга оказала существенное влияние на код, который я пишу на Ruby. Он стал более функциональным, более удобочитаемым, и в нем стало меньше повторяющихся фрагментов. Я реже использую изменяемые переменные и успешнее справляюсь с задачами, применяя блоки кода и функции высшего порядка. Я также использую некоторые приемы, необычные для Ruby, но они делают мой код более кратким и выразительным.

В лучшем же случае вы начнете новую карьеру. Каждые десять лет происходит смена парадигмы программирования. Когда язык Java стал слишком тесным для меня, я начал экспериментировать с Ruby, чтобы лучше понять используемые в нем подходы к разработке веб-приложений. После нескольких успешных побочных проектов я продолжил свою карьеру в этом направлении и никогда не жалел об этом. Моя карьера программиста на Ruby начиналась с простых экспериментов и выросла в то, что выросла.

Где получить помощь в трудный момент

Многие читатели этой книги слишком молоды, чтобы помнить времена, когда в последний раз произошла смена парадигм. Переход на объектно-ориентированную парадигму потерпел несколько неудачных попыток, но беда в том, что старая парадигма структурного программирования просто не справлялась со все возрастающей сложностью требований, предъявляемых к современному программному обеспечению. Успех языка Java послужил серьезным толчком в этом направлении, и новая парадигма заработала. Многие разработчики вынуждены были полностью менять свои навыки, образ мышления, используемые инструменты и подходы к проектированию приложений.

Возможно, мы находимся уже на полпути к следующей трансформации. Но на этот раз толчком к ней будет служить изменение архитектуры компьютеров. Пять языков из семи в этой книге реализуют превосходные модели параллельных вычислений. (Исключениями являются Ruby и Prolog.) Не важно, собираетесь вы поменять свой

язык программирования в ближайшее время или нет, я все же рискну предположить, что некоторые языки из этой книги покажутся вам весьма привлекательными. Попробуйте отложенные операции в Io, акторы в Scala или философию Erlang «позвольте приложению потерпеть неудачу». Разберитесь с тем, как программисты на Haskell обходятся без изменяемых переменных или как Clojure использует механизм версионирования для решения проблем, связанных с многозадачностью.

Интересные решения на языке Erlang можно также найти в некоторых облачных базах данных. Доктор Джо Армстронг (Joe Armstrong) создал этот язык на основе языка Prolog.

1.4. Не покупайте эту книгу

...пока не прочтете этот раздел и не согласитесь со мной. Я собираюсь заключить соглашение с вами. Вы должны согласиться с тем, что основное внимание будет уделяться самим языкам программирования, а не тонкостям их установки. Со своей стороны я постараюсь рассказать вам как можно больше за короткое время. В вашем распоряжении имеется поисковая система Google, поэтому вы не должны полагаться на мою помощь в установке, но когда вы прочтете книгу, объем ваших знаний увеличится, потому что я собираюсь копать достаточно глубоко.

Имейте также в виду, что семь языков – это достаточно честолюбивая цель для нас обоих. Как читателю вам придется настроить свой мозг на знакомство с семью разными видами синтаксиса, четырьмя парадигмами программирования, четырьмя десятилетиями разработки языков и многим другим. Как автор я должен буду охватить широкий спектр тем. Я изучил некоторые из этих языков в процессе работы над данной книгой, и, чтобы охватить наиболее важные особенности каждого языка, мне придется пойти на некоторые упрощения.

Здесь рассказывается не только о синтаксисе, но и о многом другом

Чтобы по-настоящему понять замыслы разработчиков языков, у вас должно быть желание пойти дальше знакомства с основами их синтаксиса. То есть вам придется писать программы посложнее, чем «Hello, World» или даже вычисление чисел Фибоначчи. При знакомстве с языком Ruby вы займетесь метапрограммированием. При

знакомстве с Prolog напишете программу для игры в судоку. А при знакомстве с Erlang – программу мониторинга, которая будет определять момент завершения другого процесса и запускать третий или сообщать пользователю об ошибке.

Своим решением пойти дальше освещения основ я беру на себя обязательство и предлагаю соглашение. Обязательство: я не буду ограничиваться поверхностным освещением. И соглашение: я хочу, чтобы вы признали, что я не в состоянии охватить все основы, которые можно найти в специализированных книгах. Я редко использую механизм обработки исключений, кроме случаев, когда это является основополагающей особенностью языка. Я не буду углубляться в тонкости моделей пакетов, потому что мы будем иметь дело с очень маленькими проектами, и эти знания нам не потребуются. Я не буду заниматься описанием примитивов, не нужных для решения простых задач, которые я буду ставить перед вами.

Здесь не описывается порядок установки

Платформа – вот одна из самых больших моих проблем. У меня был опыт непосредственного общения с читателями различных книг, использующими три разные версии Windows, Mac OS X и как минимум пять разных версий Unix. Еще я видел множество комментариев в различных форумах, где упоминается еще большее разнообразие платформ. Освещение семи языков на семи платформах – это практически неодолимая тема для одного автора и, может быть, даже для нескольких авторов. Я не в состоянии описать установку семи языков, поэтому я не буду даже пытаться сделать это.

Я надеюсь, что вас не интересует устаревшее руководство по установке. Языки и платформы постоянно развиваются. Я скажу вам, куда обратиться, чтобы узнать, как установить язык, и назову номер версии, которую я использую. Используя эту информацию, вы сможете найти свежие инструкции по установке из тех же источников, что и все остальные. Я не смогу дать здесь исчерпывающие инструкции по установке.

Это не справочник по программированию

Я старался создать достаточно полный обзор языков программирования. В некоторых случаях мне даже удалось получить интервью у их создателей. Я уверен, что эта книга очень хорошо передает дух каждого рассматриваемого в ней языка. Тем не менее постарайтесь понять, что я не в состоянии написать исчерпывающие руководства по

использованию всех семи языков. В этом смысле я хотел бы провести аналогию с языками человеческого общения.

Знание языка на уровне, достаточном для туриста, значительно отличается от знания языка на уровне его носителя. Я бегло говорю на английском и с запинками на испанском. Я знаю несколько фраз еще на трех языках. Я смогу заказать рыбу в Японии. Я смогу спросить, как пройти в туалетную комнату в Италии. Но я знаю свои границы. С точки зрения программирования, я бегло говорю на Basic, C, C++, Java, C#, JavaScript, Ruby и нескольких других языках. Я говорю с запинками на десятках других языков, включая и языки, охватываемые этой книгой. Я не являюсь квалифицированным специалистом по шести языкам из этого списка. Последние пять лет я пишу почти исключительно на Ruby. Но я не смогу рассказать вам, как написать веб-сервер на Io или базу данных на Erlang.

Я потерпел бы неудачу, если попытался бы написать исчерпывающий справочник по каждому из этих языков. Я мог бы написать отдельное руководство по программированию на любом из языков, описываемых здесь, но оно получилось бы по объему ничуть не меньше, чем эта книга. Я представлю вашему вниманию достаточно начальной информации. Я покажу вам примеры программ на каждом из языков. Я тщательно проверю все мои примеры, чтобы они компилировались и выполнялись без сучка и задоринки. Но я не смогу помочь вам в ваших попытках приступить к самостоятельному программированию, даже если захотел бы.

Все языки в этом списке имеют исключительно доброжелательные сообщества. И это был один из критериев отбора. Каждый пример я постараюсь сопроводить разделом, в котором буду просить вас найти дополнительные ресурсы. Сделано это преднамеренно – попробовав несколько раз выполнить поиск самостоятельно, вы будете чувствовать себя гораздо увереннее.

Я буду постоянно подталкивать вас

В этой книге я собираюсь пойти чуть дальше, чем в двадцатиминутном руководстве. Вы не хуже меня знакомы с Google и сможете без труда найти простые примеры для каждого из обсуждаемых языков. Я проведу для вас экскурс в каждый из языков. Буду давать вам решать небольшие задачи и по одному программному проекту каждую неделю. Это будет непросто, зато интересно.

Если просто читать эту книгу, вы лишь познакомитесь с несколькими разновидностями синтаксиса, и не более. Если вы будете искать

в Интернете ответы на вопросы, прежде чем опробовать примеры у себя, то наверняка потерпите неудачу. Вы должны сначала попробовать выполнить примеры, понимая, что в некоторых случаях будете ошибаться. Синтаксис всегда дается проще, чем внутренняя философия языка.

Если после прочтения этого описания вы почувствовали неуверенность, я предлагаю выбрать другую книгу, а эту положить обратно на полку – она едва ли вам понравится. Возможно, вам лучше подойдут семь отдельных книг по программированию. Но если вас захватила перспектива быстро окунуться в программный код, тогда давайте двигаться дальше.

1.5. Заключительное замечание

В этой точке мне хотелось написать что-нибудь ободряющее, но все сводится к единственному слову.

Развлекайтесь!

Глава 2

Ruby

Чтобы выпить лекарство, ложку сахара добавь.

Мэри Поппинс

Если вы выбрали эту книгу, скорее всего, у нас с вами есть кое-что общее: знакомство с новыми языками программирования захватывает нас. На мой взгляд, изучение языка программирования сродни изучению характера. В течение своей карьеры я попробовал множество языков. Подобно человеку, каждый язык обладает своими отличительными особенностями. Язык Java напоминает мне богатого адвоката. Это был забавный язык, пока он был молод, но сейчас он высасывает радость программирования на 100 миль вокруг, подобно черной дыре. Visual Basic напоминает мне крашеную блондинку. Она не собирается заниматься проблемой глобального потепления, но падка на новые прически и готова болтать на эту тему часами. На протяжении всей книги я буду сравнивать языки с популярными персонажами и надеюсь, что эти сравнения помогут вам хоть немного раскрыть характер каждого языка.

Встречайте: язык Ruby – один из моих любимчиков. Иногда немного эксцентричный, но всегда прекрасный, слегка таинственный и абсолютно волшебный. Он напоминает мне Мэри Поппинс – британскую няню[1]. Были времена, когда в большинстве своем няни напоминали C-подобные языки – жуткие, но чертовски эффективные создания, выстреливавшие ложкой рыбьего жира на ночь. С добавлением ложки сахара все изменилось. Мэри Поппинс подняла воспитание детей на новый уровень, привнеся в него элемент игры. Ruby сделал практически то же самое, добавив синтаксического сахара[2] намного

---

[1] Персонаж из кинофильма «Mary Poppins». Режиссер Роберт Стивенсон (Robert Stivenson). 1964; Лос-Анжелес, Калифорния: Walt Disney Video, 2004. (http://cinema.mosfilm.ru/films/film/Meri-Poppins-do-svidaniya/meri-poppins-do-svidaniya-1-seriya/ – *Прим. перев.*)

[2] Под синтаксическим сахаром понимаются особенности языка, которые упрощают чтение и написание кода и дают возможность выразить одни и те же действия разными способами.

больше, чем одну ложку. Мац (Matz), создатель Ruby, не беспокоился об эффективности языка. Он оптимизировал эффективность труда программистов.

2.1. Краткая история

Юкихиро Мацумото (Yukihiro Matsumoto) создал язык Ruby в 1993 году. Большинство называют автора просто Мац (Matz). Ruby – интерпретирующий, объектно-ориентированный язык с динамической типизацией, относящийся к категории так называемых языков сценариев. «Интерпретирующий» означает, что код на языке Ruby выполняется интерпретатором, то есть он не компилируется в машинный код. «Динамическая типизация» означает, что типы переменных определяются на этапе выполнения, а не на этапе компиляции. Благодаря этому достигается высокая гибкость языка, хотя и ценой безопасности во время выполнения, но об этом мы поговорим позже. «Объектно-ориентированный» означает, что язык поддерживает инкапсуляцию (данные и их поведение объединяются вместе), наследование через классы (типы объектов образуют иерархические деревья классов) и полиморфизм (объекты могут принимать разные формы). Ruby терпеливо ждал подходящего момента и вышел на сцену в 2006 году в составе фреймворка Rails. Для многих программистов, блуждавших в корпоративных (enterprise) джунглях почти десять лет, программирование снова превратилось в забаву. Ruby не слишком эффективен, с точки зрения скорости выполнения, но он весьма положительно сказался на эффективности труда программистов.

Интервью с Юкихиро Мацумото (Мац)

Мацумото любезно пригласил меня посетить город Мацуэ в Японии. У нас появилась возможность поговорить об истории происхождения Ruby, и он согласился ответить на некоторые вопросы для этой книги.

Брюс: Что побудило вас написать Ruby?

Мац: Начав экспериментировать с компьютерами, я заинтересовался языками программирования. Они не только дают возможность писать программы, но и формируют ваши представления о программировании. Поэтому в течение долгого времени я с увлечением изучал множество языков программирования. Я даже написал несколько собственных языков, но все они были игрушечными, и среди них не было ни одного настоящего.

В 1993 году, когда я познакомился с языком Perl, у меня возникло непреодолимое желание написать объектно-ориентированный язык, объединяющий в себе лучшие качества Lisp, Smalltalk и Perl и помогающий повышать нашу продуктивность. В том же году я приступил к созданию своего языка и назвал его Ruby. Главной целью для меня было развлечение. Сначала это было всего лишь хобби. Я пытался создать язык, соответствующий моим наклонностям. Так или иначе, этот язык пришелся по вкусу другим программистам во всем мире. В конечном итоге он приобрел популярность, о которой я даже не мечтал.

Брюс: Что больше всего вам нравится в этом языке?

Мац: Мне нравится, как он превращает программирование в приятное занятие. В частности, больше всего мне нравятся блоки. Они представляют собой дрессированные функции высшего порядка, открывающие широкие возможности в области предметно-ориентированного программирования.

Брюс: Что бы вы изменили в языке, будь у вас возможность вернуться назад?

Мац: Я убрал бы поддержку потоков выполнения и добавил акторы и другие, более совершенные механизмы поддержки параллельных вычислений.

В процессе чтения данной главы, независимо от того, знаете вы язык Ruby или нет, постарайтесь заметить компромиссы, на который пошел Мац. Поищите тот самый синтаксический сахар – маленькие особенности, нарушающие основные правила языка, – чтобы сделать его более дружественным по отношению к программистам. Попробуйте заметить, где Мац использовал блоки кода для достижения бесподобного эффекта в коллекциях, и не только. И попытайтесь понять, где автору пришлось идти на компромиссы между простотой и безопасностью или продуктивностью и производительностью.

Итак, начнем. Взгляните на следующий фрагмент кода на языке Ruby:

```
>> properties = ['object oriented' , 'duck typed' , 'productive' , 'fun' ]
=> ["object oriented" , "duck typed" , "productive" , "fun" ]
>> properties.each {|property| puts "Ruby is #{property}." }
Ruby is object oriented.
Ruby is duck typed.
Ruby is productive.
Ruby is fun.
=> ["object oriented" , "duck typed" , "productive" , "fun" ]
```

Ruby – это язык, вновь научивший меня улыбаться. Динамичный по своей природе, он имеет сплоченное и дружелюбное сообщество пользователей. Все реализации языка выпускаются с открытыми исходными текстами. Основная коммерческая поддержка исходит от маленьких компаний, и это обстоятельство защитило Ruby от излишних ограничений, ставших бичом в некоторых других областях. Ruby медленно проникал в корпоративный сектор, но теперь он надежно удерживает занятые позиции благодаря своей продуктивности, особенно в области разработки веб-приложений.

2.2. День 1: Поиск няни

Следует отметить, что, кроме того что она волшебница, Мэри Поппинс – еще и великий педагог. Приступая к изучению нового языка программирования, наша задача заключается в том, чтобы узнать, как с его помощью выполнить уже знакомую нам работу. Воспринимайте это знакомство с Ruby как диалог. Будет ли этот диалог течь свободно или в нем будут возникать неловкие паузы? Какая модель программирования используется в качестве основы? Как интерпретируются типы данных? Давайте начнем с поиска ответов на некоторые вопросы.

Молниеносный тур

Как уже было обещано, я не буду даже пытаться провести через процедуру установки Ruby, в котором, впрочем, нет ничего сложного. Просто перейдите на страницу http://www.ruby-lang.org/ru/downloads/, найдите свою платформу и установите версию Ruby 1.8.6 или выше. В этой главе я пользуюсь версией Ruby 1.8.7 – версия 1.9 имеет некоторые небольшие отличия. Для пользователей Windows существует самоустанавливающийся дистрибутив, запускаемый одним щелчком мыши, а пользователи OS X Leopard или более новых версий могут найти дистрибутив Ruby в виде дисков Xcode.

Чтобы проверить установку, просто введите команду `irb`. Если в результате вы не увидите сообщения об ошибке, значит, вы готовы продолжить чтение оставшейся части главы. Если это не так – не расстраивайтесь. Серьезные проблемы при установке возникают достаточно редко. Поэтому просто обратитесь за помощью к Google.

Использование Ruby в консоли

Если вы этого еще не сделали, введите команду `irb`. В результате должен запуститься интерактивный сеанс Ruby. В этом сеансе вы вводите команду и тут же получаете результат. Попробуйте ввести следующее:

```
>> puts 'hello, world'
hello, world
=> nil
>> language = 'Ruby'
=> "Ruby"
>> puts "hello, #{language}"
hello, Ruby
=> nil
>> language = 'my Ruby'
=> "my Ruby"
>> puts "hello, #{language}"
hello, my Ruby
=> nil
```

Если вы еще не знакомы с Ruby, этот короткий пример многое может рассказать вам о языке. Вы уже знаете, что Ruby может действовать как интерпретатор. Ruby практически всегда действует как интерпретатор, однако некоторые разработчики могут использовать виртуальные машины, компилирующие исходный программный код на Ruby в выполняемый байт-код. Здесь не объявляется никаких переменных. Все команды в примере возвращают некоторые значения, даже при том, что я не просил Ruby, чтобы он возвращал что-либо. Фактически каждый фрагмент кода на Ruby возвращает некоторый результат.

Из примера также видно, что в Ruby поддерживается по меньшей мере два типа строк. Первый – строки в одиночных кавычках, которые интерпретируются буквально, и второй тип – строки в двойных кавычках, допускающие возможность дополнительной интерпретации своего содержимого. Первое, на что обращает внимание интерпретатор Ruby в таких строках, – замещаемые параметры. В данном примере Ruby замещает параметр `#{language}` его значением. Продолжим.

Модель программирования

Один из первых вопросов о языке, на который следует ответить: «Какая модель программирования используется?» Ответ на этот вопрос не всегда очевиден. Вероятно, вы знакомы с процедурными языками, такими как C, Fortran или Pascal. Большинство из нас в настоящее время пользуется объектно-ориентированными языками, но многие из этих языков поддерживают и процедурные элементы. Например, число 4 в языке Java не является объектом. Возможно, вы приобрели эту книгу, чтобы познакомиться поближе с языками функционального программирования. Некоторые из таких языков, например Scala, являются смесью разных моделей программирования, добавляя под-

держку объектно-ориентированных понятий. Кроме того, существуют десятки других моделей программирования. В языках программирования, ориентированных на выполнение операций со стеком, таких как PostScript или Forth, главной особенностью является использование одного или более стеков. Языки логического программирования, такие как Prolog, опираются на определения правил. Языки, основанные на прототипах, такие как Io, Lua и Self, в качестве основы для создания объектов и даже выстраивания иерархий наследования используют сами объекты, а не классы.

Ruby – исключительно объектно-ориентированный язык. В этой главе вы увидите, насколько глубоко это понятие укоренилось в языке Ruby. Давайте рассмотрим несколько простейших объектов:

```
>> 4
=> 4
>> 4.class
=> Fixnum
>> 4 + 4
=> 8
>> 4.methods
=> ["inspect", "%", "<<", "singleton_method_added", "numerator", ...
"*", "+", "to_i", "methods", ...
]
```

Я опустил некоторые из методов ради экономии места, но если вы опробуете данный пример, то вы получите полный список. Все сущее в языке Ruby является объектами, даже каждое отдельно взятое число. Число – это объект, относящийся к классу `Fixnum` и обладающий методом с именем `methods`, который возвращает массив методов (интерпретатор Ruby представляет массивы, заключая их в квадратные скобки). Вызвать любой метод объекта можно с помощью оператора «точка».

Условные конструкции

Программы пишутся, чтобы принимать решения, поэтому важно понимать, что способ принятия решений в языке является одним из основных понятий, формирующих приемы программирования на данном языке. В Ruby, как и в большинстве объектно-ориентированных и процедурных языков, имеется несколько способов принятия решений. Взгляните на следующие выражения:

```
>> x = 4
=> 4
>> x < 5
```

```
=> true
>> x <= 4
=> true
>> x > 4
=> false
>> false.class
=> FalseClass
>> true.class
=> TrueClass
```

Итак, в Ruby существуют выражения, которые могут возвращать значения `true` или `false`. Самое интересное, что значения `true` и `false` также являются объектами. С их помощью можно организовать условное выполнение фрагментов кода:

```
>> x = 4
=> 4
>> puts 'This appears to be false.' unless x == 4
=> nil
>> puts 'This appears to be true.' if x == 4
This appears to be true.
=> nil
>> if x == 4
>>   puts 'This appears to be true.'
>> end
This appears to be true.
=> nil
>> unless x == 4
>>   puts 'This appears to be false.'
>> else
?>   puts 'This appears to be true.'
>> end
This appears to be true.
=> nil
>> puts 'This appears to be true.' if not true
=> nil
>> puts 'This appears to be true.' if !true
=> nil
```

Я люблю Ruby за простоту условных конструкций. Пр использовании операторов `if` и `unless` можно применять блочную форму (`if` *условие*, *инструкции*, `end`) или однострочную (*инструкция* `if` *условие*). Кому-то однострочная версия может показаться отталкивающей, но что касается меня, то, на мой взгляд, она позволяет выразить свою мысль единственной строкой кода:

```
order.calculate_tax unless order.nil?
```

Несомненно, то же самое можно записать в блочной форме, но при этом придется добавить лишние конструкции, загромождающие

мысль, которую проще выразить единственной строкой. Когда вы научитесь выражать свои мысли одной строкой, ваш код будет читаться намного проще. Мне также нравится сама идея введения оператора `unless` (если не). То же самое условие можно выразить с помощью оператора `not` или `!`, но применение `unless` делает выражение идеи более наглядным.

То же относится к операторам циклов `while` и `until`:

```
>> x = x + 1 while x < 10
=> nil
>> x
=> 10
>> x = x - 1 until x == 1
=> nil
>> x
=> 1
>> while x < 10
>>   x = x + 1
>>   puts x
>> end
2
3
4
5
6
7
8
9
10
=> nil
```

Обратите внимание, что знак = является оператором присваивания, а два знака == проверяют равенство операндов. В языке Ruby каждый объект поддерживает понятие равенства. Числа равны, если равны их значения.

В качестве условных выражений можно также использовать любые значения, отличные от `true` и `false`:

```
>> puts 'This appears to be true.' if 1
This appears to be true.
=> nil
>> puts 'This appears to be true.' if 'random string'
(irb):31: warning: string literal in condition
This appears to be true.
=> nil
>> puts 'This appears to be true.' if 0
This appears to be true.
=> nil
```

```
>> puts 'This appears to be true.' if true
This appears to be true.
=> nil
>> puts 'This appears to be true.' if false
=> nil
>> puts 'This appears to be true.' if nil
=> nil
```

Любые значения, кроме `nil` и `false`, интерпретируются как `true`. Программисты на C и C++, обратите внимание – значение `0` интерпретируется как `true`!

Логические операторы Ruby действуют точно так же, как в C, C++, C# и Java, за некоторыми небольшими исключениями. Оператор `and` (и его альтернативная форма `&&`) выполняет логическую операцию И. Оператор `or` (и его альтернативная форма `||`) выполняет логическую операцию ИЛИ. Вычисление условных выражений продолжается интерпретатором лишь до тех пор, пока результат всего выражения не станет очевиден. Для условного выполнения целых выражений используйте операторы `&` и `|`. Следующий пример демонстрирует принцип действия всех этих операторов:

```
>> true and false
=> false
>> true or false
=> true
>> false && false
=> false

>> true && this_will_cause_an_error
NameError: undefined local variable or method 'this_will_cause_an_error'
    for main:Object
        from (irb):59
>> false && this_will_not_cause_an_error
=> false
>> true or this_will_not_cause_an_error
=> true
>> true || this_will_not_cause_an_error
=> true
>> true | this_will_cause_an_error
NameError: undefined local variable or method 'this_will_cause_an_error'
    for main:Object
        from (irb):2
        from :0
>> true | false
=> true
```

Здесь нет никакого волшебства. В Ruby часто используются версии команд, вычисляемые по короткой схеме.

«Утиная» типизация

Теперь познакомимся с моделью типов в языке Ruby. Первое, что следует знать, – как Ruby защищает от ошибочного использования типов. Здесь я имею в виду безопасность типов. В строго типизированных языках каждая операция проверяет типы своих операндов перед выполнением. Такие проверки могут выполняться при передаче кода интерпретатору или компилятору или во время его выполнения. Взгляните на следующий листинг:

```
>> 4 + 'four'
TypeError: String can't be coerced into Fixnum
        from (irb):51:in '+'
        from (irb):51

>> 4.class
=> Fixnum
>> (4.0).class
=> Float
>> 4 + 4.0
=> 8.0
```

Итак, Ruby является языком со строгим контролем типов[1], в том смысле, что он сообщает об ошибке, если типы операндов оказываются несовместимыми. Проверка типов операндов производится во время выполнения, а не во время компиляции. Чтобы доказать это, я покажу вам определение функции чуть раньше, чем следовало бы. Ключевое слово `def` определяет новую функцию, но не выполняет ее. Введите следующий код:

```
>> def add_them_up
>>   4 + 'four'
>> end
=> nil
>> add_them_up
TypeError: String can't be coerced into Fixnum
        from (irb):56:in '+'
        from (irb):56:in 'add_them_up'
        from (irb):58
```

---

[1] Здесь я немного погрешил против истины, но совсем немного. Двумя примерами ниже вы увидите, как я изменяю класс значения во время выполнения. Теоретически пользователь может изменить класс до неузнаваемости и обмануть механизм контроля типов, поэтому нельзя сказать, что Ruby является строго типизированным языком, в полном смысле. Но по большей части Ruby действует как строго типизированный язык.

Итак, Ruby не проверяет типы операндов, пока действительно не попытается выполнить операцию. Эта особенность называется *динамической типизацией*. Она имеет свои недостатки: вы не сможете обнаружить все ошибки в коде, как это позволяют компиляторы и другие инструменты в языках со статической системой типов. Но система типов в Ruby имеет свои достоинства. Классы не обязательно должны иметь общего предка, чтобы использоваться в одних и тех же операциях:

```
>> i = 0
=>0
>> a = ['100', 100.0]
=> ['100', 100.0]
>> while i < 2
>>   puts a[i].to_i
>>   i = i + 1
>> end
100
100
```

Вы только что наблюдали «утиную» типизацию в действии. Первый элемент массива имеет тип `String`, а второй – тип `Float`. Один и тот же код преобразует каждый элемент в целочисленное значение с помощью метода `to_i`. Механизм «утиной» типизации не заботится об исходном типе оригинального значения. Если он ходит как утка и крякает как утка, это – утка[1]. В данном случае под «крякает» подразумевается метод `to_i`.

Поддержка «утиной» типизации приобретает особенно большое значение в исключительно объектно-ориентированных архитектурах. Важнейшим принципом философии проектирования подобных архитектур является опора на интерфейсы объектов, а не на их реализацию. Если вы используете «утиную» типизацию, вам не составит труда придерживаться данной философии. Если объект имеет методы `push` и `pop`, его можно интерпретировать как стек.

Что мы узнали в первый день

К настоящему моменту мы познакомились только с самыми основами. Ruby – это интерпретирующий, объектно-ориентированный язык. Все сущее в нем является объектами, и мы легко можем добраться до любых частей этих объектов, таких как методы или класс. В языке

---

[1] http://ru.wikipedia.org/wiki/Утиная_типизация – *Прим. перев.*

Ruby используется механизм «утиной» типизации, и Ruby действует подобно языкам со строгим контролем типов, хотя некоторые академики могли бы оспорить последнее утверждение. Это – свободный и открытый язык, который дает вам возможность изменить в нем все, что угодно, вплоть до базовых классов, таких как `NilClass` и `String`. А теперь я дам вам задание для самостоятельного решения.

День 1: задания для самостоятельного решения

Итак, закончился первый день знакомства с языком Ruby. Я предлагаю вам самостоятельно написать программный код. Я не прошу вас написать законченную программу, я лишь хочу, чтобы вы выполнили несколько фрагментов на Ruby в интерактивной оболочке `irb`. Ответы на вопросы можно найти за пределами книги.

Найдите:

- описание Ruby API;
- свободную электронную версию книги «Programming Ruby: The Pragmatic Programmer's Guide» [TFH08];
- метод, выполняющий подстановку в строках;
- информацию о поддержке регулярных выражений в Ruby;
- информацию о диапазонах в Ruby.

Практические задания:

- выведите строку «Hello, world»;
- определите индекс слова «Ruby» в строке «Hello, Ruby»;
- выведите свое имя десять раз;
- выведите строку «This is sentence number 1» десять раз так, чтобы вместо числа 1 выводились числа от 1 до 10;
- запустите программу на Ruby, хранящуюся в файле;
- дополнительное задание: если вам не терпится написать что-нибудь самому, напишите программу, загадывающую случайное число и дающую игроку, запустившему ее, возможность угадать его, сообщая в ответ на ввод числа о том, является ли оно больше или меньше загаданного числа (подсказка: `rand(10)` генерирует случайные числа от 0 до 9, а `gets` читает строку с клавиатуры, которую можно преобразовать в целое число).

2.3. День 2: Спускаемся с небес

Одной из самых захватывающих сцен в фильме о Мэри Поппинс было ее появление. Она плыла по небу под своим зонтиком. Мои дети никогда не поймут, что такого особенного в этой сцене. Сегодня вам

предстоит испытать волшебство языка Ruby. Вы будете учиться использовать стандартные строительные блоки – объекты, коллекции и классы. Вы также познакомитесь с простейшими случаями применения блоков кода. Итак, приготовьтесь прикоснуться к волшебству.

Определение функций

В отличие от Java и C#, вам не придется конструировать целый класс, чтобы определить функцию. Функцию можно определить прямо в консоли:

```
>> def tell_the_truth
>>   true
>> end
```

Каждая функция возвращает какое-нибудь значение. Если возвращаемое значение не определено явно, функция будет возвращать результат последнего выражения, вычисленного перед выходом. Как и все сущее в языке Ruby, эта функция является объектом.

Позднее мы узнаем, как передавать функции другим функциям в качестве параметров.

Массивы

Массивы в Ruby являются основным представителем упорядоченных коллекций. В версии Ruby 1.9 появились также упорядоченные хэши, тем не менее массивы остаются основными упорядоченными коллекциями. Взгляните:

```
>> animals = ['lions', 'tigers', 'bears']
=> ["lions", "tigers", "bears"]
>> puts animals
lions
tigers
bears
=> nil
>> animals[0]
=> "lions"
>> animals[2]
=> "bears"
>> animals[10]
=> nil
>> animals[-1]
=> "bears"
>> animals[-2]
=> "tigers"
>> animals[0..1]
```

```
=> ['lions', 'tigers']
>> (0..1).class
=> Range
```

Как видите, коллекции в Ruby дают некоторую свободу обращения с ними. Если обратиться к отсутствующему элементу массива, Ruby просто вернет `nil`. Массивы также обладают некоторыми особенностями, которые не делают их более мощными, зато упрощают операции над ними. Выражение `animals[-1]` вернет первый элемент массива с конца, `animals[-2]` – второй элемент с конца и т. д. Эти особенности называются *синтаксическим сахаром*, они были добавлены исключительно для удобства. Выражение `animals[0..1]` тоже может выглядеть как синтаксический сахар, но в действительности это не так. Выражение `0..1` – это экземпляр класса `Range`, подразумевающий коллекцию всех целых чисел в диапазоне от 0 до 1 включительно.

Массивы способны хранить значения разных типов:

```
>> a[0] = 0
NameError: undefined local variable or method 'a' for main:Object
    from (irb):23
>> a = []
=> []
```

Ой! Я попробовал использовать переменную как массив еще до того, как она стала массивом. Эта ошибка подсказывает, как в действительности работают массивы и хэши в Ruby. В действительности конструкция `[]` является методом класса `Array`:

```
>> [1].class
=> Array
>> [1].methods.include?('[]')
=> true
>> # используйте [1].methods.include?(:[]) в ruby 1.9
```

Итак, конструкции `[]` и `[]=` являются всего лишь синтаксическим сахаром, упрощающим доступ к элементам массивов. Чтобы избежать ошибки, с которой мы столкнулись выше, необходимо сначала создать пустой массив и только потом выполнять операции с ним:

```
>> a[0] = 'zero'
=> "zero"
>> a[1] = 1
=> 1
>> a[2] = ['two', 'things']
=> ["two", "things"]
>> a
=> ["zero", 1, ["two", "things"]]
```

Массивы необязательно должны быть однородными.

```
>> a = [[1, 2, 3], [10, 20, 30], [40, 50, 60]]
=> [[1, 2, 3], [10, 20, 30], [40, 50, 60]]
>> a[0][0]
=> 1
>> a[1][2]
=> 30
```

Многомерные массивы — это всего лишь массивы массивов.

```
>> a = [1]
=> [1]
>> a.push(1)
=> [1, 1]
>> a = [1]
=> [1]
>> a.push(2)
=> [1, 2]
>> a.pop
=> 2
>> a.pop
=>1
```

Массивы имеют весьма богатый API. Массив можно использовать как очередь, связанный список, стек или множество. А теперь познакомимся с другим типом коллекций в Ruby – хэшем.

Хэши

Напомню, что коллекции – это хранилища объектов. В хэше каждому хранимому объекту присваивается метка. Метка называется ключом, а объект – его значением. Таким образом, хэш является хранилищем пар ключ/значение:

```
>> numbers = {1 => 'one', 2 => 'two'}
=> {1=>"one", 2=>"two"}
>> numbers[1]
=> "one"
>> numbers[2]
=> "two"
>> stuff = {:array => [1, 2, 3], :string => 'Hi, mom!'}
=> {:array=>[1, 2, 3], :string=>"Hi, mom!"}
>> stuff[:string]
=> "Hi, mom!"
```

Ничего сложного. Хэши во многом подобны массивам, только вместо целочисленных индексов при работе с ними можно использовать произвольные ключи. Обратите внимание на последний хэш – здесь

я впервые использовал *символическое имя*. Символическое имя – это идентификатор, начинающийся с двоеточия, например: `:symbol`. Символические имена отлично подходят для именования элементов хэшей. Несмотря на то что две строки с одинаковыми значениями могут быть физически разными объектами, идентичные символические имена всегда ссылаются на один и тот же физический объект. Убедиться в этом можно, попытавшись получить уникальный идентификатор объекта несколько раз, как показано ниже:

```
>> 'string'.object_id
=> 3092010
>> 'string'.object_id
=> 3089690
>> :string.object_id
=> 69618
>> :string.object_id
=> 69618
```

Хэши иногда используются в самых необычных ситуациях. Например, Ruby не поддерживает именованных параметров, однако их можно имитировать с помощью хэша. Добавьте немного синтаксического сахара, и вы получите кое-что любопытное:

```
>> def tell_the_truth(options={})
>>   if options[:profession] == :lawyer
>>     'it could be believed that this is almost certainly not false.'
>>   else
>>     true
>>   end
>> end
=> nil
>> tell_the_truth
=> true
>> tell_the_truth :profession => :lawyer
=> "it could be believed that this is almost certainly not false."
```

Этот метод принимает один необязательный параметр. Если вызвать его без аргумента, параметр `options` будет инициализирован пустым хэшем. Если передать ему ключ `:profession` со значением `:lawyer`, он вернет строку. Результат будет не совсем истинным, но в этом нет ничего страшного, потому что система все равно будет интерпретировать его как истину. Обратите внимание, что в данном случае не потребовалось заключать аргумент в квадратные скобки. Эти скобки необязательны, если хэш является последним аргументом в вызове функции. Так как элементами массива, ключами и значениями хэша может быть почти все, что угодно, с их помощью можно конструиро-

вать невероятно сложные структуры данных, но истинная мощь языка проявляется при использовании блоков кода.

Блоки кода и инструкция yield

Блок кода – это функция без имени. Его можно передать в виде параметра другой функции или методу. Например:

```
>> 3.times {puts 'hiya there, kiddo'}
hiya there, kiddo
hiya there, kiddo
hiya there, kiddo
```

Код в фигурных скобках называется *блок кода*. `times` – это метод класса `Fixnum`, который просто выполняет все, что ему будет передано, *некоторое* число раз, где *некоторое* – это значение числа, являющегося экземпляром класса `Fixnum`. Блоки кода можно заключать в фигурные скобки `{}` или в операторы `do/end`. Когда блок кода умещается в одну строку, его принято заключать в фигурные скобки, а операторы `do/end` используются для оформления блоков кода, располагающихся на нескольких строках. Блоки кода могут принимать параметры:

```
>> animals = ['lions and ', 'tigers and', 'bears', 'oh my']
=> ["lions and ", "tigers and", "bears", "oh my"]
>> animals.each {|a| puts a}
lions and
tigers and
bears
oh my
```

Этот пример демонстрирует широту возможностей блоков кода. Он определяет действия, какие должен выполнить интерпретатор Ruby для каждого элемента в коллекции. В данном случае Ruby выполнит итерации по всем элементам и выведет каждый из них в отдельности. Чтобы вы могли понять, что собственно происходит за кулисами, ниже приводится упрощенная реализация метода `times`:

```
>> class Fixnum
>>   def my_times
>>     i = self
>>     while i > 0
>>       i = i - 1
>>       yield
>>     end
>>   end
>> end
=> nil
>> 3.my_times {puts 'mangy moose'}
```

```
mangy moose
mangy moose
mangy moose
```

Этот код открывает существующий класс и добавляет в него новый метод. В данном случае метод с именем `my_times` выполняет цикл данное число раз, вызывая блок кода с помощью инструкции `yield`. Блоки также могут передаваться как обычные параметры. Взгляните на следующий пример:

```
>> def call_block(&block)
>>   block.call
>> end
=> nil
>> def pass_block(&block)
>>   call_block(&block)
>> end
=> nil
>> pass_block {puts 'Hello, block'}
Hello, block
```

Этот прием позволяет передавать выполняемый код между программными компонентами. Блоки могут применяться не только для выполнения в ходе итераций. В Ruby с их помощью можно определять отложенные действия...

```
execute_at_noon { puts 'Beep beep... time to get up'}
```

выполнять некоторые действия по условию...

```
...некоторый код...
in_case_of_emergency do
  use_credit_card
  panic
end

def in_case_of_emergency
  yield if emergency?
end
...еще код...
```

принудительно выполнять некоторые операции...

```
within_a_transaction do
  things_that
  must_happen_together
end

def within_a_transaction
  begin_transaction
```

```
  yield
  end_transaction
end
```

и многое другое. Вам встретятся библиотеки на Ruby, использующие блоки для построчной обработки файлов, для реализации взаимодействий по протоколу HTTP и для выполнения сложных операций с коллекциями. Ruby – большой любитель блоков.

Запуск файлов сценариев на Ruby

Примеры программного кода будут постепенно усложняться, из-за чего работать с ними в интерактивной оболочке становится неудобно. Вы еще будете использовать консоль для опробования коротких фрагментов, но в основном весь код вы будете помещать в файлы. Создайте файл с именем `hello.rb` и поместите в него любой код по своему выбору, например:

```
puts 'hello, world'
```

Сохраните файл в текущем каталоге и запустите его на выполнение из командной строки:

```
batate$ ruby hello.rb
hello, world
```

Немногие используют полноценные интегрированные среды разработки для программирования на Ruby, большинству вполне достаточно простого текстового редактора. Лично я предпочитаю пользоваться редактором TextMate, однако расширения с поддержкой Ruby имеются также для vi, emacs и многих других популярных редакторов. Памятуя об этом, можно приступать к созданию строительных блоков для программ на языке Ruby.

Определение классов

Как и в Java, C# и C++, в языке Ruby также имеются классы и объекты. Представьте себе формы для выпечки печенья и само печенье. Так вот классы – это формы для «выпечки» объектов. Разумеется, Ruby также поддерживает наследование. В отличие от C++, класс в языке Ruby может иметь только одного родителя, называемого *суперклассом*. Чтобы увидеть механизм наследования в действии, введите в консоли следующие команды:

```
>> 4.class
=> Fixnum
>> 4.class.superclass
```

```
=> Integer
>> 4.class.superclass.superclass
=> Numeric
>> 4.class.superclass.superclass.superclass
=> Object
>> 4.class.superclass.superclass.superclass.superclass
=> nil
```

Итак, объекты создаются на основе классов. Объект `4` создан на основе класса `Fixnum`, который наследует класс `Integer`, `Numeric` и в конечном итоге `Object`.

Взгляните на рис. 2.1, где изображена эта иерархия наследования. Все классы в Ruby в конечном итоге наследуют класс `Object`. Класс `Class` наследует класс `Module`. Экземпляры класса `Class` служат шаблонами для объектов. В нашем случае `Fixnum` – это экземпляр класса `Class`, а `4` – экземпляр класса `Fixnum`. Каждый из этих классов сам является объектом:

```
>> 4.class.class
=> Class
>> 4.class.class.superclass
=> Module
>> 4.class.class.superclass.superclass
=> Object
```

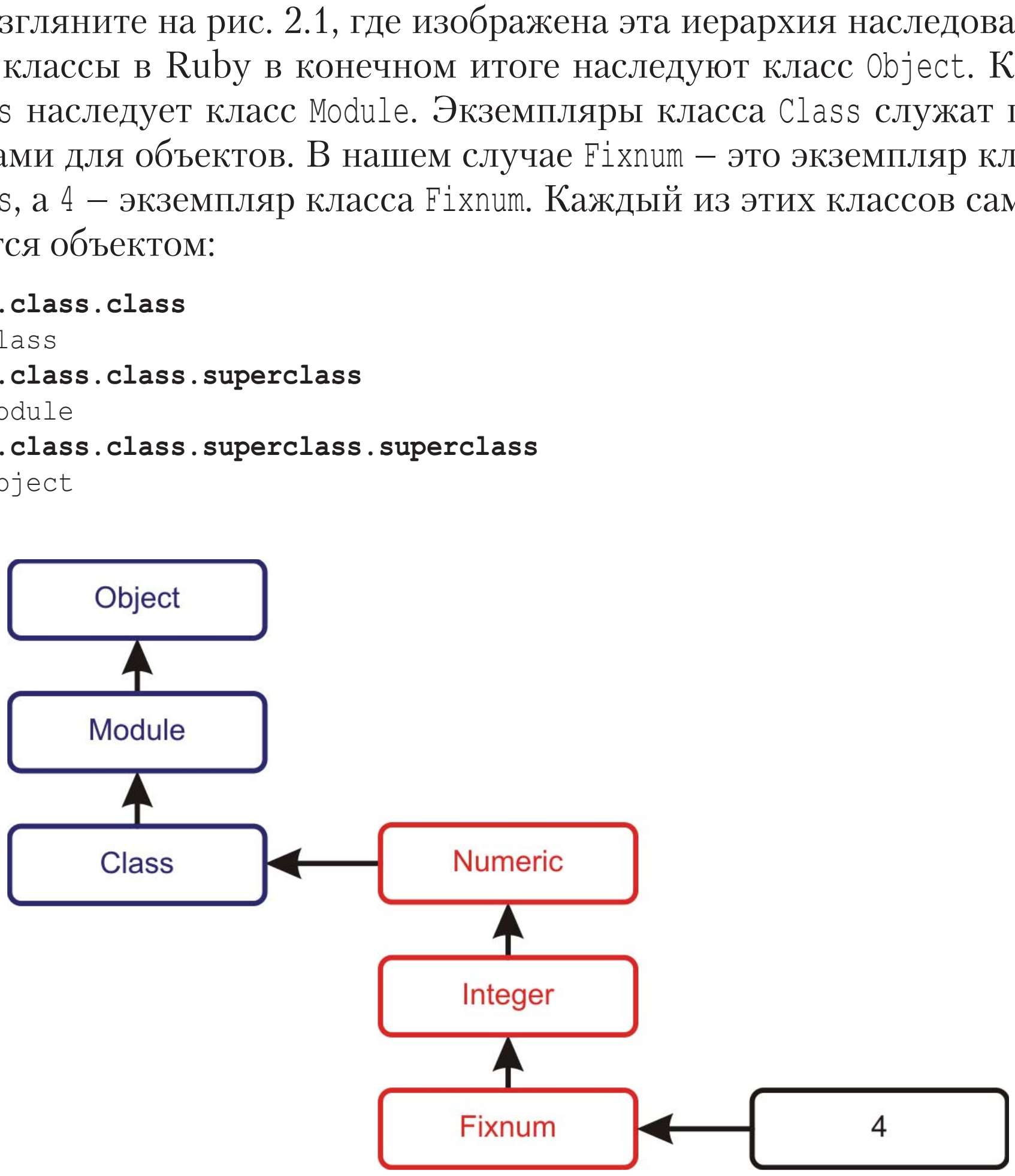

Рис. 2.1 ❖ Метамодель Ruby

То есть `Fixnum` наследует класс `Class`. С этого момента объектная модель начинает становиться все более запутанной. Класс `Class` на-

следует класс Module, а Module наследует Object. В конечном счете все объекты в Ruby имеют общего предка, Object.

ruby/tree.rb
http://media.pragprog.com/titles/btlang/code/ruby/tree.rb

```
class Tree
  attr_accessor :children, :node_name

  def initialize(name, children=[])
    @children = children
    @node_name = name
  end

  def visit_all(&block)
    visit &block
    @children.each {|c| c.visit_all &block}
  end

  def visit(&block)
    block.call self
  end
end

ruby_tree = Tree.new( "Ruby" ,
  [Tree.new("Reia" ),
   Tree.new("MacRuby" )])

puts "Visiting a node"
ruby_tree.visit {|node| puts node.node_name}
puts

puts "visiting entire tree"
ruby_tree.visit_all {|node| puts node.node_name}
```

Этот класс реализует простое дерево. В нем имеются три метода – initialize, visit и visit_all – и две переменные экземпляров, children и node_name. Метод initialize имеет особое значение. Ruby будет вызывать его при создании нового экземпляра.

Здесь следует остановиться на некоторых правилах и соглашениях, принятых в языке Ruby. Имена классов начинаются с заглавной буквы и обычно записываются в «верблюжьей» (CamelCase) нотации. Имена переменных экземпляров (хранящие уникальные значения для каждого экземпляра) должны начинаться с одного символа @, а переменные класса (хранящие значения, используемые всеми объектами класса) должны начинаться с двух символов @@. Имена переменных экземпляров и методов обычно начинают с прописной буквы

и используют форму_записи_с_подчеркиванием. Имена констант записываются ТОЛЬКО_ЗАГЛАВНЫМИ_БУКВАМИ. В этом примере определяется класс дерева. Каждое дерево обладает двумя переменными экземпляра: @children и @node_name. Имена функций и методов, которые обычно используются в условных конструкциях, принято завершать знаком вопроса (if test?).

Ключевое слово attr определяет переменные экземпляров. Существует несколько версий этого ключевого слова. Наиболее часто используются attr (определяет переменную экземпляра и метод с тем же именем для доступа к ней) и attr_accessor (определяет переменную экземпляра, метод чтения этой переменной и метод записи в нее).

Наша компактная программа обладает немалыми возможностями. В ней используются блоки и рекурсия, позволяющие любому пользователю выполнить обход всех узлов в дереве. Каждый экземпляр класса Tree хранит один узел дерева. Метод initialize дает возможность определить начальные значения для атрибутов children и node_name. Метод visit вызывает переданный ему блок кода. Метод visit_all вызывает метод visit для своего узла, а затем рекурсивно вызывает visit_all для каждого дочернего узла.

Код, следующий за определением класса, демонстрирует его использование. Он создает дерево, посещает один узел и затем выполняет обход всех узлов. Если запустить этот сценарий, он выведет следующее:

```
Visiting a node
Ruby

visiting entire tree
Ruby
Reia
MacRuby
```

Классы – это только часть уравнения. Выше уже упоминались модули. Давайте вернемся назад и познакомимся с ними поближе.

Подмешивание

Механизм наследования в объектно-ориентированных языках используется для распространения общности поведения на похожие объекты. Чтобы обеспечить сходство в поведении разнородных объектов, либо используют наследование от нескольких классов (множественное наследование), либо ищут другие решения. Как показывает опыт, множественное наследование сопровождается массой

проблем. В Java для решения этой задачи используются интерфейсы. В Ruby – модули. Модуль – это коллекция функций и констант. Когда модуль включается в класс, его функции и константы становятся частью класса.

Рассмотрим пример модуля, добавляющего метод `to_f` в произвольный класс:

ruby/to_file.rb
http://media.pragprog.com/titles/btlang/code/ruby/to_file.rb

```
module ToFile
  def filename
    "object_#{self.object_id}.txt"
  end

  def to_f
    File.open(filename, 'w' ) {|f| f.write(to_s)}
  end
end

class Person
  include ToFile
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def to_s
    name
  end
end

Person.new('matz' ).to_f
```

Здесь определяется модуль с двумя методами. Метод `to_f` записывает результат вызова метода `to_s` в файл с именем, возвращаемым методом `filename`. Любопытно отметить, что метод `to_s` используется в модуле, но реализован в классе! При этом метод используется раньше его определения. Модуль очень тесно взаимодействует с классом, включающим его. Модули часто зависят от нескольких методов классов. В Java такая зависимость оформляется явно – посредством реализации интерфейсов в классе. В Ruby зависимости обрабатываются неявно, посредством механизма «утиной» типизации.

Особенности реализации класса `Person` не представляют никакого интереса, и в этом вся суть. Класс `Person` подключает модуль, и дело

в шляпе! Способность писать в файл фактически не имеет никакого отношения к классу `Person`. Мы добавляем возможность сохранять содержимое в файл, подмешивая ее. Точно так же можно подмешать в класс `Person` другие модули, создавать от него подклассы, и каждый подкласс будет обладать возможностями всех подмешиваемых компонентов, не заботясь о тонкостях их реализации. Теперь можно использовать упрощенный механизм единственного наследования, чтобы определить основной класс, и затем включать в него дополнительные возможности с помощью модулей. Такой стиль программирования, впервые появившийся в языке Flavors, а затем заимствованный языками от Smalltalk до Python, называется *подмешиванием* (mixin). Программные компоненты, несущие в себе подмешиваемые особенности, не всегда называют модулями, но суть их от этого не меняется. Единственное наследование в комбинации с механизмом подмешивания позволяет компоновать функциональные возможности по своему усмотрению без лишних хлопот.

Модули, перечисления и множества

Двумя важнейшими подмешиваемыми компонентами в Ruby являются модули `enumerable` и `comparable`. Класс, поддерживающий возможность итераций, должен реализовать метод `each`, а класс, поддерживающий возможность сравнения, должен реализовать оператор <=>. Оператор <=>, называемый «кораблико́м», реализует простое сравнение и должен возвращать –1, если правый операнд больше левого, 1 – если меньше, и 0 – если операнды равны. В обмен на реализацию этих методов модули `enumerable` и `comparable` предоставляют множество методов, удобных при работе с коллекциями. Выполните следующий код в консоли:

```
>> 'begin' <=> 'end'
=> -1
>> 'same' <=> 'same'
=> 0
>> a = [5, 3, 4, 1]
=> [5, 3, 4, 1]
>> a.sort
=> [1, 3, 4, 5]
>> a.any? {|i| i > 6}
=> false
>> a.any? {|i| i > 4}
=> true
>> a.all? {|i| i > 4}
=> false
```

```
>> a.all? {|i| i > 0}
=> true
>> a.collect {|i| i * 2}
=> [10, 6, 8, 2]
>> a.select {|i| i % 2 == 0 } # even
=> [4]
>> a.select {|i| i % 2 == 1 } # odd
=> [5, 3, 1]
>> a.max
=> 5
>> a.member?(2)
=> false
```

Метод any? возвращает true, если указанное условие выполняется хотя бы для одного (любого – any) элемента; метод all? возвращает true, если условие выполняется для всех (all) элементов. Так как оператор «кораблик» реализован для целых чисел в классе Fixnum, появляется возможность выполнять сортировку (метод sort) элементов массива, а также находить минимальное и максимальное значения (методы min и max соответственно).

Имеется также возможность использовать операции над множествами. Методы collect и map применяют указанную функцию к каждому элементу и возвращают массив результатов. Метод find выполняет поиск первого элемента, соответствующего условию, а методы select и find_all возвращают все элементы, соответствующие заданному условию. С помощью метода inject можно вычислить сумму или произведение элементов списка:

```
>> a
=> [5, 3, 4, 1]
>> a.inject(0) {|sum, i| sum + i}
=> 13
>> a.inject {|sum, i| sum + i}
=> 13
>> a.inject {|product, i| product * i}
=> 60
```

Метод inject кажется непонятным на первый взгляд, но в нем нет ничего необычного. Он принимает блок кода с двумя аргументами и выражение. Блок кода будет выполнен для каждого элемента списка, при этом каждый элемент будет передан блоку во втором аргументе. Первый аргумент – это результат предыдущего выполнения блока кода. Так как при первом обращении результат отсутствует, есть возможность передать начальное значение в виде аргумента метода inject. (Если начальное значение не указано, inject будет использо-

вать первый элемент коллекции.) Взгляните на аналогичный пример, куда был добавлен вывод дополнительной информации:

```
>> a.inject(0) do |sum, i|
?>   puts "sum: #{sum}  i: #{i}  sum + i: #{sum + i}"
?>   sum + i
?>end
sum: 0  i: 5  sum + i: 5
sum: 5  i: 3  sum + i: 8
sum: 8  i: 4  sum + i: 12
sum: 12  i: 1  sum + i: 13
```

Как и ожидалось, результат, выведенный в предыдущей строке, всегда оказывается первым значением в следующей. С помощью метода `inject` можно подсчитывать количество слов в предложениях, находить самые длинные слова в абзацах и многое другое.

Что мы узнали во второй день

Вы получили шанс увидеть, как выглядит синтаксический сахар Ruby, а также познакомились с некоторыми волшебными особенностями языка. Сначала мы посмотрели, насколько гибким может быть Ruby. Коллекции оказались чрезвычайно просты: всего два типа коллекций с многоуровневыми API, реализующими операции с ними. Производительность приложений вторична. Основная цель Ruby – высокая продуктивность программиста. Модуль `enumerable` наглядно демонстрирует, насколько хорошо спроектирован Ruby. Объектно-ориентированная модель на основе единственного наследования не нова, но ее реализация имеет интуитивно понятную архитектуру и снабжена массой полезных особенностей. Такой уровень абстракции несколько улучшает язык, но дальше мы познакомимся с еще более серьезными улучшениями.

День 2: задания для самостоятельного решения

Ниже предлагаются немного более сложные задания для самостоятельного решения. Вы уже знаете о языке Ruby чуть больше, поэтому приступим без лишних предисловий. Следующие задания заставят вас проявить свои аналитические возможности.

Узнайте:

- как обращаться к файлам без применения блоков кода. В чем заключаются преимущества блоков кода?
- как преобразовать хэш в массив? Можно ли преобразовать массив в хэш?

- можно ли выполнять итерации по элементам хэшей?
- массивы в Ruby можно использовать как стеки. Какие другие распространенные структуры данных поддерживаются массивами?

Практические задания:

- реализуйте вывод массива, содержащего 16 чисел, по четыре числа сразу, только с помощью метода `each`. А затем реализуйте то же самое с помощью метода `each_slice` из модуля `Enumerable`;
- класс `Tree` получился довольно интересным, но он не позволяет создавать деревья с более очевидной структурой. Добавьте в метод инициализации поддержку вложенных структур данных в виде хэшей и массивов. То есть должна поддерживаться возможность инициализировать дерево, как: `{'grandpa' => {'dad' => {'child 1' => {}, 'child 2' => {} }, 'uncle' => {'child 3' => {}, 'child 4' => {} } } };`
- напишите простой инструмент поиска, который отыскивал бы в файле строки, содержащие указанную фразу, и выводил бы их на экран. Для этого вам потребуется написать простое регулярное выражение, читающее целые строки из файла. (В Ruby это делается на удивление просто.) При желании добавьте вывод номеров строк.

2.4. День 3: Большие перемены

Главное достоинство Мэри Поппинс в том, что она занималась домашним хозяйством эффективнее, превратив его в забаву и изменив представление людей об этом занятии. Вы можете пойти проторенным путем, используя в сценариях на языке Ruby приемы, проверенные при программировании на других языках. Но, меняя свои взгляды на язык, вы начинаете замечать новые, необычные возможности, и программирование вновь превращается в забаву. В каждой главе в этой книге мы будем знакомиться с нетривиальной задачей, удивительно легко решаемой на описываемом языке. В главе о языке Ruby мы познакомимся с метапрограммированием.

Под метапрограммированием понимается создание программ, которые создают другие программы. Фреймворк ActiveRecord, образующий ядро Rails, использует возможности метапрограммирования для реализации дружественного языка определения классов, связанных с таблицами баз данных. Например, класс `Department` на языке фреймворка ActiveRecord мог бы выглядеть так:

```
class Department < ActiveRecord::Base
  has_many :employees
  has_one :manager
end
```

has_many и has_one – это методы на языке Ruby, добавляющие переменные экземпляров и методы, необходимые для установления отношения has_many. Это определение класса читается как обычный текст на английском языке – в нем отсутствуют лишние синтаксические элементы, которые обычно используются в других фреймворках, реализующих взаимодействия с базами данных. Рассмотрим еще некоторые инструменты, которые можно использовать для метапрограммирования.

Открытые классы

Вы уже познакомились с открытыми классами. В Ruby поддерживается возможность в любой момент изменить определение любого класса, обычно с целью добавить в него новые возможности. Ниже приводится отличный пример из фреймворка Rails, демонстрирующий добавление метода NilClass:

ruby/blank.rb
http://media.pragprog.com/titles/btlang/code/ruby/blank.rb

```
class NilClass
  def blank?
    true
  end
end

class String
  def blank?
    self.size == 0
  end
end

["" , "person" , nil].each do |element|
  puts element unless element.blank?
end
```

При первом использовании ключевое слово class определяет новый класс; если класс уже определен, при последующих использованиях это ключевое слово производит изменение указанного класса. В данном примере мы добавляем метод с именем blank? в два существующих класса: NilClass и String. Получая в программе некоторую строку, часто бывает желательно проверить, не является ли она пус-

той строкой. Большинство строк имеет некоторое значение, но они могут быть также пустыми и даже иметь значение nil. Это небольшое улучшение позволяет мне быстро проверить соответствие двум случаям, когда строка может считаться пустой, потому что в любом из этих случаев метод blank? вернет true. И не важно, ссылается ли переменная на экземпляр класса String или какой-то другой. Если он поддерживает метод blank?, этот код будет работать. Вспомните: «если он ходит как утка и крякает как утка, значит, это утка». Мне не нужно тратить лишнее время на проверку типа.

Смотрите, что тут происходит. Мы просим самый острый скальпель, и Ruby с готовностью дает его нам, а затем вскрываем классы String и NilClass. Эту возможность можно полностью запретить, например переопределив метод Class.new, но при этом будет утрачена свобода действий. Обладая такой свободой, которая позволяет в любой момент переопределять классы или объекты, можно создавать удивительно легко читаемый код. Однако, получая свободу, мы принимаем на себя и груз ответственности.

Открытые классы очень удобны, особенно для тех, кто занимается разработкой собственного, предметно-ориентированного языка. Часто бывает полезно предоставить в распоряжение пользователя специализированные синтаксические элементы. Например, взгляните на API представления расстояния в разных единицах измерения, первоначально выраженного в дюймах:

ruby/units.rb
http://media.pragprog.com/titles/btlang/code/ruby/units.rb

```
class Numeric
  def inches
    self
  end

  def feet
    self * 12.inches
  end

def yards
  self * 3.feet
end

def miles
  self * 5280.feet
end

def back
```

```
    self * -1
  end

  def forward
    self
  end
end

puts 10.miles.back
puts 2.feet.forward
```

Открытые классы дают возможность реализовать подобную поддержку минимальным объемом кода. Однако в Ruby существуют еще более мощные методики.

Применение метода method_missing

Столкнувшись с вызовом несуществующего метода, интерпретатор Ruby вызывает специальный отладочный метод, чтобы вывести сообщение об ошибке. Это здорово упрощает отладку программ. Но иногда эту особенность используют совсем по другому назначению и с ее помощью расширяют возможности объектов. Для этого необходимо всего лишь переопределить метод `method_missing`. Взгляните на API представления римских чисел. То же самое легко можно было бы реализовать с помощью обычного метода, который вызывается, например, так: `Roman.number_for "ii"`. И в этом нет ничего плохого, потому что от нас не требуется в обязательном порядке использовать скобки или точки с запятой. Но в Ruby возможно более удачное решение:

ruby/roman.rb
http://media.pragprog.com/titles/btlang/code/ruby/roman.rb

```
class Roman
  def self.method_missing name, *args
    roman = name.to_s
    roman.gsub!("IV" , "IIII" )
    roman.gsub!("IX" , "VIIII" )
    roman.gsub!("XL" , "XXXX" )
    roman.gsub!("XC" , "LXXXX" )

    (roman.count("I" ) +
     roman.count("V" ) * 5 +
     roman.count("X" ) * 10 +
     roman.count("L" ) * 50 +
     roman.count("C" ) * 100)
  end
end
```

```
puts Roman.X
puts Roman.XC
puts Roman.XII
puts Roman.X
```

Этот код наглядно демонстрирует применение метода `method_missing`. Он прост и понятен. Сначала мы переопределяем метод `method_missing`. Он получает имя вызванного метода и его параметры. Нас интересует только имя. Сначала мы преобразуем его в объект типа `String`. Затем замещаем специальные комбинации, такие как `iv` и `ix`, более простыми для подсчета эквивалентами. Потом мы просто подсчитываем вхождение каждой отдельной римской цифры и умножаем полученное количество на ее значение. Интерфейс получился удивительно простым, сравните: `Roman.i` и `Roman.number_for "i"`.

И все же давайте посмотрим, какой ценой это достигается. В результате мы получили класс, более сложный в отладке, потому что теперь Ruby больше не будет предупреждать нас о вызове отсутствующего метода! Определенно для нас было бы желательно обеспечить проверку ошибок на случай передачи недопустимых римских чисел. Если не знать, что искать, вам нелегко будет отыскать, например, реализацию метода `ii` в классе `Roman`. И все же это весьма действенный инструмент в нашем арсенале. Используйте его с большой осторожностью.

Модули

Самым популярным инструментом метапрограммирования в языке Ruby являются модули. Вы буквально сможете реализовать `def` или `attr_accessor` всего несколькими строками кода в модуле. Вы можете также расширить определение любого класса. Часто этот прием используется с целью создания собственных предметно-ориентированных языков (Domain-Specific Language, DSL) для определения своих классов[1]. В этом случае методы определяются в модуле, который затем добавляет (или подмешивает) все свои методы и константы в управляемый класс.

Давайте подробно разберем пример реализации подобного суперкласса. Ниже приводится класс, который требуется реализовать

---

[1] Предметно-ориентированный язык позволяет адаптировать синтаксис основного языка для решения специализированных задач. Самым известным, пожалуй, примером реализации такого языка в мире Ruby является фреймворк ActiveRecord, определяющий собственный предметно-ориентированный язык для отображения классов в таблицы баз данных.

с применением приемов метапрограммирования. Это – простая программа, открывающая файл CSV при помощи имени класса.

ruby/acts_as_csv_class.rb
http://media.pragprog.com/titles/btlang/code/ruby/acts_as_csv_class.rb

```
class ActsAsCsv
  def read
    file = File.new(self.class.to_s.downcase + '.txt' )
    @headers = file.gets.chomp.split(', ' )

    file.each do |row|
      @result << row.chomp.split(', ' )
    end
end

def headers
  @headers
end

def csv_contents
  @result
end

  def initialize
    @result = []
    read
  end
end

class RubyCsv < ActsAsCsv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

Базовый класс определяет четыре метода. `headers` и `csv_contents` – это обычные методы доступа, возвращающие значения переменных экземпляра. Метод `initialize` инициализирует массив с результатами и читает содержимое файла. Основная работа выполняется в методе `read`. Метод `read` открывает файл, читает строку с заголовками полей и разбивает ее на отдельные поля. Затем он последовательно читает строки и помещает содержимое каждой строки в массив. Данная реализация парсинга не отличается полнотой, потому что в ней не предусмотрена обработка особых случаев, таких как наличие кавычек, но основная идея должна быть вам понятна.

Следующий шаг – взять файл и подключить это поведение к классу с методом модуля, который часто называют *макросом*. Макросы изменяют поведение классов, часто опираясь на изменения в окружении. В данном случае наш макрос открывает класс и добавляет в него поддержку файлов CSV:

ruby/acts_as_csv.rb
http://media.pragprog.com/titles/btlang/code/ruby/acts_as_csv.rb

```
class ActsAsCsv
  def self.acts_as_csv

    define_method 'read' do
      file = File.new(self.class.to_s.downcase + '.txt' )
      @headers = file.gets.chomp.split(', ' )

      file.each do |row|
        @result << row.chomp.split(', ' )
      end
    end

    define_method "headers" do
      @headers
    end

    define_method "csv_contents" do
      @result
    end

    define_method 'initialize' do
      @result = []
      read
    end
  end
end

class RubyCsv < ActsAsCsv
  acts_as_csv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

Собственно метапрограммирование имеет место в макросе `acts_as_csv`. Этот код вызовет `define_method` для всех методов, которые требуется добавить в целевой класс. Теперь, при вызове макроса `acts_as_csv`, он определит все четыре метода в целевом классе.

Итак, макрос `acts_as_csv` не делает ничего особенного, только добавляет несколько методов, которые с тем же успехом можно добавить через наследование. Данный прием не выглядит большим усовершенствованием, однако на его основе можно получить более интересные результаты. Давайте посмотрим, как то же самое можно реализовать в модуле:

ruby/acts_as_csv_module.rb
http://media.pragprog.com/titles/btlang/code/ruby/acts_as_csv_module.rb

```
module ActsAsCsv

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods
    def acts_as_csv
      include InstanceMethods
    end
  end

  module InstanceMethods

def read
  @csv_contents = []
  filename = self.class.to_s.downcase + '.txt'
  file = File.new(filename)
  @headers = file.gets.chomp.split(', ' )

  file.each do |row|
    @csv_contents << row.chomp.split(', ' )
  end
end

attr_accessor :headers, :csv_contents

def initialize
  read
end
  end
end

class RubyCsv # Никакого наследования! Простое подмешивание
  include ActsAsCsv
  acts_as_csv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

Ruby будет вызывать метод `included` при каждом подключении этого модуля к другому. Не забывайте, что класс – это модуль. В методе `included` мы расширяем целевой класс (каковым в данном случае является класс `RubyCsv`) обращением к `base`, и этот модуль добавляет методы класса в `RubyCsv`. Единственный метод класса – `acts_as_csv`. Этот метод, в свою очередь, открывает класс и включает в него все методы экземпляра. В результате мы получили программу, которая создает другие программы.

Интересно отметить, что применение всех этих приемов метапрограммирования позволяет изменять программы, исходя из окружения приложения. Фреймворк ActiveRecord использует метапрограммирование для динамического добавления методов доступа, имена которых совпадают с именами полей в базе данных. Некоторые фреймворки поддержки XML, такие как `builder`, позволяют пользователям определять собственные теги с помощью `method_missing`, чтобы обеспечить удобный синтаксис для работы с ними. Чем удобнее синтаксис, тем понятнее он тем, кто будет читать ваш код. В этом и заключается мощь языка Ruby.

Что мы узнали в третий день

В этом разделе мы узнали, как на языке Ruby определять собственные синтаксические конструкции и изменять классы «на лету». Описанные приемы относятся к категории метапрограммирования. Каждая строка кода, написанная вами, адресована не только компьютерам, но и людям. Иногда бывает сложно соблюсти баланс между функциональностью кода, который передается интерпретатору или компилятору, и простотой для понимания человеком. С помощью приемов метапрограммирования можно сократить разрыв между допустимостью синтаксиса Ruby и выражением своих намерений простым и понятным образом.

Некоторые из наиболее удачных фреймворков на языке Ruby, такие как Builder и ActiveRecord, широко используют приемы метапрограммирования для повышения удобочитаемости кода. Здесь мы использовали преимущества открытых классов для создания интерфейса «утиной» типизации в виде метода `blank?` для объектов `String` и `nil`, значительно уменьшив тем самым количество кода, загромождающего наши намерения. Мы реализовали поддержку римских чисел с помощью метода `method_missing`. И наконец, задействовали модули для определения элементов предметно-ориентированного языка, обеспечивающего возможность обработки файлов CSV.

День 3: задания для самостоятельного решения

Практические задания: добавьте в реализацию приложения обработки файлов CSV поддержку метода `each`, возвращающего объект `CsvRow`. Используйте в классе `CsvRow` метод `method_missing`, чтобы получить значение столбца по его заголовку.

Например, для файла

```
one, two
lions, tigers
```

должна иметься возможность обрабатывать его, как показано ниже:

```
csv = RubyCsv.new
csv.each {|row| puts row.one}
```

Этот код должен вывести `"lions"`.

2.5. В заключение о Ruby

Мы охватили множество базовых сведений в этой главе. Я надеюсь, вы смогли уловить сравнение с Мэри Поппинс. Много раз на конференциях, посвященных Ruby, я слышал от людей, что они любят Ruby за то, что он превращает программирование в развлечение. В индустрии программирования, заключенной в объятия C-подобных языков, таких как C++, C#, Java и другие, язык Ruby воспринимается как глоток свежего воздуха.

Сильные стороны

Ruby как исключительно объектно-ориентированный язык дает возможность интерпретировать объекты единообразным и непротиворечивым способом. «Утиная» типизация обеспечивает поддержку истинного полиморфизма, опирающегося на фактические возможности объектов, а не на иерархию наследования. А модули и открытые классы в Ruby позволяют программисту модифицировать их поведение с применением приемов, не связанных с обычным определением методов или переменных экземпляров в классах.

Ruby идеально подходит на роль языка сценариев или языка для веб-разработки, если требования к масштабированию не очень высоки. Это весьма продуктивный язык. Некоторые особенности языка, обеспечивающие высокую продуктивность, делают практически невозможным создание компилятора Ruby и повышение его производительности.

Сценарии

Ruby – фантастический язык сценариев. Он отлично подходит для создания программного кода, связывающего два приложения; веб-роботов, извлекающих из веб-страниц информацию о котировках акций или о ценах на книги; выполнения автоматизированной сборки приложений или их тестирования.

Как язык, поддерживающий большинство основных операционных систем, Ruby является отличным выбором для разработки системных сценариев. Для языка имеется множество разнообразных библиотек и расширений, которые можно использовать для реализации поддержки файлов CSV, обработки XML или взаимодействия с низкоуровневыми сетевыми API.

Веб-разработка

Rails является одним из наиболее успешных фреймворков для разработки веб-приложений. Его дизайн основан на хорошо зарекомендовавшей себя архитектуре модель–представление–контроллер (model–view–controller). Множество поддерживаемых соглашений по именованию для работы с базами данных и прикладными элементами позволяют типичному приложению обходиться лишь несколькими строками настройки. Кроме того, для фреймворка имеется множество расширений, решающих довольно сложные проблемы, связанные с промышленной эксплуатацией:

- приложения на основе Rails имеют непротиворечивую и понятную организацию;
- миграция обеспечивает возможность обработки изменений в схеме базы данных;
- несколько хорошо документированных соглашений позволяют уменьшить общий объем конфигурационного кода;
- доступно множество разнообразных расширений.

Готовность к использованию в коммерческих проектах

Я считаю высокую продуктивность Ruby and Rails одной из важнейших составляющих успеха. В середине 2000-х в Сан-Франциско нельзя было камень бросить, чтобы не попасть в кого-нибудь, занимающегося запуском нового проекта на основе Rails. Даже в настоящее время продуктивность Ruby часто является определяющим фактором для различных компаний, в том числе и для меня. Сочетание

удобного синтаксиса, обширного сообщества программистов, богатого выбора инструментов и расширений играет огромную роль. Вы без труда сможете найти пакеты на языке Ruby для поиска адресов по почтовым индексам или определения почтовых индексов в радиусе 50 миль. Вы можете обрабатывать изображения и обслуживать кредитные карты, обмениваться информацией с веб-службами и взаимодействовать с программами на других языках.

Язык Ruby и фреймворк and Ruby on Rails используется многими крупными коммерческими веб-сайтами. Первоначально Twitter был реализован на языке Ruby, высочайшая продуктивность которого обеспечила возможность роста веб-сайта до гигантских размеров. В конечном итоге ядро Twitter было переписано на языке Scala. Из этого следуют два вывода. Во-первых, Ruby – великолепный язык для создания быстро прогрессирующих программных продуктов. Во-вторых, приходится признать, что масштабируемость Ruby пока имеет определенные ограничения.

С позиции индустрии разработки программных продуктов уровня предприятия, широко использующих механизмы распределенных взаимодействий, обмена сообщениями и интернационализации, роль Ruby часто расценивается как второстепенная, но на языке Ruby вполне возможно реализовать все только что перечисленное. Иногда опасения по поводу выбора программной платформы и требования к высокой масштабируемости вполне оправданны, но слишком многие пытаются получить масштабируемость, которой с лихвой хватило бы для создания очередного интернет-аукциона eBay, хотя на самом деле она им не нужна. Часто языка Ruby более чем достаточно, учитывая ограниченное время, отпускаемое до вывода продукта на рынок.

Недостатки

В мире не существует совершенных языков. Язык Ruby тоже имеет свои ограничения. Перечислим наиболее основные из них.

Производительность

Основным недостатком Ruby является невысокая производительность. Конечно, разработчики постоянно повышают скорость работы Ruby. Версия 1.9 показывает десятикратное увеличение производительности в некоторых тестах. Новая виртуальная машина Ruby под названием Rubinius, написанная Эваном Фениксом (Evan Phoenix), включает механизм динамической компиляции. Он исследует шаблоны, используемые интерпретатором для выполнения некоторого

блока кода, и производит машинный код, который с высокой вероятностью потребуется выполнить еще раз. Данный подход отлично зарекомендовал себя в языке Ruby, знания синтаксиса которого обычно недостаточно, чтобы обеспечить компиляцию. Не забывайте, что определение любого класса может измениться в любой момент.

И все-таки Мацумото поступил достаточно мудро. Он сосредоточился на повышении продуктивности программиста, а не на производительности языка. Многие особенности, такие как открытые классы, «утиная» типизация и метод `method_missing`, оказываются более предпочтительными, чем поддержка компиляции и связанная с этим производительность программ.

Параллельные вычисления и ООП

Объектно-ориентированное программирование имеет одно важное ограничение. Сама идея основана на обертывании данных дополнительными функциональными возможностями, а данные обычно могут изменяться. Такая стратегия страдает серьезными проблемами, связанными с параллельными вычислениями. В лучшем случае поддержка параллельных вычислений встраивается непосредственно в язык. В худшем – объектно-ориентированные системы оказывается невозможно отладить и надежно протестировать в многозадачном окружении. На момент написания этих строк разработчики Rails только приступили к решению проблем, связанных с эффективным управлением параллельными вычислениями.

Безопасность типов

Я убежденный сторонник «утиной» типизации. Применение этой стратегии типизации позволяет писать краткий и выразительный код. Но «утиная» типизация имеет свою цену. Статическая система типов упрощает синтаксический анализ исходного кода и тем самым облегчает создание интегрированных сред разработки. Создать подобную среду разработки для Ruby намного сложнее, и потому большинство программистов на Ruby не используют их. Много раз я сокрушался по поводу отсутствия хорошего отладчика. И я знаю, что в этом я не одинок.

Заключительные замечания

Итак, основные достоинства Ruby – удобный синтаксис и гибкость языка. А основным недостатком является невысокая производитель-

ность, хотя для большинства приложений его производительности вполне достаточно. В общем и целом Ruby – великолепный объектно-ориентированный язык. При правильном подходе Ruby может оказаться отличным выбором. Как и любой инструмент, он предназначен для решения определенного круга задач, и если его использовать по назначению, он не разочарует вас. Кроме того, не забывайте про некоторые его волшебные возможности.

Глава 3

Io

«Вопрос не в том, что мы будем делать.
Вопрос в том, чего мы делать не собираемся».

Феррис Бьюллер (Ferris Bueller)

Встречайте: язык программирования Io. Io, как и Ruby, отличается большой гибкостью. Это еще юный, очень интересный, простой, понятный и непредсказуемый язык. Представьте себе Ферриса Бьюллера[1]. Если вы любитель веселых вечеринок, позвольте Io показать вам окрестности. Он будет пытаться сделать что-нибудь хотя бы один раз. Он может подарить вам самые забавные минуты в вашей жизни или доставить массу неприятностей, или и то и другое. В любом случае вам не будет скучно. Как следует из цитаты выше, в нем не так много сдерживающих правил.

3.1. Введение в Io

Язык Io был создан Стивом Декортом (Steve Dekorte) в 2002 году. Название языка начинается с заглавной буквы *I*, за которой следует прописная буква *o*. Io – это язык, основанный на прототипах, так же как Lua или JavaScript, в том смысле, что каждый объект является копией другого объекта.

Язык Io создавался Стивом, чтобы на его примере разобраться в механике работы интерпретаторов, и по сей день он остается очень маленьким языком. Его синтаксис можно изучить буквально за пятнадцать минут, а основные механизмы языка – за тридцать. В нем нет никаких неожиданностей. Но для изучения его библиотек потребуется больше времени. Библиотеки являются источником сложности и богатства языка.

---

[1] Персонаж из кинофильма «Ferris Bueller's Day Off». Режиссер Джон Хьюз (John Hughes). 1986; Hollywood, CA: Paramount, 1999 («Феррис Бьюллер берет выходной» – *Прим. перев.*).

В настоящее время значительная часть сообщества Io расценивает его как встраиваемый язык с небольшой виртуальной машиной и богатыми возможностями параллельного выполнения. Основными преимуществами языка являются возможность настройки синтаксиса в широких пределах, а также мощная модель параллельных вычислений. Попытайтесь сконцентрировать свое внимание на простоте синтаксиса и модели программирования, основанной на прототипах. После изучения языка Io я стал гораздо глубже понимать, как действует JavaScript.

3.2. День 1: Пропустим школу и повеселимся

Знакомство с языком Io происходит так же, как знакомство с любым другим языком. Для этого вам потребуется провести некоторое время за клавиатурой. Будет намного проще, если мы опустим нудный диалог на полпути в кабинет истории. Давайте сбежим с занятий и займемся более интересными делами.

Иногда названия сбивают с толку, но имя Io может рассказать о многом. Оно одновременно выглядит и безрассудным (пробовали ли вы «погуглить» по двум буквам «Io»?)[1], и гениальным. Название состоит всего из двух букв, причем обе они – гласные. Синтаксис языка чрезвычайно прост, как и его название. Суть синтаксиса Io состоит в объединении сообщений в цепочки, где каждое сообщение может принимать необязательные параметры в круглых скобках и возвращает некоторый объект. Все сущее в языке Io есть сообщение, возвращающее объект-приемник. В нем отсутствуют ключевые слова – только горстка специальных имен, действующих подобно ключевым словам.

В Io нет классов – только объекты. Вы будете иметь дело лишь с объектами, создавая их копии по мере необходимости. Копируемые объекты называются *прототипами*, а Io – первый и единственный язык в этой книге, основанный на прототипах. В таких языках каждый объект является копией некоторого существующего объекта, а не класса. Io достаточно близко напоминает объектно-ориентированный Lisp. Пока трудно сказать, получит ли Io широкое распростра-

---

[1] Попробуйте вместо поиска по двум буквам выполнить поиск по фразе «Io language». (А еще лучше по фразе: «Io язык программирования». – *Прим. перев.*)

нение, но, учитывая простоту его синтаксиса, можно предположить, что у него есть все шансы для этого. Библиотеки поддержки параллельных вычислений, с которыми вы познакомитесь на третий день, продуманы очень тщательно, а семантика сообщений обеспечивает особое изящество в реализации параллельных операций.

Ломаем лед

Итак, давайте установим интерпретатор и начнем вечеринку. Дистрибутив можно найти на сайте http://iolanguage.com. Загрузите и установите его. Запустите интерпретатор командой `io` и введите традиционную программу «Hello, World»:

```
Io> "Hi ho, Io" print
Hi ho, Io==> Hi ho, Io
```

Взглянув на этот код, можно точно сказать, что здесь происходит. Он отправляет сообщение `print` строке `"Hi ho, Io"`. Приемники сообщений всегда располагаются слева, а сами сообщения – справа. В этом языке вы не найдете синтаксического сахара. С его помощью вы просто будете отправлять сообщения объектам.

В языке Ruby новые объекты создаются на основе классов с помощью ключевого слова `new`. А новые типы объектов создаются за счет определения классов. В Io отсутствует грань между классами и объектами. Новые объекты создаются путем копирования существующих. Существующие объекты играют роль прототипов:

```
batate$ io
Io 20110905
Io> Vehicle := Object clone
==>  Vehicle_0x1003b61f8:
  type              = "Vehicle"
```

Здесь `Object` – это корневой объект. Мы посылаем ему сообщение `clone` и получаем новый объект. Далее этот объект присваивается переменной `Vehicle`. В данном случае `Vehicle` не является классом. Это – не шаблон, на основе которого создаются новые объекты. Это – самый обычный объект, основанный на прототипе `Object`. Попробуем выполнить с ним некоторые операции:

```
Io> Vehicle description := "Something to take you places"
==> Something to take you places
```

Объекты имеют слоты. Коллекцию слотов можно рассматривать как аналог хэша. Обращение к каждому слоту выполняется посредством ключа. Присвоить значение слоту можно с помощью операто-

ра :=. Если слот отсутствует в объекте, интерпретатор Io создаст его. Для присваивания можно также использовать оператор =. Но в этом случае, если указанный слот отсутствует, Io возбудит исключение. В примере выше мы создали слот с именем description.

```
Io> Vehicle description = "Something to take you far away"
==> Something to take you far away
Io> Vehicle nonexistingSlot = "This won't work."
  Exception: Slot nonexistingSlot not found.
    Must define slot using := operator before updating.
  ---------
  message 'updateSlot' in 'Command Line' on line1
```

Получить значение слота можно, отправив имя требуемого слота объекту:

```
Io> Vehicle description
==> Something to take you far away
```

В действительности объект – это нечто большее, чем простая коллекция слотов. Мы можем получить список всех существующих слотов в объекте Vehicle, как показано ниже:

```
Io> Vehicle slotNames
==> list("type", "description")
```

Здесь мы отправили сообщение slotNames объекту Vehicle и получили список имен слотов. В данный момент в объекте имеются два слота. Слот description мы уже видели, но в объекте имеется также слот type. Этот слот поддерживается всеми объектами:

```
Io> Vehicle type
==> Vehicle
Io> Object type
==> Object
```

Мы вернемся к типам несколькими абзацами ниже. А пока просто помните, что слот type представляет разновидность данного объекта. Имейте в виду, что тип объекта – это не его класс. Итак, к настоящему моменту мы узнали, что:

- объекты создаются копированием других объектов;
- объекты – это коллекции слотов;
- получить значение слота можно, отправив сообщение.

Вы уже наверняка поняли, насколько прост и приятен язык Io. Но усядьтесь поудобнее. Мы только начали знакомство с ним. Давайте перейдем к знакомству с моделью наследования.

Объекты, прототипы и наследование

В этом разделе мы познакомимся с моделью наследования. Учитывая, что автомобиль является транспортным средством, попробуем смоделировать объект `ferrari`, являющийся экземпляром автомобиля. В объектно-ориентированном языке для этого пришлось бы создать иерархию, изображенную на рис. 3.1.

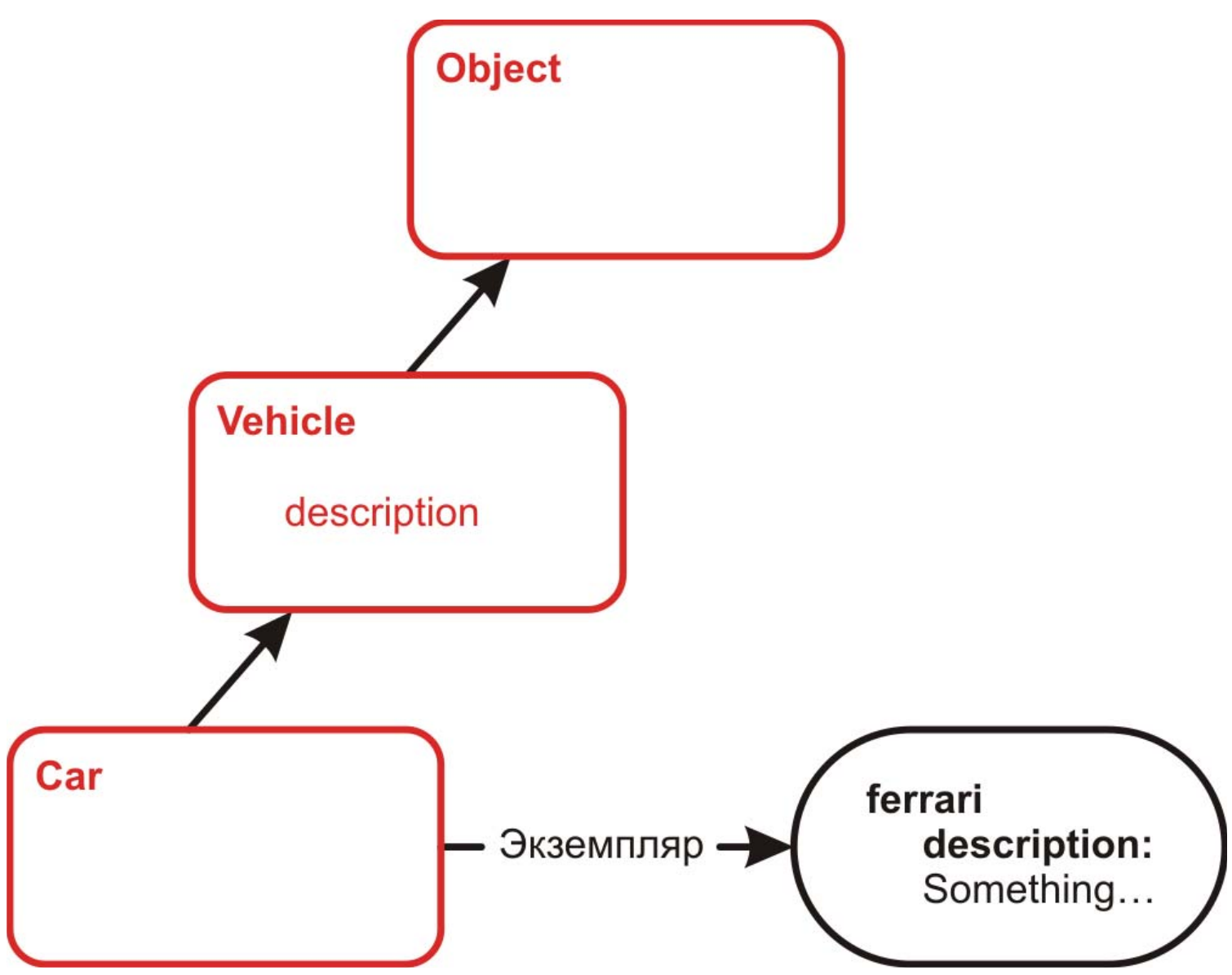

Рис. 3.1 ❖ Иерархия наследования объектов

Посмотрим, как та же самая задача решается на языке, основанном на прототипах. Для этого потребуется создать несколько дополнительных объектов:

```
Io> Car := Vehicle clone
==> Car_0x100473938:
  type              = "Car"

Io> Car slotNames
==> list("type")

Io> Car type
==> Car
```

Говоря языком Io, мы создали новый объект с именем `Car`, послав сообщение `clone` прототипу `Vehicle`. Давайте отправим сообщение `description` объекту `Car`:

```
Io> Car description
==> Something to take you far away
```

В объекте Car отсутствует слот description. Поэтому Io просто передал сообщение description прототипу Vehicle, где и обнаружился данный слот. Это – поразительно простая и мощная особенность. Создадим теперь еще один автомобиль, но на этот раз присвоим его переменной ferrari:

```
Io> ferrari := Car clone
==> Car_0x1004f43d0:

Io> ferrari slotNames
==> list()
```

Ага! В этом объекте отсутствует слот type. В соответствии с соглашениями имена типов в языке Io начинаются с буквы в верхнем регистре. Если теперь попытаться обратиться к слоту type, интерпретатор вернет значение слота type из прототипа:

```
Io> ferrari type
==> Car
```

Именно так работает модель представления объектов в языке Io. Объекты – это всего лишь контейнеры слотов. Чтобы получить значение слота, нужно отправить его имя объекту. Если слот отсутствует в данном объекте, Io попытается обратиться к родителю. Вот, собственно, и все, что вам нужно знать. В Io нет ни классов, ни метаклассов. Вам не придется описывать модули или интерфейсы. Вам достаточно будет одних только объектов, как показано на рис. 3.2.

Роль типов в Io играют соглашения по именованию. Идиоматически, если имя объекта начинается с буквы в верхнем регистре, он будет интерпретироваться как тип, то есть Io добавит в него слот type. Любые копии этого типа, имена которых начинаются с буквы в нижнем регистре, просто будут использовать слот type своего родителя. Типы – это лишь инструменты, помогающие программисту на Io организовать свой код.

Если необходимо, чтобы объект ferrari был типом, его имя должно начинаться с заглавной буквы, как показано ниже:

```
Io> Ferrari := Car clone
==> Ferrari_0x9d085c8:
type = "Ferrari"

Io> Ferrari type
==> Ferrari
```

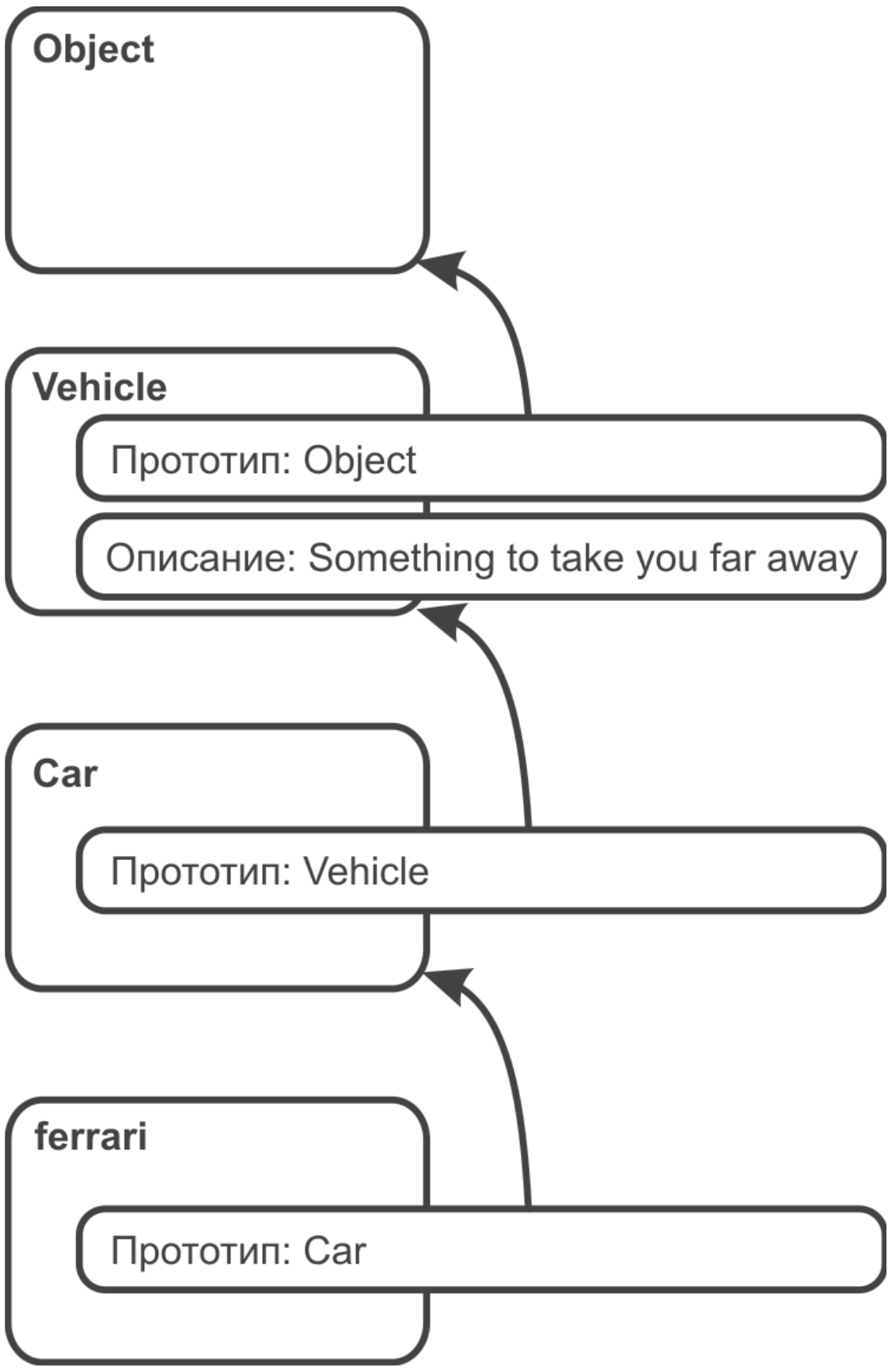

Рис. 3.2 ❖ Модель наследования в Io

```
Io> Ferrari slotNames
==> list("type")
Io> ferrari slotNames
==> list()
Io>
```

Обратите внимание, что в объекте `ferrari` отсутствует слот `type`, а в объекте `Ferrari` он имеется. Различия между типами и экземплярами: в Io используется простое соглашение по именованию, а не какая-то синтаксическая конструкция. Кроме наличия слота `type`, во всех остальных отношениях объекты совершенно неотличимы друг от друга.

В Ruby и Java классы являются шаблонами, используемыми для создания объектов. Инструкция `bruce = Person.new` создаст новый объект `bruce` типа `Person`. Классы и объекты в этих языках – совершенно

разные сущности. Но в языке Io все совсем иначе. Инструкция `bruce := Person clone` создаст на основе прототипа `Person` копию с именем `bruce`. Обе сущности, `bruce` и `Person`, являются объектами. Объект `Person` в данном случае играет роль типа, потому что имеет слот `type`. Во всех остальных отношениях объекты `Person` и `bruce` совершенно идентичны. А теперь перейдем к знакомству с поведением объектов.

Методы

Методы в языке Io определяются очень просто:

```
Io> method("So, you've come for an argument." println)
==> method(
    "So, you've come for an argument." println
)
```

Метод – это объект типа, как любой другой аналогичный объект. Вы можете получить его тип:

```
Io> method() type
==> Block
```

Так как метод является объектом, его можно присвоить слоту:

```
Io> Car drive := method("Vroom" println)
==> method(
    "Vroom" println
)
```

Если слоту присвоен метод, обращение к этому слоту приводит к вызову метода:

```
Io> ferrari drive
Vroom
==> Vroom
```

Хотите верьте, хотите нет, но теперь вы обладаете полной информацией об основных принципах организации программного кода на языке Io. Вспоминайте чаще об этом. Вы знаете основы синтаксиса языка. Вы можете определять типы и создавать объекты. Вы можете добавлять в объекты данные и методы, присваивая соответствующие компоненты их слотам. Все остальное связано с изучением библиотек.

Давайте копнем чуть глубже. Итак, вы можете получить содержимое слота независимо от того, является ли он переменной или методом:

```
Io> ferrari getSlot("drive")
==> method(
    "Vroom" println
)
```

Метод `getSlot` вернет содержимое слота родителя, если такой слот отсутствует в данном объекте:

```
Io> ferrari getSlot("type")
==> Car
```

Вы можете получить прототип данного объекта:

```
Io> ferrari proto
==> Car_0x100473938:
  drive           = method(...)
  type            = "Car"

Io> Car proto
==> Vehicle_0x1003b61f8:
  description         = "Something to take you far away"
  type                = "Vehicle"
```

В данном случае возвращаются прототипы, использовавшиеся для создания объектов `ferrari` и `Car`. Дополнительно выводятся слоты, созданные вашим кодом.

В языке Io существует основное пространство имен `Lobby`, содержащее все именованные объекты. Все операции присваивания, выполняемые в консоли, а также некоторые другие сохраняют объекты в пространстве имен `Lobby`. Получить его содержимое можно, как показано ниже:

```
Io> Lobby
==> Object_0x1002184e0:
  Car             = Car_0x100473938
  Lobby           = Object_0x1002184e0
  Protos          = Object_0x1002184e0
  Vehicle         = Vehicle_0x1003b61f8
  exit            = method(...)
  ferrari         = Car_0x1004f43d0
  forward         = method(...)
```

В этом списке можно видеть методы `exit` и `forward`, объект `Protos`, а также объекты, созданные нами.

Парадигма программирования на основе прототипов выглядит достаточно простой и понятной. Тем не менее приведу некоторые основные правила:

- все *сущности* в программе являются объектами;
- все *взаимодействия* с объектами являются сообщениями;
- в Io отсутствуют классы – все объекты создаются копированием других объектов, называемых *прототипами*;
- объекты помнят свои прототипы;

- объекты имеют слоты;
- слоты содержат объекты, включая объекты-методы;
- сообщение возвращает содержимое слота или вызывает метод, хранящийся в слоте;
- если объект не в состоянии ответить на сообщение, он переадресует его своему прототипу.

Это – подавляющее большинство правил. Благодаря возможности получить или изменить любой слот или любой объект появляется возможность использовать приемы метапрограммирования. Но, прежде чем приступить к этой теме, необходимо познакомиться с еще одним понятием: коллекциями.

Списки и отображения

В языке Io имеется несколько типов коллекций. Список – это упорядоченная коллекция объектов любых типов. Прототипом всех списков является объект `List`, а прототипом всех коллекций пар ключ/значение является объект `Map`. Создать список можно, как показано ниже:

```
Io> toDos := list("find my car", "find Continuum Transfunctioner")
==> list("find my car", "find Continuum Transfunctioner")

Io> toDos size
==>2

Io> toDos append("Find a present")
==> list("find my car", "find Continuum Transfunctioner", "Find a present")
```

Поддерживается также более краткий синтаксис представления списков. Объект `Object` имеет метод `list`, который преобразует свои аргументы в список. Метод `list` является удобным инструментом создания списков:

```
Io> list(1, 2, 3, 4)
==> list(1, 2, 3, 4)
```

Объект `List` имеет несколько удобных вспомогательных методов для выполнения математических операций, а также для представления списков в виде структур других типов, например в виде стека:

```
Io> list(1, 2, 3, 4) average
==> 2.5

Io> list(1, 2, 3, 4) sum
==> 10

Io> list(1, 2, 3) at(1)
```

```
==>2

Io> list(1, 2, 3) append(4)
==> list(1, 2, 3, 4)

Io> list(1, 2, 3) pop
==>3

Io> list(1, 2, 3) prepend(0)
==> list(0, 1, 2, 3)

Io> list() isEmpty
==> true
```

Еще одним основным классом коллекций в Io является объект `Map`. Отображения в языке Io действуют подобно хэшам в языке Ruby. Так как в Io отсутствует синтаксический сахар, для работы с отображениями придется использовать прикладной интерфейс, который выглядит, как показано ниже:

```
Io> elvis := Map clone
==> Map_0x115f580:

Io> elvis atPut("home", "Graceland")
==> Map_0x115f580:

Io> elvis at("home")
==> Graceland

Io> elvis atPut("style", "rock and roll")
==> Map_0x115f580:

Io> elvis asObject
==> Object_0x11c1d90:
  home              = "Graceland"
  style             = "rock and roll"

Io> elvis asList
==> list(list("style", "rock and roll"), list("home", "Graceland"))

Io> elvis keys
==> list("style", "home")

Io> elvis size
==>2
```

При близком рассмотрении хэш больше напоминает объект в языке Io, где ключами являются слоты, которые связаны с некоторыми значениями. Весьма интересной в этом свете представляется возможность быстрого преобразования комбинации слотов в объект.

Теперь, после знакомства с основными коллекциями, у вас может появиться желание попробовать их в деле. Но для этого сначала нужно познакомиться с управляющими конструкциями, опирающимися на логические значения.

true, false, nil и одиночные объекты

Условные конструкции в языке Io очень похожи на аналогичные конструкции в других объектно-ориентированных языках. Например:

```
Io> 4 < 5
==> true
Io> 4 <= 3
==> false
Io> true and false
==> false
Io> true and true
==> true
Io> true or true
==> true
Io> true or false
==> true
Io> 4 < 5 and 6 > 7
==> false
Io> true and 6
==> true
Io> true and 0
==> true
```

Все достаточно просто. Обратите внимание, что число 0 имеет истинное значение, как в языке Ruby, а не ложное, как в языке C. Так что же из себя представляет значение `true`?

```
Io> true proto
==> Object_0x200490:
                  = Object_()
  !=              = Object_!=()
...
```

```
Io> true clone
==> true
Io> false clone
==> false
Io> nil clone
==> nil
```

Как интересно, однако! Значения `true`, `false` и `nil` являются объектами-одиночками (singletons). Попытки создать их копии просто возвращают сами объекты. То же самое легко можно реализовать самому.

Попробуем создать собственный объект-одиночку:

```
Io> Highlander := Object clone
==> Highlander_0x378920:
  type              = "Highlander"

Io> Highlander clone := Highlander
==> Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"
```

Здесь мы просто переопределили метод `clone`, чтобы он возвращал сам объект `Highlander`, а не передавал сообщение вверх по дереву наследования. Теперь, при попытке создать копию объекта `Highlander`, мы будем получать следующий результат:

```
Io> Highlander clone
==> Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"
Io> fred := Highlander clone
==> Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"

Io> mike := Highlander clone
==> Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"

Io> fred == mike
==> true
```

Две копии представляют один и тот же объект. Но для других объектов это не так:

```
Io> one := Object clone
==> Object_0x356d00:

Io> two := Object clone
==> Object_0x31eb60:

Io> one == two
==> false
```

Теперь в программе может существовать только один объект `Highlander`. Это решение выглядит простым и элегантным, но немножко неожиданным. Теперь вы знаете достаточно много, чтобы действовать радикально, например вы способны подменить метод `clone` объекта и превратить его в объект-одиночку.

Но будьте осторожны. Язык Io можно любить или ненавидеть, но нельзя отрицать, что он является достаточно интересным языком. Io позволяет изменить практически любой слот любого объекта, даже определяемого самим языком. Ниже приводится пример того, чего делать нежелательно:

```
Object clone := "hosed"
```

После переопределения метода `clone` объекта `Object` никто не сможет создавать объекты. И эту проблему не удастся исправить. Останется только завершить процесс. Но вы можете также реализовать еще более интересное поведение. Так как у нас имеется полный доступ к операторам и слотам, составляющим любой объект, мы можем сконструировать предметно-ориентированный язык всего лишь несколькими строками кода.

Прежде чем завершить первый день, давайте послушаем, что говорит создатель языка.

Интервью со Стивом Декортом

Стив Декорт (Steve Dekorte) работает независимым консультантом в Сан-Франциско. Он любезно согласился рассказать мне об истории создания Io.

Брюс Тейт: Как у вас родилась идея создать язык Io?

Стив Декорт: В 2002 году мой друг Дрю Нельсон (Dru Nelson) создал язык Cel (прототипом для которого послужил язык Self) и предложил мне высказать свое мнение о его реализации. На тот момент я достаточно плохо представлял себе, как работают языки программирования, чтобы сказать что-то полезное, поэтому я решил написать свой, маленький язык программирования, чтобы лучше разобраться в основных проблемах. В результате появился язык Io.

Брюс Тейт: Что в Io кажется вам наиболее привлекательным?

Стив Декорт: Больше всего мне нравятся простота и единство синтаксиса и семантики. Эта простота помогает понять, что делает код. У меня жутко плохая память. Я постоянно забываю синтаксические и семантические правила языка C, и я постоянно вынужден вспоминать их. (От редактора: Стив написал Io на языке C.) Но при использовании Io мне не приходится делать этого.

Например, достаточно просто взглянуть на код, такой как `people select(age > 20) map(address) println`, чтобы понять, что он делает. Он фильтрует список людей по возрасту, получает адреса и выводит их.

Чем проще семантика языка, тем он гибче. Благодаря простоте вы сможете составлять композиции, которых даже не понимали на момент создания языка. Например, существуют игры, такие как головоломки, предполагающие наличие конечного решения, а есть игры, не имеющие логического конца. Такие бесконечные игры особенно интересны, потому что позволяют создавать ситуации, которые создатели игры даже представить не могли. В этом смысле Io является такой бесконечной игрой.

Иногда в других языках их авторы предусматривают синтаксические сокращения. Это ведет к появлению дополнительных правил парсинга. При программировании на любом языке у вас в голове должен работать свой собственный парсер. Чем сложнее язык, тем больше правил его парсинга приходится держать в голове. Чем больший объем работы приходится выполнять парсеру, тем напряженнее приходится вам думать.

Брюс Тейт: Какие наиболее существенные ограничения языка Io вы могли бы назвать?

Стив Декорт: За гибкость языка Io пришлось заплатить его производительностью. Однако в нем имеются некоторые важные достоинства (такие как сопрограммы, асинхронные сокеты и поддержка архитектуры SIMD[1]), благодаря которым в отдельных случаях программы могут выполняться даже быстрее, чем программы на языке C, написанные с применением традиционных механизмов управления потоками выполнения и сокетами.

Одним из недостатков является также отсутствие синтаксических конструкций, упрощающих чтение кода. У меня были подобные проблемы с языком Lisp, поэтому я их прекрасно вижу и понимаю. Дополнительный синтаксис мог бы упростить чтение кода. Новые пользователи иногда жалуются, что Io имеет слишком бедный синтаксис, но потом жалобы обычно прекращаются.

Брюс Тейт: С каким из самых необычных применений Io вам приходилось сталкиваться?

Стив Декорт: До меня дошли слухи, что Io используется на спутниках, в маршрутизаторах, как язык управления настройками, в игровых программах, как язык сценариев. Кроме того, известно, что Io используется компанией Pixar. Они отмечали этот факт в своем блоге.

---

[1] Single Instruction Multiple Data – один поток команд и множество потоков данных. – *Прим. перев.*

Итак, первый день закончен, и пришло время сделать перерыв. Теперь вы можете приостановиться и попробовать применить полученные знания на практике.

Что мы узнали в первый день

Вы получили немалый объем информации о языке Io. Теперь вы знакомы с основными чертами его характера. Язык на основе прототипов имеет очень простой синтаксис, с помощью которого можно даже перекраивать базовые компоненты самого языка. В языке полностью отсутствует синтаксический сахар. Отчасти такой минимализм делает программный код менее читаемым.

Однако бедность синтаксиса имеет свои преимущества. Из-за отсутствия синтаксического сахара отпадает необходимость изучать дополнительные специальные правила и исключения из них. Научившись читать одно предложение, вы сможете прочитать их все. А дальше вам остается только накапливать свой словарный запас.

Вам как начинающему изучать новый язык достаточно лишь разобраться:

- с простыми синтаксическими правилами;
- сообщениями;
- прототипами;
- библиотеками.

День 1: задания для самостоятельного решения

Поиск информации по языку Io может оказаться не самым простым делом, потому что слово «Io» имеет множество разных значений. Я рекомендую «гуглить» по фразе «Io language» (язык программирования Io).

Найдите:

- примеры решения задач на языке Io;
- форумы сообщества пользователей Io, где можно было бы получить ответы на свои вопросы;
- руководство по идиоматическим приемам программирования на Io.

Ответьте на следующие вопросы.

- Вычислите `1 + 1` и затем `1 + "one"`. Как вы считаете – Io является языком со строгим или со слабым контролем типов? Подкрепите свой ответ кодом.
- Значение `0` в языке интерпретируется как истинное или как ложное? А пустая строка? А значение `nil`? Подкрепите свой ответ кодом.

- Как узнать, определен ли слот в данном объекте или является собственностью прототипа?
- Чем отличаются операторы = (равно), := (двоеточие-равно) и ::= (двоеточие-двоеточие-равно)? В каких случаях используется каждый из них?

Практические задания:

- запустите файл с программой на языке Io;
- выполните код в слоте, заданном его именем.

Поэкспериментируйте со слотами и прототипами. Убедитесь, что понимаете, как действуют прототипы.

3.3. День 2: Сосисочный король

Вспомним ненадолго о Феррисе Бьюллере. В фильме студент колледжа выдавал себя за сосисочного короля из Чикаго. Он получил столик в хорошем ресторане, потому что был готов нарушить правила. Если у вас есть опыт работы на языке Java и он вам нравится, вы могли бы подумать, что слишком много свободы не всегда хорошо и, возможно, Бьюллер заслуживает, чтобы его прогнали. В этом случае, программируя на Io, просто расслабьтесь и воспользуйтесь его преимуществами. Если у вас есть опыт работы на языке Perl, розыгрыш Бьюллера может прийтись вам по вкусу, так как, возможно, вы привыкли к свободе самовыражения. В этом случае, чтобы программировать на Io, вам придется научиться ограничивать себя. Во второй день мы займемся исследованием особенностей применения слотов и сообщений для реализации прикладной логики.

Условные конструкции и циклы

Ни одна условная инструкция в языке Io не имеет синтаксического сахара. Особенности их использования просты и легко запоминаются, но читаются они немножко сложнее, чем могли бы. Бесконечный цикл реализуется очень просто. Нажмите комбинацию **Control**+**C**, чтобы прервать его:

```
Io> loop("getting dizzy..." println)
getting dizzy...
getting dizzy...
...
getting dizzy.^C
IoVM:
        Received signal. Setting interrupt flag.
...
```

Бесконечные циклы нередко используются в различных конструкциях конкурентного выполнения, но в обычной практике чаще используются циклы с условием продолжения, такие как `while`. Цикл `while` принимает условие и сообщение, которое требуется вычислить. Имейте в виду, что точка с запятой в языке Io служит для объединения двух разных сообщений:

```
Io> i := 1
==> 1
Io> while(i <= 11, i println; i = i + 1); "This one goes up to 11"
println
1
2
...
10
11
This one goes up to 11
```

То же самое можно было бы реализовать с применением цикла `for`. Цикл `for` принимает имя счетчика, начальное значение, конечное значение, необязательный шаг наращивания и сообщение с отправителем.

```
Io> for(i, 1, 11, i println); "This one goes up to 11" println
1
2
...
10
11
This one goes up to 11
==> This one goes up to 11
```

И с необязательным шагом наращивания счетчика:

```
Io> for(i, 1, 11, 2, i println); "This one goes up to 11" println
1
3
5
7
9
11
This one goes up to 11
==> This one goes up to 11
```

В действительности число параметров может быть произвольным. Нужен дополнительный параметр? Язык Io позволит вам передать его. Это может показаться удобным, но будьте внимательны – здесь нет компилятора, который подбирал бы за вами ваши ошибки:

```
Io> for(i, 1, 2, 1, i println, "extra argument")
1
2
==> 2
Io> for(i, 1, 2, i println, "extra argument")
2
==> extra argument
```

В первом случае строка "extra argument" действительно является дополнительным параметром. Во втором случае был опущен необязательный шаг наращивания счетчика, что вызвало смещение всех остальных параметров влево. В результате строка "extra argument" превратилась в сообщение, а величину шага стало определять выражение i println, возвращающее i. Если такая строка кода окажется в глубине сложного пакета, она может вызвать целую цепь логических ошибок, причину которых найти будет очень непросто. Io дает вам свободу, но иногда этой свободы оказывается слишком много.

Управляющая конструкция if реализована в виде функции и имеет форму: if(condition, true code, false code). Функция выполнит true code, если условие condition истинно; в противном случае она выполнит false code:

```
Io> if(true, "It is true.", "It is false.")
==> It is true.
Io> if(false) then("It is true") else("It is false")
==> nil
Io> if(false) then("It is true." println) else("It is false." println)
It is false.
==> nil
```

Потратив некоторое время на управляющие конструкции, можно попытаться использовать их для реализации своих собственных операторов.

Операторы

Подобно объектно-ориентированным языкам, языки программирования, основанные на прототипах, позволяют вам добавлять синтаксический сахар в виде реализации собственных *операторов*. Операторы в Io являются специальными методами, такими как + и /, имеющими особую форму. В языке Io есть возможность вывести таблицу доступных операторов:

```
Io> OperatorTable
==> OperatorTable_0x100296098:
Operators
```

```
0    ? @ @@
1    **
2    % * /
3    +-
4    << >>
5    < <= > >=
6    != ==
7    &
8    ^
9    |
10   && and
11   or ||
12   ..
13   %= &= *= += -= /= <<= >>= ^= |=
14   return

Assign Operators
  ::= newSlot
  := setSlot
  = updateSlot

To add a new operator: OperatorTable addOperator("+", 4)
    and implement the + message.
To add a new assign operator: OperatorTable addAssignOperator(
  "=", "updateSlot") and implement the updateSlot message.
```

Как видите, присваивание может выполняться несколькими различными операторами. Число слева отражает приоритет оператора. Операторы с меньшим уровнем приоритета выполняются первыми. То есть оператор `+` выполняется раньше оператора `==`, а оператор `*` выполняется раньше оператора `+`, как и следовало ожидать. Изменить порядок выполнения операторов можно с помощью скобок `()`. Давайте определим оператор ИСКЛЮЧАЮЩЕЕ ИЛИ. Оператор `xor` должен возвращать `true`, если только один из операндов имеет истинное значение. В противном случае он должен возвращать `false`. Сначала добавим оператор в таблицу:

```
Io> OperatorTable addOperator("xor", 11)
==> OperatorTable_0x100296098:
Operators
  ...
  10   && and
  11   or xor ||
  12   ..
  ...
```

Как видите, новый оператор появился на предопределенном ему месте. Далее нужно реализовать метод `xor` в объектах `true` и `false`:

```
Io> true xor := method(bool, if(bool, false, true))
==> method(bool,
    if(bool, false, true)
)
Io> false xor := method(bool, if(bool, true, false))
==> method(bool,
    if(bool, true, false)
)
```

Для простоты мы использовали решение «в лоб». Новый оператор действует именно так, как и предполагалось:

```
Io> true xor true
==> false
Io> true xor false
==> true
Io> false xor true
==> true
Io> false xor false
==> false
```

В конечном счете выражение `true xor true` интерпретируется как `true xor(true)`. Присутствие метода в таблице операторов определяет его приоритет и обеспечивает упрощенный синтаксис его использования.

Операторы присваивания находятся в отдельной таблице, и действуют они несколько иначе. Операторы присваивания играют роль сообщений. Примеры их использования приводятся в разделе 3.4 «Предметно-ориентированные языки» ниже. А пока это все, что я хотел сказать об операторах. Давайте перейдем к сообщениям и познакомимся с возможностью создания собственных управляющих конструкций.

Сообщения

Когда я работал над этой главой, один из членов сообщества Io помогал мне разобраться с непонятными для меня моментами. Он говорил: «Брюс, поймите главное – в языке Io все сущности являются сообщениями». В программном коде на Io все, кроме комментариев и запятых (`,`) между аргументами, является сообщениями. Все. Овладение языком Io также означает овладение приемами работы с сообщениями, помимо простых вызовов методов. Одной из важнейших особенностей языка является поддержка механизма рефлексии (исследования внутренней структуры) сообщений – с его помощью можно получить любые характеристики сообщения и действовать соответственно.

Любое сообщение состоит из трех компонентов: отправитель, получатель и аргументы. Отправитель посылает сообщение получателю. Получатель выполняет его.

Доступ к любой метаинформации о любом сообщении можно получить с помощью метода `call`. Для демонстрации создадим пару объектов: объект `postOffice`, получающий сообщения:

```
Io> postOffice := Object clone
==> Object_0x100444b38:

Io> postOffice packageSender := method(call sender)
==> method(
    call sender
)
```

и объект `mailer`, осуществляющий их доставку:

```
Io> mailer := Object clone
==> Object_0x1005bfda0:

Io> mailer deliver := method(postOffice packageSender)
==> method(
    postOffice packageSender
)
```

Итак, у нас имеется слот `deliver`, отправляющий сообщение `packageSender` объекту `postOffice`. Теперь у нас имеется объект `mailer`, осуществляющий доставку сообщения:

```
Io> mailer deliver
==> Object_0x1005bfda0:
  deliver         = method(...)
```

Метод `deliver` – это объект, отправляющий сообщение. Мы можем также узнать получателя:

```
Io> postOffice messageTarget := method(call target)
==> method(
    call target
)

Io> postOffice messageTarget
==> Object_0x1004ce658:
  messageTarget   = method(...)
  packageSender   = method(...)
```

Достаточно просто. Получателем является объект `postOffice`, как видно из имен слотов. А теперь определим оригинальное имя сообщения и аргументы:

```
Io> postOffice messageArgs := method(call message arguments)
==> method(
    call message arguments
)
Io> postOffice messageName := method(call message name)
==> method(
    call message name
)
Io> postOffice messageArgs("one", 2, :three)
==> list("one", 2, : three)
Io> postOffice messageName
==> messageName
```

Итак, в языке Io имеется множество методов, составляющих механизм рефлексии сообщений. Рассмотрим следующий вопрос: «Как Io вычисляет сообщения?»

В большинстве языков аргументы передаются как значения на стеке. Например, Java сначала вычисляет значения всех параметров, а затем помещает полученные значения на стек. Io действует иначе. Он передает само сообщение и его контекст. А вычислением сообщений занимаются приемники. Благодаря этому с помощью сообщений можно реализовать собственные управляющие конструкции. Вспомните метод `if`. Он имеет следующую реализацию: `if(booleanExpression, trueBlock, falseBlock)`. Допустим, что нам потребовалось реализовать конструкцию `unless`. Вот как это можно сделать:

io/unless.io
http://media.pragprog.com/titles/btlang/code/io/unless.io

```
unless := method(
    (call sender doMessage(call message argAt(0))) ifFalse(
    call sender doMessage(call message argAt(1))) ifTrue(
    call sender doMessage(call message argAt(2)))
)

unless(1 == 2, write("One is not two\n" ), write("one is two\n" ))
```

Это очень любопытный пример, поэтому вглядитесь в него внимательнее. Метод `doMessage` можно считать аналогом функции `eval` в Ruby, но более низкоуровневым. Функция `eval` в Ruby выполняет строку как программный код, а метод `doMessage` выполняет произвольное сообщение. Io интерпретирует параметры сообщения, но откладывает их привязку и выполнение. В типичных объектно-ориентированных языках интерпретатор или компилятор должен вычислить все аргументы, выполнить все блоки кода и поместить полученные значения на стек. В Io этого вообще не происходит.

Представьте, что объект `westley` посылает объекту `princessButtercup` сообщение `unless(trueLove, ("It is false" println), ("It is true" println))`. В результате выполняется следующая последовательность действий:

- объект `westley` посылает предыдущее сообщение;
- Io принимает интерпретированное сообщение и контекст (отправителя, получателя и сообщение) и помещает их на стек;
- затем `princessButtercup` вычисляет сообщение. В нем отсутствует слот `unless`, поэтому Io начинает восхождение по цепочке прототипов, пока не найдет слот `unless`;
- далее начинается выполнение сообщения `unless`. Сначала Io выполняет `call sender doMessage(call message argAt(0))`. Этот код упрощается до `westley trueLove`. Если вы видели фильм «The Princess Bride»[1], вы знаете, что `westley` имеет слот `trueLove`[2], хранящий значение `true`;
- результатом вычисления сообщения является истина, поэтому выполняется третий блок кода, что упрощает выражение до `westley ("It is true" println)`.

Мы воспользовались тем обстоятельством, что Io не выполняет аргументов для реализации управляющей конструкции `unless`. Эта особенность открывает очень широкие возможности. До сих пор мы рассматривали только одну сторону механизма рефлексии: позволяющую исследовать сообщения. С другой стороны уравнения находится состояние. Под состоянием мы подразумеваем слоты объектов.

Рефлексия

В Io имеется набор простых методов, позволяющих узнать, что происходит внутри слотов. Ниже демонстрируется пример применения некоторых из них. Код в примере создает пару объектов и затем выполняет подъем вверх по цепочке прототипов с помощью метода `ancestors`:

io/animals.io

http://media.pragprog.com/titles/btlang/code/io/animals.io

```
Object ancestors := method(
        prototype := self proto
        if(prototype != Object,
        writeln("Slots of " , prototype type, "\n---------------" )
        prototype slotNames foreach(slotName, writeln(slotName))
```

---

[1] http://ru.wikipedia.org/wiki/Принцесса-невеста. – *Прим. перев.*

[2] То есть Уэстли любит принцессу по-настоящему. – *Прим. перев.*

```
        writeln
        prototype ancestors))

Animal := Object clone
Animal speak := method(
            "ambiguous animal noise" println)
Duck := Animal clone
Duck speak := method(
            "quack" println)

Duck walk := method(
             "waddle" println)

disco := Duck clone
disco ancestors
```

Код получился не очень сложным. Сначала он создает прототип `Animal` и на его основе создает экземпляр `Duck` с методом `speak`. Прототипом объекта `disco` является `Duck`. Метод `ancestors` выводит слоты прототипа объекта и затем вызывает метод `ancestors` прототипа. Имейте в виду, что объект может иметь более одного прототипа, но в нашем примере это обстоятельство не учитывается. Для экономии места в книге мы прервем рекурсию перед выводом всех слотов в прототипе `Object`. Запустить этот пример можно командой `io animals.io`:

```
Slots of Duck
---------------
speak
walk
type

Slots of Animal
---------------
speak
type
```

В выводе нет никаких сюрпризов. Все объекты имеют прототипы, а эти прототипы являются объектами и имеют слоты. В языке Io механизм рефлексии делится на две части. Первая из них, как было показано в примере с передачей сообщений, служит для исследования сообщений, а вторая – для исследования объектов и их слотов.

Что мы узнали во второй день

Второй день должен стать для вас днем больших открытий. Теперь вы знаете достаточно, чтобы суметь самостоятельно решать простые задачи с небольшой помощью документации. Вы узнали, как принимать решения, определять методы, использовать структуры данных и основные

управляющие конструкции. В примерах, представленных в этом разделе, мы опробовали язык Io в деле. Продолжайте изучать язык Io. Он понравится вам еще больше, когда мы приступим к знакомству с такими его возможностями, как метапрограммирование и многозадачность.

День 2: задания для самостоятельного решения

Практические задания:

- последовательность чисел Фибоначчи начинается с двух 1. Каждое последующее число является суммой двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 21 и т. д. Напишите программу поиска *n*-го числа Фибоначчи. Вызов `fib(1)` должен возвращать число 1, а вызов `fib(4)` – число 3. Дополнительно решите эту задачу с применением цикла и рекурсии;
- как можно было бы изменить оператор /, чтобы он возвращал 0, если знаменатель равен нулю?
- напишите программу, вычисляющую сумму всех чисел в двумерном массиве;
- добавьте в список слот `myAverage`, вычисляющий среднее значение для чисел в списке. Что произойдет, если в списке не будет обнаружено ни одного числа? (Дополнительно: реализуйте возбуждение исключения, если хотя бы один элемент в списке не является числом.);
- напишите прототип двумерного списка. Метод `dim(x, y)` должен создавать список из `y` списков длиной `x` каждый. Метод `set(x,y,value)` должен устанавливать значение элемента списка, а метод `get(x, y)` – возвращать это значение;
- дополнительно: напишите метод `transpose`, преобразующий список так, что `(new_matrix get(y, x)) == matrix get(x, y)`;
- запишите матрицу в файл и прочитайте ее из файла;
- напишите программу, дающую 10 попыток угадать случайное число от 1 до 100. Можете добавить в программу вывод подсказок «больше» и «меньше».

3.4. День 3: На параде и в других неожиданных местах

Первые несколько дней знакомства с Io я пребывал в растерянности, но спустя пару недель я уже хихикал как школьница, когда язык демонстрировал мне свои странности. Он напоминал мне Ферриса,

появляющегося в новостях, на стадионе, на параде – везде, где его совсем не ожидали увидеть. В конечном итоге я научился получать от Io именно то, что мне было нужно, – этот язык изменил мой образ мышления.

Предметно-ориентированные языки

Практически каждый, кто активно использует язык Io, начинает понимать широту его возможностей в области создания предметно-ориентированных языков. Джереми Треганн (Jeremy Tregunna), один из наиболее активных разработчиков Io, рассказал мне о реализации подмножества языка C на Io, занимающей всего 40 строк кода! Так как этот пример пока еще слишком сложен для нас, мы познакомимся с другими твореними Джереми. Ниже приводится реализация API для работы с телефонными номерами.

Допустим, что нам нужно реализовать поддержку списков телефонных номеров в следующем виде:

```
{
    "Bob Smith": "5195551212",
    "Mary Walsh": "4162223434"
}
```

Существует множество способов решения задачи управления таким списком. Первые два из них – парсинг и интерпретация. Под парсингом подразумеваются разработка алгоритма распознавания различных синтаксических элементов и его реализация в форме, понятной интерпретатору Io. Но эта задача для другого дня. Было бы гораздо интереснее реализовать интерпретацию данных как хэша Io. Для этого придется изменить сам язык Io. По окончании задуманного Io будет принимать подобные списки и на их основе создавать хэши!

Ниже показано, как эта проблема была решена Джереми с помощью Криса Кепплера (Chris Kappler), обновившего данный пример до последней версии Io:

io/phonebook.io

http://media.pragprog.com/titles/btlang/code/io/phonebook.io

```
OperatorTable addAssignOperator(":" , "atPutNumber" )
curlyBrackets := method(
  r := Map clone
  call message arguments foreach(arg,
       r doMessage(arg)
       )
  r
```

```
)
Map atPutNumber := method(
  self atPut(
       call evalArgAt(0) asMutable removePrefix("\"") removeSuffix(" \""
),
       call evalArgAt(1))
)
s := File with("phonebook.txt" ) openForReading contents
phoneNumbers := doString(s)
phoneNumbers keys println
phoneNumbers values println
```

Этот код немного сложнее, чем встречавшийся нам до сих пор, но вы уже знаете основы синтаксиса языка. Давайте разберем его:

```
OperatorTable addAssignOperator(":" , "atPutNumber" )
```

Первая строка добавляет новый оператор в таблицу операторов присваивания. После этого, встретив двоеточие (`:`), интерпретатор Io будет интерпретировать его как вызов метода `atPutNumber`, в первом аргументе которому будет передаваться имя (то есть строка), а во втором – значение. Таким образом, любая пара `key : value` будет интерпретироваться как `atPutNumber("key", value)`. Идем дальше:

```
curlyBrackets := method(
  r := Map clone
  call message arguments foreach(arg,
       r doMessage(arg)
       )
  r
)
```

Парсер вызывает метод `curlyBrackets` всякий раз, когда встречает фигурные скобки (`{}`). Внутри этого метода мы создаем пустое отображение (map). Затем выполняем `call message arguments foreach(arg, r doMessage(arg))` для каждого аргумента. Это – самая важная строка в программе! Давайте разберемся с ней.

Конструкция `call message` извлекает все, что находится между фигурными скобками. Далее выполняются итерации по телефонным номерам в списке с помощью метода `forEach`. Для каждой записи с именем и номером выполняется `r doMessage(arg)`. Например, для первой записи выполняется выражение `r "Bob Smith": "5195551212"`. Поскольку двоеточие (`:`) является оператором, вызывающим метод `atPutNumber`, конструкция разворачивается в выражение `r atPutNumber("Bob Smith", "5195551212")`. Которое в конечном итоге интерпретируется как

```
Map atPutNumber := method(
  self atPut(
```

```
        call evalArgAt(0) asMutable removePrefix("\"") removeSuffix(" \"" ),
        call evalArgAt(1))
)
```

Напомню, что пара `key : value` интерпретируется как вызов `atPutNumber("key", value)`. В данном случае аргумент `key` уже является строкой, поэтому мы отбрасываем окружающие кавычки. Метод `atPutNumber` просто вызывает метод `atPut` целевого объекта, каковым является `self`, отбрасывая кавычки в первом аргументе. Так как сообщения являются неизменяемыми, чтобы отбросить кавычки, необходимо преобразовать сообщение в изменяемое значение.

Используется описанный выше код следующим образом:

```
s := File with("phonebook.txt" ) openForReading contents
phoneNumbers := doString(s)
phoneNumbers keys println
phoneNumbers values println
```

Синтаксис языка Io прост и понятен. Вам нужно лишь разобраться с происходящим в библиотеках. В данном случае мы столкнулись с двумя новыми библиотеками. Сообщение `doString` интерпретирует телефонную книгу как программный код, `File` — это прототип для работы с файлами, `with` принимает имя файла и возвращает объект файла, `openForReading` открывает этот файл и так же возвращает объект файла, а `contents` возвращает содержимое файла. А весь вместе, этот код читает телефонную книгу и интерпретирует ее как программный код.

Далее фигурные скобки создают отображение. Каждая строка в файле, имеющая вид `"string1" : "string2"`, добавляется в отображение вызовом `atPut("string1", "string2")`, и в результате мы получаем хэш с телефонными номерами. Итак, благодаря возможности переопределять в языке Io все, что угодно, от операторов до синтаксических конструкций, мы можем конструировать собственные предметно-ориентированные языки для обработки своей информации.

Теперь в вашей голове должны появиться идеи, как можно было бы изменить синтаксис Io. А что вы думаете о динамических особенностях языка? Давайте перейдем к следующему разделу, где рассматривается эта тема.

Аналог метода method_missing в языке Io

Давайте уточним, как протекает выполнение кода в Io. Вся обработка сообщений сосредоточена в объекте `Object`. Когда объекту отправляется некоторое сообщение, выполняются следующие действия:

- вычисляются аргументы;
- извлекаются имя, приемник и отправитель сообщения;
- выполняется попытка прочитать в объекте-получателе слот с именем, совпадающим с именем сообщения;
- если искомый слот имеется, возвращаются данные из него или вызывается метод, хранящийся в нем;
- если слот отсутствует, сообщение передается прототипу.

Это – основы работы механизма наследования в Io. Обычно программисту не приходится вмешиваться в его работу.

Но такая возможность существует. Она реализована в виде сообщения `forward`, которое действует так же, как метод `method_missing` в Ruby. В Io отсутствуют классы, поэтому изменение `forward` затронет также основы поведения объектов. Это напоминает жонглирование острыми ножами, стоя на высокоподнятом канате. Весьма эффектный трюк, если овладеть им в совершенстве, так давайте приступим!

XML – отличный способ представления структурированных данных, но синтаксис этого языка разметки оставляет желать лучшего. У вас может появиться желание сконструировать нечто, что позволит представлять XML-данные в виде кода на Io.

Например, чтобы получить возможность выразить следующий фрагмент:

```
<body>
<p>
This is a simple paragraph.
</p>
</body>
```

в виде:

```
body(
    p("This is a simple paragraph.")
)
```

Назовем новый язык LispML и будем использовать сообщение `forward` как своеобразный метод `method_missing`:

io/builder.io
http://media.pragprog.com/titles/btlang/code/io/builder.io

```
Builder := Object clone

Builder forward := method(
  writeln("<" , call message name, ">" )
  call message arguments foreach(
        arg,
```

```
        content := self doMessage(arg);
        if(content type == "Sequence" , writeln(content)))
  writeln("</" , call message name, ">" ))

Builder ul(
        li("Io" ),
        li("Lua" ),
        li("JavaScript" ))
```

Разберем эту реализацию по частям. Прототип `Builder` – это рабочая лошадка. Он переопределяет метод `forward`, который перехватывает обращения к несуществующим методам. В первую очередь он выводит открывающий тег. Далее задействуется механизм рефлексии для анализа сообщения. Если сообщение является строкой, Io распознает его как последовательность и `Builder` выводит строку без кавычек. В заключение `Builder` выводит закрывающий тег.

Результат получается именно таким, как мы и ожидали:

```
<ul>
<li>
Io
</li>
<li>
Lua
</li>
<li>
JavaScript
</li>
</ul>
```

Я должен сказать, что не уверен, действительно ли язык LispML можно считать большим улучшением традиционного языка XML, но сам пример весьма поучителен. Мы всего лишь изменили порядок работы механизма наследования в одном из прототипов. Любой экземпляр прототипа `Builder` будет обладать тем же поведением. Используя этот прием, можно создать новый язык с синтаксисом Io, но совершенно иным поведением, определив собственный прототип и породив от него все остальные свои прототипы. Можно даже переопределить прототип `Object` так, чтобы его метод `clone` возвращал ваш объект.

Параллельные вычисления

В Io имеется великолепная библиотека поддержки многозадачности. Главными ее компонентами являются сопрограммы (coroutines), акторы (actors) и отложенные вычисления (futures) .

Сопрограммы

Основу поддержки параллельных вычислений составляют сопрограммы. Сопрограммы дают возможность добровольно приостанавливать и возобновлять выполнение процесса. Сопрограмму можно рассматривать как функцию с многочисленными точками входа и выхода. В каждой точке выхода из сопрограммы выполнение процесса приостанавливается, и управление передается другому процессу. Имеется возможность асинхронной передачи сообщений добавлением приставки `@` или `@@` перед именем сообщения. В первом случае возвращается отложенное задание (об этом рассказывается ниже), а во втором – `nil` и обработка сообщения запускаются в отдельном потоке выполнения. Например, взгляните на следующую программу:

io/coroutine.io
http://media.pragprog.com/titles/btlang/code/io/coroutine.io

```
vizzini := Object clone
vizzini talk := method(
            "Fezzik, are there rocks ahead?" println
            yield
            "No more rhymes now, I mean it." println
            yield)

fezzik := Object clone

fezzik rhyme := method(
            yield
            "If there are, we'll all be dead." println
            yield
            "Anybody want a peanut?" println)

vizzini @@talk; fezzik @@rhyme

Coroutine currentCoroutine pause
```

`fezzik` и `vizzini` – это независимые экземпляры `Object` с сопрограммами. Методы `talk` и `rhyme` вызываются асинхронно. Они выполняются параллельно, добровольно передавая управление друг другу, отправляя сообщение `yield`. Последняя пауза выполняется с целью ожидания завершения обработки всех асинхронных сообщений, после чего программа завершает работу. Сопрограммы отлично подходят для решения задач, где достаточно кооперативной мультизадачности. В этом примере два процесса прекрасно координируют взаимную работу, воспроизводя поэтическое произведение:

```
batate$ io code/io/coroutine.io
Fezzik, are there rocks ahead?
If there are, we'll all be dead.
No more rhymes now, I mean it.
Anybody want a peanut?
Scheduler: nothing left to resume so we are exiting
```

В Java и C-подобных языках используется философия конкурентного выполнения, которая называется *вытесняющая многозадачность* (preemptive multitasking). Когда эта стратегия объединяется с объектами, имеющими изменяемое состояние, получаются программы с плохо предсказуемым поведением, которые практически невозможно надежно протестировать с применением современных технологий тестирования, используемых в большинстве коллективов разработчиков. Сопрограммы выгодно отличаются в этом смысле. Применяя сопрограммы, приложения могут добровольно отдавать управление в наиболее подходящие для этого моменты времени. Клиент может отдать управление на время ожидания ответа от сервера. Рабочие процессы могут брать паузы после обработки очередного элемента из очереди заданий.

Сопрограммы являются основными строительными блоками более высокоуровневых абстракций, таких как акторы (actors). Акторы можно рассматривать как универсальные примитивы поддержки параллельных вычислений, позволяющие посылать сообщения и обрабатывать их, а также создавать другие акторы. Сообщения, которые принимают акторы, являются асинхронными. В языке Io актор помещает входящее сообщение в очередь и обрабатывает содержимое очереди с применением сопрограмм.

В следующем подразделе мы познакомимся с акторами. Не надейтесь, что программировать их будет просто.

Акторы

Акторы имеют огромное теоретическое преимущество перед потоками выполнения. Акторы изменяют собственное состояние и взаимодействуют с другими акторами только через управляемые очереди. Потоки выполнения могут изменять состояние друг друга без ограничений. Потоки подвержены проблеме *состояния гонки* (race condition), когда два потока одновременно обращаются к одному и тому же ресурсу, приводящей к непредсказуемым результатам.

В языке Io отправка асинхронного сообщения любому объекту превращает его в актор. Это, собственно, все, что можно сказать о под-

держке акторов. Рассмотрим простой пример. Сначала создадим два объекта с именами `faster` и `slower`:

```
Io> slower := Object clone
==> Object_0x1004ebb18:

Io> faster := Object clone
==> Object_0x100340b10:
```

Теперь добавим в каждый из них метод `start`:

```
Io> slower start := method(wait(2); writeln("slowly"))
==> method(
    wait(2); writeln("slowly")
)
Io> faster start := method(wait(1); writeln("quickly"))
==> method(
    wait(1); writeln("quickly")
)
```

Мы можем вызвать методы последовательно, в одной строке кода:

```
Io> slower start; faster start
slowly
quickly
==> nil
```

Они вызываются последовательно, потому что первое сообщение должно быть обработано до того, как начнется обработка второго. Но мы легко можем запустить каждое сообщение в отдельном потоке выполнения, добавив приставку `@@` к именам методов. В результате строка кода немедленно вернет значение `nil`:

```
Io> slower @@start; faster @@start; wait(3)
quickly
slowly
```

Здесь была добавлена дополнительная задержка, чтобы все потоки выполнения успели завершиться до того, как завершится сама программа. Мы запустили два потока выполнения, превратив оба объекта в акторы, просто отправив им асинхронные сообщения!

Отложенные вычисления

Обзор поддержки параллельных вычислений мы завершим знакомством с механизмом отложенных вычислений, или отложенных заданий (futures). Отложенное задание (future) – это объект результата, который немедленно возвращает управление в ответ на асинхронное сообщение. Так как для обработки сообщения может потребо-

ваться некоторое время, отложенное задание превратится в готовый результат, как только он станет доступен. Если попытаться запросить результат отложенного задания еще до того, как он будет вычислен, процесс заблокируется до окончания вычислений. Допустим, что имеется метод, выполняющий продолжительные вычисления:

```
futureResult := URL with("http://google.com/") @fetch
```

Мы можем запустить этот метод и тут же продолжить заниматься другими делами, пока результаты его работы не станут доступны:

```
writeln("Do something immediately while fetch goes on in background...")
// ...
```

И затем использовать результаты отложенных вычислений:

```
writeln("This will block until the result is available.")
// эта строка будет выполнена немедленно

writeln("fetched ", futureResult size, " bytes")
// эта строка может заблокировать процесс, пока не завершатся
// отложенные вычисления

// после чего Io выведет значение
==> 1955
```

Фрагмент `futureResult` вернет объект отложенного задания немедленно. В Io отложенные задания не проксируются другими объектами! Поэтому обращение к отложенному заданию будет заблокировано, пока не будет получен результат вычислений. Возвращаемым значением является объект `Future`, пока результат не будет вычислен, после чего все ссылки на него будут указывать на объект результата. В консоль будет выведено строковое значение, возвращаемое последней инструкцией.

Механизм отложенных вычислений также включает возможность автоматического определения состояний взаимоблокировки. К тому же он прост и понятен.

Теперь, когда вы получили представление о поддержке параллельных вычислений, вы можете по достоинству оценить язык. Давайте на этом завершим третий день, чтобы вы могли на практике опробовать новые знания.

Что мы узнали в третий день

В этом разделе вы познакомились с весьма нетривиальными возможностями Io. Во-первых, мы немного изменили синтаксис языка, добавив в него поддержку литералов хэшей в фигурных скобках.

С этой целью мы добавили оператор в таблицу операторов и связали его с хэш-таблицами. Затем мы сконструировали генератор разметки XML, использующий аналог метода `method_missing` для вывода элементов XML.

Потом мы написали некоторый код, использующий сопрограммы для параллельного выполнения вычислений. Сопрограммы отличаются от примитивов поддержки параллельных вычислений, используемых в таких языках, как Ruby, C и Java, тем, что могут изменять только собственное состояние, благодаря чему многозадачные программы становятся более предсказуемыми и в меньшей степени требуют применения блокировок, которые обычно являются узким местом многопоточных приложений.

Мы посылали асинхронные сообщения, превращая тем самым наши объекты в акторы. При этом нам не пришлось менять синтаксис сообщений. В заключение мы познакомились с отложенными вычислениями и их особенностями в языке Io.

День 3: задания для самостоятельного решения

Практические задания:

- добавьте в программу-генератор разметки XML вывод пробелов для оформления отступов;
- реализуйте поддержку синтаксиса литералов списков в квадратных скобках;
- добавьте в программу-генератор разметки XML обработку атрибутов: если первым аргументом является отображение (в фигурных скобках), программа должна выводить его содержимое как атрибуты, например:

```
book({"author": "Tate"}...) would print <book author="Tate">:
```

3.5. В заключение об Io

Io – замечательный язык, он отлично подходит для изучения принципов программирования на языках, основанных на прототипах. Синтаксис языка удивительно прост, но его семантика обладает невероятной широтой. Языки на основе прототипов обеспечивают инкапсуляцию данных и поведение подобно объектно-ориентированным языкам. Но механизм наследования в них реализуется намного проще. В Io нет ни модулей, ни классов. Каждый объект наследует свойства и методы непосредственно от своего прототипа.

Сильные стороны

Языки на основе прототипов обычно более удобны. Они позволяют изменять любые слоты любых объектов. Io обладает огромной гибкостью, давая возможность изменять его синтаксис по мере необходимости. Как и в случае с языком Ruby, чтобы сделать язык Io таким динамичным, его создателю пришлось пойти на компромиссы, касающиеся потери производительности, по крайней мере в однопоточных приложениях. Мощные, современные библиотеки поддержки параллельных вычислений превращают Io в отличный инструмент параллельной обработки. Рассмотрим поближе наиболее сильные стороны Io.

Размер интерпретатора

Интерпретатор языка Io имеет очень небольшой размер. Большинство промышленных приложений на Io выполняется во встраиваемых системах. И это вполне закономерно, потому что язык очень маленький, мощный и гибкий, а виртуальная машина легко переносится в различные операционные окружения.

Простота

Io имеет особенно компактный синтаксис. Изучить его можно в самые кратчайшие сроки. Как только вы научитесь понимать базовый синтаксис языка, вам останется лишь познакомиться со структурой библиотек языка. Я быстро нашел свой путь к метапрограммированию, буквально в течение первого месяца изучения языка. В языке Ruby то же самое заняло немного больше времени. При программировании на Java я потратил много месяцев, прежде чем я нашел применение приемам метапрограммирования.

Гибкость

В Io используется модель «утиной» типизации и разрешается изменять любые слоты любых объектов в любой момент времени. Такая свобода означает, что вы можете изменить даже самые основные правила языка в угоду своему приложению. Вы легко можете реализовать обработку обращений к несуществующим методам посредством слота `forward`. Вы можете также переопределить ключевые конструкции языка, непосредственно изменяя их слоты. Вы можете даже определять свои синтаксические конструкции.

Параллельные вычисления

В отличие от Java и Ruby, в Io используются более современные механизмы параллельных вычислений. Акторы (actors), отложенные вычисления (futures) и сопрограммы (coroutines) значительно облегчают создание многопоточных приложений, более простых в тестировании и обладающих более высокой производительностью. Io также придает особое значение изменяемым данным и помогает избежать их использования. Наличие всех этих особенностей, реализованных в виде стандартных библиотек, упрощает изучение более надежной модели параллельных вычислений. Позднее, при знакомстве с другими языками, мы еще не раз вернемся к этим понятиям. В частности, с акторами мы еще встретимся при знакомстве с языками Scala, Erlang и Haskell.

Недостатки

Язык Io имеет и свои недостатки. Свобода и гибкость имеют свою цену. Кроме того, так как вокруг Io сложилось не такое многочисленное сообщество, как вокруг других языков, описываемых в этой книге, выбор данного языка для реализации некоторых проектов является довольно рискованным. Давайте поближе рассмотрим проблемы, сопутствующие языку Io.

Синтаксис

В языке Io практически полностью отсутствует синтаксический сахар. Простота синтаксиса – обоюдоострое оружие. С одной стороны ясность синтаксиса облегчает изучение языка Io. Но, как известно, все имеет свою цену. Простота синтаксиса часто не позволяет кратко выразить сложные понятия. Иначе говоря, при чтении программы на языке Io вы легко сможете понять, как она работает, но в то же время вам сложнее будет разобраться в том, что именно программа делает.

Возьмем для сравнения язык Ruby. В первый момент конструкция `array[-1]` на языке Ruby может показаться непонятной, пока вы не узнаете, что под индексом `-1` подразумевается последний элемент массива. Вам также потребуется узнать, что `[]` – это метод, возвращающий значение элемента с указанным индексом. Когда вы изучите эти понятия, то сможете улавливать смысл программного кода с первого взгляда. В Io все с точностью до наоборот. Вам практически не требуется изучать синтаксис языка, чтобы понимать написанный на нем программный код, но вам придется потрудиться, чтобы усвоить некоторые понятия, которые иначе могли бы быть выражены с применением синтаксического сахара.

Соблюдение баланса синтаксического сахара в языке – весьма сложная задача. Стоит добавить слишком много сахара, как язык становится сложным в изучении и запоминании. Если добавить недостаточно сахара, программный код станет менее выразительным и на его отладку может потребоваться затратить гораздо больше энергии. Но как бы то ни было, синтаксис в значительной степени определяется личными предпочтениями создателей. Мац любит много сахара, а Стив – нет.

Сообщество

В настоящее время вокруг языка Io собралось очень небольшое сообщество пользователей. Вы не всегда сможете найти библиотеки для Io, которые давно уже существуют в других языках. Не менее сложно будет найти программистов, которые могли бы оказать вам помощь. Эти проблемы несколько смягчаются наличием хорошего интерфейса с языком C и простым синтаксисом. Хороший программист на JavaScript сможет освоить Io очень быстро. Но наличие слишком маленького сообщества определенно является главным недостатком, сдерживающим развитие любого нового языка. Либо Io получит новые привлекательные особенности, либо он останется малозаметным языком на второстепенных ролях.

Производительность

Обсуждение производительности в отрыве от других проблем, касающихся параллельных вычислений и архитектуры программ, обычно бессмысленно, но я должен отметить, что в Io имеется множество особенностей, замедляющих скорость выполнения однопоточных приложений. Проблема низкой производительности несколько смягчается наличием в Io поддержки параллельных вычислений, тем не менее вы должны постоянно помнить об этом ограничении.

Заключительные замечания

В общем и целом мне понравился язык Io. Простота синтаксиса и маленький объем интерпретатора заинтриговали меня. Мне также кажется, что в основе Io, как и Lisp, лежит доминирующая философия простоты и гибкости. Следуя этой философии, Стиву Декорту удалось создать язык, подобный языку Lisp, но основанный на прототипах. Я думаю, у языка есть потенциал. Как и Ферриса Бьюллера, его ждет яркое и полное приключений будущее.

Глава 4

Prolog

«Салли Диббс, Диббс Салли. 461-0192».

Раймонд

Ах, Prolog! Иногда потрясающе умный, но иногда просто обескураживающий. С его помощью можно получать удивительные ответы, но только если знать, как правильно задавать вопросы. Мне он напоминает Раймонда из фильма «Человек дождя»[1]. Я помню, как Раймонд, главный герой фильма, отбарабанил телефонный номер Салли Диббс, прочитав телефонную книгу прошлой ночью, не задумываясь о том, нужно ли это кому-нибудь. В отношении Раймонда, как и в отношении Prolog, я часто спрашиваю сам себя: «Как он узнал об этом?» – и: «Почему он этого не знает?» Он может фонтанировать знаниями, если задавать вопросы правильно.

Язык Prolog радикально отличается от языков программирования, рассматривавшихся выше. Оба языка, Io и Ruby, называют *императивными языками*. Программа на императивном языке подобна рецепту. В ней вы точно описываете, какие действия должен выполнить компьютер. Высокоуровневые императивные языки могут позволять объединять длинные инструкции в одну, но, в общем и целом, на таких языках вы фактически объединяете список ингредиентов и пошаговые инструкции по приготовлению желаемого блюда.

Мне потребовалось несколько недель экспериментов с языком Prolog, прежде чем я смог приступить к этой главе. Для знакомства с языком я использовал несколько руководств, включая руководство Дж. Р. Фишера (J. R. Fisher)[2], где нашел множество примеров, и руководство А. Эби (A. Aaby)[3], которое помогло мне усвоить структуру языка и его терминологию.

---

[1] «Rain Man». DVD. Режиссер Барри Левинсон (Barry Levinson). 1988; Лос-Анджелес, Калифорния: MGM, 2000. (http://ru.wikipedia.org/wiki/Человек_дождя. – *Прим. перев.*)

[2] http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html.

[3] http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html.

Prolog – декларативный язык. Вы обозначаете некоторые факты и выводы и позволяете ему сделать выводы за вас. Он напоминает хорошего пекаря. Вы описываете, какой пирог хотелось бы получить, а пекарь сам подбирает ингредиенты и выпекает пирог. На языке Prolog не требуется знать, как решить ту или иную задачу. Компьютер сам будет принимать необходимые решения.

Выполнив поиск в Интернете, вы легко найдете пример реализацию игры в судоку, занимающую не более 20 строк кода, решение задачи сборки кубика Рубика и множества других известных головоломок, таких как Ханойские башни (кстати, занимает всего десяток строк кода). Prolog был одним из первых успешных языков логического программирования. Вы записываете логические правила, а Prolog определяет, являются ли они истинными. Правила могут быть неполными, и Prolog попытается восполнить эту неполноту.

4.1. О языке Prolog

Язык Prolog, созданный в 1972 году Аланом Кулмероэ (Alain Colmerauer) и Филиппом Расселом (Phillipe Roussel), является языком логического программирования и получил особую популярность в решении задач обработки естественных языков. В настоящее время этот почтенный язык является основой для решения самых разнообразных задач, от планирования до экспертных систем. Этот язык, основанный на системе правил, можно использовать для выражения логических утверждений и получения ответов на вопросы. Подобно языку SQL, Prolog работает с базами данных, но роль данных в них играют логические утверждения и отношения между ними. Как и SQL, Prolog делится на две части: одна служит для определения данных, а другая – для выполнения запросов. Данные в языке Prolog имеют форму логических правил. Ниже перечислены основные строительные блоки языка.

- *Факты*. Факт – это базовое утверждение о некотором мире. (Ребенок – как поросенок; поросенок любит грязь.)
- *Правила*. Правила играют роль суждений о фактах в данном мире. (Животное обожает грязь, если это – поросенок.)
- *Запросы*. Запрос – это вопрос, касающийся данного мира. (Любит ли ребенок грязь?)

Факты и правила составляют *базу знаний*. Компилятор Prolog компилирует базу знаний в форму, позволяющую эффективно выполнять запросы. По мере знакомства с примерами вы будете использо-

вать Prolog для накопления собственной базы знаний. Затем, опять же с помощью Prolog, вы будете обращаться к этой базе знаний, чтобы получить ответы на свои вопросы.

Но хватит теории. Давайте перейдем к практике.

4.2. День 1: Отличный водитель

В фильме «Человек дождя» Раймонд говорит своему брату, что он – отличный водитель, подразумевая, что ему разрешалось водить машину медленно по дорожке. Он умеет использовать все органы управления – рулевое колесо, тормоза, педаль акселератора, – хотя и с определенными ограничениями. На сегодня перед нами стоит такая же ограниченная задача. Мы обозначим на языке Prolog некоторые факты, определим некоторые правила и попробуем выполнить простейшие запросы. Синтаксис языка Prolog, как и языка Io, чрезвычайно прост. Вы сможете изучить его очень быстро. Но самое интересное начнется потом, когда вы научитесь комбинировать понятия. Если вы впервые знакомитесь с языком Prolog, предупреждаю, что вам придется изменить свой образ мышления, иначе вы потерпите неудачу. Однако оставим пока детальное обсуждение языка.

Будем действовать по порядку. Сначала нужно установить язык. Работая над этой книгой, я использовал версию GNU Prolog 1.3.1. Будьте внимательны, диалекты могут иметь существенные отличия. Я буду стараться не использовать специфические особенности, характерные для какой-то одной версии, но если вы установите другую версию Prolog, вам придется самим разобраться с различиями между диалектами. Далее описываются приемы использования языка, общие для любых версий.

Факты

В некоторых языках регистр первого символа в идентификаторах не имеет никакого значения, и выбор регистра определяется предпочтениями программиста, но в Prolog регистр первого символа играет важную роль. Если идентификатор начинается с буквы нижнего регистра, он интерпретируется как *атом* (atom) – фиксированное значение, подобное символу (ключевому слову) в языке Ruby. Если идентификатор начинается с буквы верхнего регистра или символа подчеркивания, он интерпретируется как *переменная*. Значения переменных могут изменяться; значения атомов – нет. Давайте созда-

дим простую базу знаний, содержащую несколько фактов. Откройте текстовый редактор и введите в него следующий текст:

prolog/friends.pl
http://media.pragprog.com/titles/btlang/code/prolog/friends.pl

```
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).

friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

Этот файл содержит базу знаний с фактами и правилами. Первые три строки – это факты, а последняя строка – правило. Факты отражают непосредственные наблюдения, сделанные в нашем мире. Правила являются логическими выводами о нашем мире. Пока забудем о последней строке и сосредоточимся на первых трех. Каждая из тих строк является фактом. `wallace`, `grommit` и `wendolene` – это атомы. Их можно читать как: «wallace likes cheese» (Уоллес любит сыр), «grommit likes cheese» (Громит любит сыр) и «wendolene likes sheep» (Венделин любит овец)[1]. Давайте задействуем эти факты.

Запустите интерпретатор Prolog. Если вы используете GNU Prolog, введите команду `gprolog`. Затем загрузите файл следующей командой:

```
| ?- ['friends.pl'].
compiling /Users/batate/prag/Book/code/prolog/friends.pl for byte code...
/Users/batate/prag/Book/code/prolog/friends.pl compiled, 4 lines read-
997 bytes written, 11 ms

yes

| ?-
```

Если интерпретатор Prolog не ожидает промежуточного ввода, он выводит ответ `yes` (да) или `no` (нет). В данном случае файл загружен благополучно, поэтому интерпретатор вывел `yes`. Теперь можно попробовать задать несколько вопросов. Наиболее простыми являются вопросы о фактах, требующие односложного ответа – «да» или «нет». Вот некоторые из них:

```
| ?- likes(wallace, sheep).

no
```

---

[1] Пример создан по мотивам мультфильма «Невероятные приключения Уоллеса и Громита: Стрижка под ноль». http://ru.wikipedia.org/wiki/Невероятные_приключения_Уоллеса_и_Громита:_Стрижка_«под_ноль» – *Прим. перев.*

```
| ?- likes(grommit, cheese).

yes
```

Эти вопросы просты и понятны. Любит ли Уоллес овец? (Нет.) Любит ли Громит сыр? (Да.) Пока ничего интересного: Prolog просто повторяет факты. Однако дело приобретает совсем другой оборот, как только вы начинаете подключать логику. Давайте познакомимся с выводами.

Простые выводы и переменные

Попробуем применить правило `friend` (друг):

```
| ?- friend(wallace, wallace).

no
```

Итак, Prolog просматривает правила, которые мы передали ему, и отвечает `yes` или `no`. Однако в действительности здесь кроется нечто большее, чем кажется на первый взгляд. Рассмотрим правило `friend`:

Говоря человеческим языком, чтобы субъект `X` мог считаться другом (friend) субъекта `Y`, `X` и `Y` должны быть разными субъектами. Взгляните на первую часть справа от `:-`, которая называется *подцелью* (subgoal). Последовательность символов `\+` представляет логическое отрицание, то есть `\+(X = Y)` означает «X не равно Y».

Попробуем выполнить несколько запросов:

```
| ?- friend(grommit, wallace).

yes
| ?- friend(wallace, grommit).

yes
```

Говоря человеческим языком, `X` может считаться другом (friend) `Y`, если известно, что субъект `X` любит нечто `Z` и субъект `Y` любит то же самое нечто `Z`. Оба героя, Уоллес и Громит, любят сыр, поэтому в ответ на эти запросы возвращается утвердительный ответ.

Продолжим исследование кода. В этих запросах `X` не равно `Y`, то есть первая подцель доказана. Далее запрос проверит вторую и третью подцели, `likes(X, Z)` и `likes(Y, Z)`. Громит и Уоллес оба любят сыр, то есть вторая и третья подцели также оказываются доказанными. Попробуем выполнить другой запрос:

```
| ?- friend(wendolene, grommit).

no
```

В данном случае Prolog пришлось опробовать несколько возможных значений для X, Y и Z:

- wendolene, grommit и cheese;
- wendolene, grommit и sheep.

Ни одна из комбинаций не удовлетворяет вторую и третью подцели одновременно, когда Венделина любит Z и Громит любит Z. Поэтому логический механизм ответил no – они не являются друзьями.

Давайте формализуем терминологию. Итак, строка...

```
friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

...определяет правило на языке Prolog с тремя переменными – X, Y и Z. Мы будем называть это правило friend/2, сокращенно от: «правило friend с двумя параметрами». Это правило включает три подцели, разделенные запятыми. Чтобы запрос к этому правилу вернул утвердительный ответ, все три подцели также должны вернуть утвердительный ответ. То есть наше правило означает следующее: «X может считаться другом Y, если X и Y не одно и то же, и оба, X и Y, любят Z».

Восполнение неполноты

Мы использовали интерпретатор Prolog, чтобы получить однослож-ные ответы на свои вопросы – yes или no, однако он способен на большее. В этом разделе мы задействуем логический механизм для поиска всех возможных совпадений. Для этого нужно указать *переменную* в запросе.

Взгляните на следующую базу знаний:

prolog/food.pl
http://media.pragprog.com/titles/btlang/code/prolog/food.pl

```
food_type(velveeta, cheese).
food_type(ritz, cracker).
food_type(spam, meat).
food_type(sausage, meat).
food_type(jolt, soda).
food_type(twinkie, dessert).

flavor(sweet, dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).
```

В нашем распоряжении имеется несколько фактов. Некоторые из них, такие как food_type(velveeta, cheese), определяют конкретный

тип продукта питания. Другие, такие как `flavor(sweet, dessert)`, определяют вкусовые характеристики. Наконец, у нас имеется правило с именем `food_flavor`, которое позволяет определить вкус (`flavor`) продукта. Продукт `X` имеет вкус `Y`, если он относится к типу `Z`, и этот тип `Z` имеет этот же вкус. Скомпилируем ее:

```
| ?- ['code/prolog/food.pl'].
compiling /Users/batate/prag/Book/code/prolog/food.pl for byte code...
/Users/batate/prag/Book/code/prolog/food.pl compiled,
12 lines read - 1557 bytes written, 15 ms

(1 ms) yes
```

и зададим несколько вопросов:

```
| ?- food_type(What, meat).

What = spam ?;

What = sausage ?;

no
```

Здесь мы наблюдаем кое-что интересное! Мы попросили Prolog «найти значение для переменной `What`, удовлетворяющее условию `food_type(What, meat)`». Prolog нашел один продукт, `spam` (колбасный фарш). Введя точку с запятой (`;`), мы тем самым предложили интерпретатору Prolog найти еще один продукт, и он вернул нам `sausage` (сосиски). Найти эти значения не составляло труда, так как запрос опирался на основные факты. Когда мы снова предложили интерпретатору продолжить поиск, он вывел `no`. Такое поведение может показаться немного противоречивым. Для удобства, если Prolog в состоянии определить отсутствие других альтернатив, он сразу выведет `yes`. Если Prolog не сможет сразу же проверить наличие других вариантов, отвечающих условиям запроса, без выполнения дополнительных вычислений, он предложит ввести символ продолжения и в конце выведет `no`. Это действительно удобная особенность. Если интерпретатор Prolog сможет немедленно дать всю информацию, он сделает это. Попробуем выполнить еще пару запросов:

```
| ?- food_flavor(sausage, sweet).

no
| ?- flavor(sweet, What).

What = dessert ?;
```

```
What = soda

yes
```

Как видите, сосиски (sausage) не обладают сладким (sweet) вкусом. А какие продукты имеют сладкий вкус? Десерт (dessert) и газировка (soda). Однако все это – факты. А способен ли Prolog самостоятельно делать выводы? Давайте посмотрим:

```
| ?- food_flavor(What, savory).

What = velveeta ?;

What = spam ?;

What = sausage ? ;

no
```

Напомню, что food_flavor(X, Y) – это правило, а не факт. В этом примере мы попросили Prolog найти все продукты «имеющие острый (savory) вкус». Чтобы сделать вывод, интерпретатору пришлось объединить известные факты о продуктах, типах и их вкусовых характеристиках. Логический механизм выполнил перебор возможных комбинаций и отобрал те, что соответствуют правилу.

Раскрашивание карты

Давайте попробуем использовать ту же идею для раскрашивания карты. Для большей наглядности рассмотрим пример из реальной жизни. Итак, нам требуется раскрасить карту юго-востока Соединенных Штатов. На карте нанесены границы штатов, как показано на рис. 4.1, и нам нужно раскрасить их так, чтобы участки одного цвета не имели общих границ.

Вот как выглядит простая база знаний, отражающая эти простые факты:

prolog/map.pl
http://media.pragprog.com/titles/btlang/code/prolog/map.pl

```
different(red, green). different(red, blue).
different(green, red). different(green, blue).
different(blue, red). different(blue, green).

coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
```

Рис. 4.1 ❖ Карта юго-востока Соединенных Штатов

```
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).
```

У нас имеются три цвета. Мы сообщаем интерпретатору Prolog наборы различающихся цветов, которые он должен использовать для раскрашивания карты. Далее следует правило. Правило `coloring` сообщает, какие штаты имеют общие границы. Вот и все. Попробуем раскрасить карту:

```
| ?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).

Alabama = blue
Florida = green
Georgia = red
Mississippi = red
Tennessee = green?
```

И правда, в данном случае есть вариант раскрашивания карты с пятью штатами тремя цветами. Вы можете получить другие возможные комбинации, нажав клавишу `a`. С помощью всего десятка строк

кода мы получили желаемый результат. Логика поиска до смешного проста – даже ребенок справится с этой задачей. Но рано или поздно наступит момент, когда вы зададитесь вопросом...

А где сама программа?

У нас есть алгоритм! Попробуйте решить эту задачу на любом процедурном языке по своему выбору. Получится ли ваше решение простым для понимания? Подумайте о том, что необходимо сделать, чтобы решить эту сложную логическую проблему на таком языке, как Ruby или Io. Ниже приводится одна из возможных последовательностей действий:

- выработать и организовать логику решения;
- выразить логику в виде программы;
- найти все возможные решения;
- оценить найденные решения.

И вы вынуждены будете писать эту программу снова и снова. Prolog позволяет выразить логику в виде фактов и выводов и затем задавать интересующие вас вопросы. Он не требует описывать пошаговые рецепты решения задач. Prolog не является языком описания алгоритмов решения логических задач. Prolog – это язык описания вашего мира и представления логических задач, которые ваш компьютер может попытаться решить.

Так давайте заставим компьютер работать!

Унификация, часть 1

Итак, пришло время приостановиться и вновь окунуться в теорию. Давайте коснемся проблемы унификации. В некоторых языках программирования присутствует операция присваивания значения переменной. В Java или Ruby, например, инструкция `x = 10` присвоит значение `10` переменной `x`. Такое объединение двух структур имеет своей целью сделать их идентичными. Рассмотрим следующую базу данных:

prolog/ohmy.pl
http://media.pragprog.com/titles/btlang/code/prolog/ohmy.pl

```
cat(lion).
cat(tiger).

dorothy(X, Y, Z) :- X = lion, Y = tiger, Z = bear.
twin_cats(X, Y) :- cat(X), cat(Y).
```

В данном примере оператор = означает *унификацию* (unify), проверяет совпадение обеих сторон. У нас имеются два факта: львы (lions) являются кошками (cats) и тигры (tigers) являются кошками. В базе знаний присутствуют также два правила. В правиле `dorothy/3` параметрами `X`, `Y` и `Z` являются `lion` (лев), `tiger` (тигр) и `bear` (медведь) соответственно. В правиле `twin_cats/2` оба параметра, `X` и `Y`, являются кошками (`cat`). С помощью этой базы знаний мы можем пролить некоторый свет на унификацию.

Для начала попробуем воспользоваться первым правилом. Я скомпилировал базу знаний и затем выполнил простой запрос без параметров:

```
| ?- dorothy(lion, tiger, bear).

yes
```

Напомню, что унификация означает: «найти такие значения, которые обеспечат соответствие обеих сторон». Справа Prolog связывает `X`, `Y` и `Z` со значениями `lion`, `tiger` и `bear`. Они соответствуют значениям слева, то есть унификация увенчалась успехом. В результате Prolog ответил `yes`. Этот случай достаточно прост, но мы можем немного расширить его. Унификация способна работать с обеих сторон. Давайте попробуем:

```
| ?- dorothy(One, Two, Three).

One = lion
Three = bear
Two = tiger

yes
```

Этот пример имеет еще один уровень косвенности. В подцелях Prolog унифицирует (сопоставляет) `X`, `Y` и `Z` со значениями `lion`, `tiger` и `bear`. Слева Prolog унифицирует `X`, `Y` и `Z` со значениями `One`, `Two` и `Three` и затем выводит результат.

Теперь перейдем к последнему правилу, `twin_cats/2`. Это правило утверждает, что `twin_cats(X, Y)` истинно, если можно доказать, что оба параметра, `X` и `Y`, являются кошками. Попробуем:

```
| ?- twin_cats(One, Two).

One = lion
Two = lion?
```

В первой попытке Prolog сообщает, что оба значения, `lion` и `lion`, являются кошками. Давайте посмотрим, как он пришел к этому выводу.

1. Мы выполнили запрос `twin_cats(One, Two)`. Prolog связал `One` с `X` и `Two` с `Y`. Чтобы выполнить запрос, Prolog должен проверить подцели.
2. Первая подцель `cat(X)`.
3. В базе знаний имеются два факта, удовлетворяющие этой подцели, `cat(lion)` и `cat(tiger)`. Prolog опробует первый факт, связывает `X` с `lion` и переходит ко второй подцели.
4. Теперь Prolog должен проверить подцель `cat(Y)`. В этом случае он идет тем же путем, что и при проверке первой подцели, и выбирает `lion`.
5. В результате обе подцели удовлетворены и проверка соответствия правилу оканчивается успехом. Prolog сообщает значения для `One` и `Two`, при которых проверка правила увенчалась успехом, и в конце выводит `yes`.

Итак, мы получили первое решение. Иногда одного решения бывает вполне достаточно. Иногда бывает желательно получить несколько решений. Мы можем теперь последовательно получить другие решения, вводя символ точки с запятой (`;`), или вывести сразу все решения, введя символ `a`.

```
Two = lion ?a

One = lion
Two = tiger

One = tiger
Two = lion

One = tiger
Two = tiger

(1 ms) yes
```

Обратите внимание, что Prolog вывел список всех возможных комбинаций `X` и `Y`, учитывая информацию, имеющуюся в подцелях и фактах. Как будет показано далее, унификация позволяет также выполнять более сложные сопоставления, опираясь на структуру данных. Впрочем, для первого дня достаточно. Обсуждение более сложных тем мы отложим на второй день.

Практическое применение языка Prolog

Довольно странно видеть «программу» в таком представлении. Программируя на языке Prolog, нечасто можно увидеть подробный рецепт. Обычно программа на Prolog – это общее описание пирога, который вы вынете из духовки в конце. Приступив к изучению Prolog, я получил возможность познакомиться с человеком, использующим этот язык в своей повседневной практике. Я побеседовал Брайаном Тарбоксом (Brian Tarbox), использующим этот язык логического программирования в своем проекте исследования дельфинов.

Интервью с Брайаном Тарбоксом, исследователем дельфинов

Брюс: Расскажите, как вы начинали осваивать Prolog?

Брайан: Изучать Prolog я начал в конце 1980-х, когда учился в аспирантуре Гавайского университета в Маноа. Я тогда работал в лаборатории морских млекопитающих в городе Кевало Базин (Kewalo Basin) и занимался исследованиями когнитивных способностей дельфинов-белобочек. В то время я обратил внимание на большое количество дискуссий между сотрудниками лаборатории по поводу того, как мыслят дельфины. Мы работали в основном с дельфином по кличке Акиакамаи (Akeakamai), или просто Аки. Чаще всего споры начинались со слов: «Аки эту ситуацию видит, скорее всего, так...».

Я подумал, что было бы неплохо, если бы моя кандидатская диссертация содержала действующую модель, соответствующую нашим представлениям о том, как Аки понимает окружающий мир. Хотя бы в той части, где мы проводили исследования. Если бы с помощью действующей модели нам удалось предсказать фактическое поведение Аки, мы получили бы некоторую точку опоры для своих теоретических выкладок.

Prolog – замечательный язык, но его нельзя использовать бездумно, иначе результаты могут оказаться самыми обескураживающими. Я вспоминаю один из своих первых экспериментов с языком Prolog, когда среди прочих написал инструкцию `x = x + 1`. В ответ интерпретатор сообщил мне «no». В моем понимании язык не может просто так сказать «no». Он может дать ошибочный ответ или потерпеть неудачу при попытке скомпилировать исходный код, но я никогда не встречался с языком, который просто отвечал бы мне. Поэтому я обратился в службу поддержки и сказал, что Prolog ответил мне «no», когда я попытался изменить значение переменной. В ответ меня спросили: «А зачем вам понадобилось изменить значение переменной?» В мое

понимание тогда не укладывалось, что могут существовать языки, не позволяющие изменять переменные. В один прекрасный момент, разобравшись с особенностями языка Prolog, вы начинаете понимать, что переменные в этом языке либо имеют определенное значение, либо просто не существуют, но в то время меня это выбило из колеи.

Брюс: Для каких целей вы используете Prolog?

Брайан: Мною были разработаны две основные системы: система моделирования поведения дельфина и система планирования экспериментов. Лаборатория должна была ежедневно проводить четыре эксперимента с каждым из четырех дельфинов. Важно понимать, что на исследование дельфинов выделяются не очень большие деньги. Каждый дельфин участвует в четырех разных экспериментах, и для проведения каждого эксперимента требуются разные специалисты. На некоторые вакансии, такие как дрессировщики дельфинов, может быть принято не более двух-трех человек. Для других видов работ, таких как регистрация данных, требуется еще несколько человек, которых, кстати, нужно обучать. Для проведения большинства экспериментов требуется от шести до десяти специалистов. В нашем распоряжении было несколько аспирантов, студентов и добровольцев-экологов. У каждого из них был свой распорядок дня и свой уровень знаний. Поэтому для составления оптимального графика проведения экспериментов с учетом всех нестыковок у нас существовал отдельный человек, занятый полный рабочий день.

Я решил попробовать написать систему планирования на языке Prolog. Как оказалось, язык словно специально создавался для решения таких задач. Я собрал множество фактов, описывающих навыки всех сотрудников, расписание каждого из них и необходимость привлечения к каждому эксперименту. После этого мне оставалось лишь сказать интерпретатору Prolog «составить план такого-то эксперимента». Для всех задач, перечисленных в эксперименте, интерпретатор отыскивал подходящих специалистов с требуемыми навыками и включал их в список участников. Составление плана продолжалось вплоть до удовлетворения всех условий или обнаружения невозможности найти оптимальное решение. В последнем случае интерпретатор возвращался назад и пробовал другую комбинацию. В конечном итоге он либо находил оптимальный график, либо сообщал, что эксперимент провести невозможно из-за имеющихся ограничений.

Брюс: Можете ли вы привести примеры фактов, правил или утверждений, касающихся дельфинов, которые были бы интересны нашим читателям?

Брайан: Однажды сложилась такая ситуация, когда система моделирования поведения дельфинов помогла нам понять действия Аки. Аки умел отвечать на «предложения», подаваемые ему на языке жестов, такие как «прыгнуть через обруч» или «коснуться хвостом мяча справа». Мы подавали ему команды, и он выполнял их.

Частью моих исследований была попытка научить его новым словам, таким как «не». В этом контексте команда «коснуться не мяча» означала бы «коснуться чего угодно, только не мяча». Для Аки это оказалось сложной задачей, тем не менее определенный прогресс все же наблюдался. Однако в какой-то момент Аки начал просто погружаться в глубину, когда мы подавали ему такую команду. Мы не понимали, в чем причина. Эта ситуация нас обескураживала, потому что мы не могли спросить у дельфина, почему он так поступает. Тогда мы поставили задачу перед системой моделирования и получили любопытный результат. Дельфины – очень умные животные, но в большинстве случаев они пытаются найти самое простое решение задачи. Мы сформулировали аналогичную эвристику в системе моделирования. Оказалось, что язык жестов, который понимает Аки, включает «слово», обозначающее одно из окон в бассейне. Большинство дрессировщиков забывают об этом слове, так как оно очень редко используется. Система моделирования обнаружила, что слово «окно» отлично подходит под условие «не мяч». Оно так же отлично подходит под условие «не обруч», «не бочка» и «не диск». Мы старались препятствовать выработке шаблонных решений, изменяя в каждом эксперименте набор объектов, находящихся в бассейне, но, как вы понимаете, мы не могли убрать окно. Как оказывается, получив команду, Аки погружался на дно бассейна и касался окна, которого я не мог видеть!

Брюс: Что больше всего вам нравится в языке Prolog?

Брайан: Меня очень привлекает модель декларативного программирования. Вообще говоря, если вы можете описать задачу, значит, вы наполовину уже решили ее. При использовании большинства языков мне приходилось спорить с компьютером, высказывая в какой-то момент: «Ты же знаешь, что я имею в виду, так сделай это!» Символичными в этом смысле являются ошибки компилятора C/C++, такие как «отсутствует точка с запятой». Если компилятор понимает, что здесь должна быть точка с запятой, так почему бы просто не вставить ее? В языке Prolog, напротив, от меня требуется всего лишь описать задачу, например: «Найди день, удовлетворяющий таким-то условиям», – и все.

Брюс: С какой самой большой проблемой вам пришлось столкнуться?

Брайан: С бескомпромиссностью Prolog при решении задач, по крайней мере тех, которые мне приходилось решать. Система планирования экспериментов могла по полчаса перебирать различные варианты и в конце выдать готовый план на день или просто вывести «no». Под «no» в данном случае предполагается, что системе не удалось найти вариант, полностью удовлетворяющий всем условиям. Однако она не сообщала, какие именно условия не удается удовлетворить, и не предлагала никаких частичных решений.

Как видите, логическое программирование является очень мощной концепцией. От вас не требуется предоставить пошаговое решение. Вы должны лишь описать задачу на логическом языке. Начните с фактов и выводов и позвольте интерпретатору Prolog сделать все остальное. Программы на языке Prolog находятся на самом высоком уровне абстракции. Системы планирования и моделирования, упоминавшиеся выше, являются отличными примерами решения задач с помощью Prolog.

Что мы узнали в первый день

Сегодня мы познакомились с основными строительными блоками языка Prolog. Вместо составления подробных инструкций решения задачи мы лишь описываем имеющиеся у нас знания, используя голую логику, а Prolog берет на себя тяжелый труд сплести эти знания в единое полотно и найти решение. Мы создаем базу знаний и затем просто выполняем запросы к ней.

Создав несколько баз знаний, мы скомпилировали их и затем просто пытались выполнять запросы. Мы опробовали запросы двух видов. Запросы первого вида определяли факты, а Prolog сообщал нам, являются эти факты истинными или ложными. Запросы второго вида включали одну или более переменных. Обрабатывая эти запросы, Prolog просматривал все возможные комбинации фактов, при которых ответ на запрос получался бы истинным.

Мы также узнали, что Prolog обрабатывает правила, просматривая встречающиеся в них утверждения. При этом Prolog пытается удовлетворить все подцели во всех утверждениях, опробуя различные комбинации переменных. Все программы на Prolog работают именно так.

В разделах, следующих ниже, мы займемся исследованием еще более сложных выводов. Кроме того, мы познакомимся с математиче-

скими операциями и более сложными структурами данных, такими как списки, а также с различными стратегиями итераций по элементам списков.

День 1: задания для самостоятельного решения

Найдите:

- какие-нибудь бесплатные руководства по языку Prolog;
- форум поддержки (можно несколько);
- электронное руководство по вашей версии Prolog.

Практические задания:

- создайте простую базу знаний, где были бы перечислены ваши любимые книги и авторы;
- найдите все книги в своей базе знаний, написанные одним автором;
- создайте базу знаний, где были бы перечислены музыканты, инструменты, на которых они играют, и жанры, в которых они выступают;
- найдите всех музыкантов, играющих на гитаре.

4.3. День 2: Пятнадцать минут до «Народного суда»

Телепередача «Народный суд» была навязчивой идеей главного персонажа из фильма «Человек дождя». Подобно большинству аутистов, Раймонд зациклен на всем, что ему знакомо, поэтому у него был свой «пунктик» – просмотр телепередачи «Народный суд». Приступая к изучению этого загадочного языка, вы должны быть готовы, что у вас так же внезапно что-то будет «щелкать» в голове. Возможно, сейчас у кого-то из вас уже щелкнула мысль, что вы на правильном пути, но если это не так – наберитесь терпения. Сегодня у нас определенно не более «пятнадцати минут до Народного суда». Держитесь крепче. Мы пополним свой арсенал новыми инструментами, познакомимся с рекурсией, математическими операциями и списками. Итак, вперед!

Рекурсия

Ruby и Io – это императивные языки программирования. На этих языках приходится обстоятельно описывать каждый свой шаг. Prolog, напротив, первый из декларативных языков, с которыми мы

познакомимся в этой книге. Имея дело с коллекциями, такими как списки или деревья, программисты часто предпочитают использовать рекурсию вместо итераций. Мы тоже познакомимся с рекурсией и будем использовать ее для решения некоторых задач, основанных на простых выводах, а затем попробуем применить тот же прием для работы со списками и выполнения математических вычислений.

Взгляните на следующую базу знаний. Она содержит сведения об обширном генеалогическом дереве Уолтонов, персонажей телесериала, выпущенного в 1963 году. В ней отражены отношения отца, из которых выводятся отношения для предка. Так как под предком может подразумеваться отец, дед или прадед, нам необходимы вложенные правила или возможность выполнения итераций. Поскольку мы имеем дело с декларативным языком, используем прием вложения. Одно из утверждений в правиле `ancestor` будет использовать `ancestor`. В этом случае `ancestor(Z, Y)` является рекурсивной подцелью. Вот эта база знаний:

prolog/family.pl
http://media.pragprog.com/titles/btlang/code/prolog/family.pl

```
father(zeb,          john_boy_sr).
father(john_boy_sr, john_boy_jr).

ancestor(X, Y) :-
    father(X, Y).
ancestor(X, Y) :-
    father(X, Z), ancestor(Z, Y).
```

`father` – это базовое множество фактов, обеспечивающих поддержку рекурсивной подцели. Правило `ancestor/2` содержит два утверждения. Когда правило состоит из нескольких утверждений, достаточно выполнения только одного из них, чтобы все правило считалось выполненным. Запятые между подцелями можно рассматривать как оператор `and`, а точки между утверждениями – как оператор `or`. Первое утверждение гласит: «`X` является предком (`ancestor`) для `Y`, если `X` является отцом (`father`) для `Y`». То есть имеется прямая связь. Опробовать это правило можно следующим образом:

```
| ?- ancestor(john_boy_sr, john_boy_jr).

true ?

no
```

Prolog сообщил `true`, то есть `john_boy_sr` действительно является предком для `john_boy_jr`. Это первое утверждение зависит от факта.

Второе утверждение более сложное: `ancestor(X, Y) :- father(X, Z), ancestor(Z, Y)`. Оно гласит: «`X` является предком (`ancestor`) для `Y`, если можно доказать, что `X` является отцом (`father`) для `Z`, а `Z` является предком (`ancestor`) для `Y`».

Уф-ф. Давайте попробуем задействовать второе утверждение:

```
| ?- ancestor(zeb, john_boy_jr).
true?
```

Да, `zeb` является предком для `john_boy_jr`. Как обычно, в запрос можно подставить переменные:

```
| ?- ancestor(zeb, Who).

Who = john_boy_sr ? a

Who = john_boy_jr

no
```

Как видите, `zeb` является предком и для `john_boy_jr`, и для `john_boy_sr`. Предикат `ancestor` способен действовать и в обратном направлении:

```
| ?- ancestor(Who, john_boy_jr).

Who = john_boy_sr ? a

Who = zeb

(1 ms) no
```

Это замечательная особенность, потому что она позволяет использовать данное правило в двух целях: для поиска предков и для поиска потомков.

Но будьте внимательны, используя рекурсивные подцели, потому что каждая рекурсивная подцель занимает некоторое пространство на стеке, которое не бесконечно. В декларативных языках эта проблема часто решается путем оптимизации *хвостовой рекурсии* (tail recursion). Если есть возможность поместить рекурсивную подцель в конец правила, Prolog сможет оптимизировать рекурсивные вызовы и тем самым избежать исчерпания пространства, выделенного для стека. В нашем случае рекурсивная подцель `ancestor(Z, Y)` находится в конце правила, следовательно, нам не грозит проблема исчерпания памяти. Если ваша программа на Prolog начнет завершаться аварийно

из-за нехватки пространства на стеке, знайте, что настал момент попытаться оптимизировать рекурсивные правила.

А теперь перейдем к знакомству со списками и кортежами.

Списки и кортежи

Списки и кортежи играют важную роль в Prolog. Список можно определить как `[1, 2, 3]`, а кортеж – как `(1, 2, 3)`. Списки – это контейнеры переменной длины, а кортежи – контейнеры фиксированной длины. Списки и кортежи раскрываются во всей своей широте, если рассматривать их в терминах унификации.

Унификация, часть 2

Напомню, что когда Prolog пытается унифицировать переменные, он старается добиться совпадения левой и правой частей. Два кортежа могут совпадать, если они имеют одинаковое число элементов, и каждый элемент может быть унифицирован. Рассмотрим пару примеров:

```
| ?- (1, 2, 3) = (1, 2, 3).

yes
| ?- (1, 2, 3) = (1, 2, 3, 4).

no
| ?- (1, 2, 3) = (3, 2, 1).

no
```

Два кортежа считаются унифицируемыми, если все их элементы являются унифицируемыми. Первые два кортежа в примере точно соответствуют друг другу, вторые два кортежа имеют разное число элементов, и третьи два кортежа имеют разные элементы в одинаковых позициях. Теперь введем несколько переменных:

```
| ?- (A, B, C) = (1, 2, 3).

A =1
B =2
C = 3

yes
| ?- (1, 2, 3) = (A, B, C).

A =1
B =2
C = 3

yes
```

```
| ?- (A, 2, C) = (1, B, 3).

A =1
B =2
C = 3

yes
```

В действительности не важно, с какой стороны присутствуют переменные. Они считаются унифицируемыми, если Prolog в состоянии обеспечить совпадение элементов слева и справа. Теперь обратимся к спискам. Списки могут действовать подобно кортежам:

```
| ?- [1, 2, 3] = [1, 2, 3].

yes
| ?- [1, 2, 3] = [X, Y, Z].

X =1
Y =2
Z = 3

yes
| ?- [2, 2, 3] = [X, X, Z].

X =2
Z = 3

yes
| ?- [1, 2, 3] = [X, X, Z].

no
| ?- [] = [].
```

Последние два примера особенно интересны. Списки `[X, X, Z]` и `[2, 2, 3]` унифицируются по той причине, что интерпретатору Prolog удалось добиться соответствия `X = 2`. Унификация `[1, 2, 3] = [X, X, Z]` потерпела неудачу, потому что переменная `X` использовалась для унификации с первой и второй позициями, значения которых отличаются. Списки обладают возможностями, которые не поддерживаются кортежами. Списки можно разложить с помощью конструкции `[Head|Tail]`. При унификации списка с помощью такой конструкции элемент `Head` будет связан с первым элементом списка, а `Tail` – с оставшейся частью списка, как показано ниже:

```
| ?- [a, b, c] = [Head|Tail].

Head =a
```

```
Tail = [b,c]

yes
```

Конструкция `[Head|Tail]` не унифицируется только с пустым списком – списки с одним элементом прекрасно унифицируются:

```
| ?- [] = [Head|Tail].
no
| ?- [a] = [Head|Tail].

Head =a
Tail = []

yes
```

Можно использовать и более сложные комбинации:

```
| ?- [a, b, c] = [a|Tail].

Tail = [b,c]

(1 ms) yes
```

Prolog обнаружит совпадение элемента `a` и унифицирует остальную часть списка с переменной `Tail`. Остаток «хвоста» этого списка, в свою очередь, можно разбить на «голову» и «хвост»:

```
| ?- [a, b, c] = [a|[Head|Tail]].

Head =b
Tail = [c]

yes
```

Или извлечь третий элемент:

```
| ?- [a, b, c, d, e] = [_, _|[Head|_]].

Head = c

yes
```

Символ подчеркивания (_) играет роль шаблонного символа и может унифицироваться с чем угодно. В общем случае он интерпретируется как «любое значение, находящееся в этой позиции». В примере выше мы предложили интерпретатору Prolog пропустить первые два элемента и разбить остаток на «голову» и «хвост». В переменную

Head был записан третий элемент, а заключительный символ подчеркивания (_) поглотил «хвост» – остаток исходного списка.

Этих примеров должно быть достаточно, чтобы уловить основную идею. Унификация – мощный инструмент, а применение к спискам и кортежам еще больше расширяет его возможности.

Теперь у вас должно сложиться некоторое представление об основных структурах данных в языке Prolog и о том, как действует механизм унификации. Давайте попробуем объединить эти элементы с правилами и утверждениями для выполнения математических операций.

Списки и математические операции

Своим следующим примером я надеюсь продемонстрировать вам один из способов использования рекурсии для работы со списками и выполнения математических операций. В этих примерах вычисляются число элементов, сумма и среднее арифметическое. Вся работа выполняется с помощью пяти правил.

prolog/list_math.pl
http://media.pragprog.com/titles/btlang/code/prolog/list_math.pl

```
count(0, []).
count(Count, [Head|Tail]) :- count(TailCount, Tail), Count is TailCount
+ 1.

sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.

average(Average, List) :- sum(Sum, List), count(Count, List), Average is
Sum/Count.
```

Правило count – самое простое. Используется оно следующим образом:

```
| ?- count(What, [1]).

What = 1 ? ;

no
```

Правила до смешного просты. Число элементов в пустом списке равно 0. Число элементов в непустом списке равно числу элементов в «хвосте» плюс 1. Рассмотрим подробнее, как работает этот пример.

- Мы выполнили запрос count(What, [1]), который не унифицируется с первым правилом, потому что список не пуст. Далее

выполняется попытка удовлетворить цели во втором правиле, `count(Count, [Head|Tail])`. Механизм унификации связывает `What` с `Count`, `Head` с `1` и `Tail` с пустым списком `[]`.

- После унификации первая цель приобретает вид `count (TailCount, [])`. Интерпретатор, пытаясь проверить верность этой подцели, унифицирует ее с помощью первого правила. В результате переменная `TailCount` связывается со значением `0`. Первое правило удовлетворено, поэтому далее интерпретатор переходит к проверке второй цели.
- На этот раз он вычисляет `Count is TailCount + 1`. Переменная `TailCount` связана со значением `0`, поэтому переменная `Count` связывается с результатом выражения `0 + 1`, то есть 1.

Вот и все. Мы определили не рекурсивную процедуру, а логические правила. Следующий пример – сложение элементов списка. Взгляните еще раз на соответствующие правила:

```
sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.
```

Этот код действует точно так же, как и правило `count`. Он тоже состоит из двух утверждений: базового случая и рекурсивного случая. Используется данное правило похожим образом:

```
| ?- sum(What, [1, 2, 3]).

What = 6 ? ;

no
```

Если заглянуть внутрь интерпретатора и посмотреть, как действует правило `sum`, можно заметить, что оно работает подобно рекурсивной процедуре в императивных языках. Для пустого списка `sum` возвращает ноль; для непустого списка `sum` выполняет сложение `Head` и результат применения правила `sum` к `Tail`.

Однако существует и иная интерпретация происходящего. В действительности мы не сообщали интерпретатору, как вычислить сумму. Мы просто описали сумму как набор правил и целей. Чтобы удовлетворить некоторые из целей, логический механизм должен удовлетворить некоторые подцели. Декларативная интерпретация гласит так: «сумма элементов пустого списка равна нулю, а сумма элементов непустого списка равна `Total`, где `Total` есть сумма головы и хвоста». Мы заменили рекурсию удовлетворением целей и подцелей.

Аналогично число элементов в пустом списке равно нулю; число элементов в непустом списке равно один («голова») плюс число элементов в `Tail` (в «хвосте»).

Эти правила могут служить основой для других правил. Например, на основе правил `sum` и `count` можно реализовать вычисление среднего арифметического:

```
average(Average, List) :- sum(Sum, List), count(Count, List), Average is
Sum/Count.
```

То есть среднее арифметическое (`average`) для списка `List` равно `Average`, если:

- сумма (`sum`) этого списка (`List`) равна `Sum`;
- число элементов (`count`) в этом списке (`List`) равно `Count`;
- среднее (`Average`) равно `Sum/Count`.

Действует это правило, в точности как ожидалось:

```
| ?- average(What, [1, 2, 3]).

What = 2.0 ? ;

no
```

Использование правил в обоих направлениях

К настоящему моменту у вас должно сложиться достаточно четкое представление о том, как действует рекурсия. А теперь поговорим о встроенном правиле `append`. Правило `append(List1, List2, List3)` выполняется, если `List3` – это `List1` + `List2`. Это очень мощное правило, которое можно использовать в самых разных ситуациях.

Ниже представлены короткие фрагменты кода, которым можно найти множество применений. Вот детектор лжи:

```
| ?- append([oil], [water], [oil, water]).

yes
| ?- append([oil], [water], [oil, slick]).

no
```

А вот конструктор списков:

```
| ?- append([tiny], [bubbles], What).

What = [tiny,bubbles]

yes
```

Следующий фрагмент находит разность списков:

```
| ?- append([dessert_topping], Who, [dessert_topping, floor_wax]).

Who = [floor_wax]

yes
```

А этот вычисляет число возможных перестановок:

```
| ?- append(One, Two, [apples, oranges, bananas]).

One = []
Two = [apples,oranges,bananas] ? a

One = [apples]
Two = [oranges,bananas]

One = [apples,oranges]
Two = [bananas]

One = [apples,oranges,bananas]
Two = []

(1 ms) no
```

Итак, для одного правила мы нашли четыре разных применения. Кому-то может показаться, что определения подобных правил должны содержать массу кода. Давайте посмотрим, так ли это. Перепишем правило `append` на языке Prolog, но дадим ему имя `concatenate`. Выполним для этого следующие шаги:

1. Запишем правило `concatenate(List1, List2, List3)`, объединяющее пустой список со списком `List1`.
2. Добавим правило, добавляющее один элемент из списка `List1` в список `List2`.
3. Добавим правило, добавляющее два и три элемента из списка `List1` в список `List2`.
4. Посмотрим, можно ли обобщить полученные правила.

Приступим. Наш первый шаг – реализация объединения пустого списка со списком `List1`. Это правило проще простого:

prolog/concat_step_1.pl
http://media.pragprog.com/titles/btlang/code/prolog/concat_step_1.pl

```
concatenate([], List, List).
```

Правило `concatenate` соблюдается, если первый параметр является списком, а другие два параметра равны.

Вот как оно действует:

```
| ?- concatenate([], [harry], What).

What = [harry]

yes
```

Следующий шаг – добавить правило, добавляющее первый элемент из списка `List1` в начало списка `List2`:

prolog/concat_step_2.pl
http://media.pragprog.com/titles/btlang/code/prolog/concat_step_2.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
```

Правило `concatenate(List1, List2, List3)` разрывает список `List1` на «голову» и «хвост», при этом «хвост» становится пустым списком. Затем третий элемент так же разрывается на «голову» и «хвост», и в голову третьего элемента вставляется «голова» из списка `List1`, а в хвост вставляется список `List2`. Не забудьте скомпилировать базу знаний. Вот как она действует:

```
| ?- concatenate([malfoy], [potter], What).

What = [malfoy,potter]

yes
```

Теперь можно определить следующую пару правил, объединяющих списки с двумя и тремя элементами. Они работают точно так же:

prolog/concat_step_3.pl
http://media.pragprog.com/titles/btlang/code/prolog/concat_step_3.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
concatenate([Head1|[Head2|[]]], List, [Head1, Head2|List]).
concatenate([Head1|[Head2|[Head3|[]]]], List, [Head1, Head2,
Head3|List]).
```

```
| ?- concatenate([malfoy, granger], [potter], What).

What = [malfoy,granger,potter]

yes
```

Итак, мы имеем базовый случай и стратегию, согласно которой каждая подцель сокращает первый список и наращивает третий. Второй список остается постоянным. У нас уже достаточно информации,

чтобы обобщить результаты. Ниже приводится определение правила `concatenate` с использованием вложенных правил:

prolog/concat.pl
http://media.pragprog.com/titles/btlang/code/prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
  concatenate(Tail1, List, Tail2).
```

Этот короткий блок кода имеет невероятно простую интерпретацию. Первое утверждение гласит: «результатом объединения пустого списка со списком `List` является сам список `List`». Второе утверждение гласит: «результатом объединения списков `List1` и `List2` является список `List3`, если голова списка `List1` совпадает с головой списка `List3`, и при этом объединение “хвоста” списка `List1` со списком `List2` дает “хвост” списка `List3`». Простота и элегантность этого решения лишний раз доказывает мощь языка Prolog.

Давайте посмотрим, как интерпретатор выполнит запрос `concatenate([1, 2], [3], What)`. Мы посмотрим, как выполняется унификация на каждом шаге. Помните о вложенности правил, именно из-за этого при проверке каждой подцели мы будем работать с отдельной копией переменных. Каждую копию я буду отмечать отдельной буквой, чтобы вам было проще следить. На каждом проходе я буду показывать, что происходит, когда Prolog попытается проверить очередную подцель.

- Начнем:

  ```
  concatenate([1, 2], [3], What)
  ```

- Первое правило неприменимо, потому что `[1, 2]` не является пустым списком. В результате унификации мы получаем:

  ```
  concatenate([1|[2]], [3], [1|Tail2-A]) :- concatenate([2], [3],
  [Tail2-A])
  ```

  Унифицируется все, кроме второго «хвоста». Теперь переходим к целям. Унифицируем правую сторону.

- Здесь выполняется попытка применить правило `concatenate ([2], [3], [Tail2-A])`. В результате мы получаем:

  ```
  concatenate([2|[ ]], [3], [2| Tail2-B ]) :- concatenate([],
  [3],Tail2-B)
  ```

  Обратите внимание, что список `Tail2-B` — это «хвост» списка `Tail2-A`. Это не то же самое, что исходный список `Tail2`. Но сейчас нам снова нужно унифицировать правую сторону.

- `concatenate([], [3],Tail2-C) :- concatenate([], [3], [3]).`
- Итак, мы знаем, что `Tail2-C` – это список `[3]`. Теперь можно вернуться тем же путем, каким мы пришли сюда. Следите за третьим параметром: на каждом шаге происходит включение в `Tail2`. `Tail2-C` – это `[3]`, соответственно, `[2|Tail2-C]` – это `[2, 3]`, и, наконец, `[1|Tail2-B]` – это `[1, 2, 3]`. В результате получается `[1, 2, 3]`.

Prolog проделал огромный объем работы. Пройдитесь через этот список еще и еще раз, пока не поймете происходящее окончательно. Унификация вложенных подцелей является основой решения сложных задач.

Мы рассмотрели одну из богатейших функций в языке Prolog. Потратьте немного времени, чтобы исследовать представленные решения, и убедитесь, что понимаете их.

Что мы узнали во второй день

В этом разделе мы познакомились с основными строительными блоками, используемыми в языке Prolog для организации данных: списками и кортежами. Мы также научились вкладывать правила, что позволило нам описывать задачи, решаемые в других языках с помощью итераций. Мы детальнее рассмотрели механизм унификации в Prolog и порядок сопоставления обеих сторон в операторах `:-` и `=`. Создавая правила, мы описываем логические правила, а не алгоритмы, и позволяем интерпретатору Prolog найти свой путь к решению.

Мы также познакомились с особенностями использования простейших арифметических действий и вложенных подцелей для вычисления сумм и средних значений.

В заключение мы коснулись вопросов использования списков. Мы увидели, как выполнить присваивание элементов списка переменным, но, что особенно важно, мы узнали, как извлечь «голову» и «хвост» списка с использованием конструкции `[Head|Tail]`. Мы использовали этот прием для рекурсивного обхода списка. Все эти строительные блоки служат основой для решения сложных задач, с которыми мы столкнемся в третий день.

День 2: задания для самостоятельного решения

Найдите:

- какие-либо реализации поиска чисел Фибоначчи и вычисления факториалов. Разберитесь, как они работают;

- активное сообщество пользователей Prolog. Узнайте, какие задачи они решают.

Если вы готовы выполнить более сложные задания, попробуйте справиться со следующими задачами:

- найдите решение головоломки «Ханойские башни». Разберитесь, как оно работает;
- узнайте, какие проблемы могут возникать с выражениями, начинающимися со слова «не»? Почему в Prolog следует проявлять особую осторожность при использовании отрицания?

Практические задания:

- выполните перестановку элементов списка в обратном порядке;
- найдите наименьший элемент списка;
- отсортируйте элементы в списке.

4.4. День 3: Взорвем Лас-Вегас

Сейчас вы должны лучше понимать, почему я выбрал фильм «Человек дождя», точнее аутиста-эрудита, для проведения параллелей с языком Prolog. Кому-то будет сложно это понять, но иногда довольно забавно думать о программировании с такой точки зрения. Один из моих любимых эпизодов в фильме «Человек дождя» – когда брат Раймонда понял, что тот может «считать карты». Раймонд и его брат поехали в Лас-Вегас и просто сорвали банк. В этом разделе мы посмотрим на Prolog со стороны, которая заставит вас улыбнуться. Примеры кода в этой главе кажутся одинаково сумасшедшими и восхитительными. Далее будут представлены решения двух знаменитых головоломок, которые относятся к разряду систем с ограничениями, достаточно легко решаемых с помощью Prolog.

Возможно, у вас появится желание самостоятельно решить некоторые из этих головоломок. В этом случае попробуйте представить не пошаговое решение, а описание известных вам правил игры. Мы начнем с игры в судоку с небольшим игровым полем, а затем вы получите шанс расширить его. Потом мы перейдем к классической задаче «Восемь ферзей».

Решение судоку

Программирование игры судоку казалось мне чем-то невообразимо таинственным. Типичное игровое поле в судоку – квадрат 9×9, разделенный на меньшие квадраты со стороной в 3 клетки. Некоторые клетки изначально содержат цифры от 1 до 9, другие остаются пусты-

ми. От игрока требуется заполнить свободные клетки цифрами от 1 до 9 так, чтобы в каждой строке, в каждом столбце и в каждом малом квадрате 3×3 каждая цифра встречалась бы только один раз.

Мы напишем решение судоку для игрового поля 4×4. Суть от этого не меняется, однако решение получится короче. Начнем с описания мира, как мы его понимаем. Итак, у нас имеется игровое поле с четырьмя строками, четырьмя столбцами и четырьмя малыми квадратами 2×2. Следующая таблица иллюстрирует размещение малых квадратов с номерами от 1 до 4 в пределах игрового поля:

```
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
```

Сначала определимся, как мог бы выглядеть запрос. Это довольно просто. У нас имеются головоломка (`Puzzle`) и решение (`Solution`), соответственно, запрос будет иметь вид `sodoku(Puzzle, Solution)`. Пользователь может передавать головоломку в виде списка, с символами подчеркивания вместо ячеек с неизвестными цифрами, например:

```
sodoku([ _, _, 2, 3,
         _, _, _, _,
         _, _, _, _,
         3, 4, _, _],
         Solution).
```

Если решение существует, Prolog выведет его. Когда я решал эту головоломку на Ruby, мне пришлось продумать алгоритм ее решения. Чтобы реализовать решение на Prolog, этого не требуется. Нам просто нужно описать правила игры. Вот они:

- в случае правильного решения числа в решении и в самой головоломке должны совпадать;
- игровое поле содержит 16 ячеек со значениями от 1 до 4;
- игровое поле содержит четыре строки, четыре столбца и четыре малых квадрата;
- в случае правильного решения строки, столбцы и малые квадраты не должны содержать повторяющихся цифр.

Начнем с самого первого пункта. Числа в решении и в головоломке должны совпадать:

prolog/sudoku4_step_1.pl
http://media.pragprog.com/titles/btlang/code/prolog/sudoku4_step_1.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle.
```

Мы фактически достигли некоторого прогресса. Наш «решатель судоку» работает для случаев, когда в игровом поле отсутствуют незаполненные ячейки:

```
| ?- sudoku([4, 1, 2, 3,
             2, 3, 4, 1,
             1, 2, 3, 4,
             3, 4, 1, 2], Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

Формат вывода не самый привлекательный, но смысл понятен. Мы получили обратно 16 цифр, строку за строкой. Попробуем укротить свою жадность:

```
| ?- sudoku([1, 2, 3], Solution).

Solution = [1,2,3]

yes
```

Теперь используется недопустимое игровое поле, но решатель сообщает, что решение верное. Очевидно, что нам следует ограничить размер игрового поля 16 элементами. Кроме того, у нас имеется и другая проблема. Значения в ячейках могут быть любыми:

```
| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

Solution = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]

yes
```

Допустимое решение может содержать только числа от 1 до 4. Эта проблема приводит нас к двум выводам. Во-первых, наша реализация может возвращать недопустимые решения. Во-вторых, интерпретатор не имеет достаточного объема информации, чтобы проверить допустимость значений в ячейках. Иными словами, множество результатов *необоснованно*. Это означает, что мы должны добавить дополнительные правила, ограничивающие допустимые значения в ячейках, чтобы Prolog мог проверить их.

Давайте решим эти проблемы, введя в игру следующее правило. Второе правило в списке выше гласит: «игровое поле содержит 16 ячеек со значениями от 1 до 4». В GNU Prolog имеется встроенный предикат `fd_domain(List, LowerBound, UpperBound)`, позволяющий выразить ограничение на диапазон допустимых значений. Этот предикат

возвращает истинное значение, если все значения в списке `List` находятся в диапазоне от `LowerBound` до `UpperBound` включительно. Нам остается только убедиться, что все значения в `Puzzle` находятся в диапазоне от 1 до 4.

prolog/sudoku4_step_2.pl
http://media.pragprog.com/titles/btlang/code/prolog/sudoku4_step_2.pl

```
sudoku(Puzzle, Solution) :-
    Solution = Puzzle,
    Puzzle = [S11, S12, S13, S14,
              S21, S22, S23, S24,
              S31, S32, S33, S34,
              S41, S42, S43, S44],
    fd_domain(Puzzle, 1, 4).
```

Правило унифицирует `Puzzle` со списком из 16 переменных, а предикат `fd_domain` ограничивает значения в ячейках числами от 1 до 4. Теперь запрос будет терпеть неудачу, если игровое поле содержит недопустимые значения:

```
| ?- sudoku([1, 2, 3], Solution).

no

| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

no
```

Вот мы и подобрались к главной части решения. Третье правило гласит: «игровое поле содержит четыре строки, четыре столбца и четыре малых квадрата». Нам нужно «разрезать» игровое поле на строки, столбцы и квадраты. Теперь вам должно быть понятно, почему в предыдущем правиле мы дали переменным именно такие имена. Ниже приводится самый простой способ описать строки:

```
Row1 = [S11, S12, S13, S14],
Row2 = [S21, S22, S23, S24],
Row3 = [S31, S32, S33, S34],
Row4 = [S41, S42, S43, S44],
```

столбцы:

```
Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],
```

и квадраты:

```
Square1 = [S11, S12, S21, S22],
```

```
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44].
```

Теперь, когда мы нарезали игровое поле на фрагменты, можно перейти к определению следующего правила. Итак, решение считается допустимым, если в каждой строке, в каждом столбце и в каждом малом квадрате каждая цифра встречается только один раз. Для проверки присутствия повторяющихся элементов мы воспользуемся встроенным предикатом, имеющимся в GNU Prolog. Предикат `fd_all_different(List)` возвращает истинное значение, если все элементы в списке `List` имеют разные значения. Определим правило, выполняющее проверку всех строк, столбцов и квадратов:

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).
```

Это правило истинно, если все списки содержат только разные элементы. Первое утверждение гласит, что пустой список является допустимым. Второе утверждение гласит, что список является допустимым, если первый список содержит только разные элементы, и остальные списки также являются допустимыми.

Нам осталось лишь вызвать правило `valid(List)`:

```
valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

Хотите верьте, хотите нет, но мы закончили! Данный набор правил способен решать судоку с игровым полем 4×4:

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

В более дружественном представлении это решение выглядит так:

4 1 2 3
2 3 4 1
1 2 3 4
3 4 1 2

Ниже приводится полный текст программы, от начала до конца:

prolog/sudoku4.pl
http://media.pragprog.com/titles/btlang/code/prolog/sudoku4.pl

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).

sudoku(Puzzle, Solution) :-
        Solution = Puzzle,

        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],

        fd_domain(Solution, 1, 4),

        Row1 = [S11, S12, S13, S14],
        Row2 = [S21, S22, S23, S24],
        Row3 = [S31, S32, S33, S34],
        Row4 = [S41, S42, S43, S44],

        Col1 = [S11, S21, S31, S41],
        Col2 = [S12, S22, S32, S42],
        Col3 = [S13, S23, S33, S43],
        Col4 = [S14, S24, S34, S44],

        Square1 = [S11, S12, S21, S22],
        Square2 = [S13, S14, S23, S24],
        Square3 = [S31, S32, S41, S42],
        Square4 = [S33, S34, S43, S44],

        valid([Row1, Row2, Row3, Row4,
               Col1, Col2, Col3, Col4,
               Square1, Square2, Square3, Square4]).
```

Если вы еще не осознали, в каких областях можно применять Prolog, этот пример должен дать вам толчок в правильном направлении. А где же программа? А мы, собственно, и не писали программу. Мы описали правила игры: игровое поле содержит 16 ячеек с числами от 1 до 4, при этом строки, столбцы или квадраты не должны содержать повторяющихся значений. Для решения задачи пришлось написать всего пару десятков строк кода, и при этом нам не потребовалось знание стратегий решения судоку. В практических заданиях, предлагаемых в конце дня, вы получите шанс написать реализацию для решения судоку с игровым полем 9×9. Это будет совсем несложно.

Данная головоломка является отличным примером задач, с успехом решаемых на языке Prolog. У нас имеется ряд ограничений, которые легко выразить, но сложно разрешить. Давайте рассмотрим еще одну головоломку, с более существенными ограничениями: задачу «Восемь ферзей».

Восемь ферзей

Задача «Восемь ферзей» заключается в том, чтобы разместить на шахматной доске восемь ферзей, причем так, чтобы ни на одной вертикали, горизонтали или диагонали не находилось более одного ферзя. Кому-то эта задача может показаться тривиально простой, как детская забава. Но, с другой стороны, горизонтальные, вертикальные и диагональные линии можно рассматривать как ограниченные ресурсы. В промышленности имеется масса подобных задач, которые решаются как система с ограничениями. Давайте посмотрим, как она решается на языке Prolog.

Прежде всего посмотрим, как мог бы выглядеть запрос. Позицию каждого ферзя на шахматной доске можно выразить как кортеж `(Row, Col)` с координатами по горизонтали и вертикали. Шахматную доску можно представить как список `Board` кортежей. Запрос `eight_queens(Board)` возвращает положительный ответ, только если положения ферзей на доске удовлетворяют условиям задачи. Итак, наш запрос мог бы выглядеть так:

```
eight_queens([(1, 1), (3, 2), ...]).
```

Теперь посмотрим, какие цели должны быть удовлетворены. Если вы пожелаете сыграть в эту игру, не заглядывая в решение, просто ознакомьтесь с правилами ниже. Я не буду демонстрировать полное решение до самого конца этой главы.

- На доске имеются восемь ферзей.
- Каждый ферзь контролирует горизонталь и вертикаль с порядковыми номерами от 1 до 8.
- На одной горизонтали не может находиться более одного ферзя.
- На одной вертикали не может находиться более одного ферзя.
- На одной диагонали (с юго-востока на северо-запад) не может находиться более одного ферзя.
- На одной диагонали (с северо-востока на юго-запад) не может находиться более одного ферзя.

С горизонталями и вертикалями все просто, однако с диагоналями нужно быть внимательнее. Каждый ферзь контролирует одновремен-

но две диагонали, одна тянется слева и снизу (северо-запад) вправо и вверх (юго-восток), а другая тянется справа и снизу (северо-восток) влево и вверх (юго-запад), как показано на рис. 4.2. Но, как бы то ни было, эти правила легко выразить в коде.

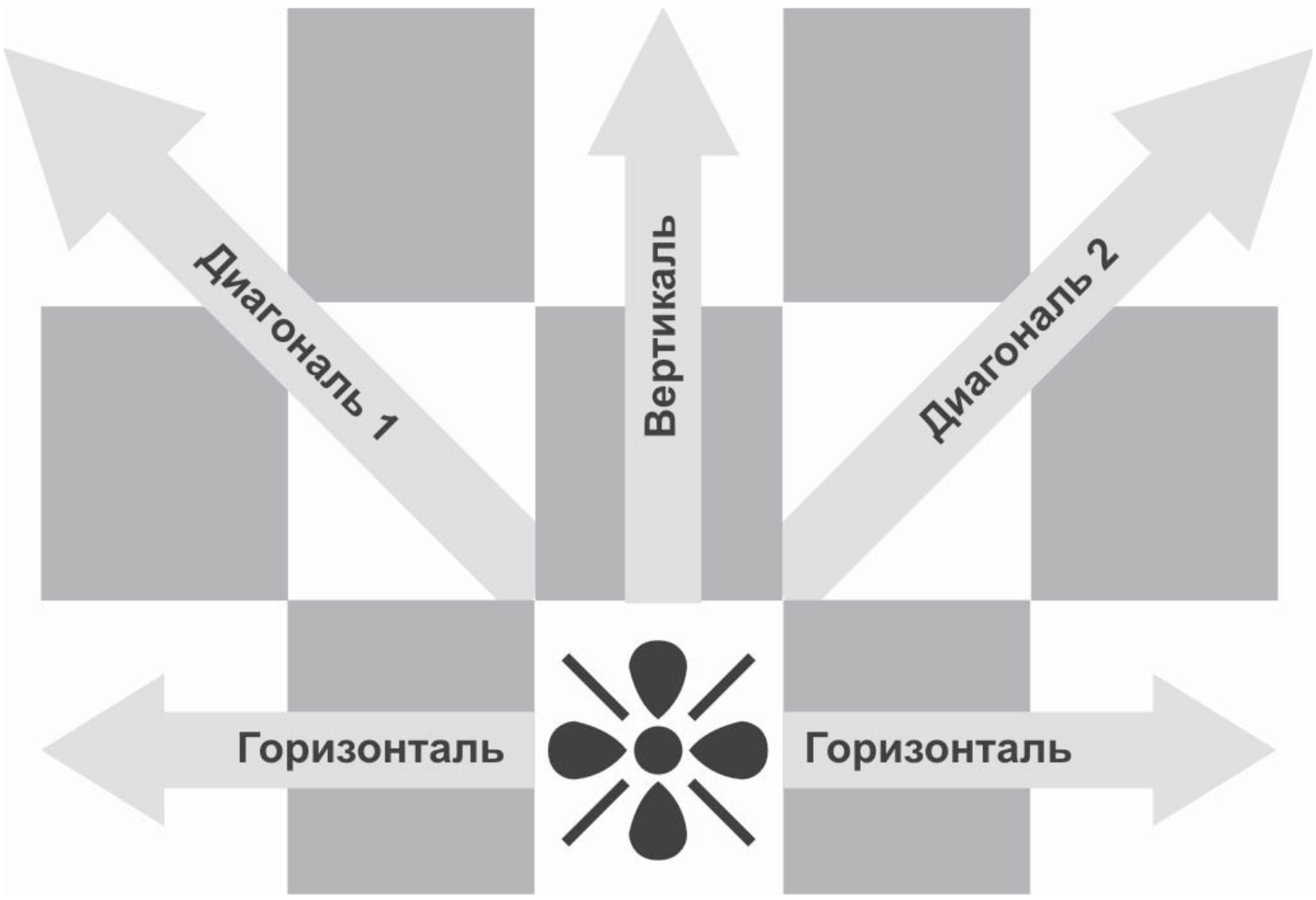

Рис. 4.2 ❖ Правила для задачи «Восемь ферзей»

Начнем опять с первого правила. Итак, у нас имеется восемь ферзей. Это означает, что список должен содержать восемь элементов. Выразить это правило достаточно просто. Мы можем использовать предикат `count`, представленный выше в этой главе, или воспользоваться встроенным предикатом `length`. Предикат `length(List, N)` возвращает истинное значение, если список `List` содержит `N` элементов. На этот раз я не буду демонстрировать работу каждой цели в отдельности, а ознакомлю вас со всеми целями по порядку, а в конце представлю законченное решение. Вот первая цель:

```
eight_queens(List) :- length(List, 8).
```

Далее нужно гарантировать допустимость координат каждого ферзя в списке. Следующее правило проверяет это условие:

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).
```

Предикат `member` действует именно так, как можно было бы догадаться; он проверяет вхождение в указанное множество. Координаты

ферзя считаются допустимыми, если каждая из координат является числом в диапазоне от 1 до 8. Теперь определим правило, проверяющее допустимость координат всех ферзей на шахматной доске:

```
valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).
```

Пустая шахматная доска считается допустимой. Допустимой также считается шахматная доска, первый ферзь на которой имеет допустимые координаты и остальная часть шахматной доски является допустимой.

Двигаемся дальше. Следующее правило проверяет присутствие на одной горизонтали не более одного ферзя. Чтобы учесть накладываемые ограничения, нам необходимо разбить программу на фрагменты, которые помогли бы описать проблему: что является горизонталью, вертикалью и диагональю? Начнем с горизонталей. Напишем функцию `rows(Queens, Rows)`, возвращающую истинное значение, если `Rows` является списком координат `Row` всех ферзей.

```
rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).
```

Здесь требуется немного напрячь свое воображение. Результатом применения `rows` к пустому списку будет пустой список, а результатом `rows(Queens, Rows)` будет `Rows`, если координата `Row` первого ферзя в списке соответствует первому элементу в `Rows`, а результатом применения `rows` к «хвосту» списка `Queens` является «хвост» списка `Rows`. Если вы ничего не поняли, пройдитесь по нескольким тестовым спискам. К счастью, вертикали обрабатываются точно так же, только вместо горизонтальных координат используются вертикальные:

```
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).
```

Логика работы осталась той же самой, только вместо первого элемента кортежа используется второй.

Идем дальше. Нам нужно пронумеровать диагонали. Проще всего это сделать с применением операций вычитания и сложения. Если принять север и запад за 1, тогда диагонали, простирающиеся с северо-запада на юго-восток, можно пронумеровать значениями `Col - Row`. Следующий предикат отбирает такие диагонали:

```
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
```

```
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

Данное правило работает точно так же, как правила `rows` и `cols`, но у нас есть еще одно ограничение: `Diagonal is Col - Row`. Обратите внимание, что это – не унификация! Это – предикат, позволяющий убедиться в обоснованности решения. Наконец, отберем диагонали, простирающиеся с юго-запада на северо-восток:

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).
```

Формула может показаться кому-то странной, но попробуйте несколько значений, пока не убедитесь, что ферзи с одинаковыми суммами своих координат находятся на одной диагонали. Теперь у нас имеются правила, позволяющие описать горизонтали, вертикали и диагонали. Нам осталось убедиться только в том, что на доске нет двух ферзей, у которых совпадали бы эти горизонтали, вертикали и диагонали.

Все это вы можете увидеть в окончательном решении. Проверки горизонталей, вертикалей и диагоналей выполняются последними восемью утверждениями.

prolog/queens.pl
http://media.pragprog.com/titles/btlang/code/prolog/queens.pl

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
    rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
    cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col - Row,
    diags1(QueensTail, DiagonalsTail).

diags2([], []).
```

```
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col + Row,
    diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
    length(Board, 8),
    valid_board(Board),

    rows(Board, Rows),
    cols(Board, Cols),
    diags1(Board, Diags1),
    diags2(Board, Diags2),

    fd_all_different(Rows),
    fd_all_different(Cols),
    fd_all_different(Diags1),
    fd_all_different(Diags2).
```

Теперь можно запустить программу, которая будет выполняться.., и выполняться.., и выполняться. Существует слишком много комбинаций, которые придется опробовать интерпретатору. Однако, если задуматься, можно обнаружить, что при любом раскладе на каждой горизонтали может находиться только один ферзь. Мы можем помочь интерпретатору быстрее найти решение, передав в запрос такую подсказку:

```
| ?- eight_queens([(1, A), (2, B), (3, C), (4, D), (5, E), (6, F), (7,
G), (8, H)]).

A =1
B =5
C =8
D =6
E =3
F =7
G =2
H = 4?
```

У нас получилось, но программа все равно работает слишком долго. Мы можем вообще убрать перебор горизонталей и упростить API. Ниже приводится оптимизированная версия:

prolog/optimized_queens.pl
http://media.pragprog.com/titles/btlang/code/prolog/optimized_queens.pl

```
valid_queen((Row, Col)) :- member(Col, [1,2,3,4,5,6,7,8]).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).
```

```
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
    cols(QueensTail, ColsTail).
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col - Row,
    diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col + Row,
    diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
    Board = [(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8,
_)],
    valid_board(Board),

    cols(Board, Cols),
    diags1(Board, Diags1),
    diags2(Board, Diags2),

    fd_all_different(Cols),
    fd_all_different(Diags1),
    fd_all_different(Diags2).
```

Мы внесли здесь одно важное изменение – организовали сопоставление `Board` с последовательностью `(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)` и тем самым существенно уменьшили общее количество перестановок. Мы также убрали все правила, связанные с горизонталями. На моем древнем MacBook окончательное решение вычисляется менее чем за три минуты.

Мы снова добились отличного результата, построив очень компактную базу знаний. Мы всего лишь описали правила игры и применили немного логики, чтобы ускорить поиск решения. Решая правильные задачи, я нашел свой путь в Prolog.

Что мы узнали в третий день

Сегодня мы применили свои знания языка Prolog для решения некоторых классических головоломок. Задачи с ограничениями имеют множество схожих черт с задачами, решаемыми в промышленности. Нам оказалось достаточно перечислить ограничения, чтобы найти решение. Нам и в голову не пришло бы пытаться написать соединение девяти таблиц на императивном языке SQL, но мы с легкостью решили две логические задачи именно таким способом.

Сначала мы занялись решением головоломки судоку. Решение на Prolog получилось удивительно простым. Мы отобразили 16 переменных в строки, столбцы и квадраты. Затем описали правила игры, наложив ограничения уникальности на строки, столбцы и квадраты. После этого Prolog методично опробовал все варианты и выдал готовое решение. Для построения интуитивно понятного API мы использовали шаблонные символы и переменные, но мы никак не обозначили пути решения задачи.

Затем мы решили задачу «Восемь ферзей». И снова мы всего лишь определили правила игры и позволили интерпретатору самому найти решение. Эта классическая задача, имеющая 92 возможных решения, потребовала большого количества вычислений, но даже наш упрощенный подход позволил нам решить ее за несколько минут.

Я все еще не знаю всех тонкостей игры в судоку, но благодаря Prolog эти знания мне не понадобились. Мне достаточно было знать правила игры.

День 3: задания для самостоятельного решения

Найдите:

- в Prolog имеется несколько инструментов ввода/вывода, найдите предикаты, позволяющие выводить переменные;
- найдите способ использования предикатов вывода, чтобы с их помощью выводились только успешные решения; как они работают?

Практические задания:

- измените реализацию решения судоку так, чтобы она могла обрабатывать игровое поле 6×6 (с квадратами 3×2) и игровое поле 9×9;
- добавьте в реализацию решения судоку вывод только успешных решений.

Если вам нравится решать головоломки, вы не сможете не влюбиться в Prolog. Желающие погрузиться глубже в решение головоломок, начните с задачи «Восемь ферзей»:

- решите задачу «Восемь ферзей» на основе списка ферзей. Представьте каждого ферзя числом от 1 до 8 и используйте его порядковый номер в списке как номер горизонтали, в которой он находится.

4.5. В заключение о Prolog

Prolog – один из старейших языков программирования среди представленных в этой книге, но идеи, заложенные в него, остаются актуальными и в наши дни. Название Prolog происходит от «Programming with Logic» (логическое программирование). Мы использовали Prolog для обработки правил, состоящих из утверждений, которые, в свою очередь, состоят из последовательностей целей.

Процесс программирования на Prolog состоит из двух основных этапов. Сначала создается база знаний, состоящая из логических фактов и выводов о предметной области. Затем база знаний компилируется и интерпретатору задаются вопросы, касающиеся предметной области. Некоторые запросы могут иметь форму утверждений, и на эти утверждения Prolog будет давать односложные ответы `yes` (да) или `no` (нет). Другие запросы могут содержать переменные, и в ответ на эти запросы Prolog будет стараться заполнить переменные такими значениями, чтобы получались утвердительные ответы.

Вместо простого присваивания Prolog использует особую процедуру, которая называется *унификация*. Она обеспечивает совпадение переменных с обеих сторон. Иногда Prolog вынужден опробовать множество различных комбинаций значений в переменных, чтобы добиться успеха.

Сильные стороны

Язык Prolog можно применять для решения самых разных задач, от составления расписания движения воздушных судов до реализации финансовых инструментов. Язык Prolog сложен в изучении, но возможность решать труднейшие задачи делает освоение Prolog и других языков, подобных ему, стоящим делом.

Вспомните Брайана Тарбокса (Brian Tarbox), занимающегося исследованием дельфинов. Ему удалось на основе простых выводов о своей предметной области объяснить сложное поведение дельфина. Он также создал на Prolog систему планирования экспериментов, исходящую из весьма ограниченных ресурсов. Существует несколько областей, где Prolog активно используется в наши дни.

Обработка текстов на естественных языках

Prolog одним из первых был использован для распознавания фраз на естественном языке. В частности, существуют модели на языке Prolog, которые могут принимать фразы и с применением базы знаний,

состоящей из фактов и выводов, преобразовывать выражения на сложном и неточном языке в конкретные инструкции для компьютеров.

Игры

Игры с годами становятся все более сложными, особенно игры, моделирующие поведение конкурентов или врагов. Модели на языке Prolog легко могут выражать поведение других персонажей в игре. С помощью Prolog можно также придавать разным врагам разные особенности поведения, что позволяет сделать игру более близкой к реальности.

Семантическая паутина

Семантическая паутина (Semantic Web) – направление развития Всемирной паутины, целью которого является представление информации в виде, пригодном для машинной обработки. Разработчик может предоставить базовое описание ресурса на языке описания ресурсов (Resource Description Framework, RDF). Сервер может собирать из этих описаний базу знаний. А уже эта база знаний, совместно с возможностью обработки фраз на естественном языке, может использоваться конечными пользователями для выполнения запросов. В настоящее время существует множество пакетов на Prolog, реализующих эту функциональность в контексте веб-сервера.

Искусственный интеллект

Под искусственным интеллектом (ИИ) понимаются наука и технология внедрения интеллекта в машины. Машинный интеллект может иметь разные формы, но в общем случае все сводится к изменению поведения некоторого агента на основе сложнейших правил. Prolog выделяется в этой области среди других языков, особенно когда используются конкретные правила, основанные на формальной логике. По этой причине Prolog иногда называют *языком логического программирования* (logic programming language).

Планирование

Prolog прекрасно подходит для планирования в среде с ограниченными ресурсами. Многие используют Prolog для создания своих систем планирования.

Недостатки

Язык Prolog было создан довольно давно, тем не менее он имеет серьезные ограничения.

Практичность

Хотя в основной своей области Prolog превосходно справляется с разнообразными задачами, он все же занимает весьма узкую нишу, нишу логического программирования. Это не универсальный язык. Кроме того, ему свойственны ограничения, обусловленные архитектурой языка.

Очень большие множества данных

В Prolog используется механизм принятия решений с поиском в глубину, проверяющий все возможные комбинации переменных во всем множестве правил. Многие языки и компиляторы оптимизируют этот процесс. И все же эта стратегия остается весьма требовательной к вычислительным ресурсам, особенно с ростом объемов обрабатываемых данных. Данная особенность вынуждает пользователей Prolog изучать особенности функционирования языка, чтобы сохранить объемы данных в разумных пределах.

Смешение императивной и декларативной моделей

Как и при использовании многих языков, входящих в семейство языков функционального программирования, особенно тех, что прочно опираются на рекурсию, программируя на Prolog, необходимо понимать, как он обрабатывает рекурсивные правила. Часто для решения даже умеренно больших задач приходится использовать правила с хвостовой рекурсией. Довольно легко можно написать приложение на Prolog, способное обрабатывать лишь до смешного малые объемы данных. Вам необходимо иметь глубокое понимание особенностей работы Prolog, чтобы писать эффективные правила, позволяющие обрабатывать самые разные объемы данных.

Заключительные замечания

Исследуя различные языки в процессе работы над этой книгой, я частo с изумлением обнаруживал, что много лет подряд «заколачивал шурупы молотком». Prolog в этом отношении явился наиболее ярким примером моего развития. Если вам доведется столкнуться с задачей, которая легко может быть решена на языке Prolog, воспользуйтесь этим. В таких ситуациях часто бывает удобно использовать этот язык, основанный на правилах, в комбинации с другими универсальными языками, подобно тому, как вы используете SQL в программах на Ruby или Java. Если вам удастся построить такую комбинацию, вы наверняка окажетесь в выигрыше в долгосрочной перспективе.

Глава 5

Scala

«Мы не овцы».

Эдвард Руки-ножницы

К настоящему моменту мы познакомились с тремя языками и с тремя разными парадигмами программирования. Язык Scala будет четвертым. Это – гибридный язык, в том смысле что он создавался с целью заполнить пустоту между парадигмами программирования. В данном случае – пустоту между объектно-ориентированными языками, такими как Java, и функциональными, такими как Haskell. В этом смысле язык Scala больше походит на Франкенштейна. Но еще лучше сравнить его с Эдвардом Руки-ножницы (Edward Scissorhands)[1].

Главным персонажем в этом фантастическом фильме Тима Бартона (Tim Burton) был Эдвард, наполовину человек, наполовину робот, с ножницами вместо рук. Это один из самых любимых мною персонажей. Эдвард – очаровательный персонаж. Иногда он бывает неуклюжим, иногда очень ловким, но он всегда удивительно выразителен. Иногда с помощью своих ножниц он творил невероятное. Иногда проявлял крайнюю неуклюжесть. Он часто оставался непонятым и обвинялся в «отступлении от добродетелей». В один из таких моментов он с детской непосредственностью отвечает: «Мы не овцы».

5.1. О языке Scala

С ростом сложности требований, предъявляемых к компьютерным программам, возрастают и требования к языкам программирования. Через каждые примерно двадцать лет старые парадигмы перестают удовлетворять новым требованиям к организации и выражению идей. На сцене появляются новые парадигмы, но этот процесс очень непрост.

---

[1] «Edward Scissorhands». DVD. Режиссер Тим Бартон (Tim Burton). 1990; Беверли Хиллс, Калифорния: 20th Century Fox, 2002. (http://ru.wikipedia.org/wiki/Эдвард_Руки-ножницы. – *Прим. перев.*)

Появление каждой новой парадигмы вызывает целую волну появления новых языков программирования. Первые языки часто оказываются удивительно продуктивными, но крайне непрактичными. Вспомните объектно-ориентированный Smalltalk или функциональный Lisp. Затем в языках, прежде поддерживавших другие парадигмы, начинают появляться особенности, позволяющие людям использовать новые понятия, оставаясь при этом в рамках безопасной старой парадигмы. Язык Ada, например, обеспечил поддержку некоторых основных объектно-ориентированных приемов, таких как инкапсуляция, оставаясь по сути процедурным языком. В некоторый момент появляется гибридный язык, соединяющий старую и новую парадигмы, как это получилось, например, с языком C++. Затем на сцену выходят коммерческие разработки, такие как Java или C#. И наконец, возникают зрелые реализации новой парадигмы.

Близость с Java...

Язык Scala – не просто мост, соединяющий две парадигмы, но нечто большее. Он предлагает тесную интеграцию с Java и дает пользователям возможность защитить свои инвестиции:

- программы на языке Scala выполняются под управлением виртуальной машины Java, поэтому они могут вводиться в эксплуатацию в уже имеющихся окружениях;
- программы на языке Scala способны непосредственно использовать библиотеки Java, поэтому разработчики могут пользоваться существующими фреймворками и прежними своими наработками;
- как и Java, язык Scala имеет статическую систему типов, поэтому при программировании на этих языках можно использовать общую философию;
- синтаксис языка Scala довольно близок к синтаксису языка Java, поэтому разработчики осваивают его достаточно быстро;
- язык Scala поддерживает и объектно-ориентированную, и функциональную парадигмы программирования, благодаря чему программисты могут постепенно учиться применять идеи функционального программирования в своем коде.

Но без рабской преданности

Некоторые языки довольно далеко отходят от своих предков, расширяя тесные рамки устаревших понятий. Несмотря на существенное

сходство с Java, язык Scala имеет некоторые важные отличия, получившие широкое признание, которые перечислены ниже.

- *Вывод типов*. В программах на языке Java необходимо явно указывать тип каждой переменной, аргумента или параметра. Язык Scala, напротив, старается выводить типы переменных везде, где это возможно.
- *Поддержка функционального программирования*. Язык Scala вводит важнейшие понятия функционального программирования в мир Java. В частности, новый язык позволяет на основе существующих функций создавать новые функции. К числу понятий, с которыми вы познакомитесь в этой главе, относятся: блоки кода, функции высшего порядка и обширная библиотека коллекций. Scala – это больше, чем простой синтаксический сахар.
- *Неизменяемые переменные*. Язык Java поддерживает неизменяемые переменные, но для этого требуется использовать специальные модификаторы. В этой главе вы увидите, что язык Scala, напротив, вынуждает явно принимать решение о том, какие переменные будут изменяемыми. Эти решения оказывают сильное влияние на поведение приложений в многозадачном контексте.
- *Дополнительные программные конструкции*. Scala вводит дополнительные надстройки над основополагающими понятиями. В этой главе вы познакомитесь с акторами, применяемыми для организации параллельного выполнения, с коллекциями в стиле языка Ruby, с функциями высшего порядка и с механизмами обработки XML.

Прежде чем двинуться дальше, необходимо познакомиться с побудительными мотивами, положенными в основу языка Scala. Мы познакомимся с создателем языка и узнаем, как он пришел к идее объединить две парадигмы программирования.

Интервью с создателем Scala, Мартином Одерски

Мартин Одерски (Martin Odersky), создатель языка Scala, профессор швейцарской высшей технической школы Лозанны (École Polytechnique Fédérale de Lausanne, EPFL), одного из двух швейцарских федеральных технологических институтов. Участвовал в разработке спецификации обобщенных типов Java (Java Generics Specification) и является создателем оптимизирующего компилятора `javac`. Кроме того, им была написана одна из лучших книг о программировании

на языке Scala «Programming in Scala: A Comprehensive Step-by-Step Guide» [OSV08]. Вот какая беседа у нас с ним состоялась:

Брюс: Как вы пришли к идее написать язык Scala?

Доктор Одерски: Я был убежден, что сочетание функциональной и объектно-ориентированной парадигм будет иметь большую практическую ценность. Я был удручен пренебрежительным отношением сторонников функционального программирования к объектно-ориентированной парадигме и верой объектно-ориентированных программистов в то, что функциональная парадигма представляет лишь академический интерес. Мне захотелось показать, что эти две парадигмы можно объединить и получить в результате новый, мощный сплав. Мне также хотелось создать язык, на котором лично мне будет удобно писать программы.

Брюс: Что больше всего вам нравится в вашем языке?

Доктор Одерски: Мне нравится, что он позволяет программистам легко и свободно выражать свои мысли и в то же время дает надежную опору через свою систему типов.

Брюс: Для решения каких типов задач лучше всего подходит этот язык?

Доктор Одерски: В действительности Scala – это универсальный язык. Я не встречался с задачами, которые не мог бы решить с его помощью. И все же хочу отметить, что главным преимуществом Scala перед другими основными языками программирования является поддержка функционального программирования. То есть везде, где предпочтительнее использовать функциональный подход, Scala предстает во всей своей красе, будь то многозадачное приложение, веб-приложение с поддержкой XML или реализация предметно-ориентированного языка.

Брюс: Что бы вы изменили в языке, будь у вас возможность начать все заново?

Доктор Одерски: Механизм вывода типов переменных в Scala в общем и целом работает неплохо, но он имеет некоторые ограничения. Если бы я мог начать все сначала, я попробовал бы реализовать более мощный механизм. Может быть, это можно сделать и сейчас, но согласитесь, что менять что-то в обширной кодовой базе намного сложнее.

Шумиха вокруг Scala продолжает нарастать, и этому в немалой степени способствовал перенос реализации обработки сообщений

в Twitter с языка Ruby на язык Scala. Объектно-ориентированные особенности языка позволяют безболезненно перейти на него с языка Java, но наиболее привлекательной чертой Scala все же является поддержка функционального программирования. Исключительно функциональные языки поддерживают стиль программирования, имеющий строгие математические основы. Типичный функциональный язык обладает следующими характеристиками:

- функциональные программы состоят из функций;
- любая функция всегда возвращает некоторое значение;
- вызов функции с одними и теми же аргументами всегда будет возвращать одно и то же значение;
- функциональные программы исключают возможность изменения информации о состоянии. После начального присваивания переменной некоторого значения она остается неизменной.

Строго говоря, язык Scala не является исключительно функциональным языком, так же как C++ не является исключительно объектно-ориентированным языком. Он поддерживает изменяемые переменные, вследствие чего функции могут возвращать разные значения для одного и того же набора входных аргументов. (Например, методы чтения и записи свойств в большинстве объектно-ориентированных языков с легкостью могут нарушать это правило.) Но он также предлагает инструменты, дающие разработчикам возможность использовать функциональные абстракции там, где они имеют смысл.

Функциональное программирование и параллельные вычисления

Самой большой проблемой, с которой сталкиваются программисты при разработке многозадачных приложений на объектно-ориентированных языках, является *изменчивость состояния*, то есть данные могут измениться в любой момент в другом потоке выполнения. Значение любой переменной может быть изменено после инициализации. Изменчивость для параллельных вычислений – это как доктор Зло для Остина Пауэрса. Если два разных потока выполнения могут одновременно изменять одни и те же данные, появляется вероятность нарушения целостности этих данных, причем такие ситуации очень сложно выявить при тестировании, поскольку они носят нерегулярный характер. В базах данных эта проблема решается с помощью транзакций и блокировок. В объектно-ориентированных языках предусмотрены инструменты управления доступом к совместно

используемым данным, но обычно программисты используют эти инструменты не лучшим образом, даже если знают, как надо их применять.

Языки функционального программирования решают эти проблемы путем устранения изменяемого состояния из уравнения. Язык Scala не вынуждает полностью избавляться от изменяемого состояния, но дает инструменты, позволяющие программистам выражать свои идеи в функциональном стиле.

Пользуясь языком Scala, вам не придется выбирать между Smalltalk и Lisp. Так давайте объединим объектно-ориентированный и функциональный миры в программном коде на языке Scala.

5.2. День 1: Дом на холме

В фильме «Эдвард Руки-ножницы» имелся дом на холме. В прошлом этот дом был странным, но очень красивым местом, а сейчас в нем царят разруха и запустенье. Ветер гуляет в оконных проемах, и комнаты уже не такие, какими они были прежде. Дом, когда-то очень удобный для его обитателей, сейчас превратился в холодные и непривлекательные развалины. На лице объектно-ориентированной парадигмы тоже стали видны следы времени, особенно они заметны в самых ранних реализациях объектно-ориентированных языков. Язык Java со своими устаревшими системами статических типов и поддержки параллельных вычислений давно уже нуждается в реконструкции. В этом разделе мы будем говорить прежде всего о Scala, подчеркивая параллели между объектно-ориентированной парадигмой и домом на холме.

Программы на языке Scala выполняются под управлением виртуальной машины Java (Java Virtual Machine, JVM). Я не собираюсь давать вам исчерпывающий обзор языка Java; эту информацию вы сможете почерпнуть в другом месте. Вы познакомитесь с некоторыми идеями из мира Java, перекочевавшими в язык Scala, но я постараюсь свести к минимуму их влияние, чтобы вам не пришлось изучать два языка сразу. А теперь установите Scala. В процессе работы над этой книгой я пользовался версией 2.7.7.final.

Типы данных в Scala

После установки Scala выполните в консоли команду `scala`. Если все в порядке, вы увидите приглашение к вводу `scala>`. Теперь давайте введем немного кода.

```
scala> println("Hello, surreal world")
Hello, surreal world

scala> 1 + 1
res8: Int =2

scala> (1).+(1)
res9: Int =2

scala> 5 + 4 * 3
res10: Int = 17

scala> 5.+(4.*(3))
res11: Double = 17.0

scala> (5).+((4).*(3))

res12: Int = 17
```

Итак, целые числа являются объектами. В Java я потратил немало нервов из-за преобразования между типами `Int` (примитивы) и `Integer` (объекты). В языке Scala все элементы являются объектами, за редким исключением. Это важное отличие от статической системы типов в объектно-ориентированных языках. Давайте посмотрим, как в Scala обрабатываются строки:

```
scala> "abc".size
res13: Int =3
```

Итак, строка тоже является самым обычным объектом, с небольшой примесью синтаксического сахара. Попробуем вызвать конфликт типов:

```
scala> "abc" + 4
res14: java.lang.String = abc4

scala> 4 + "abc"
res15: java.lang.String = 4abc

scala> 4 + "1.0"
res16: java.lang.String = 41.0
```

М-да. Это не совсем то, что мы ожидали. Scala преобразует целые числа в строки. Попробуем ужесточить конфликт:

```
scala> 4 * "abc"
<console>:5: error: overloaded method value * with alternatives
  (Double)Double <and>
  (Float)Float <and>
  (Long)Long <and>
  (Int)Int <and>
```

```
(Char)Int <and>
(Short)Int <and>
(Byte)Int
cannot be applied to (java.lang.String)
     4 * "abc"
       ^
```

Ага, как раз то, что нужно. Scala действительно является строго типизированным языком. В нем имеется механизм вывода типов, поэтому в большинстве случаев он способен автоматически определять типы переменных, пользуясь синтаксическими подсказками, но, в отличие от Ruby, Scala проверяет типы на этапе компиляции. Фактически, когда вы вводите программный код в консоли Scala, он компилируется и только потом выполняется.

Небольшое примечание. В примерах выше видно, что в качестве результатов возвращаются строки Java. Эта тема подробно освещается во многих статьях и книгах о языке Scala, но мы не будем останавливаться на этом факте, а продолжим исследование программных конструкций, которые, как мне кажется, будут вам более интересны. По пути я укажу некоторые книги, где подробно рассматриваются вопросы интеграции с Java. А пока просто имейте в виду, что во многих случаях Scala пользуется стратегией совместного использования одних и тех же типов в двух языках. Как следствие в Scala используются те же элементарные типы, что и в Java (такие как `java.lang.String`), там, где это уместно. Доверьтесь мне и примите предлагаемые мною упрощения.

Выражения и условные конструкции

Теперь быстро пробежимся по синтаксису языка на примерах. Ниже представлено несколько выражений на языке Scala, возвращающих значение `true/false`:

```
scala> 5 < 6
res27: Boolean = true

scala> 5 <= 6
res28: Boolean = true

scala> 5 <= 2
res29: Boolean = false

scala> 5 >= 2
res30: Boolean = true

scala> 5 != 2
res31: Boolean = true
```

Пока ничего интересного, Мы видим привычный C-подобный синтаксис, поддерживаемый некоторыми другими языками, рассматривавшимися выше. Давайте попробуем задействовать выражение в инструкции `if`:

```
scala> val a = 1
a: Int =1

scala> val b = 2
b: Int =2

scala> if ( b < a) {
     |   println("true")
     | } else {
     |   println("false")
     | }
false
```

Мы присвоили значения двум переменным и сравнили их в инструкции `if/else`. Рассмотрим внимательнее инструкцию присваивания значения переменной. Прежде всего обратите внимание на отсутствие объявления типа. В отличие от Ruby, в языке Scala типы присваиваются переменным во время компиляции. А в отличие от Java, типы в языке Scala могут определяться из контекста (то есть выводиться), поэтому нет нужды вводить полное объявление переменной, такое как `val a : Int = 1`, хотя это и не возбраняется.

Далее, отметьте, что объявления переменных в языке Scala начинаются с ключевого слова `val`. Переменные могут также объявляться с помощью ключевого слова `var`. `val` объявляет неизменяемые переменные, а `var` – нет. Мы еще вернемся к этой теме ниже.

В Ruby число `0` интерпретируется как `true`. В C число `0` интерпретируется как `false`. В обоих языках значение `nil` интерпретируется как `false`. Давайте посмотрим, как обстоят дела с этим в языке Scala:

```
scala> Nil
res3: Nil.type = List()

scala> if(0) {println("true")}
<console>:5: error: type mismatch;
 found   : Int(0)
 required: Boolean
       if(0) {println("true")}
          ^

scala> if(Nil) {println("true")}
<console>:5: error: type mismatch;
```

```
found   : Nil.type (with underlying type object Nil)
required: Boolean
      if(Nil) {println("true")}
         ^
```

Итак, значение `Nil` – это пустой список, и в условных инструкциях нельзя даже проверить `Nil` или `0`. Такое поведение обусловлено строгой философией статической типизации, которой следует Scala. `Nil` и числа не являются логическими значениями, поэтому их нельзя использовать в качестве логических значений. Теперь, когда мы познакомились с простыми выражениями и основными условными конструкциями, можно перейти к циклам.

Циклы

Ниже приводится пара чуть более сложных программ, которые надо будет запускать из файлов, а не в консоли. Запуск программ на языке Scala производится так же, как запуск программ на Ruby или Io:

```
scala path/to/program.scala.
```

Во второй день вы увидите еще несколько способов организации итераций по множеству значений, когда мы приступим к знакомству с блоками кода. А пока сосредоточимся на императивном подходе к выполнению циклов, который очень близко напоминает подход в языке Java.

Мои войны с системой статических типов

Некоторые начинающие программисты часто путают понятия «статическая типизация» и «строгая типизация». В общих чертах языки со строгим контролем типов проверяют совместимость типов операндов и генерируют сообщения об ошибках или выполняют необходимые преобразования, если типы оказываются несовместимыми. Языки Java и Ruby, например, являются языками со строгим контролем типов. (Я понимаю, что это слишком упрощенное описание.) Языки Ассемблера и C, напротив, являются языками со слабым контролем типов. Компиляторы могут не проверять тип данных, хранящихся в памяти.

Статическая и динамическая типизация – это совершенно иное понятие. Языки со статической типизацией поддерживают полиморфизм на уровне внутренней структуры типов. Они отличают настоящих генетических (статических) уток от подделок, которые ходят и крякают как утки. Языки со статической типизацией обладают важным преимуществом, так как компиляторы и другие инструменты имеют дополнительную информацию, помогающую обнаруживать ошибки на ранних стадиях. Но за это приходится платить дополнительными усилиями и мириться с некоторыми ограничениями. Вы как разработчик часто будете ощущать на себе цену, которую приходится платить за статическую типизацию.

Свое первое объектно-ориентированное приложение я написал на Java. Я перебрал множество разных фреймворков, пытаясь освободиться от оков ста-

тической системы типов в Java. Промышленность вложила сотни миллионов долларов в три версии Enterprise Java Beans, Spring, Hibernate, JBoss и аспектно-ориентированное программирование, чтобы сделать некоторые модели программирования более податливыми. Мы старались сделать модель поддержки типов в Java более динамичной, и с каждым шагом сопротивление увеличивалось все больше и больше. В конце концов у нас даже создалось ощущение, что мы имеем дело с соперничающей культурой, а не со средой программирования. По истории выхода моих книг четко прослеживается мой путь от фреймворков с повышенной динамичностью к динамическим языкам.

Итак, предвзятое отношение к статической типизации сформировалось у меня в пору моих войн с языком Java. Знакомство с языком Haskell и его превосходной системой статических типов помогло мне изменить свое мнение. Моя совесть чиста. Вы пригласили кабинетного политика на этот случайный обед, но я приложу все силы, чтобы сохранить легкость и непринужденность нашего общения.

Первый способ – простой цикл `while`:

scala/while.scala
http://media.pragprog.com/titles/btlang/code/scala/while.scala

```
def whileLoop {
    var i = 1
    while(i <= 3){
        println(i)
        i +=1
    }
}

whileLoop
```

Здесь определяется функция. Программисты на Java могут заметить отсутствие спецификатора `public`. В Scala область видимости `public` используется по умолчанию, то есть эта функция будет доступна из любой точки программы.

Внутри функции объявляется простой цикл `while`, который просто ведет счет от 1 до 3. Переменная `i` изменяется в цикле, поэтому она объявлена как `var`. Далее следует объявление инструкции `while` в стиле Java. Код в фигурных скобках выполняется, пока условное выражение не вернет значение `false`. Запустить этот пример можно, как показано ниже:

```
batate$ scala code/scala/while.scala
1
2
3
```

Цикл `for` также действует подобно аналогичному циклу в языках Java и C, но имеет несколько иной синтаксис:

scala/for_loop.scala
http://media.pragprog.com/titles/btlang/code/scala/for_loop.scala

```
def forLoop {
    println( "for loop using Java-style iteration" )
    for(i <- 0 until args.length){
        println(args(i))
    }
}
```

```
forLoop
```

Сначала указывается аргумент цикла, который является переменной, за ним следуют оператор `<-` и диапазон изменения аргумента цикла в форме `initialValue until endingValue`. В данном случае выполняются итерации по аргументам командной строки:

```
batate$ scala code/scala/forLoop.scala its all in the grind
for loop using Java-style iteration
its
all
in
the
grind
```

Как и в Ruby, циклы можно использовать для итераций по элементам коллекции. Например, цикл `foreach`, напоминающий цикл `each` в Ruby:

scala/ruby_for_loop.scala
http://media.pragprog.com/titles/btlang/code/scala/ruby_for_loop.scala

```
def rubyStyleForLoop {
    println( "for loop using Ruby-style iteration" )
    args.foreach { arg =>
        println(arg)
    }
}
```

```
rubyStyleForLoop
```

Здесь `args` – это список с аргументами командной строки. Все элементы списка поочередно будут передаваться в блок кода. В данном случае `arg` – это один аргумент из списка аргументов. В Ruby то же самое действие можно выполнить инструкцией `args.each {|arg| println(arg) }`. Синтаксис определения одного аргумента несколько отличается, но сама идея остается неизменной. Ниже демонстрируется пример запуска этого фрагмента:

```
batate$ scala code/scala/ruby_for_loop.scala freeze those knees
chickadees
for loop using Ruby-style iteration
freeze
those
knees
chickadees
```

Позднее вы поймаете себя на том, что используете этот способ реализации итераций намного чаще остальных. Но сейчас нас интересует не это, поэтому давайте пока отложим дальнейшее обсуждение циклов.

Диапазоны и кортежи

Как и Ruby, Scala поддерживает диапазоны значений. Запустите консоль и введите следующие фрагменты:

```
scala> val range = 0 until 10
range: Range = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> range.start
res2: Int =0

scala> range.end
res3: Int = 10
```

Диапазоны в языке Scala действуют точно так же, как в Ruby. Они позволяют определить шаг приращения:

```
scala> range.step
res4: Int =1

scala> (0 to 10) by 5
res6: Range = Range(0, 5, 10)

scala> (0 to 10) by 6
res7: Range = Range(0, 6)
```

Эквивалентом диапазона `1..10` в Ruby является диапазон `1 to 10` в Scala, а эквивалентом диапазона `1...10` является диапазон `1 until 10`. Ключевое слово `to` используется для создания диапазонов, включающих верхнюю границу:

```
scala> (0 until 10 by 5)
res0: Range = Range(0, 5)
```

Диапазоны дают возможность определить направление явно:

```
scala> val range = (10 until 0) by -1
range: Range = Range(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

Но направление не определяется автоматически:

```
scala> val range = (10 until 0)
range: Range = Range()

scala> val range = (0 to 10)
range: Range.Inclusive = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

По умолчанию используется шаг, равный 1, независимо от граничных значений диапазона. Диапазоны могут определяться не только целыми числами:

```
scala> val range = 'a' to 'e'
range: RandomAccessSeq.Projection[Char] = RandomAccessSeq.Projection(a,
b, c, d, e)
```

Scala автоматически выполняет некоторые преобразования типов. Фактически, когда определяется инструкция `for`, в действительности определяется диапазон.

В языке Scala, как и в Prolog, поддерживаются кортежи. Кортеж – это фиксированное множество объектов. Эту структуру данных можно найти также во многих функциональных языках. Объекты в кортеже необязательно должны принадлежать к одному типу. В исключительно функциональных языках программисты часто выражают объекты и их атрибуты в виде кортежей. Например:

```
scala> val person = ("Elvis", "Presley")
person: (java.lang.String, java.lang.String) = (Elvis,Presley)

scala> person._1
res9: java.lang.String = Elvis

scala> person._2
res10: java.lang.String = Presley

scala> person._3
<console>:6: error: value _3 is not a member of (java.lang.String, java.
lang.String)
       person._3
              ^
```

Для присваивания сразу нескольких значений в Scala используются кортежи, а не списки:

```
scala> val (x, y) = (1, 2)
x: Int = 1
y: Int =2
```

Так как кортежи имеют фиксированную длину, компилятор Scala оказывается в состоянии выполнить статическую проверку типов, опираясь на тип каждого значения в кортеже:

```
scala> val (a, b) = (1, 2, 3)
<console>:15: error: constructor cannot be instantiated to expected
type;
 found   : (T1, T2)
 required: (Int, Int, Int)
       val (a, b) = (1, 2, 3)
           ^
<console>:15: error: recursive value x$1 needs type
       val (a, b) = (1, 2, 3)
            ^
```

Получив некоторые базовые сведения, давайте объединим их и попробуем определить несколько классов.

Классы в Scala

Простейшие классы, имеющие только атрибуты, но не имеющие ни методов, ни конструкторов, в Scala могут объявляться одной строкой кода:

```
class Person(firstName: String, lastName: String)
```

Чтобы определить простой класс значений, не требуется определять его тело. Класс `Person`, объявленный выше, является общедоступным и содержит атрибуты `firstName` и `lastName`. Такой класс можно использовать прямо в консоли:

```
scala> class Person(val firstName: String, val lastName: String)
defined class Person

scala> val gump = new Person("Forrest", "Gump")
gump: Person = Person@7c6d75b6
```

Но чаще требуется нечто большее. Объектно-ориентированные классы смешивают данные и методы, определяющие их поведение. Давайте сконструируем полноценный класс. Назовем его `Compass`. Изначально компас направлен на север. Мы будем сообщать компасу о необходимости повернуть влево или вправо на 90 градусов и соответствующим образом изменить значение атрибута, определяющего направление. Ниже приводится полное определение класса:

scala/compass.scala
http://media.pragprog.com/titles/btlang/code/scala/compass.scala

```
class Compass{

  val directions = List("north" , "east" , "south" , "west" )
  var bearing =0

  print("Initial bearing: " )
  println(direction)

  def direction() = directions(bearing)

  def inform(turnDirection: String) {
    println("Turning " + turnDirection + ". Now bearing " + direction)
  }

  def turnRight() {
    bearing = (bearing + 1) % directions.size
    inform("right" )
  }

  def turnLeft() {
    bearing = (bearing + (directions.size - 1)) % directions.size
    inform("left" )
  }
}

val myCompass = new Compass

myCompass.turnRight
myCompass.turnRight

myCompass.turnLeft
myCompass.turnLeft
myCompass.turnLeft
```

Синтаксис определения классов достаточно прямолинеен, но имеет пару интересных особенностей. За определение переменных экземпляра (по крайней мере тех, что не передаются конструктору) и методов отвечает конструктор. В отличие от Ruby, в Scala все определения методов включают типы и имена параметров. Начальный блок кода не принадлежит никакому методу. Давайте разбираться:

```
class Compass{

  val directions = List("north" , "east" , "south" , "west" )
  var bearing =0

  print("Initial bearing: " )
  println(direction)
```

Весь блок кода, следующий за заголовком определения класса, в действительности является телом конструктора. В данном случае конструктор имеет список `List` направлений `directions` и текущий азимут `bearing`, который является всего лишь индексом в списке `directions`. Попытки выполнить поворот будут изменять азимут. Ниже определяется несколько вспомогательных методов, сообщающих текущий азимут:

```
def direction() = directions(bearing)

def inform(turnDirection: String) {
  println("Turning " + turnDirection + ". Now bearing " + direction)
}
```

Тело конструктора простирается на определения методов. Метод `direction` возвращает элемент списка `directions` с индексом `bearing`. В языке Scala поддерживается удобный альтернативный синтаксис определения однострочных методов, в соответствии с которым разрешается опускать фигурные скобки, ограничивающие тело метода.

Метод `inform` выводит сообщение всякий раз, когда пользователь выполняет поворот. Он принимает единственный параметр – направление поворота. Этот метод ничего не возвращает. Теперь рассмотрим методы, реализующие повороты.

```
def turnRight() {
  bearing = (bearing + 1) % directions.size
  inform("right" )
}

def turnLeft() {
  bearing = (bearing + (directions.size - 1)) % directions.size
  inform("left" )
}
```

Методы `turnLeft` и `turnRight` изменяют атрибут `bearing`, исходя из направления поворота. Оператор `%` выполняет деление по модулю. (Он производит целочисленное деление, отбрасывает частное и возвращает только остаток.) В результате поворот вправо добавляет единицу к азимуту, поворот влево вычитает единицу, после чего полученное значение переносится в начало или в конец допустимого диапазона.

Вспомогательные конструкторы

Только что вы познакомились с конструктором по умолчанию. Это блок кода, инициализирующий атрибуты и методы класса. Однако

имеется возможность определять альтернативные конструкторы. Взгляните на следующий класс Person с двумя конструкторами:

scala/constructor.scala
http://media.pragprog.com/titles/btlang/code/scala/constructor.scala

```
class Person(firstName: String) {
  println("Outer constructor" )
  def this(firstName: String, lastName: String){
    this(firstName)
    println("Inner constructor" )
  }
  def talk() = println("Hi" )
}

val bob = new Person("Bob" )
val bobTate = new Person("Bob" , "Tate" )
```

Класс имеет конструктор с единственным параметром, firstName, и метод с именем talk. Обратите внимание на метод this. Это – второй конструктор. Он принимает два параметра, firstName и lastName. В самом начале метод вызывает первичный конструктор this, передавая единственный параметр firstName.

Код, следующий за определением класса, создает два экземпляра класса Person – сначала с помощью первичного конструктора, а затем с помощью вспомогательного:

```
batate$ scala code/scala/constructor.scala
Outer constructor
Outer constructor
Inner constructor
```

Вот, собственно, и все. Вспомогательные конструкторы играют важную роль, позволяя реализовать широкий спектр шаблонов использования класса. А теперь посмотрим, как создавать методы классов.

Расширение классов

Пока что мы не видели в классах ничего необычного. Мы создали пару простых классов, не обладающих ничем, кроме атрибутов и методов. В этом разделе мы рассмотрим некоторые из способов взаимодействий с классами.

Объекты-компаньоны и методы класса

В Java и Ruby методы класса и методы экземпляра создаются в пределах общего объявления класса. В Java методы класса определяются

с помощью ключевого слова `static`. В Ruby используется форма `def self.class_method`. В языке Scala не применяется ни одна из этих стратегий. В определении класса объявляются только методы экземпляра. Когда для класса может быть создан только один экземпляр, такой класс объявляется с помощью ключевого слова `object` вместо `class`. Например:

scala/ring.scala
http://media.pragprog.com/titles/btlang/code/scala/ring.scala

```
object TrueRing {
  def rule = println("To rule them all" )
}

TrueRing.rule
```

Определение объекта `TrueRing` действует точно так же, как определение любого другого класса, но оно создает объект-одиночку (singleton object). В языке Scala допускается определять классы и объекты-одиночки с одинаковыми именами. Таким способом можно создавать методы класса внутри объявления объекта-одиночки, а методы экземпляров – внутри определения класса. В данном примере метод `rule` является методом класса. Такие объекты-одиночки в языке Scala называются объектами-компаньонами (companion objects).

Наследование

Наследование в Scala организуется довольно просто. Ниже приводится пример наследования класса `Person` классом `Employee`. Обратите внимание, что в классе `Employee` имеется дополнительное поле `number`, хранящее идентификационный номер работника:

scala/employee.scala
http://media.pragprog.com/titles/btlang/code/scala/employee.scala

```
class Person(val name: String) {
  def talk(message: String) = println(name + " says " + message)
  def id(): String = name
}

class Employee(override val name: String,
                        val number: Int) extends Person(name){
  override def talk(message: String){
    println(name + " with number " + number + " says " + message)
  }

  override def id():String = number.toString
```

```
}

val employee = new Employee("Yoda" , 4)
employee.talk("Extend or extend not. There is no try." )
```

В этом примере мы расширили базовый класс `Person` классом `Employee`. В новый класс `Employee` была добавлена новая переменная экземпляров `number`, а также переопределен метод `talk`. Основные необычные синтаксические конструкции здесь сосредоточены в определении конструктора класса. Обратите внимание на необходимость явно перечислить все параметры конструктора наследуемого класса `Person`, правда, при этом типы параметров можно опустить.

Ключевое слово `override` обязательно должно предшествовать элементам класса, которые наследуются из базового класса и переопределяются в текущем. Это ключевое слово убережет вас от создания новых методов в случае опечатки в именах. В целом объектно-ориентированная модель не преподносит никаких неожиданностей, но иногда я чувствую себя как Эдвард, пытающийся осторожно погладить хрупкую детскую игрушку. Идем дальше....

Трейты

Каждый объектно-ориентированный язык должен каким-то способом попытаться решить проблему использования одного объекта в разных ипостасях. Объект может представлять хранимый набор данных. Было бы нежелательно, чтобы объект знал, как сохранять двоичные блоки данных, например в MySQL. В языке C++ эта проблема решается с помощью множественного наследования, в Java используются интерфейсы, в Ruby – примесные классы, а в Scala используются трейты (traits). Трейт в языке Scala можно сравнить с примесным классом в Ruby, реализованным с помощью модулей. Или, если хотите, трейт можно считать неким подобием интерфейса в языке Java, но обладающего реализацией. Проще говоря, трейты можно считать классами с неполной реализацией. В идеале каждый трейт должен реализовать только какую-то одну важную функцию. Ниже приводится пример добавления трейта `Nice` в класс `Person`:

scala/nice.scala
http://media.pragprog.com/titles/btlang/code/scala/nice.scala

```
class Person(val name:String)

trait Nice {
  def greet() = println("Howdily doodily." )
```

```
}

class Character(override val name:String) extends Person(name) with Nice

val flanders = new Character("Ned" )
flanders.greet
```

В первой строке здесь определяется класс `Person`. Это очень простой класс, обладающий единственным атрибутом `name`. Далее следует определение трейта `Nice`. По сути, это – примесный класс. Он имеет единственный метод `greet`. Затем следует определение класса `Character`, подмешивающее трейт `Nice`. Теперь клиенты могут вызывать метод `greet` у любых экземпляров класса `Character`. Ниже приводится листинг, демонстрирующий работу этого примера:

```
batate$ scala code/scala/nice.scala
Howdily doodily.
```

Как видите, ничего сложного. Мы можем взять трейт `Nice` с методом `greet` и подмешать его в любой класс, чтобы снабдить его методом `greet`.

Что мы узнали в первый день

Сегодня мы охватили достаточно большой объем сведений из-за необходимости познакомиться с двумя разными парадигмами программирования, поддерживаемыми в одном языке. Как показали сегодняшние исследования, Scala поддерживает объектно-ориентированные концепции, программы на этом языке выполняются под управлением JVM и могут использовать существующие библиотеки для Java. Синтаксис языка Scala имеет много общего с синтаксисом языка Java и также является статически типизированным языком со строгим контролем типов. Но Мартин Одерски (Martin Odersky) создавал язык Scala с целью объединения двух парадигм программирования: объектно-ориентированной и функциональной. Функциональные концепции, с которыми мы познакомимся во второй день, упрощают создание многозадачных приложений.

Система статических типов в языке Scala дополнительно поддерживает возможность вывода типов. От пользователя не требуется все время явно указывать типы в объявлениях переменных, потому что компилятор Scala часто способен самостоятельно автоматически определять типы, пользуясь синтаксическими подсказками. Компилятор способен также выполнять приведение типов, например преоб-

разовывать целые числа в строки, обеспечивая неявное преобразование типов, когда это имеет смысл.

Выражения в языке Scala действуют практически так же, как в других языках, но они накладывают некоторые дополнительные ограничения. Большинство условных инструкций принимает только значения логического типа – значения `0` и `Nil` вообще не могут в них использоваться, так как они не могут замещаться значениями `true` или `false`. Циклы и управляющие конструкции в языке Scala тоже не имеют существенных отличий. Кроме того, Scala поддерживает некоторые дополнительные типы данных, такие как кортежи (списки фиксированной длины, способные одновременно хранить объекты разных типов) и диапазоны (фиксированные, упорядоченные последовательности чисел).

Классы в Scala близко напоминают классы в Java, но они не поддерживают методы класса. Для добавления методов класса в Scala используются *объекты-компаньоны* с именами, совпадающими с именами соответствующих классов. Аналогом примесных классов в Ruby и интерфейсов в Java в Scala являются трейты.

Второй день мы полностью посвятим функциональным особенностям языка Scala. Мы познакомимся с блоками кода, коллекциями, неизменяемыми переменными и некоторыми встроенными методами, такими как `foldLeft`.

День 1: задания для самостоятельного решения

В первый день мы познакомились со многими основами языка Scala, но это была достаточно хорошо известная нам территория. Многие объектно-ориентированные понятия должны быть хорошо знакомы вам. Следующие далее упражнения чуть сложнее тех, что предлагались выше в этой книге, но я уверен – вы справитесь с ними.

Найдите:

- описание Scala API;
- сравнительное описание Java и Scala;
- обсуждение `val` и `var`.

Практические задания:

- напишите игру «крестики-нолики», позволяющую вставлять в клетки игрового поля символы X, O и пробелы и определяющую победителя или ничью; используйте классы, если посчитаете необходимым;
- дополнительное задание: добавьте в игру возможность игры человека с человеком.

5.3. День 2: Обрезка кустарников и другие новые хитрости

В фильме «Эдвард Руки-ножницы» все самое интересное начинается, когда Эдвард обнаруживает, что вдали от дома на холме его уникальные способности могут обеспечить ему особое положение в обществе.

Любой, имеющий более или менее продолжительный опыт программирования, мог наблюдать ход истории развития языков прежде. Когда объектно-ориентированная парадигма была в диковинку, массы программистов не смогли принять Smalltalk, потому что сама парадигма была еще слишком нова. Им необходим был язык, который позволил бы продолжать программировать в процедурном стиле и попутно экспериментировать с объектно-ориентированными идеями. В C++ новые объектно-ориентированные особенности благополучно сосуществовали с процедурными особенностями языка C. В результате программисты получили возможность пользоваться новыми инструментами в старом, привычном контексте.

Теперь настал момент опробовать Scala как функциональный язык. Некоторые из его новых возможностей поначалу будут казаться неудобными, но идеи, заложенные в них, чрезвычайно важны. Они образуют основу для конструкций поддержки параллельного выполнения, с которыми мы познакомимся в третий день. Начнем со знакомства с простой функцией:

```
scala> def double(x:Int):Int = x * 2
double: (Int)Int

scala> double(4)
res0: Int =8
```

Определение функции мало чем отличается от определения функции в Ruby. Определение функций и методов в Scala начинается с ключевого слова `def`. За именем функции следуют имена параметров и их типы. Вслед за списком параметров может следовать необязательное определение типа возвращаемого значения. Впрочем, компилятор Scala часто оказывается в состоянии сам определить тип возвращаемого значения.

Чтобы вызвать функцию, достаточно просто указать ее имя и список аргументов. Обратите внимание, что, в отличие от Ruby, круглые скобки в вызовах функций являются обязательными.

Это был пример определения однострочного метода. Объявления методов также могут иметь блочную форму:

```
scala> def double(x:Int):Int = {
     |   x * 2
     | }
double: (Int)Int

scala> double(6)
res3: Int = 12
```

В данной форме знак = после объявления типа `Int` возвращаемого значения является обязательным. Забывчивость в этом случае может привести к неприятностям. В этом примере можно наблюдать все основные элементы объявлений функций. Иногда вам будут встречаться некоторые незначительные отклонения, такие как отсутствие параметров, но именно эта форма будет встречаться вам чаще всего.

Теперь перейдем к переменным, которые вы наверняка будете использовать внутри функций. Вам следует внимательно изучить особенности жизненного цикла переменных, если хотите понять модель функционального программирования.

var и val

Scala основана на виртуальной машине Java и имеет тесные связи с Java. Такая архитектура языка накладывает определенные ограничения. Но, с другой стороны, возможность пользоваться наработками, сделанными в последние 15–20 лет, дает языку Scala некоторые преимущества. Вы увидите, что в Scala основной упор сделан на упрощение разработки многозадачных приложений. Но никакие, даже самые лучшие в мире механизмы параллельных вычислений не помогут вам, если вы не будете следовать основным архитектурным принципам. Изменяемое состояние – зло. Всегда используйте неизменяемые переменные, если есть вероятность появления конфликтов, порождаемых изменяемостью состояния. В Java для этой цели можно использовать ключевое слово `final`. В Scala неизменяемые переменные объявляются с помощью ключевого слова `val`:

```
scala> var mutable = "I am mutable"
mutable: java.lang.String = I am mutable

scala> mutable = "Touch me, change me..."
mutable: java.lang.String = Touch me, change me...

scala> val immutable = "I am not mutable"
immutable: java.lang.String = I am not mutable

scala> immutable = "Can't touch this"
```

```
<console>:5: error: reassignment to val
       immutable = "Can't touch this"
                 ^
```

Итак, переменные, объявленные с помощью ключевого слова `var`, позволяют изменять их значения; переменные, объявленные с помощью ключевого слова `val`, не дают такой возможности. В консоли, даже при использовании ключевого слова `val`, разрешается многократно переопределять одну и ту же переменную. Это сделано для удобства. Но за пределами консоли попытка переопределить значение `val` приведет к ошибке.

Изменяемые переменные, объявляемые как `var`, были введены в язык Scala для поддержки программирования в традиционном, императивном стиле, но пока вы только изучаете этот язык, старайтесь избегать применения `var`, так как это упростит реализацию многозадачного кода. Эта базовая философия является ключевым элементом, отличающим функциональное программирование от объектно-ориентированного: *изменяемость состояния ограничивает параллельные вычисления*.

А теперь перейдем к моей самой любимой особенности функциональных языков — коллекциям.

Коллекции

Разработчиками функциональных языков накоплен богатейший опыт реализации эффективных коллекций. Один из первых функциональных языков, Lisp, был основан на идее представления всего сущего в виде списков. Даже название этого языка происходит от «LISt Processing» (обработка списков). Функциональные языки позволяют легко конструировать сложнейшие структуры, содержащие данные и программный код. Основной разновидностью коллекций в языке Scala являются списки, множества и ассоциативные массивы (иногда их называют отображениями).

Списки

В большинстве функциональных языков основной структурой данных является список. В Scala списки, значения типа `List`, являются упорядоченными коллекциями элементов одного типа, с произвольным доступом. Попробуйте ввести следующие списки в консоли:

```
scala> List(1, 2, 3)
res4: List[Int] = List(1, 2, 3)
```

В первую очередь обратите внимание на возвращаемое значение: `List[Int] = List(1, 2, 3)`. Здесь виден не только тип самой структуры данных, но и типы элементов внутри списка. Ниже показано, как выглядит список строк:

```
scala> List("one", "two", "three")
res5: List[java.lang.String] = List(one, two, three)
```

Если вам кажется, что здесь наблюдается влияние языка Java, то вы будете правы. В языке Java имеется особенность под названием Generics, позволяющая включать в один массив или список элементы разных типов. Давайте посмотрим, что получится, если попытаться создать список, содержащий строки и целые числа:

```
scala> List("one", "two", 3)
res6: List[Any] = List(one, two, 3)
```

В результате получился список значений типа `Any`, который является всеобъемлющим типом данных в языке Scala. Ниже показано, как обращаться к элементам списка:

```
scala> List("one", "two", 3)(2)
res7: Any =3

scala> List("one", "two", 3)(4)
java.util.NoSuchElementException: head of empty list
        at scala.Nil$.head(List.scala:1365)
        at scala.Nil$.head(List.scala:1362)
        at scala.List.apply(List.scala:800)
        at .<init>(<console>:5)
        at .<clinit>(<console>)
        at RequestResult$.<init>(<console>:3)
        at RequestResult$.<clinit>(<console>)
        at RequestResult$result(<console>)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Met...
```

Операция доступа к списку реализована в виде функции, поэтому вместо квадратных скобок `[]` используются круглые `()`. Нумерация индексов списков в Scala начинается с 0, то есть так же, как в Java и Ruby. В отличие от Ruby, попытка обратиться к элементу за пределами списка вызывает исключение.

Можете попробовать использовать отрицательные индексы. Ранние версии возвращали первый элемент списка:

```
scala> List("one", "two", 3)(-1)
res9: Any = one

scala> List("one", "two", 3)(-2)
```

```
res10: Any = one

scala> List("one", "two", 3)(-3)
res11: Any = one
```

Так как такое поведение противоречит исключению `NoSuchElement`, порождаемому в ответ на попытку использовать слишком большой индекс, в версии 2.8.0 оно было исправлено, и теперь генерируется исключение `java.lang.IndexOutOfBoundsException`.

И последнее замечание. Значение `Nil` в Scala – это пустой список:

```
scala> Nil
res33: Nil.type = List()
```

Мы будем использовать списки как основной строительный блок, когда перейдем к обсуждению блоков кода, а пока отложим списки в сторону, потому что далее я хочу познакомить вас с двумя другими типами коллекций.

Множества

Множество напоминает список, но, в отличие от списков, множества являются неупорядоченными коллекциями. Создать новое множество можно с помощью ключевого слова `Set`:

```
scala> val animals = Set("lions", "tigers", "bears")
animals: scala.collection.immutable.Set[java.lang.String]=
    Set(lions, tigers, bears)
```

Добавление элементов в множество и удаление их из множества выполняются очень просто:

```
scala> animals + "armadillos"
res25: scala.collection.immutable.Set[java.lang.String]=
    Set(lions, tigers, bears, armadillos)

scala> animals - "tigers"
res26: scala.collection.immutable.Set[java.lang.String] =
    Set(lions, bears)
```

```
scala> animals + Set("armadillos", "raccoons")
<console>:6: error: type mismatch;
 found : scala.collection.immutable.Set[java.lang.String]
 required: java.lang.String
       animals + Set("armadillos", "raccoons")
                 ^
```

Имейте в виду, что операции с множествами являются неразрушающими. Каждая операция создает новое множество, не изменяя

существующее. По умолчанию множества являются неизменяемыми структурами данных. Как показано в примерах выше, с помощью операторов + и - можно добавлять в множества и удалять из множеств отдельные элементы, но с их помощью нельзя выполнять операции над двумя множествами, как это допускается в Ruby. Для получения объединения или разности множеств в Scala следует использовать ++ и --:

```
scala> animals ++ Set("armadillos", "raccoons")
res28: scala.collection.immutable.Set[java.lang.String]=
    Set(bears, tigers, armadillos, raccoons, lions)

scala> animals -- Set("lions", "bears")
res29: scala.collection.immutable.Set[java.lang.String] =
    Set(tigers)
```

Для вычисления пересечения двух множеств (множество общих элементов) можно использовать оператор **[1]:

```
scala> animals ** Set("armadillos", "raccoons", "lions", "tigers")
res1: scala.collection.immutable.Set[java.lang.String] =
    Set(lions, tigers)
```

В отличие от списков, множества являются неупорядоченными коллекциями. Это влияет на правила определения равенства множеств и списков:

```
scala> Set(1, 2, 3) == Set(3, 2, 1)
res36: Boolean = true

scala> List(1, 2, 3) == List(3, 2, 1)
res37: Boolean = false
```

На этом закончим обсуждение множеств и перейдем к ассоциативным массивам.

Ассоциативные массивы

Ассоциативный массив (тип `Map`) – это множество пар ключ/значение, подобно типу `Hash` в Ruby. Синтаксис ассоциативных массивов многим из вас покажется знакомым:

```
scala> val ordinals = Map(0 -> "zero", 1 -> "one", 2 -> "two")
ordinals: scala.collection.immutable.Map[Int,java.lang.String]=
    Map(0 -> zero, 1 -> one, 2 -> two)

scala> ordinals(2)
res41: java.lang.String = two
```

---

[1] В версиях Scala 2.8.0 и выше используйте оператор `&`, так как оператор ** в них объявлен устаревшим.

Подобно спискам и множествам, ассоциативные массивы в Scala определяются с помощью ключевого слова `Map`. Ключи и значения разделяются оператором `->`. Мы только что воспользовались синтаксическим сахаром языка Scala, упрощающим создание ассоциативных массивов. Давайте воспользуемся другой разновидностью ассоциативных массивов и определим типы ключей и значений явно:

```
scala> import scala.collection.mutable.HashMap
import scala.collection.mutable.HashMap

scala> val map = new HashMap[Int, String]
map: scala.collection.mutable.HashMap[Int,String] = Map()

scala> map += 4 -> "four"

scala> map += 8 -> "eight"

scala> map
res2: scala.collection.mutable.HashMap[Int,String]=
    Map(4 -> four, 8 -> eight)
```

Прежде всего мы импортировали библиотеки поддержки изменяемого типа `HashMap` в Scala. Значения внутри такого ассоциативного массива могут изменяться. Далее, мы объявили неизменяемую переменную `map`. Это означает, что *ссылка* на ассоциативный массив не может изменяться. Обратите внимание: здесь мы также указали типы ключей и значений. Наконец, мы добавили в массив несколько пар ключ/значение и вывели результат.

Следующий пример показывает, что произойдет, если попытаться добавить в массив данные других типов:

```
scala> map += "zero" -> 0
<console>:7: error: overloaded method value += with alternatives (Int)
map.MapTo
    <and> ((Int, String))Unit cannot be applied to ((java.lang.String, Int))
       map += "zero" ->0
           ^
```

Как и ожидалось, была сгенерирована ошибка. Проверка типов выполняется на этапе компиляции, если это возможно, но она также выполняется и на этапе выполнения. Итак, мы познакомились с основами коллекций, а теперь перейдем к обсуждению более тонких особенностей.

Типы Any и Nothing

Прежде чем перейти к знакомству с анонимными функциями, необходимо поближе познакомиться с иерархией классов в языке Scala.

При организации взаимодействий с программным кодом на Java из программ на языке Scala вам обязательно потребуется знакомство с иерархией классов в Java. Однако вам потребуется также некоторое знакомство с типами в языке Scala. Тип `Any` – это корневой класс в иерархии классов языка Scala. Он часто является источником непонимания, поэтому запомните, что любой тип в языке Scala является потомком класса `Any`.

Аналогично `Nothing` является подтипом любого типа. То есть функция, возвращающая коллекцию, может вернуть `Nothing`, и это значение будет соответствовать данной функции. На рис. 5.1 изображена диаграмма, отражающая все эти взаимоотношения между типами. Любой тип наследует `Any`, а тип `Nothing` наследует любые типы.

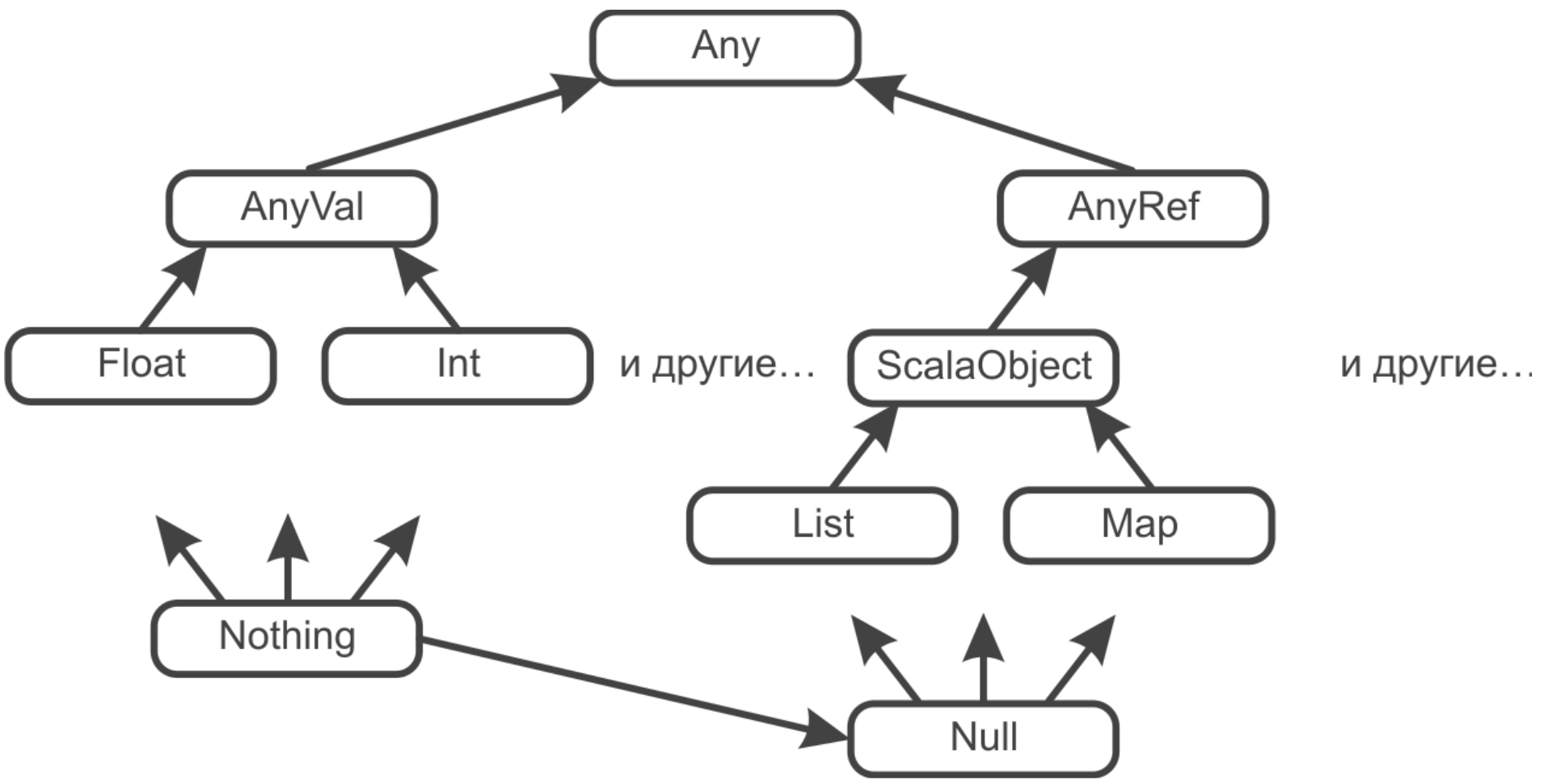

Рис. 5.1 ❖ Типы Any и Nothing

В Scala существует несколько нюансов, касающихся понятия «пустое значение» (nil). `Null` – это трейт, а `null` – его экземпляр, действующий подобно `null` в Java и означающий пустое значение. Пустая коллекция – это `Nil`. Напротив, `Nothing` – это трейт, являющийся подтипом любого типа. Тип `Nothing` не имеет экземпляров, поэтому его нельзя разыменовать, как `Null`. Например, метод, возбуждающий исключение `Exception`, имеет возвращаемое значение типа `Nothing` в том смысле, что вообще не имеет значения.

Помните эти правила, и удача не отвернется от вас. Теперь вы готовы приступить к выполнению операций с коллекциями с помощью функций высшего порядка.

Коллекции и функции

Так как мы приступаем к знакомству с языками, основывающимися на функциональном подходе к программированию, я хочу формализовать некоторые понятия, с которыми мы будем работать далее. Первым таким понятием являются *функции высшего порядка* (higher-order functions).

Коллекции в языке Scala становятся особенно интересными при использовании их совместно с функциями высшего порядка. Так же как Ruby и Io, Scala позволяет передавать функции в цикл `foreach`. Суть в том, что `foreach` в действительности является функцией высшего порядка. Для непосвященных отмечу, что функцией высшего порядка называется функция, возвращающая или принимающая другие функции. Если говорить точнее, функцией высшего порядка называется функция, которая принимает другие функции в виде параметров или возвращает функцию в виде возвращаемого значения. В связи с этим создание функций, использующих другие функции, является важнейшим понятием в семействе функциональных языков, влияющим на приемы программирования.

Язык Scala обладает мощной поддержкой функций высшего порядка. У нас не так много времени, чтобы вдаваться в обсуждение таких тем, как частично примененные функции (partially applied functions) или карринг (currying), но мы познакомимся с особенностями передачи простых функций, которые часто называют *блоками кода*, в виде параметров. Вы узнаете, как присваивать функции любым переменным или параметрам, как передавать их другим функциям и возвращать из функций. Основное внимание мы уделим передаче анонимных функций во входных параметрах как одного из наиболее интересных методов обработки коллекций.

foreach

Первая функция, которую мы исследуем, – функция `foreach` – основной инструмент выполнения итераций в Scala. Как и в языке Io, метод коллекций `foreach` принимает блок кода в качестве параметра. В Scala этот блок кода выражается в форме `variableName => yourCode`, как показано ниже:

```
scala> val list = List("frodo", "samwise", "pippin")
list: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.foreach(hobbit => println(hobbit))
frodo
```

```
samwise
pippin
```

Здесь `hobbit => println(hobbit)` – это анонимная функция, то есть функция без имени. В объявлении присутствует аргумент, слева от `=>`, и программный код справа. Функция `foreach` будет вызывать анонимную функцию и передавать ей каждый элемент, имеющийся в списке. Как вы уже наверняка догадались, тот же самый прием можно использовать для обработки множеств и ассоциативных массивов, только при работе с этими двумя типами коллекций порядок обхода элементов не гарантируется:

```
scala> val hobbits = Set("frodo", "samwise", "pippin")
hobbits: scala.collection.immutable.Set[java.lang.String]=
    Set(frodo, samwise, pippin)

scala> hobbits.foreach(hobbit => println(hobbit))
frodo
samwise
pippin

scala> val hobbits = Map("frodo" -> "hobbit",
  "samwise" -> "hobbit", "pippin" -> "hobbit")
hobbits: scala.collection.immutable.Map[java.lang.String,java.lang.
String] =
    Map(frodo -> hobbit, samwise -> hobbit, pippin -> hobbit)

scala> hobbits.foreach(hobbit => println(hobbit))
(frodo,hobbit)
(samwise,hobbit)
(pippin,hobbit)
```

Разумеется, при выполнении итераций по содержимому ассоциативного массива функции будут передаваться кортежи. Как вы наверняка помните, к элементам кортежа можно обращаться следующим способом:

```
scala> hobbits.foreach(hobbit => println(hobbit._1))
frodo
samwise
pippin

scala> hobbits.foreach(hobbit => println(hobbit._2))
hobbit
hobbit
hobbit
```

С помощью анонимных функций можно выполнять намного более сложные операции, чем простой обход значений. Далее мы познако-

мимся с дополнительными особенностями, а затем я познакомлю вас с другими, не менее интересными приемами использования функций в сочетании с коллекциями.

Подробнее о методах списков

Далее я коротко перечислю некоторые основные методы типа `List`. Эти методы пригодятся вам для реализации обхода элементов списков вручную. Сначала познакомимся с методами, позволяющими проверить наличие элементов в списке и длину списка:

```
scala> list
res23: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.isEmpty
res24: Boolean = false

scala> Nil.isEmpty
res25: Boolean = true

scala> list.length
res27: Int = 3

scala> list.size
res28: Int = 3
```

Обратите внимание, что размер списка можно проверить с помощью двух методов, `length` и `size`. Кроме того, запомните, что `Nil` – это пустой список. Как и в Prolog, в языке Scala имеется возможность выделить «голову» и «хвост» списка, что может пригодиться в рекурсивных алгоритмах.

```
scala> list.head
res34: java.lang.String = frodo

scala> list.tail
res35: List[java.lang.String] = List(samwise, pippin)

scala> list.last
res36: java.lang.String = pippin

scala> list.init
res37: List[java.lang.String] = List(frodo, samwise)
```

Вот это сюрприз! Методы `head` и `tail` можно использовать для организации рекурсивного обхода с начала списка, а методы `last` и `init` – с конца. Мы еще вернемся к проблеме рекурсии. А пока закончим

знакомство с основами демонстрацией нескольких вспомогательных методов:

```
scala> list.reverse
res29: List[java.lang.String] = List(pippin, samwise, frodo)

scala> list.drop(1)
res30: List[java.lang.String] = List(samwise, pippin)

scala> list
res31: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.drop(2)
res32: List[java.lang.String] = List(pippin)
```

Эти методы действуют именно так, как можно было бы догадаться из их названий. Метод `reverse` возвращает список, элементы в котором следуют в обратном порядке, а метод `drop(n)` возвращает список, из которого удалены первые `n` элементов. Ни один из этих методов не изменяет оригинального списка.

count, map, filter и другие методы

Как и в Ruby, в языке Scala имеется множество других функций для выполнения операций со списками. Списки можно фильтровать по определенному условию, сортировать с учетом заданных критериев, создавать другие списки на основе имеющихся элементов и вычислять агрегатные значения:

```
scala> val words = List("peg", "al", "bud", "kelly")
words: List[java.lang.String] = List(peg, al, bud, kelly)

scala> words.count(word => word.size > 2)
res43: Int = 3

scala> words.filter(word => word.size > 2)
res44: List[java.lang.String] = List(peg, bud, kelly)

scala> words.map(word => word.size)
res45: List[Int] = List(3, 2, 3, 5)

scala> words.forall(word => word.size > 1)
res46: Boolean = true

scala> words.exists(word => word.size > 4)
res47: Boolean = true

scala> words.exists(word => word.size > 5)
res48: Boolean = false
```

В этом примере мы сначала создали список. Затем подсчитали количество слов в списке, содержащих более двух символов. Метод `count` вызывает блок кода `word => word.size > 2`, вычисляет значение выражения `word.size > 2` для каждого элемента списка и подсчитывает количество элементов, для которых это выражение вернуло значение `true`.

Точно так же `words.filter(word => word.size > 2)` возвращает список всех слов, содержащих более двух символов, действуя подобно методу `select` в языке Ruby. Используя тот же шаблон, метод `map` создает список значений размеров всех слов в оригинальном списке, метод `forall` возвращает `true`, если блок кода вернет `true` для каждого элемента в списке, а метод `exists` вернет `true`, если блок кода вернет `true` хотя бы для одного элемента.

Иногда бывает желательно обобщить некоторую операцию, выполняемую блоком кода. Например, ниже показано, как можно отсортировать список традиционным способом:

```
scala> words.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).
toLowerCase)
res49: List[java.lang.String] = List(al, bud, kelly, peg)
```

Здесь используется блок кода, принимающий два параметра, `s` и `t`. Метод `sort`[1] позволяет сравнивать два аргумента любым желаемым способом. В примере выше мы приводим символы к нижнему регистру[2] и сравниваем их. В результате получается сортировка без учета регистра символов. Этот же метод можно использовать для сортировки элементов списка по их размерам:

```
scala> words.sort((s, t) => s.size < t.size)
res50: List[java.lang.String] = List(al, bud, peg, kelly)
```

Благодаря блоку кода можно организовать сортировку[3] по любому желаемому критерию. А теперь познакомимся с более сложным методом – `foldLeft`.

foldLeft

Метод `foldLeft` в языке Scala близко напоминает метод `inject` в Ruby. Этому методу передаются начальное значение и блок кода.

---

1 В версии Scala 2.8.0 метод `sort` объявлен устаревшим. Используйте вместо него метод `sortWith`.

2 В версии Scala 2.8.0 метод `toLowerCase` объявлен устаревшим. Используйте вместо него метод `toLower`.

3 В версии Scala 2.8.0 метод `sort` объявлен устаревшим. Используйте вместо него метод `sortWith`.

Метод `foldLeft` будет передавать блоку кода каждый элемент массива и еще одно значение. Вторым значением может быть или начальное значение (для первого вызова), или результат выполнения блока кода (для последующих вызовов). Ниже приводятся примеры использования двух версий метода. Первая версия – оператор `/:`, который вызывается, как `initialValue /: codeBlock`:

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> val sum = (0 /: list) {(sum, i) => sum + i}
sum: Int =6
```

Мы уже рассматривали, как действует эта операция в Ruby, однако не будет лишним вспомнить порядок ее работы.

- Мы вызываем оператор и передаем ему начальное значение и блок кода. Блок кода принимает два аргумента, `sum` и `i`.
- В первой итерации оператор `/:` передаст блоку кода начальное значение 0 и первый элемент списка `list`, `1`. `sum` имеет значение `0`, `i`-й (то есть 0-й) элемент имеет значение `1`, соответственно, результатом блока кода будет `0 + 1`, то есть значение `1`.
- В следующей итерации оператор `/:` передаст блоку кода `1`, результат, полученный в предыдущей итерации, и значение следующего элемента списка. Так как прежний результат имеет значение `1`, а следующий элемент списка имеет значение `2`, результатом блока кода будет `3`.
- Наконец, в последней итерации оператор `/:` передаст блоку кода `3`, результат, полученный в предыдущей итерации, и значение `3` последнего элемента списка, результатом блока кода будет `6`.

Синтаксис другой версии `foldLeft` может показаться странным. Эта версия создана с применением такого приема, как *карринг* (currying). Функциональные языки поддерживают карринг, чтобы дать возможность преобразовывать функции с несколькими параметрами в несколько функций с собственными списками параметров. Поближе с этим приемом мы познакомимся в главе 8 «Haskell». А пока просто знайте, что под покровом карринга скрывается не единственная функция, а целая композиция из функций. Однако, несмотря на различия в механике и синтаксисе, результат получается тем же:

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> list.foldLeft(0)((sum, value) => sum + value)
res54: Int =6
```

Обратите внимание, что в вызов функции `list.foldLeft(0)((sum, value) => sum + value)` передаются два списка параметров. Это является следствием карринга, о котором упоминалось выше. Версии этого метода вы увидите во всех остальных языках, обсуждаемых далее в данной книге.

Что мы узнали во второй день

В первый день мы знакомились с объектно-ориентированными особенностями языка, многие из которых многим из вас уже знакомы. Во второй день язык Scala был представлен, как язык функционального программирования.

Мы начали со знакомства с простой функцией. Язык Scala имеет гибкий синтаксис определения функций. Компилятор часто оказывается в состоянии автоматически определить тип возвращаемого значения. Тело функции может иметь однострочную форму или форму блока, и списки параметров могут изменяться.

Затем были перечислены разные коллекции. В языке Scala поддерживаются три основных типа коллекций: списки, ассоциативные массивы и множества. Множество – это коллекция уникальных объектов. Список – это упорядоченная коллекция объектов. Наконец, ассоциативный массив – это массив пар ключ/значение. Как и в Ruby, в Scala так же поддерживается возможность объединять блоки кода с коллекциями различных типов. Мы на скорую руку познакомились с некоторыми методами из API-коллекций, наиболее характерными для парадигмы функционального программирования.

Списки в языке Scala имеют Lisp-подобные методы `head` и `tail`, позволяющие извлекать «голову» (первый элемент) и «хвост» (остальную часть) списка. Мы также познакомились с методами `count`, `empty` и `first`, назначение которых очевидно из названий. Но самые мощные методы, имеющиеся в арсенале, способны принимать блоки кода.

Мы узнали, как выполнять итерации с помощью `foreach` и как использовать `filter` для выбора из списка элементов, отвечающих определенным критериям. Мы также познакомились с применением `foldLeft` для накопления результатов, получаемых в процессе итераций через коллекцию, например для получения суммы всех элементов списка.

Функциональное программирование в значительной степени заключается в управлении коллекциями с применением высокоуровневых конструкций вместо итераций в стиле языка Java. Мы продолжим приобретать эти навыки на следующий день, когда приступим

к знакомству с инструментами параллельного программирования, приемами обработки XML и напишем простой практический пример.

День 2: задания для самостоятельного решения

Теперь, после еще более глубокого погружения в Scala, вы начнете замечать его функциональные аспекты. Всякий раз имея дело с функциями, вам придется сталкиваться с коллекциями. Упражнения ниже помогут вам лучше освоить коллекции и некоторые функции.

Найдите:

- обсуждение операций с файлами в Scala;
- чем отличаются замыкания от блоков кода;

Практические задания:

- с помощью `foldLeft` вычислите сумму длин строк в списке;
- определите трейт `Censor` с методом, замещающим бранные слова *Shoot* и *Darn* словами *Pucky* и *Beans*; для хранения бранных слов и их альтернатив используйте ассоциативный массив;
- реализуйте загрузку бранных слов и их альтернатив из файла.

5.4. День 3: Художественная стрижка

Непосредственно перед кульминацией в фильме «Эдвард Руки-ножницы» Эдвард учится владеть своими ножницами, как истинный художник. Он подстригает кусты, придавая им форму динозавров, делает стрижки женщинам, действуя почти как сам сэр Видал Сассун (Vidal Sassoon), и даже выстригает из кустарника скульптурную композицию, отображающую семью, которая его приютила. В языке Scala мы встретились с некоторыми не совсем удобными особенностями, но когда этот язык используется по правильному назначению, он показывает захватывающие возможности. Такие сложные задачи, как интерпретация данных в формате XML и параллельные вычисления, в Scala становятся почти рутиной. Так давайте же познакомимся с этими возможностями.

XML

Одной из типичных задач современного программирования является обработка данных в формате XML (Extensible Markup Language – расширяемый язык разметки). Язык Scala сделал огромный шаг вперед в этом направлении, превратив элементы языка XML в обычные программные конструкции. Вы можете выражать данные в формате XML так же просто, как любые строки:

```
scala> val movies =
     | <movies>
     |     <movie genre="action">Pirates of the Caribbean</movie>
     |     <movie genre="fairytale">Edward Scissorhands</movie>
     | </movies>
movies: scala.xml.Elem =
<movies>
          <movie genre="action">Pirates of the Caribbean</movie>
          <movie genre="fairytale">Edward Scissorhands</movie>
</movies>
```

Определив переменную `movies` как фрагмент разметки XML, вы сможете обращаться к различным ее элементам непосредственно.

Например, извлечь весь вложенный текст можно, как показано ниже:

```
scala> movies.text
res1: String=

          Pirates of the Caribbean
          Edward Scissorhands
```

Как видите, получить весь вложенный текст не составляет никакого труда. Но мы не ограничены одной только возможностью извлечения всего текста сразу. Мы можем проявить большую избирательность. В Scala встроен язык запросов, напоминающий XPath – язык поиска элементов разметки XML. Но, так как комбинация символов // в Scala начинает однострочные комментарии, вместо нее используются \ и \\. Символ обратного слэша используется для поиска узлов верхнего уровня, например:

```
scala> val movieNodes = movies \ "movie"
movieNodes: scala.xml.NodeSeq=
  <movie genre="action">Pirates of the Caribbean</movie>
  <movie genre="fairytale">Edward Scissorhands</movie>
```

В этом примере выполняется поиск элементов `movie` в разметке XML. Существует также возможность поиска элементов по индексам:

```
scala> movieNodes(0)
res3: scala.xml.Node = <movie genre="action">Pirates of the Caribbean</
movie>
```

Здесь мы извлекли элемент с нулевым порядковым номером, то есть элемент с текстом `Pirates of the Caribbean`. Существует также возможность извлекать атрибуты отдельных узлов XML, применяя символ `@`. Например, извлечь атрибут `genre` из первого элемента можно следующим образом:

```
scala> movieNodes(0) \ "@genre"
res4: scala.xml.NodeSeq = action
```

В примерах выше демонстрируется лишь вершина айсберга возможностей, но основная идея, я думаю, вам понятна. Если сюда добавить еще возможность сопоставления с образцом в стиле языка Prolog, картина становится еще более захватывающей. Далее мы рассмотрим пример сопоставления образцов с простыми строками.

Сопоставление с образцом

Средства сопоставления с образцом позволяют выполнять код, исходя из наличия определенных данных. Программисты на Scala часто используют сопоставление с образцом, особенно при обработке разметки XML или передаче сообщений между потоками выполнения.

Ниже приводится простейшая форма сопоставления с образцом:

scala/chores.scala
http://media.pragprog.com/titles/btlang/code/scala/chores.scala

```
def doChore(chore: String): String = chore match {
    case "clean dishes" => "scrub, dry"
    case "cook dinner" => "chop, sizzle"
    case _ => "whine, complain"
}

println(doChore("clean dishes" ))
println(doChore("mow lawn" ))
```

Здесь определяются два вида работ по хозяйству: мойка посуды (clean dishes) и приготовление пищи (cook dinner). Каждый вид работ оформлен в виде блока кода. В данном случае блоки кода просто возвращают строки. Последний вид работ мы определили с помощью универсального образца (_). Scala выполнит блок кода, соответствующий первому найденному совпадению, или вернет строку «whine, complain» (пожаловаться), если совпадение не будет найдено, как показано ниже:

```
>> scala chores.scala
scrub, dry
whine, complain
```

Ограничители

Операция сопоставления с образцом имеет также некоторые дополнительные особенности. В языке Prolog сопоставление с образцом часто связывается с некоторыми условиями. Чтобы реализовать вы-

численение факториала числа на языке Scala, необходимо определить условие в ограничителе (guard) внутри выражения `match`:

scala/factorial.scala
http://media.pragprog.com/titles/btlang/code/scala/factorial.scala

```
def factorial(n: Int): Int = n match {
    case 0 => 1
    case x if x > 0 => factorial(n - 1) *n
}

println(factorial(3))
println(factorial(0))
```

Первый образец совпадает со значением `0`, но во втором используется ограничитель `if x > 0`. Он совпадает с любым значением `x`, если выполняется условие `x > 0`. В ограничителях можно указывать самые разные условия. В выражениях сопоставления с образцом можно также использовать регулярные выражения и типы. Ниже, при обсуждении приемов параллельного программирования, вы увидите пример, в котором определяются пустые классы и используются в качестве сообщений.

Регулярные выражения

Scala обладает первоклассной поддержкой регулярных выражений. Строковый метод `.r` может преобразовать любую строку в объект регулярного выражения. В следующем примере демонстрируется применение регулярного выражения, совпадающего с буквой `F` в любом регистре, верхнем или нижнем, если она находится в начале строки.

```
scala> val reg = """^(F|f)\w*""".r
reg: scala.util.matching.Regex = ^(F|f)\w*

scala> println(reg.findFirstIn("Fantastic"))
Some(Fantastic)

scala> println(reg.findFirstIn("not Fantastic"))
None
```

В этом примере сначала определяется простая строка, для чего используется форма определения с тройными кавычками """, позволяющая вставлять в программный код многострочный текст. Затем вызывается метод `.r`, преобразующий строку в объект регулярного выражения. После этого вызовом метода `findFirstIn` этого объекта выполняется поиск первого совпадения.

```
scala> val reg = "the".r
reg: scala.util.matching.Regex = the

scala> reg.findAllIn("the way the scissors trim the hair and the shrubs")
res9: scala.util.matching.Regex.MatchIterator = non-empty iterator
```

В этом примере мы конструируем регулярное выражение и с помощью его метода `findAllIn` находим все совпадения со словом «the» в строке «the way the scissors trim the hair and the shrubs». При желании можно было бы выполнить обход списка совпадений с помощью `foreach`. Фактически этих сведений уже достаточно, чтобы начать пользоваться регулярными выражениями. Регулярные выражения можно использовать в выражениях сопоставления с образцом (`match`) вместо обычных строк.

Обработка XML и сопоставление с образцом

Синтаксис обработки XML и сопоставления с образцом в Scala позволяет получать довольно интересные комбинации. Можно выполнить обход элементов XML в файле и выполнить определенные операции, опираясь на типы элементов. Например, рассмотрим следующий XML-файл со списком фильмов:

scala/movies.scala
http://media.pragprog.com/titles/btlang/code/scala/movies.scala

```
val movies = <movies>
    <movie>The Incredibles</movie>
    <movie>WALL E</movie>
    <short>Jack Jack Attack</short>
    <short>Geri's Game</short>
</movies>

(movies \ "_" ).foreach { movie =>
    movie match{
        case <movie>{movieName}</movie> => println(movieName)
        case <short>{shortName}</short> => println(shortName + " (short)" )
    }
}
```

В этом примере выполняется запрос на извлечение всех узлов дерева. Затем с помощью выражения сопоставления с образцом извлекается содержимое узлов `short` и `movie`. Мне нравится, как поддержка синтаксиса XML, сопоставление с образцом и XQuery-подобный язык в Scala превращают решение типичных задач обработки данных в рутину. Программисты могут добиться желаемого результата почти без усилий.

Итак, мы познакомились с основными возможностями выражений сопоставления с образцом. В следующем разделе мы познакомимся с поддержкой параллельных вычислений.

Параллельные вычисления

Особенности поддержки параллельных вычислений являются одним из важнейших аспектов языка Scala. Основными здесь являются акторы (actors) и механизм передачи сообщений. Акторы имеют очереди сообщений и пулы потоков выполнения. При передаче сообщения в актор (с применением оператора !) происходит добавление объекта в очередь сообщений этого актора. Актор последовательно извлекает сообщения и выполняет соответствующие им действия. Часто для классификации сообщений и выполнения требуемых операций акторы используют выражение сопоставления с образцом. Рассмотрим программу `kids`:

scala/kids.scala
http://media.pragprog.com/titles/btlang/code/scala/kids.scala

```
import scala.actors._
import scala.actors.Actor._

case object Poke
case object Feed

class Kid() extends Actor{
  def act() {
    loop {
      react{
        case Poke => {
          println("Ow..." )
          println("Quit it..." )
        }
        case Feed =>{
          println("Gurgle..." )
          println("Burp..." )
        }
      }
    }
  }
}

val bart = new Kid().start
val lisa = new Kid().start
println("Ready to poke and feed..." )
bart ! Poke
lisa ! Poke
bart ! Feed
lisa ! Feed
```

В этой программе два простых объекта-одиночки (singletons) с именами `Poke` и `Feed`. Они не реализуют ничего, а просто играют роль сообщений. Основой программы является класс `Kid`. Это – актор, то есть он запускает потоки выполнения из пула и извлекает сообщения из очереди. После обработки предыдущего сообщения тут же осуществляется переход к следующему. Основой реализации является простой цикл `loop`. Внутри цикла определена конструкция `react`, которая извлекает сообщения. Обработка сообщений выполняется с помощью выражения сопоставления с образцом, предусматривающим возможность существования всего двух сообщений, `Poke` и `Feed`.

Остальной код в сценарии создает пару экземпляров класса `Kid` и выполняет операции с ними, отправляя им сообщения `Poke` и `Feed`. Запустить программу можно, как показано ниже:

```
batate$ scala code/scala/kids.scala
Ready to poke and feed...
Ow...
Quit it...
Ow...
Quit it...
Gurgle...
Burp...
Gurgle...
Burp...

batate$ scala code/scala/kids.scala
Ready to poke and feed...
Ow...
Quit it...
Gurgle...
Burp...
Ow...
Quit it...
Gurgle...
Burp...
```

Я запустил приложение дважды, чтобы показать, что экземпляры класса `Kid` действительно действуют параллельно, – обратите внимание на разный порядок вывода сообщений. Акторы могут также выполнять некоторые действия по тайм-ауту (`reactWithin`), то есть если в течение заданного интервала времени не было получено ни одного сообщения. Кроме того, существуют методы `receive` (блокирует выполнение потока) и `receiveWithin` (блокирует выполнение потока не более чем на заданный промежуток времени).

Параллельные вычисления в действии

Поскольку в реальной жизни спрос на имитацию Симпсонов[1] не особенно велик, попробуем создать что-нибудь более практичное. Ниже приводится пример приложения `sizer`, определяющего размеры веб-страниц. В нем указываются адреса нескольких страниц, размеры которых требуется определить. Так как для загрузки веб-страниц требуется некоторое время, в значительной степени обусловленное задержками в сети, было решено извлекать их параллельно, с помощью акторов. Взгляните сначала на полный текст программы, а затем мы подробно рассмотрим некоторые ее фрагменты:

scala/sizer.scala
http://media.pragprog.com/titles/btlang/code/scala/sizer.scala

```
import scala.io._
import scala.actors._
import Actor._

object PageLoader {
  def getPageSize(url : String) = Source.fromURL(url).mkString.length
}

val urls = List("http://www.amazon.com/" ,
                "http://www.twitter.com/" ,
                "http://www.google.com/" ,
                "http://www.cnn.com/" )

def timeMethod(method: () => Unit) = {
  val start = System.nanoTime
  method()
  val end = System.nanoTime
  println("Method took " + (end - start)/1000000000.0 + " seconds." )
}

def getPageSizeSequentially() = {
  for(url <- urls){
    println("Size for " + url + ": " + PageLoader.getPageSize(url))
  }
}

def getPageSizeConcurrently() = {
  val caller = self

  for(url <- urls) {
```

[1] В примере выше создавались имитации двух детей из семейки Симпсонов, Лизы и Барта (http://ru.wikipedia.org/wiki/Симпсоны). – *Прим. перев.*

```
    actor { caller ! (url, PageLoader.getPageSize(url))}
  }

  for(i <- 1 to urls.size) {
    receive{
      case (url, size) =>
        println("Size for " + url + ": " + size)
    }
  }

  println("Sequential run:" )
  timeMethod { getPageSizeSequentially}

  println("Concurrent run" )
  timeMethod { getPageSizeConcurrently}
```

Итак, начнем разбираться с самого начала. Программа импортирует несколько библиотек поддержки акторов (`actors`) и ввода/вывода (`io`), с помощью которых будут выполняться параллельные HTTP-запросы. Далее вычисляется размер страницы с указанным адресом URL:

```
object PageLoader {
  def getPageSize(url : String) = Source.fromURL(url).mkString.length
}
```

Затем создается неизменяемый список с несколькими адресами URL. После этого определяется метод, засекающий время выполнения веб-запросов:

```
def timeMethod(method: () => Unit) = {
  val start = System.nanoTime
  method()
  val end = System.nanoTime
  println("Method took " + (end - start)/1000000000.0 + " seconds." )
}
```

Далее выполняются веб-запросы двумя разными методами. Первый выполняет запросы последовательно, с помощью цикла `for`.

```
def getPageSizeSequentially() = {
  for(url <- urls){
    println("Size for " + url + ": " + PageLoader.getPageSize(url))
  }
}
```

Второй запускает асинхронные запросы:

```
def getPageSizeConcurrently() = {
  val caller = self

  for(url <- urls) {
```

```
    actor { caller ! (url, PageLoader.getPageSize(url))}
  }

  for(i <- 1 to urls.size) {
    receive{
      case (url, size) =>
        println("Size for " + url + ": " + size)
    }
  }
```

Нам известно, что актор будет получать фиксированное множество сообщений. В первом цикле `for` мы отправляем четыре асинхронных запроса. Это происходит более или менее мгновенно. Затем, с помощью `receive`, мы просто принимаем четыре сообщения. Этот метод выполняет всю основную работу. Завершается программа инструкциями, выполняющими тестирование:

```
println("Sequential run:" )
timeMethod { getPageSizeSequentially}

println("Concurrent run" )
timeMethod { getPageSizeConcurrently}
```

Вот что получилось на моем компьютере:

```
>> scala sizer.scala
Sequential run:
Size for http://www.amazon.com/: 81002
Size for http://www.twitter.com/: 43640
Size for http://www.google.com/: 8076
Size for http://www.cnn.com/: 100739
Method took 6.707612 seconds.
Concurrent run
Size for http://www.google.com/: 8076
Size for http://www.cnn.com/: 100739
Size for http://www.amazon.com/: 84600
Size for http://www.twitter.com/: 44158
Method took 3.969936 seconds.
```

Как и следовало ожидать, метод, выполняющий запросы параллельно, требует меньше времени. На этом мы завершаем знакомство с одной из самых интересных особенностей языка Scala. А теперь повторим, что мы узнали в третий день.

Что мы узнали в третий день

Третий день получился коротким, но весьма насыщенным. Мы написали пару многопоточных программ, познакомились с возможностью непосредственной обработки данных в формате XML, приемами пе-

редачи сообщений акторам, механизмом сопоставления с образцом и с регулярными выражениями.

В этой главе мы изучили четыре основные конструкции, опирающиеся друг на друга. Во-первых, мы познакомились с поддержкой использования разметки XML непосредственно в программном коде на языке Scala. С ее помощью можно обращаться к отдельным элементам или атрибутам, используя XQuery-подобный синтаксис.

Затем вашему вниманию был представлен механизм сопоставления с образцом. На первый взгляд он показался похожим на простую инструкцию `case`, но когда мы познакомились с ограничителями (guards), поддержкой регулярных выражений и возможностью сопоставления с типами, широта возможностей этого механизма стала более чем очевидной.

Потом мы переключились на поддержку параллельных вычислений. В примерах мы использовали такой инструмент, как акторы (actors). Акторы – это объекты со встроенной поддержкой параллельных вычислений. Типичный подход к применению акторов заключается в использовании инструкции цикла `loop`, обертывающей метод `react` или `receive`, выполняющий основную работу или принимающий сообщения из очереди. Наконец, мы подробнее исследовали механизм сопоставления с образцом. В качестве сообщений мы использовали простые классы – маленькие, легковесные, надежные и простые в обращении. Если вместе с сообщениями понадобится передавать какие-нибудь параметры, можно просто добавить атрибуты в определения классов, как это было сделано с адресами URL в приложении `sizer`.

Как и все остальные языки, описываемые в этой книге, Scala обладает намного более широкими возможностями, чем мне удалось показать в этой главе. Поддержка взаимодействий с классами Java проникает в язык гораздо глубже, чем было продемонстрировано здесь, а кроме того, в Scala поддерживается множество других сложных понятий, таких как карринг (currying), о которых я упомянул лишь вскользь. Тем не менее вы увидели достаточно, чтобы у вас проснулся интерес к дальнейшим исследованиям.

День 3: задания для самостоятельного решения

Итак, теперь вы познакомились с некоторыми расширенными особенностями, которые может предложить Scala. И можете попробовать заняться самостоятельным исследованием этого языка. Как обычно, это потребует от вас приложения определенных усилий.

Узнайте:

- что случится в программе `sizer`, если не создавать новый актор для каждой ссылки на страницу? Как это скажется на скорости выполнения приложения?

Практические задания:

- добавьте в приложение `sizer` сообщение для подсчета количества ссылок на странице;
- дополнительное задание: добавьте в приложение `sizer` извлечение ссылок из указанной страницы и их загрузку, например чтобы программа `sizer` вычислила размер страницы «google.com», а также всех страниц, на которые она ссылается.

5.5. В заключение о Scala

Мы рассмотрели язык Scala более подробно, чем другие языки, представленные выше, потому что Scala поддерживает две парадигмы программирования. Объектно-ориентированные возможности языка Scala позволяют позиционировать его как альтернативу языку Java. В отличие от Ruby и Io, в Scala используется стратегия статической типизации. Синтаксис Scala очень многое заимствовал из языка Java, включая фигурные скобки и использование конструкторов.

Scala также предлагает мощную поддержку функциональных концепций и неизменяемых переменных. В языке предусмотрены развитые механизмы поддержки параллельных вычислений и обработки данных в формате XML, востребованных в самых разнообразных промышленных приложениях, в настоящее время реализованных на Java.

Множество инструментов функционального программирования в Scala гораздо шире, чем можно было представить в этой главе. Я не затронул здесь такие конструкции, как карринг (currying), полные замыкания (full closures), функции с несколькими списками параметров и обработку исключений, но все это – важнейшие концепции, увеличивающие гибкость и возможности Scala.

Давайте взглянем на некоторые сильные и слабые стороны языка.

Основные сильные стороны

Основные преимущества Scala заключаются в поддержке двух парадигм программирования, прекрасно интегрирующихся с окружением Java, а также в некоторых архитектурных решениях, примерами которых могут служить акторы, сопоставление с образцом и встроенная поддержка XML. Давайте раскроем эти пункты подробнее.

Параллельные вычисления

Подход к решению проблем параллельных вычислений, предпринятый в языке Scala, является важным усовершенствованием в параллельном программировании. Модель акторов и поддержка пулов потоков выполнения являются долгожданными улучшениями, а возможность проектировать приложения без использования изменяемого состояния абсолютно бесценна.

Парадигма акторов, с которой мы впервые столкнулись во время знакомства с языком Io, и теперь в языке Scala, проста и понятна разработчикам и хорошо изучена в академическом сообществе. Обоим языкам, Java и Ruby, не помешали бы некоторые усовершенствования в этой области.

Совершенная модель параллельных вычислений – лишь одно из множества преимуществ. Когда объекты используют общие данные, вы должны приложить все силы, чтобы обеспечить неизменность значений. Языки Io и Scala изначально поддерживают такую возможность, предлагая библиотеки и ключевые слова для обеспечения неизменяемости. Неизменяемость является одной из важнейших характеристик языка, помогающих улучшить архитектуру программного кода, выполняющегося в многозадачном окружении.

Наконец, синтаксис передачи сообщений в языке Scala сильно напоминает аналогичный синтаксис языка Erlang, с которым мы познакомимся в следующей главе. Это является серьезным усовершенствованием в сравнении со стандартными библиотеками Java поддержки многопоточного выполнения.

Эволюционное развитие на основе Java

Язык Scala изначально обладает мощным сообществом потенциальных пользователей – пользователей Java. Приложения на Scala способны использовать библиотеки Java как непосредственно, так и через специальные прокси-объекты, генерируемые автоматически, то есть способность к взаимодействиям находится на превосходном уровне. Механизм вывода типов является долгожданным усовершенствованием архаичной системы типов языка Java. Лучший способ образовать новое сообщество программистов – полностью охватить существующее. Разработчики Scala пошли верным путем, предложив более компактный вариант Java, и эта идея достойна всемерной поддержки.

Кроме того, Scala предлагает сообществу Java множество новых особенностей. Например, блоки кода – стандартные конструкции

языка – прекрасно уживаются с базовыми библиотеками поддержки коллекций. Scala предлагает также поддержку примесных классов в форме трейтов. Сопоставление с образцом – еще одна интереснейшая возможность. Благодаря этим и другим возможностям программисты на Java получают современный язык программирования, который можно использовать, даже не касаясь парадигм функционального программирования.

Добавьте к упомянутым особенностям функциональные конструкции, и вы сможете значительно улучшить свои приложения. Обычно программный код на языке Scala получается более компактным в сравнении с эквивалентным кодом на Java, и это весьма немаловажно. Хороший язык программирования должен позволять выражать более сложные идеи меньшим количеством строк кода. Scala дает вам такую возможность.

Предметно-ориентированные языки

Гибкий синтаксис и поддержка перегрузки операторов в Scala делают его идеальным для создания предметно-ориентированных языков. Как и в Ruby, операторы в языке Scala являются самыми обычными методами, большинство из которых можно переопределить. Кроме того, необязательность пробелов, точек и точек с запятой увеличивает многообразие синтаксических форм. Все это в совокупности с простотой примесных классов составляет богатый арсенал средств, столь желанных для разработчиков предметно-ориентированных языков.

XML

Поддержка формата XML интегрирована непосредственно в синтаксис языка Scala. Механизм сопоставления с образцом превращает парсинг значительных блоков разметки XML почти что в рутину. Возможность использования XPath-подобного синтаксиса для исследования разметки XML со сложной структурой позволяет писать компактный и легкочитаемый код. Это преимущество особенно востребовано в сообществе Java, весьма широко использующем формат XML.

Простота перехода

Появление любой новой парадигмы программирования требует наличия своеобразного моста, по которому легко можно было бы перейти к ней. Scala как раз является таким мостом. Переход на функциональную модель программирования почти неизбежен, потому что она лучше других справляется с проблемами параллельных вычис-

лений, а развитие микропроцессорной техники как раз идет по пути распараллеливания. Scala поможет разработчикам постепенно прийти к этой модели.

Недостатки

Мне нравятся многие идеи, заложенные в язык Scala, но, на мой взгляд, синтаксис порой выглядит слишком сложным и академическим. Даже при том, что отношение к синтаксису – это во многом вопрос личных предпочтений, Scala действительно имеет более сложный синтаксис, чем многие другие языки, по крайней мере на первый взгляд. Я также считаю, что некоторые компромиссные решения, которые делают Scala таким эффективным мостом к новой парадигме, также снижают его привлекательность. Я вижу всего три основных недостатка, но все они достаточно велики.

Статическая типизация

Статическая типизация – естественное решение для функциональных языков программирования, но статическая типизация в стиле объектно-ориентированного языка Java – это что-то невообразимое. Удовлетворение хотелок компилятора и без того ложится тяжким бременем на плечи разработчиков. Но со статической типизацией это бремя возрастает еще больше. Она оказывает большое влияние на код, синтаксис и архитектуру программ. Изучая Scala, я поймал себя на том, что мне постоянно приходится бороться с синтаксисом, чтобы воплотить желаемую архитектуру. Трейты несколько облегчают ситуацию, но мне кажется, что баланс между гибкостью программиста и стремлением выполнить как можно больше проверок на этапе компиляции далек от идеала.

Далее в этой книге вы увидите, на что должна быть похожа статическая система типов в настоящем функциональном языке программирования на примере Haskell. Не обремененная необходимостью поддержки двух парадигм программирования, система типов становится более текучей и удобной в использовании, обеспечивает лучшую поддержку полиморфизма и предъявляет меньше требований к программисту.

Синтаксис

Как мне кажется, синтаксис языка Scala страдает излишней академичностью и сложен для беглого восприятия. Я долго колебался, стоит ли выражать это свое мнение в печатном издании, потому что

восприятие синтаксиса – вещь весьма субъективная, тем не менее отдельные элементы синтаксиса действительно оставляют желать лучшего. В некоторых ситуациях Scala следует соглашениям, принятым в Java, таким как использование конструкторов. Вместо `Person.new` следует писать `new Person`. В других случаях Scala вводит новые соглашения, как в случае с типами аргументов. В Java заголовок определения функции записывается, например, так: `setName(String name)`, – а в Scala так: `setName(name: String)`. Объявление типа возвращаемого значения вообще смещено в конец объявления, в отличие от Java, где тип указывается в начале. Эти маленькие отличия вынуждают меня постоянно задумываться о синтаксисе, отвлекая от решения задач. Проблема в том, что для переключения между языками Scala и Java требуется больше усилий, чем того хотелось бы.

Изменяемость

При создании языка, играющего роль моста между разными парадигмами программирования, приходится идти на компромиссы. Одним из существенных компромиссов в Scala является поддержка изменяемости переменных. Ключевым словом `var` Scala открывает ящик Пандоры, потому что изменяемость состояния программы является источником разнообразных ошибок, проявляющихся в многозадачном окружении. Но такие компромиссы неизбежны, если вы хотите привести домой ребенка, жившего в доме на холме.

Заключительные замечания

У меня остались смешанные чувства от знакомства с языком Scala. Статическая система типов обескуражила меня. В то же время Java-разработчик во мне по достоинству оценил усовершенствованную модель поддержки параллельных вычислений, механизм вывода типов и встроенный синтаксис для работы с данными в формате XML. Язык Scala определенно является большим шагом вперед.

Вне всяких сомнений, язык Scala помог бы мне повысить мою продуктивность, если бы у меня имелись существенные наработки на языке Java. Я также подумал бы об использовании Scala для реализации приложений с существенными требованиями к масштабируемости, заявляющими об обязательной поддержке параллельных вычислений. С коммерческой точки зрения, этот Франкенштейн имеет все шансы на успех, потому что является мостом между разными парадигмами программирования и полностью охватывает огромное сообщество программистов.

Глава 6

Erlang

Вы слышите, мистер Андерсон? Это – рок, неизбежность.

Агент Смит

Немногие языки обладают загадочностью Erlang, языка параллельного программирования, делающего сложное простым, а простое – сложным. Его виртуальная машина называется *BEAM*, по надежности в промышленном окружении с ней может конкурировать только виртуальная машина Java. Язык Erlang можно было бы назвать эффективным, и даже очень, но его синтаксису не хватает красоты и простоты, например языка Ruby. Представьте себе агента Смита из фильма «Матрица»[1].

Фильм «Матрица», выпущенный в 1999 году, стал классикой научно-фантастического жанра. В нем наш текущий мир был изображен как виртуальная реальность, созданная и управляемая компьютерами. Агент Смит – это программа искусственного интеллекта в матрице, имевшая удивительную способность принимать любую форму и изменять законы действительности сразу во многих местах. Он был олицетворением неизбежности.

6.1. Введение в Erlang

Это странное название является сокращением от английского *Ericsson Language* (язык компании Ericsson) и совпадает с именем датского математика Агнера Крарупа Эрланга (Agner Krarup Erlang), основателя научного направления по изучению трафика в телекоммуникационных системах и теории массового обслуживания.

В 1986 году в компьютерной лаборатории компании Ericsson Джо Армстронгом (Joe Armstrong) была создана первая версия языка Erlang, которую он продолжил развивать и дорабатывать следующие

---

[1] «The Matrix». DVD. Режиссеры Энди Вачовски (Andy Wachowski) и Лана Вачовски (Lana Wachowski). 1999; Бербанк (Калифорния): Warner Home Video, 2007. (http://ru.wikipedia.org/wiki/Матрица_(фильм). – *Прим. перев.*)

пять лет. На протяжении 90-х годов развитие языка продолжалось довольно неравномерно, но в 2000-х академические круги стали проявлять все увеличивающийся интерес к Erlang. На этом языке написаны CouchDB и SimpleDB, популярные базы данных для облачных вычислений. Кроме того, на языке Erlang реализован чат в Facebook. В последние годы отмечается непрерывный рост интереса к Erlang благодаря присущим ему характеристикам, не свойственным другим языкам: высокой степенью параллелизма и надежности.

Поддержка параллельных вычислений

Erlang – результат многолетних исследований, проводимых компанией Ericsson в области создания распределенных, отказоустойчивых приложений почти реального времени для использования в телекоммуникационной аппаратуре. Подобные системы зачастую нельзя остановить для обслуживания, а разработка программного обеспечения для них была чрезвычайно дорога. На протяжении 80-х годов в компании Ericsson проводились исследования различных языков программирования, в результате которых было установлено, что ни один из существующих языков непригоден для их нужд. В результате была начата разработка совершенно нового языка.

Erlang – это функциональный язык, одной из характерных особенностей которого является высокая надежность. Язык Erlang позволяет писать почти бесконечно надежные системы. Как вы понимаете, нельзя отключить телефон для обслуживания, и нельзя остановить Erlang, чтобы заменить целые модули. Некоторые приложения на этом языке работают годами без остановки. Но самой главной особенностью Erlang является его поддержка параллельных вычислений.

В среде специалистов по параллельным вычислениям нет единства в определении наилучших подходов. Примером может служить неутихающий спор о том, какой механизм параллельных вычислений лучше – потоки выполнения или процессы. Потоки выполнения действуют в границах общего процесса. Процессы могут иметь собственные ресурсы. Потоки в рамках одного процесса выполняются параллельно друг другу, но совместно используют одни и те же ресурсы. Обычно потоки выполнения легковеснее процессов, хотя во многом это зависит от реализации.

Отказ от модели потоков выполнения

Во многих языках, таких как Java и C, параллельные вычисления реализованы на основе модели потоков выполнения. Поддержка по-

токов менее ресурсоемка, поэтому теоретически они должны обеспечивать более высокую производительность. Недостатком потоков является совместный доступ к ресурсам, что может служить причиной создания сложных и хрупких реализаций, требующих применения блокировок, которые могут стать узким местом с точки зрения производительности. Для координации доступа двух приложений к разделяемым ресурсам в многозадачной системе требуется использовать семафоры или блокировки уровня системы. В Erlang используется иная модель параллельных вычислений. Ее цель – сделать процессы максимально легковесными.

Легковесные процессы

Вместо того чтобы продираться через тернии разделяемых ресурсов и управления доступа к ним, Erlang вводит философию легковесных процессов. Создатели Erlang приложили немало усилий, чтобы упростить создание процессов, управление ими, а также взаимодействия между процессами.

Передача распределенных сообщений является одной из основных конструкций языка, что устраняет необходимость в использовании блокировок и повышает производительность параллельных вычислений.

Как и в языке Io, в разработке Армстронга основой механизма параллельных вычислений являются акторы, поэтому ключевым понятием в нем является передача сообщений. Вы уже знакомы с синтаксисом передачи сообщений в языке Scala, который очень похож на синтаксис передачи сообщений в языке Erlang. В Scala акторы представлены объектами, опирающимися на пулы потоков выполнения. В Erlang акторы основаны на легковесных процессах. Актор читает входящие сообщения из очереди и с помощью механизма сопоставления с образцом определяет, как обрабатывать их.

Надежность

При программировании на Erlang применяются традиционные подходы к контролю за ошибками, но в обычных приложениях вам встретится не так много кода, занимающегося их обработкой, как в отказоустойчивых приложениях на других языках. Erlang следует философии «позвольте приложению потерпеть неудачу». Так как Erlang упрощает слежение за работоспособностью процесса, остановка групп взаимосвязанных процессов и их перезапуск превращаются в тривиально простую задачу.

Существует также поддержка «горячей» замены кода, когда замещаются отдельные фрагменты приложения без полной его остановки. Это значительно упрощает стратегии обслуживания в сравнении с аналогичными распределенными приложениями. Язык Erlang как нельзя лучше сочетает в себе стратегии «позвольте приложению потерпеть неудачу» обработки ошибок и горячей замены кода с легковесными процессами, которые можно запускать с минимальными издержками. Теперь вам должно быть понятно, почему приложения могут работать годами без остановки.

Поддержка параллельных вычислений в языке Erlang неотразима. В нем имеются все важнейшие примитивы – передачи сообщений, запуска процессов, мониторинга процессов. Порождаемые процессы настолько легковесны, что нет необходимости беспокоиться об ограниченности ресурсов. Разработчики языка изрядно постарались, чтобы устранить все возможные побочные эффекты, избавиться от изменяемого состояния и превратить слежение за работоспособностью процессов в простую задачу, в чем-то даже тривиальную.

Интервью с доктором Джо Армстронгом

Работая над этой книгой, я получил возможность пообщаться с некоторыми людьми из наиболее уважаемых мною, по крайней мере посредством электронной почты. Доктор Джо Армстронг (Joe Armstrong), создатель языка Erlang и автор «Programming Erlang: Software for a Concurrent World» [Arm07], находится в самом начале этого списка. Я наконец получил возможность побеседовать с первым создателем языка Erlang, родом из Стокгольма (Швеция).

Брюс: Что послужило причиной написать Erlang?

Доктор Армстронг: Случай. Я не предполагал заниматься созданием нового языка программирования. На тот момент я искал язык, который лучше всего подходил бы для разработки телекоммуникационных программных систем. Я начал возиться с языком Prolog. Это фантастический язык, но не совсем то, что мне требовалось. В ту пору, топчась на месте с языком Prolog, я подумал: «Интересно, что получится, если я изменю кое-что в этом языке?» Так я написал метаинтерпретатор, добавляющий в Prolog поддержку параллельных вычислений. Затем я добавил механизм обработки ошибок, и пошло-поехало. Спустя какое-то время этот комплект дополнений к языку Prolog получил имя Erlang. Так появился новый язык. Потом к проекту присоединились другие разработчики. По мере взросления языка мы обна-

ружили, как его можно превратить в компилирующий язык, добавили еще несколько фишек и приобрели первых пользователей, потом.

. . .

Брюс: Что больше всего нравится вам в этом языке?

Доктор Армстронг: Механизмы обработки ошибок и горячей замены кода, а также сопоставление с образцом на уровне битов. Обработка ошибок – одна из самых малопонятных областей языка, и она же больше всего отличает Erlang от других языков. Само понятие «незащищенного» программирования и философия «позвольте приложению потерпеть неудачу» полностью противоречат обычной практике, но они же позволяют писать по-настоящему короткие и красивые программы.

Брюс: Что бы вы изменили в языке, будь у вас возможность вернуться назад? (Или какие наиболее существенные ограничения Erlang вы могли бы назвать?)

Доктор Армстронг: Сложный вопрос. В разные дни я мог бы ответить на него по-разному. Было бы неплохо добавить ему мобильности, чтобы получить возможность посылать вычисления по сети. Эта возможность поддерживается с помощью библиотек, но она не является частью языка. Прямо сейчас я думаю, что было бы неплохо вернуться назад, к истокам Erlang, и добавить в язык Prolog-подобную логику предикатов, этакую смесь логики предикатов с возможностью передачи сообщений.

Также не помешало бы внести некоторые небольшие изменения, добавить хэши, модули высшего порядка и т. д.

Если бы у меня была возможность начать все сначала, я, вероятно, больше внимания уделил бы проблемам создания больших проектов с большими объемами кода, управления версиями кода, поиска кода в проектах и т. д. Когда имеется большой объем кода, программисту часто приходится вместо создания нового кода заниматься поисками нужных фрагментов и их изменением, поэтому функции поиска и фильтрации на этом этапе разработки становятся особенно востребованными. Было бы хорошо внедрить идеи, заложенные в GIT и Mercurial, непосредственно в сам язык, чтобы сделать процесс развития кода более управляемым.

Брюс: С каким из самых необычных применений Erlang вам приходилось сталкиваться?

Доктор Армстронг: Был один случай в моей практике. Не могу сказать, что я был очень удивлен, потому что знал, что рано или поздно это случится. Когда я обновил у себя версию Ubuntu до Karmic

Koala[1], я обнаружил спрятанный в ее недрах Erlang, незаметно выполняющийся в фоновом режиме. Это была поддержка базы данных CouchDB, которая и прежде присутствовала на моем компьютере. Так, крадучись, Erlang распространился по 10 миллионам компьютеров.

В этой главе мы сначала познакомимся с некоторыми основами языка Erlang. Затем опробуем в деле его функциональные возможности. И, наконец, потратим некоторое время на исследование подходов к параллельным вычислениям и поддержки отказоустойчивости. Да, друзья мои, поддержка отказоустойчивости может выглядеть очень круто!

6.2. День 1: Появление человека

Агент Смит – это программа, уничтожающая другие программы или имитируемых ими людей, которые разрушающе действуют на виртуальную реальность, известную как Матрица. Самой опасной ее чертой является сходство с человеком. В этом разделе мы рассмотрим возможности языка Erlang, которые можно использовать для создания обычных приложений. Я изо всех сил буду стараться не отклоняться от привычных вам приемов, хотя это будет нелегко.

Если вы начинали читать эту книгу, имея только опыт объектно-ориентированного программирования, возможно, вам придется нелегко, но это не смертельно. Вы уже видели блоки кода в Ruby, акторы в Io, сопоставление с образцом в Prolog и передачу сообщений в Scala. Все эти механизмы образуют основу Erlang. В этой главе вы познакомитесь с еще одной важной концепцией. Erlang – первый из по-настоящему функциональных языков. (Scala – это гибридный язык, сочетающий функциональную и объектно-ориентированную парадигмы.) Для вас это означает следующее:

- ваши программы целиком будут состоять из функций, без каких-либо объектов;
- функции обычно будут возвращать одно и то же значение при вызове с одними и теми же входными данными;
- функции обычно не будут иметь побочных эффектов, то есть они не будут изменять состояние программы;
- вы сможете присваивать значения переменным только один раз.

---

[1] Ubuntu 9.10. – *Прим. перев.*

Если первое правило хоть и выглядит трудным, но все же терпимым, то остальные три могут показаться невыполнимыми в принципе. Однако знайте, что, научившись следовать им, вы сможете писать код, который с легкостью будет поддерживать параллельные вычисления. Стоит убрать из уравнения изменяемое состояние, и реализация параллельных вычислений упростится до неузнаваемости.

Если вы были внимательны, от вас наверняка не ускользнуло слово «обычно» во втором и третьем правилах. Erlang не является исключительно функциональным языком; он допускает некоторые исключения из правил. Haskell – вот это действительно чистый функциональный язык программирования. Но вы почувствуете сильный вкус функционального программирования и в своих программах чаще будете следовать этим правилам, чем нарушать их.

Введение

Я использовал версию Erlang R13B02, однако простые примеры, что приводятся в этой главе, должны быть работоспособны в любой, достаточно свежей версии. Запустить интерактивную оболочку Erlang можно командой `erl` (или `werl` в некоторых Windows-системах), как показано ниже:

```
batate$ erl
Erlang (BEAM) emulator version 5.4.13 [source]

Eshell V5.4.13 (abort with ^G)
1>
```

Большинство начальных примеров мы будем исследовать в этой оболочке, как и в других главах. Подобно языку Java, Erlang является компилирующим языком. Скомпилировать файл можно командой `c(filename).` (обратите внимание на точку в конце). Завершить работу интерактивной оболочки можно нажатием комбинации клавиш **Control**+**C**. Итак, приступим.

Комментарии, переменные и выражения

Познакомимся с простейшими синтаксическими конструкциями. Откройте консоль и введите следующее:

```
1> % This is a comment
```

Это – комментарий. Комментарии в языке Erlang начинаются с символа `%` и простираются до конца строки. Erlang воспринимает комментарий как единственный пробел.

```
1> 2 + 2.
4
2> 2 + 2.0.
4.0
3> "string".
"string"
```

Каждая инструкция должна завершаться точкой. В языке имеются несколько простых типов данных: строки, целые и вещественные числа. А теперь взгляните на списки:

```
4> [1, 2, 3].
[1,2,3]
```

Как и в семействе языков Prolog, списки в Erlang заключаются в квадратные скобки. А вот вам маленький сюрприз:

```
4> [72, 97, 32, 72, 97, 32, 72, 97].
"Ha Ha Ha"
```

Итак, строки в действительности являются списками, и агент Смит просто посмеялся над нами. Результат выражения `2 + 2.0` сообщает нам, что Erlang поддерживает приведение простых типов. Попробуем нарушить его работу, использовав операнды несовместимых типов:

```
5> 4 + "string".
** exception error: bad argument in an arithmetic expression
 in operator +/2
    called as 4 + "string"
```

В отличие от Scala, Erlang не поддерживает приведения типов между строками и целыми числами. Попробуем выполнить присваивание переменной:

```
6> variable = 4.
** exception error: no match of right hand side value 4
```

Здесь проявилось отвратительное сходство агентов и Erlang. Иногда этот ужасный язык действует как бездушный робот. Данное сообщение об ошибке в действительности ссылается на операцию сопоставления с образцом. Оно появилось потому, что `variable` – это атом. Переменные должны начинаться с буквы верхнего регистра.

```
7> Var = 1.
1
8> Var = 2.

=ERROR REPORT==== 8-Jan-2010::11:47:46 ===
Error in process <0.39.0> with exit value: {{badmatch,2},[{erl_eval,expr,3}]}
```

```
** exited: {{badmatch,2},[{erl_eval,expr,3}]} **
8> Var.
1
```

Как видите, имена переменных начинаются с буквы верхнего регистра, а их значение не может изменяться. Присвоить значение переменной можно только один раз. Эта особенность часто приводит в замешательство даже опытных программистов, впервые столкнувшихся с функциональным языком. А теперь познакомимся с более сложными типами данных.

Атомы, списки и кортежи

В функциональных языках символы приобретают особую важность. Они являются наиболее простыми элементами данных и могут представлять все, чему вы решите присвоить имя. Символы присутствуют во всех других языках программирования, представленных в этой книге. В Erlang символы называются *атомами* (atom) и начинаются с буквы в нижнем регистре. Они представляют атомарные значения, которые можно применять для представления всего, чего угодно. Например:

```
9> red.
red
10> Pill = blue.
blue
11> Pill.
blue
```

`red` и `blue` — это атомы, произвольные имена, которые можно использовать для обозначения сущностей реального мира. Сначала мы вернули простой атом с именем `red`. Затем присвоили атом с именем `blue` переменной `Pill`. Особенно интересными атомы становятся при использовании их в структурах данных, с которыми мы познакомимся далее. А пока продолжим знакомство со списками. Список определяется как последовательность значений, заключенная в квадратные скобки:

```
13> [1, 2, 3].
[1,2,3]
14> [1, 2, "three"].
[1,2,"three"]
15> List = [1, 2, 3].
[1,2,3]
```

Синтаксис выглядит вполне знакомым. Списки в Erlang являются гетерогенными структурами данных и могут иметь произвольную

длину. Как и любой другой примитив, их можно присваивать переменным. Кортежи фактически являются списками фиксированной длины:

```
18> {one, two, three}.
{one,two,three}
19> Origin = {0, 0}.
{0,0}
```

Пока ничего необычного. Здесь можно наблюдать сильное влияние языка Prolog. Далее, когда будет рассматриваться механизм сопоставления с образцом, вы увидите, что при сопоставлении размер кортежа имеет значение – нельзя сравнить кортеж из трех элементов с кортежем из двух элементов. При сопоставлении списков длина играет менее важную роль, как и в Prolog.

В Ruby для образования связи между именами и значениями можно использовать хэши. В Erlang роль хэшей и ассоциативных массивов часто играют кортежи:

```
20> {name, "Spaceman Spiff"}.
{name,"Spaceman Spiff"}
21> {comic_strip, {name, "Calvin and Hobbes"}, {character, "Spaceman Spiff"}}.
{comic_strip,{name,"Calvin and Hobbes"},
             {character,"Spaceman Spiff"}}
```

Здесь мы использовали хэш для представления комикса. Роль ключей в таком хэше играют атомы, а роль значений – строки. Списки и кортежи могут смешиваться, как угодно, например можно создать список комиксов, каждый из которых будет представлен кортежем. А как обращаться к отдельным элементам? Если воспоминания о Prolog еще достаточно свежи у вас, ваши мысли наверняка потекли в правильном направлении – сопоставление с образцом.

Сопоставление с образцом

Если вы следовали за примерами в главе, описывающей язык Prolog, у вас должно сложиться достаточно четкое представление о сопоставлении с образцом. Я хочу указать лишь на некоторые отличительные моменты. В Prolog, когда определяется правило, сопоставление выполняется со всеми значениями, имеющимися в базе знаний, и интерпретатор проверяет все возможные комбинации. Erlang в этом отношении больше похож на Scala. Сопоставление выполняется с единственным значением. Давайте задействуем сопоставление с образцом для извлечения значений из кортежа. Допустим, что у нас имеется следующий кортеж:

```
24> Person = {person,{name,"Agent Smith"},{profession,"Killing programs"}}.
                {person,{name,"Agent Smith"},
                        {profession,"Killing programs"}}
```

Допустим также, что нам требуется присвоить значение ключа `name` переменной `Name`, а значение ключа `profession` – переменной `Profession`. Все необходимое выполнит следующая инструкция:

```
25> {person, {name, Name}, {profession, Profession}} = Person.
{person,{name,"Agent Smith"},
        {profession,"Killing programs"}}
26> Name.
"Agent Smith"
27> Profession.
"Killing programs"
```

Erlang обеспечит сопоставление структур данных, присвоив переменным значения, хранящиеся в кортежах. Атом соответствует сам себе, поэтому остается добиться только соответствия переменной `Name` и строки `"Agent Smith"`, а также переменной `Profession` и строки `"Killing programs"`. Этот механизм действует так же, как в языке Prolog, и является основной условной конструкцией.

Если у вас есть опыт использования хэшей в языке Ruby или Java, может показаться странным присутствие начального атома `person`. В Erlang часто используется несколько инструкций сопоставления для поддержки нескольких типов кортежей. Конструируя структуры данных таким способом, можно быстро выявить все кортежи `person` и оставить все остальные в стороне.

Сопоставление со списками действует точно так же, как в Prolog:

```
28> [Head | Tail] = [1, 2, 3].
[1,2,3]
29> Head.
1
30> Tail.
[2,3]
```

Просто, как «раз, два, три». Голову списка можно также связать более чем с одной переменной:

```
32> [One, Two|Rest] = [1, 2, 3].
[1,2,3]
33> One.
1
34> Two.
2
35> Rest.
[3]
```

Если в списке окажется недостаточно элементов, сопоставление выполнено не будет:

```
36> [X|Rest] = [].
** exception error: no match of right hand side value []
```

Теперь некоторые сообщения об ошибках должны казаться вам более осмысленными. Допустим, что мы забыли начать имя переменной с заглавной буквы. В результате вы получите следующее сообщение:

```
31> one = 1.
** exception error: no match of right hand side value1
```

Как уже говорилось выше, инструкция = не является простым присваиванием. В действительности это сопоставление с образцом. Вы просите Erlang сопоставить целое число 1 с атомом one, но ему это не удается.

Сопоставление на уровне битов

Иногда бывает необходимо обращаться к данным на уровне отдельных битов. Если вам требуется втиснуть больше данных в меньшую область памяти или анализировать данные в предопределенных форматах, таких как JPEG или MPEG, положение каждого бита имеет значение. Erlang дает довольно простую возможность упаковать несколько элементов данных в один байт. Для этого необходимы две операции: упаковки и распаковки. В Erlang битовые маски действуют точно так же, как любые другие типы коллекций. Чтобы упаковать структуру данных, достаточно просто сообщить Erlang, сколько битов будет занимать каждый элемент:

```
1> W = 1.
1
2> X = 2.
2
3>
3> Y = 3.
3
4> Z = 4.
4
5> All = <<W:3, X:3, Y:5, Z:5>>.
<<"(d">>
```

Скобки << и >> ограничивают двоичный шаблон. В данном случае говорится, что для хранения переменной W выделяется 3 бита, для переменной X – 5 бит и для переменной Z – тоже 5 бит. Теперь нам необходима возможность распаковывания данных. Вероятно, вы уже догадались, как выглядит синтаксис распаковывания:

```
6> <<A:3, B:3, C:5, D:5>> = All.
<<"(d">>
7> A
7> .
1
8> D.
4
```

Как и в случае с кортежами и списками, достаточно просто использовать тот же синтаксис и позволить механизму сопоставления сделать все остальное. Благодаря этим битовым операциям Erlang является удивительно эффективным языком для решения низкоуровневых задач.

Мы довольно коротко затронули основы языка, потому что вы уже знакомы со многими основными понятиями. Хотите верьте, хотите нет, но мы уже близки к завершению первого дня знакомства с Erlang, однако нам нужно рассмотреть самую важную концепцию языка – функции.

Функции

В отличие от Scala, Erlang является языком с динамической типизацией. То есть вам не придется слишком много беспокоиться о типах присваиваемых элементов данных. Как и Ruby, Erlang имеет динамическую систему типов. Типы данных в этом языке определяются во время выполнения, исходя из синтаксических подсказок, таких как кавычки или десятичные точки. Сейчас я собираюсь открыть новое окно консоли и познакомить вас с несколькими терминами. Далее мы будем сохранять свои функции в файлах с расширением `.erl`. Каждый такой файл содержит код реализации модуля, и его необходимо компилировать перед запуском. В результате компиляции создается выполняемый файл с расширением `.beam`. Скомпилированные модули `.beam` выполняются под управлением виртуальной машины, носящей название BEAM[1].

А теперь, после небольшого вступления, создадим несколько простых функций.

Создайте файл со следующим содержимым:

erlang/basic.erl
http://media.pragprog.com/titles/btlang/code/erlang/basic.erl

```
-module(basic).
-export([mirror/1]).

mirror(Anything) -> Anything.
```

---

[1] От англ. Bogdan's/Björn's Erlang Abstract Machine – абстрактная Erlang-машина Богдана/Бьярна. – *Прим. перев.*

Первая строка определяет имя модуля. Вторая строка определяет имя функции, которую предполагается использовать за пределами модуля. Функция называется `mirror`, а последовательность символов `/1` означает, что она принимает один параметр. Далее следует реализация самой функции. Как видите, здесь явно прослеживается влияние синтаксиса определения правил в языке Prolog. Определение функции начинается с ее имени, за которым следуют объявления аргументов. Символ -> просто возвращает первый аргумент.

Покончив с определением функции, откройте консоль в том же каталоге, где находится файл с исходным кодом, и скомпилируйте его:

```
4> c(basic).
{ok,basic}
```

После компиляции файла `basic.erl` в этом же каталоге должен появиться файл `basic.beam`. Запустить его можно следующим образом:

```
5> mirror(smiling_mug).
** exception error: undefined shell command mirror/1
6> basic:mirror(smiling_mug).
smiling_mug
6> basic:mirror(1).
1
```

Обратите внимание, что для вызова функции недостаточно указать только ее имя. Необходимо также добавить имя модуля и двоеточие. Эта функция настолько проста, что дальше некуда.

Отметьте также, что в параметре `Anything` у нас получилось передать значения разных типов. Erlang является динамически типизируемым языком, что, на мой взгляд, является большим плюсом. После строгого контроля типов в Scala возврат к динамической типизации сравним для меня к возврату домой после выходных, проведенных в холодной Сибири или в жаркой пустыне.

Давайте рассмотрим чуть более сложную функцию. Следующая функция определяет несколько альтернатив.

Создайте файл `matching_function.erl` со следующим содержимым:

erlang/matching_function.erl
http://media.pragprog.com/titles/btlang/code/erlang/matching_function.erl

```
-module(matching_function).
-export([number/1]).

number(one)   -> 1;
number(two)   -> 2;
number(three) -> 3.
```

И запустите его:

```
8> c(matching_function).
{ok,matching_function}
9> matching_function:number(one).
1
10> matching_function:number(two).
2
11> matching_function:number(three).
3
12> matching_function:number(four).
** exception error: no function clause matching matching_
function:number(four)
```

Это первая наша функция, выполняющая сопоставление с несколькими альтернативами. Определение каждой альтернативы включает имя функции, аргумент для сопоставления и выполняемый код, следующий за символом ->. Во всех трех предусмотренных случаях Erlang просто возвращает целое число. Последняя инструкция завершается символом точки (.), а все остальные – точкой с запятой (;).

Так же как в Io, Scala и Prolog, рекурсия играет важную роль. Хвостовая рекурсия в языке Erlang, как и в Prolog, оптимизируется. Ниже приводится ставший уже обязательным в таких случаях пример вычисления факториала:

erlang/yet_again.erl
http://media.pragprog.com/titles/btlang/code/erlang/yet_again.erl

```
-module(yet_again).
-export([another_factorial/1]).
-export([another_fib/1]).

another_factorial(0) -> 1;
another_factorial(N) -> N * another_factorial(N-1).

another_fib(0) -> 1;
another_fib(1) -> 1;
another_fib(N) -> another_fib(N-1) + another_fib(N-2).
```

Итак, это еще одна реализация вычисления факториала. Она, как и все остальные в этой книге, действует рекурсивно. Пока мы еще здесь, я заодно включил и реализацию функции вычисления чисел Фибоначчи.

Позвольте мне продемонстрировать их в действии:

```
18> c(yet_again).
{ok,yet_again}
19> yet_again:another_factorial(3).
```

```
6
20> yet_again:another_factorial(20).
2432902008176640000
21> yet_again:another_factorial(200).
788657867364790503552363213932185062295135977687173263294742533244359
449963403342920304284011984623904177212138919638830257642790242637105
061926624952829931113462857270763317237396988943922445621451664240254
033291864131227428294853277524242407573903240321257405579568660226031
904170324062351700858796178922227896237038973747200000000000000000000
000000000000000000000000
22> yet_again:another_factorial(2000).
331627509245063324117539338057632403828111720810578039457193543706038077905600822400273230859732592255402352941225834109258084817415293796131386633526343688905634058556163940605117252571870647856393544045405243957467037674108722970434684158343752431580877533645127487995436859247403767410872297043468415834375243158087753364512748799543685924740376741087229704346841583437524315808775336451274879954368592474
... и так далее, и так далее ...
000000000000000000000000000000000000000000000000000000000000000000000000
```

Ух ты! Выглядит более чем захватывающе! Теперь вы можете сами убедиться, насколько мощно может бить агент Смит, то есть Erlang. Если вы не спешили опробовать этот пример у себя, смею заверить, что результаты вычисляются в мгновение ока. Я не знаю, какой размер имеет самое большое целое число в Erlang, но рискну предположить, что оно достаточно велико.

Это было неплохое начало. Мы создали несколько простых функций и посмотрели, как они работают. А теперь пришло время подвести итоги первого дня.

Что мы узнали в первый день

Erlang – функциональный язык со строгой динамической типизацией. Он имеет простой синтаксис, совершенно отличающийся от синтаксиса типичных объектно-ориентированных языков.

В Erlang, как и в Prolog, отсутствует понятие «объект». К тому же Erlang имеет очень тесную связь с Prolog. Конструкции сопоставления с образцом и функции с несколькими точками входа должны выглядеть для вас знакомыми, и некоторые задачи точно так же решаются с помощью рекурсии. В функциональных языках не поддерживается изменяемое состояние и отсутствуют побочные эффекты. Поддержка состояния программы выглядит неудобной, но взамен вы получаете целый набор новых уловок. Вскоре вы увидите обратную сторону медали. Неизменяемое состояние и отсутствие побочных эффектов значительно упрощают реализацию параллельных вычислений.

В первый день мы познакомились с интерактивной средой и компилятором. Основное наше внимание было сосредоточено на основах языка. Мы опробовали простые выражения и создали несколько простейших функций. Функции в языке Erlang, как и правила в Prolog, могут иметь несколько точек входа. Мы также опробовали механизм сопоставления с образцом.

Помимо перечисленного, мы исследовали кортежи и списки. Кортежи в Erlang занимают ту же нишу, что и хэши в языке Ruby, и образуют фундамент для создания других структур данных. Мы узнали, как выполнять сопоставление со списками и кортежами. Новые знания помогут нам быстро вникнуть в суть поведения кортежей и механизма обмена сообщениями, о которых рассказывается в последующих разделах.

Во второй день я расскажу о дополнительных понятиях функционального программирования. Мы узнаем, как писать код для работы в мире параллельных вычислений, но не будем углубляться далеко. А пока найдите время, чтобы выполнить практические задания и вспомнить, о чем рассказывалось сегодня.

День 1: задания для самостоятельного решения

Интернет-сообщество пользователей языка Erlang растет не по дням, а по часам. Конференция в Сан-Франциско придала новый импульс развития. Но, в отличие от программистов на Io и C, вам придется активно пользоваться поисковыми системами, чтобы найти ответы на свои вопросы.

Найдите:

- официальный сайт языка Erlang;
- официальную документацию с описанием библиотек функций для Erlang;
- документацию с описанием библиотеки OTP (Open Telecom Platform – открытая телекоммуникационная платформа).

Практические задания:

- напишите функцию, реализующую рекурсивный поиск числа слов в строке;
- напишите функцию, реализующую рекурсивный счет от 1 до 10;
- напишите функцию, использующую сопоставление с образцом для выборочного вывода слов «success» и «error: message», когда исходные данные имеют вид `{error, Message}` или `success`.

6.3. День 2: Изменение формы

В этом разделе мы займемся оценкой возможностей агента Смита. Агенты в Матрице обладают нечеловеческими способностями. Они могут уклоняться от пуль и разбивать бетон кулаками. Функциональные языки находятся на более высоком уровне абстракции, чем объектно-ориентированные. Они сложнее для понимания, но позволяют выразить более сложные идеи меньшим количеством кода.

Агент Смит может также принимать обличье любого другого человека в Матрице. Это – важнейшая особенность функциональных языков. Далее мы будем учиться применять функции к спискам и преобразовывать последние в любую нужную нам форму. Хотите превратить список продуктов, которые нужно купить в магазине, в список цен? А может быть, вы хотите преобразовать список адресов URL в список кортежей, включающих эти адреса и содержимое страниц по этим адресам? Все эти задачи с легкостью решаются функциональными языками.

Управляющие структуры

Начнем с самых обыденных конструкций: простейших управляющих структур. Вы наверняка заметите, что этот раздел намного короче аналогичного раздела в главе с описанием языка Scala. Вам часто будут попадаться программы с большим количеством инструкций `case`, потому что с их помощью определяется порядок интерпретации сообщений в многозадачных приложениях. Инструкции `if` используются намного реже.

Case

Начнем с инструкции `case`. Чаще всего сопоставление с образцом выполняется в контексте вызовов функций. Эта управляющая структура служит для реализации сопоставления в любом другом контексте. Например, допустим, что имеется переменная `Animal`, и нам необходимо выполнить разные участки кода в зависимости от ее значения:

```
1> Animal = "dog".
2> case Animal of
2>     "dog" -> underdog;
2>     "cat" -> thundercat
2> end.
underdog
```

Этот пример обнаружит совпадение строки с первым образцом и вернет атом `underdog`. Как и в Prolog, можно использовать символ подчеркивания (_), которому соответствуют любые значения (примечание: переменная `Animal` все еще хранит строку `"dog"`):

```
3> case Animal of
3>     "elephant" -> dumbo;
3>     _ -> something_else
3> end.
something_else
```

Переменная `Animal` хранит значение, отличное от `"elephant"`, поэтому будет найдено совпадение с последним утверждением. Символы подчеркивания можно использовать также в любых других инструкциях сопоставления, имеющихся в языке Erlang. Я хотел бы обратить ваше внимание на один синтаксический казус. Отметьте, что все утверждения `case`, кроме последнего, заканчиваются точкой с запятой. То есть, если вы пожелаете изменить инструкцию или переупорядочить утверждения в ней, вам также потребуется привести их в соответствие с синтаксисом, хотя, как мне кажется, разработчики языка могли бы и позволить добавлять точку с запятой после последнего утверждения. Вне всяких сомнений, синтаксис выстроен вполне логично: точка с запятой служит разделителем утверждений в инструкции `case`. Просто он немного неудобен. Агент Смит просто пнул песок передо мной, дразня меня, и мне даже кажется, что я слышу его издевательский смех. Но ему придется сильно постараться, чтобы заслужить звание «Агент месяца». Давайте перейдем к более простой инструкции `if`.

If

Если инструкция `case` использует сопоставление с образцом, то инструкция `if` использует условные выражения – *ограничители* (guards). В Erlang ограничителем называется условное выражение, которое должно быть удовлетворено, чтобы обеспечить успех сопоставления. Ниже мы познакомимся с применением ограничителей в выражениях сопоставления с образцом, но чаще всего они используются в инструкции `if`. Эта инструкция начинается с ключевого слова `if`, за которым следует несколько выражений-ограничителей вида `guard -> expression`. Вот как она выглядит в общем случае:

```
if
  ProgramsTerminated > 0 ->
    success;
```

```
  ProgramsTerminated < 0 ->
    error
end.
```

Посмотрим, что произойдет, если `if` не найдет совпадения:

```
8> X = 0.
0
9> if
9>     X > 0 -> positive;
9>     X < 0 -> negative
9> end.
** exception error: no true branch found when evaluating an if expression
```

В отличие от Ruby или Io, одно из выражений должно быть истинным, потому что в действительности `if` – это функция. Инструкция обязательно должна возвращать значение. Если возможна ситуация, когда ни одно из имеющихся утверждений не будет выполняться, добавьте в конец ограничитель `true`:

```
9> if
9>     X > 0 -> positive;
9>     X < 0 -> negative;
9>     true -> zero
9> end.
```

Это правило действует только для управляющих структур. Гораздо больше пользы можно получить от функций высшего порядка и сопоставления с образцом, поэтому отложим в сторону управляющие структуры и погрузимся в функциональное программирование. Далее мы займемся исследованием функций высшего порядка и попробуем использовать их для обработки списков. Мы будем учиться с помощью функций решать все более сложные задачи.

Анонимные функции

Как вы наверняка помните, функциями высшего порядка называют такие функции, которые либо возвращают функции, либо принимают их в качестве аргументов. В языке Ruby с функциями высшего порядка используются блоки кода, особенно часто этот прием применяется для обхода списков. В Erlang допускается присваивать переменным произвольные функции и передавать их, подобно данным любых других типов.

Выше вы уже сталкивались с некоторыми из этих понятий, тем не менее далее мы рассмотрим некоторые дополнительные особенности, характерные для языка Erlang, а затем сконструируем несколько вы-

сокоуровневых абстракций. Все начинается с анонимных функций. Ниже показано, как выполняется присваивание функций переменным:

```
16> Negate = fun(I) -> -I end.
#Fun<erl_eval.6.13229925>
17> Negate(1).
-1
18> Negate(-1).
1
```

В строке 16 используется новое для нас ключевое слово `fun`. Оно определяет анонимную функцию. В данном случае функция принимает единственный аргумент `I` и возвращает отрицание `-I`. Эта функция присваивается переменной `Negate`. Еще раз повторю: переменной `Negate` присваивается не значение, возвращаемое функцией, а сама функция.

В этом примере можно видеть две важнейшие идеи. Во-первых, функции могут присваиваться переменным. Это позволяет передавать операции, как простые данные. Во-вторых, чтобы вызвать функцию, хранящуюся в переменной, достаточно указать имя переменной и список аргументов. Обратите также внимание на поддержку динамической типизации. Она избавляет от необходимости интересоваться типом возвращаемого значения функции и, соответственно, от необходимости использовать неудобный синтаксис, который мы видели, например, в Scala. Недостаток подобных функций состоит в том, что они могут терпеть неудачу. Ниже я познакомлю вас с некоторыми приемами, компенсирующими этот недостаток.

Давайте попробуем воспользоваться вновь открывшимися возможностями. Применим анонимные функции в операциях `each`, `map` и `inject`, впервые встретившихся нам во время знакомства с языком Ruby.

Списки и функции высшего порядка

Как было показано выше, списки и кортежи – это сердце и душа функционального программирования. Не случайно, что первый функциональный язык начинался со списков, а все остальное надстраивалось над ними. В этом разделе мы начнем применять к спискам функции высшего порядка.

Применение функций к спискам

К настоящему моменту основные идеи, лежащие в основе обработки списков, должны быть вам знакомы. Далее мы будем учиться при-

менять функции для этой цели. Некоторые из них, такие как `foreach`, выполняют итерации по элементам списков. Другие, такие как `filter` или `map`, возвращают списки, отфильтрованные или отображенные с помощью других функций. Более того, такие функции, как `foldl` и `foldr`, обрабатывают списки, включая в процесс промежуточные результаты, подобно функции `inject` в Ruby или `foldLeft` в Scala. Давайте откроем новое окно консоли, определим один-два списка и приступим.

Прежде всего проверим, как выполняются простые итерации. Метод `lists:foreach` принимает функцию и список. Функция может быть анонимной, как показано ниже:

```
1> Numbers = [1, 2, 3, 4].
[1,2,3,4]
2> lists:foreach(fun(Number) -> io:format("~p~n", [Number]) end, Numbers).
1
2
3
4
ok
```

Инструкция в строке 2 выглядит немного загадочной, поэтому рассмотрим ее пристальнее. В ней вызывается функция `lists:foreach`. В первом аргументе ей передается анонимная функция `fun(Number) -> io:format("~p~n", [Number]) end`. Она принимает один аргумент и выводит любое значение, какое только будет передано, вызовом функции `io:format`[1]. Наконец, во втором аргументе функции `foreach` передается список `Numbers`, созданный в строке 1. При желании анонимную функцию можно было бы определить в отдельной строке:

```
3> Print = fun(X) -> io:format("~p~n", [X]) end.
```

и упростить код, выполняющий обработку списка:

```
8> lists:foreach(Print, Numbers).
1
2
3
4
ok
```

Так выглядит реализация простых итераций. Перейдем теперь к отображению списков с помощью функций. Функция `map` действу-

---

[1] Флаг `~p` означает pretty (форматированный) вывод, `~n` – перевод строки, а `[Number]` – это список аргументов для вывода.

ет подобно функции `collect` в Ruby. Она передает значение каждого элемента списка указанной функции и конструирует список результатов. Как и `lists:foreach`, метод `lists:map` принимает функцию и список. Попробуем применить функцию `map` к нашему списку с числами и увеличить каждый элемент на единицу:

```
10> lists:map(fun(X) -> X + 1 end, Numbers).
[2,3,4,5]
```

Ничего сложного. На этот раз была использована анонимная функция `fun(X) -> X + 1 end`. Она увеличивает каждое значение на единицу, а `lists:map` конструирует список из ее результатов.

Определение функции `map` выглядит очень просто:

```
map(F, [H|T]) -> [F(H) | map(F, T)];
map(F, []) -> [].
```

Отображение функции `F` на список есть `F(head)` плюс `map(F, tail)`. Еще более короткую версию мы увидим, когда познакомимся с генераторами списков (list comprehensions).

Списки можно фильтровать, применяя к ним функции, возвращающие логические значения. Давайте определим анонимную функцию и присвоим ее переменной `Small`:

```
11> Small = fun(X) -> X < 3 end.
#Fun<erl_eval.6.13229925>
12> Small(4).
false
13> Small(1).
true
```

Теперь мы можем применить ее для фильтрации списка. Функция `lists:filter` создаст список с элементами, удовлетворяющими условию `Small`, то есть с элементами, имеющими значение меньше трех:

```
14> lists:filter(Small, Numbers).
[1,2]
```

Как видите, Erlang существенно упрощает выполнение подобных операций. При необходимости функцию `Small` можно было бы использовать для проверки списка с помощью методов `all` и `any`. Метод `lists:all` возвращает `true`, только если все элементы списка удовлетворяют критерию `Small`:

```
15> lists:all(Small, [0, 1, 2]).
true
16> lists:all(Small, [0, 1, 2, 3]).
false
```

Метод lists:any, напротив, возвращает true, если критерию удовлетворяет хотя бы один элемент списка:

```
17> lists:any(Small, [0, 1, 2, 3]).
true
18> lists:any(Small, [3, 4, 5]).
false
```

Посмотрим, какие результаты получатся при выполнении тех же операций с пустым списком:

```
19> lists:any(Small, []).
false
20> lists:all(Small, []).
true
```

Как и ожидалось, метод all вернул true (то есть все элементы списка удовлетворяют критерию, даже при том, что в списке нет ни одного элемента), а метод any вернул false (то есть в списке нет элементов, соответствующих критерию). В таких ситуациях сам критерий не имеет значения.

Существует также возможность отобрать или, наоборот, отбросить все элементы списка, находящиеся в его начале, соответствующие критерию:

```
22> lists:takewhile(Small, Numbers).
[1,2]
23> lists:dropwhile(Small, Numbers).
[3,4]
24> lists:takewhile(Small, [1, 2, 1, 4, 1]).
[1,2,1]
25> lists:dropwhile(Small, [1, 2, 1, 4, 1]).
[4,1]
```

С помощью этого приема можно, например, выбирать или отбрасывать заголовки сообщений. Давайте закончим обзор циклических операций со списками знакомством с методами foldl и foldr.

foldl

Я понимаю, что вы уже встречались с понятиями, описываемыми здесь. Если вы – Нео и сумели взять под контроль эту часть Матрицы, тогда просто ознакомьтесь с простым примером – и в бой. Другим же потребуется некоторое время, чтобы овладеть функцией foldl, поэтому я сделаю еще один заход.

Напомню, что эти функции могут пригодиться для свертывания результатов, возвращаемых некоторой функцией, при выполнении

итераций по списку. Один из аргументов служит аккумулятором (или накопителем), а другой представляет очередной элемент. Метод `lists:foldl` принимает функцию, начальное значение аккумулятора и сам список:

```
28> Numbers.
[1,2,3,4]
29> lists:foldl(fun(X, Sum) -> X + Sum end, 0, Numbers).
10
```

Чтобы упростить код, сохраним анонимную функцию в переменной и дадим ей и другим переменным имена, яснее отражающие наши намерения:

```
32> Adder = fun(ListItem, SumSoFar) -> ListItem + SumSoFar end.
#Fun<erl_eval.12.113037538>
33> InitialSum = 0.
0
34> lists:foldl(Adder, InitialSum, Numbers).
10
```

Так намного лучше. Итак, нам требуется подсчитать сумму элементов списка. В каждый вызов функции `Adder` будут передаваться сумма `SumSoFar`, накопленная к данному моменту, и очередной элемент `ListItem` из списка `Numbers`. С каждым увеличением суммы функция `lists:foldl` будет запоминать ее и передавать в следующий вызов `Adder`. В конечном итоге функция вернет накопленную сумму.

До сих пор вы видели только функции, работающие с существующими списками. Я еще не демонстрировал, как можно конструировать списки, поэтому давайте сменим направление и познакомимся с возможностью создания списков.

Дополнительные средства для работы со списками

Все понятия, касающиеся списков, что были представлены мною, являются всего лишь расширениями идей, встречавшихся нам во время знакомства с другими языками. Однако существуют более сложные понятия. До сих пор еще ни слова не было сказано о возможности конструирования списков, и пока что мы использовали лишь самые простые абстракции, имеющие форму блоков кода.

Конструирование списков

На первый взгляд может показаться, что отсутствие поддержки изменяемого состояния может существенно осложнить создание списков. В Ruby или Io можно просто последовательно добавлять новые

элементы в список. Однако существует иной способ – вернуть новый список с добавленными в него элементами. Чаще всего новые элементы добавляются в начало списка. Для этой цели можно использовать конструкцию `[H|T]`, но с правой стороны сопоставления. Следующая программа использует именно этот прием для удвоения значения каждого элемента в исходном списке:

erlang/double.erl
http://media.pragprog.com/titles/btlang/code/erlang/double.erl

```
-module(double).
-export([double_all/1]).

double_all([]) -> [];
double_all([First|Rest]) -> [First + First|double_all(Rest)].
```

Данный модуль экспортирует одну функцию с именем `double_all`. Эта функция состоит из двух утверждений. Первое говорит, что для пустого списка `double_all` должна возвращать пустой список. Это правило останавливает рекурсию.

Второе утверждение использует конструкцию `[H|T]` как в предикате сопоставления, так и в определении функции. Вы уже встречались с конструкцией, похожей на `[First|Rest]`, в левой части сопоставления. Она позволяет разбить исходный список на «голову» (первый элемент) и «хвост» (остальная часть списка).

Когда эта конструкция располагается справа, ее действие меняется на обратное – она не разбивает список, а, наоборот, собирает его. В данном примере конструкция `[First + First|double_all(Rest)]` создает список, в котором первый элемент является суммой `First + First` первого элемента в исходном списке, а остальные элементы воспроизводятся рекурсивным вызовом `double_all(Rest)`.

Скомпилируйте и запустите эту программу:

```
8> c(double).
{ok,double}
9> double:double_all([1, 2, 3]).
[2,4,6]
```

Рассмотрим еще один способ конструирования списков с помощью оператора `|`:

```
14> [1| [2, 3]].
[1,2,3]
15> [[2, 3] | 1].
[[2,3]|1]
16> [[] | [2, 3]].
```

```
[[],2,3]
17> [1 | []].
[1]
```

Здесь нет никаких сюрпризов. Второй аргумент должен быть списком. Левый операнд (чем бы он не был) просто добавляется в начало нового списка.

Давайте познакомимся с еще более сложным понятием, существующим в языке Erlang, которое называется *генераторы списков* (list comprehensions). Оно включает в себя некоторые понятия, с которыми мы уже встречались выше.

Генераторы списков

Одной из важнейших функций практически во всех функциональных языках является функция `map`. Благодаря ей списки могут изменяться, подобно сущностям в Матрице. Учитывая важность этой функции, в Erlang была добавлена более мощная и краткая ее форма, позволяющая выполнять сразу несколько трансформаций.

Начнем с того, что запустим новую консоль и попробуем задействовать старую, добрую функцию `map`:

```
1> Fibs = [1, 1, 2, 3, 5].
[1,1,2,3,5]
2> Double = fun(X) -> X * 2 end.
#Fun<erl_eval.6.13229925>
3> lists:map(Double, Fibs).
[2,2,4,6,10]
```

У нас имеются список с именем `Fibs` и анонимная функция, хранящаяся в переменной `Double`, которая удваивает все, что ей передается. Далее мы вызвали функцию `lists:map` и передали ей функцию `Double` с целью создать новый список, содержащий удвоенные элементы исходного списка. Функция `map` оказалась настолько замечательным и востребованным инструментом, что разработчики Erlang включили в язык более краткую ее форму с более удобным синтаксисом. Новая конструкция получила название *генератор списков* (list comprehension). Ниже демонстрируются операции, эквивалентные тем, что приводились выше, но использующие генератор списков:

```
4> [Double(X) || X <- Fibs].
[2,2,4,6,10]
```

Выражаясь простым языком, здесь мы сказали: «удвоить (`Double`) значение (`X`) каждого элемента (`X`) исходного списка с именем `Fibs`». При желании можно избавиться от промежуточной функции:

```
5> [X * 2 || X <- [1, 1, 2, 3, 5]].
[2,2,4,6,10]
```

Суть от этого не изменилась. Здесь вычисляется произведение `X * 2` для каждого элемента `X` исходного списка `[1, 1, 2, 3, 5]`. Эта особенность – не просто синтаксический сахар. Давайте попробуем реализовать чуть более сложный генератор списков. Начнем с краткого определения функции `map`:

```
map(F, L) -> [ F(X) || X <- L].
```

На простом языке эта инструкция гласит: «отображением некоторой функции `F` на некоторый список `L` является коллекция `F(X)`, где `X` представляет каждый элемент списка `L`». Теперь задействуем генератор списков для обработки перечня товаров, содержащего названия товаров, их количества и цены:

```
7> Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}].
[{pencil,4,0.25},{pen,1,1.2},{paper,2,0.2}]
```

Допустим, что нам требуется прибавить налог, составляющий восемь центов на каждый доллар. Для этого можно было бы сконструировать простой генератор списков, возвращающий новый перечень с суммой налога для каждого товара:

```
8> WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
8>    {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

Понятия языка Erlang, изученные нами выше, никуда не делись – здесь по-прежнему действует механизм сопоставления! Говоря простым языком, мы возвращаем список кортежей, включающих наименование товара (`Product`), цену (`Price`), количество (`Quantity`) и сумму налога (`Price * Quantity * 0.08`), для списка `Cart` кортежей вида `{Product, Quantity, Price}`. Мне очень нравится этот код! Такой синтаксис позволяет мне изменять форму списков буквально в мгновение ока.

Как еще один пример допустим, что у нас имеется перечень товаров и для наших постоянных клиентов нужно воспроизвести такой же перечень, содержащий цены с 50%-ной скидкой. Собственно, перечень можно создать так (я просто скопировал один перечень в другой, игнорируя количество каждого товара):

```
10> Cat = [{Product, Price} || {Product, _, Price} <- Cart].
[{pencil,0.25},{pen,1.2},{paper,0.2}]
```

Выражаясь простым языком, мы потребовали вернуть список кортежей с наименованиями товаров (`Product`) и ценами (`Price`) из исходного списка `Cart`, опустив в каждом исходном кортеже второй атрибут – количество товаров. Теперь можно рассчитать скидку:

```
11> DiscountedCat = [{Product, Price / 2} || {Product, Price} <- Cat].
[{pencil,0.125},{pen,0.6},{paper,0.1}]
```

Кратко, удобно, читаемо. У нас получилась великолепная абстракция.

По правде говоря, я показал вам лишь малую часть возможностей генераторов списков. В полной форме они способны на большее:

- в общем виде генератор списков имеет форму `[Expression || Clause1, Clause2, ..., ClauseN]`;
- генераторы списков могут содержать произвольное количество утверждений;
- утверждения могут быть генераторами значений или фильтрами;
- фильтр может быть логическим выражением или функцией, возвращающей логическое значение;
- генератор значений вида `Match <- List` сопоставляет образец слева с элементами списка справа.

В действительности все не так сложно. Генераторы добавляют, а фильтры удаляют значения. Здесь ярко проявляется влияние языка Prolog. Генераторы выявляют допустимые значения, а фильтры выбрасывают из списка значения, не удовлетворяющие условиям. Например:

```
22> [X || X <- [1, 2, 3, 4], X < 4, X > 1].
[2,3]
```

На простом языке это звучит так: «вернуть `X`, если `X` меньше четырех и больше одного, где `X` – это значение элемента из списка `[1, 2, 3, 4]`». Генераторов может быть несколько:

```
23> [{X, Y} || X <- [1, 2, 3, 4], X < 3, Y <- [5, 6]].
[{1,5},{1,6},{2,5},{2,6}]
```

Этот генератор списков создает кортежи `{X, Y}`, объединяя значения `X` из списка `[1, 2, 3, 4]`, которые меньше трех, со значениями `Y` из списка `[5, 6]`. В результате остается два значения `X` и два значения `Y`.

Вот, собственно, и все. Вы познакомились с основными приемами программирования последовательных вычислений на языке Erlang. А теперь прервемся и немного попрактикуемся.

Что мы узнали во второй день

Следует заметить, что мы не занимались сколько-нибудь глубокими исследованиями выражений языка Erlang или его библиотек, но вы уже обладаете достаточным объемом знаний, чтобы писать собственные функциональные программы. Этот день мы начали с исследования некоторых обычных управляющих структур, а затем быстро начали набирать темп.

Затем мы продолжили знакомство с функциями высшего порядка. Они помогли нам организовать обход элементов списка, их фильтрацию и изменение. Мы также узнали, как с помощью `foldl` производить свертку результатов.

Под конец мы перешли к знакомству с дополнительными средствами для работы со списками. Мы использовали конструкцию `[H|T]` слева от оператора сопоставления, чтобы разложить его на составляющие (первый элемент и оставшуюся часть списка). Потом мы использовали конструкцию `[H|T]` справа от оператора сопоставления или отдельно, чтобы сконструировать новый список. Далее мы исследовали генераторы списков – изящную и мощную абстракцию для быстрого преобразования списков с применением генераторов значений и фильтров.

Изученный нами синтаксис оставил смешанные впечатления. С одной стороны, он позволил нам переходить ко все более высокоуровневым абстракциям за счет совсем небольшого количества кода и благодаря динамической типизации, поддерживаемой языком Erlang. С другой стороны, мы наблюдали некоторые неудобства, особенно это касается точек с запятой в разных утверждениях инструкций `case` и `if`.

В следующем разделе мы приступим к изучению того, ради чего была вся эта суета, – параллельных вычислений.

День 2: задания для самостоятельного решения

Практические задания:

- представьте, что имеется список кортежей, состоящих из пар ключ/значение, таких как `[{erlang, "a functional language"}, {ruby, "an OO language"}]`. Напишите функцию, принимающую список и ключевое слово и возвращающую значение, соответствующее этому ключевому слову;
- представьте, что имеется список продуктов, которые нужно купить, такой как `[{item quantity price}, ...]`. Напишите генера-

тор списков, создающий список вида `[{item total_price}, ...]`, где `total_price` – это произведение цены на количество.

Дополнительное задание:

- напишите программу, которая читала бы содержимое игрового поля для игры в «крестики-нолики», представленного в виде списка или кортежа с девятью элементами, и возвращала бы победителя (`x` или `o`), если игра завершилась чьей-либо победой, значение `cat`, если победитель не определяется и может быть сделан хотя бы еще один ход, и `no_winner`, если игра завершилась вничью.

6.4. День 3: Красная таблетка

Большинство из вас уже знают об этом. Если в Матрице принять синюю таблетку, воспоминания о Матрице будут стерты и вы продолжите существование в блаженном неведении. Если принять красную таблетку, реальность откроется перед вашими глазами. Иногда реальность может быть очень жестокой.

У нас работает целая индустрия, выпускающая синие таблетки как из пулемета. Организация параллельных вычислений является достаточно сложной темой, поэтому будем двигаться не спеша. Программы, имеющие изменяемое состояние, конфликтуют между собой, когда выполняются параллельно. Функции и методы, имеющие побочные эффекты, не позволяют убедиться в правильной их работе или предсказать их результаты. Для достижения высокой производительности мы используем потоки с разделяемым состоянием, а не процессы, не имеющие общих ресурсов, из-за чего приходится защищать каждый фрагмент кода.

В результате возникает хаос. Параллельные вычисления сложны в реализации, но не потому, что они сложны сами по себе, а потому, что мы выбираем неправильную модель программирования!

Выше в этой главе я говорил, что Erlang делает некоторые простые вещи сложными. Не имея возможности создавать побочные эффекты или изменять состояние, приходится менять подходы к программированию. Вам пришлось привыкать к синтаксису в стиле языка Prolog, который многим кажется чужеродным. Но сейчас вы получаете награду. Красная таблетка, дающая знание правильной организации параллельных вычислений и способов достижения высочайшей надежности, уже не выглядит устрашающей и воспринимается, скорее, как леденец. Так примите же ее скорее.

Основные примитивы параллельных вычислений

В вашем распоряжении имеются три основных примитива параллельных вычислений, позволяющих отправлять сообщения (с помощью оператора `!`), порождать процессы (с помощью `spawn`) и принимать сообщения (с помощью `receive`). В этом разделе я покажу, как с их помощью отправлять и принимать сообщения и как они соотносятся с простой идиомой клиент-серверных вычислений.

Простой цикл приема сообщений

Итак, рассмотрим процесс, выполняющий перевод текста с одного языка на другой. Если послать процессу строку с текстом на испанском языке, он вернет строку с тем же текстом на английском. Суть стратегии заключается в том, чтобы создать процесс, принимающий и обрабатывающий сообщения в цикле.

Ниже показано, как может выглядеть простой цикл приема сообщений:

erlang/translate.erl
http://media.pragprog.com/titles/btlang/code/erlang/translate.erl

```
-module(translate).
-export([loop/0]).

loop() ->
    receive
        "casa" ->
            io:format("house~n" ),
            loop();

        "blanca" ->
            io:format("white~n" ),
            loop();

        _ ->
            io:format("I don't understand.~n" ),
            loop()
end.
```

Этот пример получился более длинным, чем предыдущие, поэтому давайте разберем его на составляющие. Первые две строки просто определяют модуль с именем `translate` и экспортируют функцию `loop`. Следующий блок кода – это определение функции `loop()`:

```
loop() ->
    ...
end.
```

Обратите внимание, что код внутри трижды вызывает функцию `loop()` и не возвращает управление. Это нормально: компилятор Erlang автоматически оптимизирует хвостовую рекурсию, поэтому накладные расходы на рекурсивные вызовы сводятся к минимуму, пока все утверждения внутри `receive` завершаются вызовом `loop()`. Мы фактически определили пустую функцию, выполняющую бесконечный цикл. Перейдем к блоку `receive`:

```
receive ->
    ...
```

Это – функция, принимающая сообщение, отправленное другим процессом. Функция `receive` действует так же, как и другие конструкции сопоставления с образцом в языке Erlang, такие как инструкция `case` и определения функций. Внутри `receive` присутствует несколько конструкций сопоставления. Рассмотрим одну из них:

```
"casa" ->
    io:format("house~n"),
    loop();
```

Это – утверждение сопоставления. Их синтаксис соответствует синтаксису инструкции `case`. Если входящее сообщение совпадет со строкой `"casa"`, Erlang выполнит следующий за ней код. Отдельные строки внутри утверждения разделяются символом запятой (`,`), а само утверждение завершается символом точки с запятой (`;`). Код в этом утверждении выводит слово *house* и затем вызывает `loop`. (Напомню, что здесь не происходит расходования памяти на стеке, потому что вызов `loop` является последней инструкцией в утверждении.) Остальные утверждения действуют аналогичным образом.

Теперь у нас имеется модуль с циклом `receive`. Можно приступать к его использованию.

Порождение процессов

Сначала скомпилируйте модуль:

```
1> c(translate).
{ok,translate}
```

Породить (то есть запустить) процесс можно с помощью функции `spawn`, которая принимает функцию. Эта функция будет выполнена в рамках нового легковесного процесса. Функция `spawn` вернет идентификатор процесса (Process ID, PID). Давайте вызовем ее и передадим ей функцию из модуля `translate`:

```
2> Pid = spawn(fun translate:loop/0).
<0.38.0>
```

Как видите, Erlang вернул идентификатор процесса <0.38.0>. В консоли идентификатор процесса выводится в угловых скобках. Мы будем рассматривать только самый простой случай запуска процессов, но вы должны знать, что существуют и другие. Процесс можно зарегистрировать под определенным именем, чтобы другие процессы могли искать его по этому имени, а не по идентификатору процесса. Можно также использовать иную версию `spawn` для запуска кода, который можно будет заменить прямо во время выполнения, то есть произвести «горячую» замену. Для запуска удаленных процессов можно использовать версию функции `spawn(Node, function)`. Впрочем, не будем углубляться в детали, так как их обсуждение выходит далеко за рамки данной книги.

Итак, мы написали модуль с блоком кода и запустили этот код в виде легковесного процесса. Теперь осталось только передать ему сообщение. Делается это с помощью третьего примитива.

Отправка сообщений

Отправка сообщений в Erlang выполняется с помощью оператора `!`, который вы уже видели в главе с обсуждением языка Scala. Вызов оператора имеет вид `Pid ! message`, где `Pid` – это идентификатор любого процесса, а `message` – любое значение, которое может быть значением простого типа, списком или кортежем. Давайте отправим сообщение:

```
3> Pid ! "casa".
"house"
"casa"
4> Pid ! "blanca".
"white"
"blanca"
5> Pid ! "loco".
"I don't understand."
"loco"
```

В каждой строке осуществляется отправка одного сообщения. В примере видно, что сначала выводится перевод, вызовом `io:format` в одном из утверждений в `receive`, а затем интерактивная оболочка выводит значение, возвращаемое выражением, каковым является отправленное сообщение.

Для передачи распределенных сообщений именованным ресурсам следует использовать синтаксис `node@server ! message`. Тема настройки

удаленного сервера выходит далеко за рамки этой книги, но заслуживает, чтобы вы самостоятельно занялись ее исследованием в конце дня. Изучив ее, вы получите возможность обрабатывать сообщения с помощью удаленного сервера.

Этот пример иллюстрирует, как объединить простые примитивы параллельных вычислений, чтобы на их основе реализовать простую асинхронную службу. Возможно, вы заметили, что здесь ничего не возвращается. В следующем разделе мы узнаем, как отправлять синхронные сообщения.

Обмен синхронными сообщениями

Некоторые многозадачные системы, такие как телефонные чаты, работают асинхронно. Система-отправитель посылает сообщение и переходит к другим делам, не дожидаясь ответа. Другие работают синхронно, как, например, веб-браузер. Браузер запрашивает страницу и ждет, пока веб-сервер не вернет ее. Давайте превратим службу перевода, которая выводит результат, в службу, действительно возвращающую строку с переводом.

Чтобы изменить модель обмена сообщениями с асинхронной на синхронную, мы выберем стратегию с промежуточным звеном.

- Каждое утверждение в `receive`, в службе обработки сообщений, должно получать кортеж с идентификатором процесса, запросившего перевод, и словом для перевода. Добавление идентификатора позволит нам организовать отправку ответа.
- Каждое утверждение в `receive` должно посылать ответ отправителю вместо вывода его в консоль.
- Вместо примитива `!` для оправки запроса и ожидания ответа мы будем использовать простую функцию, которую напишем сами.

Алгоритм действий понятен, теперь давайте посмотрим на фрагменты его реализации.

Синхронный прием

Прежде всего необходимо изменить утверждения в `receive`, чтобы обеспечить прием дополнительных параметров. В данном примере мы решили использовать кортежи. Они легко укладываются в схему сопоставления с образцом. Каждое утверждение в `receive` теперь будет выглядеть так:

```
receive
    {Pid, "casa"} ->
```

```
        Pid ! "house",
        loop();
        ...
```

Мы сопоставляем принятое сообщение с кортежем, содержащим произвольный элемент (это всегда должен быть идентификатор процесса), за которым следует слово *casa*. При наличии совпадения мы отправляем слово *house* отправителю и возвращаемся в начало цикла.

Обратите внимание на образец в этом утверждении. Это – типичная форма для блоков `receive`, когда идентификатор процесса-отправителя передается в первом элементе кортежа. Другое важное отличие от предыдущей реализации – вместо вывода перевода функция теперь посылает ответ отправителю. Изменения в службе оказались небольшими, чего нельзя сказать про отправителя.

Синхронная отправка

На стороне клиента необходимо выполнить отправку сообщения и сразу же перейти в режим ожидания ответа. Если исходить из того, что идентификатор процесса хранится в переменной `Receiver`, синхронная отправка сообщения может выглядеть как-то так:

```
Receiver ! "message_to_translate",
    receive
        Message -> do_something_with(Message)
    end
```

Поскольку отправка сообщений будет выполняться довольно часто, давайте упростим работу со службой, заключив обращение к серверу в функцию. В нашем случае эта функция будет выглядеть так:

```
translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

Объединив все эти фрагменты воедино, мы получим программу, лишь немногим сложнее предыдущей.

erlang/translate_service.erl
http://media.pragprog.com/titles/btlang/code/erlang/translate_service.erl

```
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
```

```
        {From, "casa" } ->
            From ! "house" ,
            loop();

        {From, "blanca" } ->
            From ! "white" ,
            loop();
        {From, _} ->
            From ! "I don't understand." ,
            loop()
    end.

    translate(To, Word) ->
        To ! {self(), Word},
        receive
            Translation -> Translation
        end.
```

Ниже показан порядок использования этой модели:

```
1> c(translate_service).
{ok,translate_service}
2> Translator = spawn(fun translate_service:loop/0).
<0.38.0>
3> translate_service:translate(Translator, "blanca").
"white"
4> translate_service:translate(Translator, "casa").
"house"
```

Мы просто скомпилировали код, запустили цикл и выполнили пару запросов к синхронной службе с помощью вспомогательной функции, написанной нами. Как видите, теперь процесс `Translator` возвращает переведенное слово и механизм обмена синхронными сообщениями.

Сейчас можно рассмотреть структуру простого цикла приема `receive`. Каждый процесс имеет своеобразный «почтовый ящик». Конструкция `receive` извлекает сообщения из этого почтового ящика (очереди) и передает их некоторой функции для обработки. Процессы взаимодействуют друг с другом, обмениваясь сообщениями. Доктор Армстронг не оговорился, когда назвал Erlang настоящим объектно-ориентированным языком! Он позволяет посылать сообщения и связывать их с некоторым кодом. Мы просто лишились изменяемого состояния и механизма наследования, хотя есть возможность имитировать наследование и многое другое с помощью функций высшего порядка.

До сих пор мы никак не касались темы надежности и восстановления после ошибок. В Erlang существует поддержка контролируемых исключений (checked exceptions), но я хочу провести вас другим путем и показать иной способ обработки ошибок.

Связывание процессов для повышения надежности

В этом разделе мы рассмотрим прием повышения надежности посредством связывания процессов. В Erlang существует возможность связать два процесса вместе. Всякий раз, когда один процесс в такой паре завершается, он посылает сигнал другому, связанному с ним процессу. Другой процесс может принять сигнал и среагировать в соответствии с ситуацией.

Запуск связанного процесса

Чтобы увидеть, как действует механизм связывания процессов, создадим процесс, который легко может завершиться. Я написал игру в русскую рулетку. Согласно правилам игры, имеется 6-зарядный пистолет (процесс `Gun`). Чтобы спустить курок, достаточно послать число от 1 до 6 процессу `Gun`. Стоит ввести недопустимое число, и процесс «убьет себя» (завершится):

erlang/roulette.erl
http://media.pragprog.com/titles/btlang/code/erlang/roulette.erl

```
-module(roulette).
-export([loop/0]).

% send a number, 1-6
loop() ->
    receive
        3 -> io:format("bang.~n" ), exit({roulette,die,at,erlang:ti
me()});
        _ -> io:format("click~n" ), loop()
end.
```

Реализация очень проста. У нас имеется цикл приема и обработки сообщений. При совпадении с числом `3` выполняется код `io:format("bang~n"), exit({roulette,die,at,erlang:time()};`, завершающий процесс. Все остальное – это всего лишь реализация вывода сообщения и возврат в начало цикла.

В действительности мы создали клиент-серверную программу. Роль клиента здесь играет консоль, а роль сервера – процесс `roulette`, как показано на рис. 6.1.

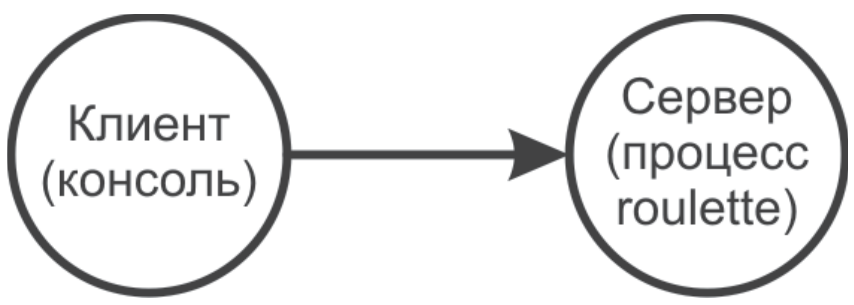

Рис. 6.1 ❖ Простая архитектура клиент-сервер

Вот как выглядит сеанс взаимодействия:

```
1> c(roulette).
{ok,roulette}
2> Gun = spawn(fun roulette:loop/0).
<0.38.0>
3> Gun ! 1.
click
1
4> Gun ! 3.
bang
3
5> Gun ! 4.
4
6> Gun ! 1.
1
```

Проблема в том, что после отправки числа 3 процесс `Gun` завершается и уже не может обработать следующие сообщения. Проверить наличие процесса можно следующим образом:

```
7> erlang:is_process_alive(Gun).
false
```

Да, процесс действительно завершился. Пришло время взять вожжи в свои руки. Мы можем исправить неприятную ситуацию. Давайте создадим процесс-монитор, который будет сообщать нам о завершении подконтрольного процесса. Как мне кажется, это будет скорее патологоанатом, а не монитор, поскольку пока нас интересует только сам факт «смерти» процесса.

Вот как выглядит реализация монитора:

erlang/coroner.erl
http://media.pragprog.com/titles/btlang/code/erlang/coroner.erl

```
-module(coroner).
-export([loop/0]).

loop() ->
    process_flag(trap_exit, true),
```

```
    receive
        {monitor, Process} ->
            link(Process),
            io:format("Monitoring process.~n" ),
            loop();

        {'EXIT', From, Reason} ->
            io:format("The shooter ~p died with reason ~p." ,
                      [From, Reason]),
            io:format("Start another one.~n" ),
            loop()
    end.
```

Как обычно, основу монитора составляет цикл `receive`. Прежде чем сделать что-то еще, программа должна зарегистрироваться как обработчик сигнала выхода. Без этого она не будет получать сообщения `EXIT`.

Затем запускается цикл приема сообщений `receive`. В данном случае конструкция `receive` принимает два вида кортежей: один из них в первом элементе содержит атом `monitor`, а второй – строку `'EXIT'`. Давайте рассмотрим этот код поближе.

```
{monitor, Process} ->
    link(Process),
    io:format("Monitoring process.~n"),
    loop();
```

Этот код связывает процесс патологоанатома с любым другим процессом, посылающим свой идентификатор в переменной `Process`. Имеется также возможность запустить уже связанный процесс, воспользовавшись функцией `spawn_link`. Если теперь подконтрольный процесс вынужден будет завершиться, он отправит сообщение данному патологоанатому. Перейдем к утверждению, перехватывающему ошибку:

```
{'EXIT', From, Reason} ->
    io:format("The shooter died. Start another one.~n"),
    loop()
end.
```

Этот код проверяет совпадение с текстом сообщения о выходе. Таковым сообщением является кортеж, содержащий три элемента: строку `'EXIT'`, идентификатор процесса `From` и информацию о причине завершения. Здесь мы выводим идентификатор процесса и причину. Ниже показано, как действует эта связка:

```
1> c(roulette).
{ok,roulette}
```

```
2> c(coroner).
{ok,coroner}
3> Revolver=spawn(fun roulette:loop/0).
<0.43.0>
4> Coroner=spawn(fun coroner:loop/0).
<0.45.0>
5> Coroner ! {monitor, Revolver}.
Monitoring process.
{monitor,<0.43.0>}
6> Revolver ! 1.
click
1
7> Revolver ! 3.
bang.
3
The shooter <0.43.0> died with reason
{roulette,die,at,{8,48,1}}. Start another one.
```

Итак, мы реализовали модель, более сложную, чем простая модель клиент-сервер. Мы добавили процесс-монитор, как показано на рис. 6.2, и теперь можем определить момент завершения процесса.

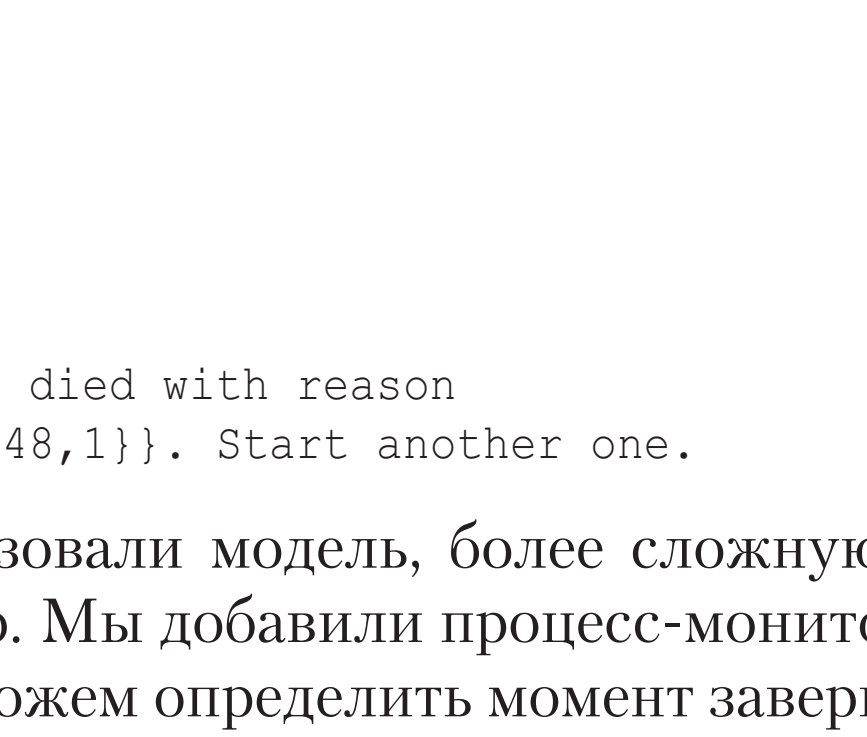

Рис. 6.2 ❖ Добавление процесса-монитора

Превращение патологоанатома во врача

Мы можем улучшить реализацию. Если зарегистрировать пистолет (любопытная игра слов получилась) под определенным именем, это избавит игроков от необходимости знать идентификатор процесса-пистолета. В этом случае можно будет перенести запуск процесса внутрь процесса-патологоанатома, и у него появится возможность повторно запускать процесс после его завершения. В результате мы увеличим надежность работы всей связки, не утруждая себя тонкостями, связанными с обработкой ошибок. Теперь наш патологоанатом уже

не просто патологоанатом, а самый настоящий врач, способный воскрешать умерших. Ниже приводится реализация нового врача:

erlang/doctor.erl
http://media.pragprog.com/titles/btlang/code/erlang/doctor.erl

```
-module(doctor).
-export([loop/0]).

loop() ->
    process_flag(trap_exit, true),
    receive
        new ->
            io:format("Creating and monitoring process.~n" ),
            register(revolver, spawn_link(fun roulette:loop/0)),
            loop();

        {'EXIT', From, Reason} ->
            io:format("The shooter ~p died with reason ~p." ,
                      [From, Reason]), io:format(" Restarting. ~n" ),
            self() ! new,
            loop()
        end.
```

Блок `receive` теперь проверяет совпадение с двумя сообщениями: с ключевым словом `new` и с кортежем `EXIT`. Оба они отличаются от сообщений, которые обрабатывались процессом-патологоанатомом. Основное волшебство творится в следующей строке кода, в блоке `new`:

```
register(revolver, spawn_link(fun roulette:loop/0)),
```

Смотреть надо в направлении изнутри наружу. Здесь с помощью `spawn_link` запускается новый процесс. Эта версия функции `spawn` сразу устанавливает связь между процессами, благодаря которой процесс-врач `doctor` будет извещаться о гибели пациента – процесса `roulette`. Затем регистрируется идентификатор процесса – устанавливается его связь с атомом `revolver`. Теперь пользователи смогут посылать сообщения процессу, используя конструкцию `revolver ! message`. Идентификатор процесса стал ненужным. Блок обработки сообщения `EXIT` также немного усложнился. В нем появилась новая строка:

```
self() ! new,
```

Она отправляет сообщение текущему процессу, в результате чего происходят запуск и регистрация нового «пистолета». Процесс игры тоже заметно упростился:

```
2> c(doctor).
{ok,doctor}
3> Doc = spawn(fun doctor:loop/0).
<0.43.0>
4> revolver ! 1.
** exception error: bad argument
    in operator !/2
        called as revolver !1
```

Как и следовало ожидать, процесс-пистолет не запустился сам собой, поэтому возникла ошибка. Давайте запустим и зарегистрируем процесс:

```
5> Doc ! new.
Creating and monitoring process.
new
6> revolver ! 1.
click
1
7> revolver ! 3.
bang.
3
The shooter <0.47.0> died with reason {roulette,die,at,{8,53,40}}.
    Restarting.
Creating and monitoring process.
8> revolver ! 4.
click
4
```

Теперь мы вынуждены делать совершенно нелепый шаг запуска процесса-пистолета `revolver`, отправляя сообщение процессу-врачу `Doc`. Мы взаимодействуем с процессом-пистолетом, отправляя сообщения посредством атома `revolver`, совершенно не беспокоясь о значении идентификатора процесса `gun`. Ниже строки 8 можно также видеть, что был запущен и зарегистрирован новый процесс `revolver`. Общая схема взаимодействий между процессами осталась практически той же, что изображена на рис. 6.2 выше, только теперь процесс-врач `doctor` играет более активную роль, чем процесс-патологоанатом.

Мы охватили только самую верхушку айсберга, но я надеюсь, что вы смогли понять, насколько язык Erlang способен упростить создание намного более надежных многозадачных систем. В своих примерах нам вообще не пришлось заниматься обработкой ошибок. Когда какой-то процесс завершается аварийно, мы можем просто запустить новый. Не составляет никакого труда реализовать мониторы, наблюдающие за работоспособностью друг друга. Фактически основные

библиотеки включают множество инструментов для разработки служб, осуществляющих мониторинг и обеспечивающих автоматический перезапуск процессов при любых ошибках в них.

Что мы узнали в третий день

В третий день мы начали полнее понимать возможности языка Erlang. Сначала мы познакомились с примитивами параллельных вычислений: `send`, `receive` и `spawn`. На их основе мы создали естественную версию службы перевода слов с одного языка на другой, чтобы проиллюстрировать работу механизма обмена сообщениями. Затем мы написали простую вспомогательную функцию, инкапсулирующую операции отправки и приема сообщений для имитации вызова удаленных процедур.

Далее мы организовали связывание процессов, чтобы показать, как один процесс может сообщать другому процессу о своем завершении. Мы также познакомились с приемами мониторинга одного процесса из другого, обеспечивающими увеличение надежности работы системы в целом. Первоначально наша система не была устойчива к ошибкам, хотя идеи, использованные при ее создании, применяются при разработке отказоустойчивых систем. Механизм распределенных взаимодействий в языке Erlang действует точно так же, как механизм взаимодействий между локальными процессами. Мы могли бы связать два процесса, выполняющихся на разных компьютерах, чтобы из одного из них постоянно следить за работой другого и своевременно получать извещения о возникающих проблемах.

Давайте попробуем применить на практике все, что мы узнали в третий день.

День 2: задания для самостоятельного решения

Упражнения, следующие ниже, относительно просты, но я добавил несколько дополнительных каверзных вопросов.

Библиотека OTP (Open Telecom Platform – открытая телекоммуникационная платформа) – очень мощный пакет инструментов для создания распределенных и многозадачных служб.

Найдите:

- службу на основе OTP, которая перезапускает процессы, если они завершаются по ошибке;
- документацию, описывающую создание простого сервера на основе OTP.

Практические задания:

- напишите монитор для службы `translate_service`, который перезапускал бы ее;
- добавьте в реализацию процесса `Doctor` перезапуск самого себя;
- напишите монитор для монитора `Doctor`; они должны перезапускать друг друга, если какой-либо из них неожиданно завершается.

Дополнительные вопросы, требующие дополнительных самостоятельных исследований:

- как создать простой сервер на основе OTP, сохраняющий сообщения в файле;
- добавьте в службу `translate_service` возможность работы в сети.

6.5. В заключение об Erlang

В начале этой главы я сказал, что Erlang делает сложное простым, а простое – сложным. Prolog-подобный синтаксис может показаться чуждым для знакомых с обширным семейством C-подобных языков, к тому же парадигма функционального программирования не лишена собственных сложностей.

Но Erlang обладает некоторыми особенностями, значимость которых будет расти с развитием вычислительной техники по пути распараллеливания вычислений. Некоторые из особенностей носят преимущественно философский характер. Легковесные процессы совершенно не согласуются с моделью потоков выполнения в Java. Философия «позвольте приложению потерпеть неудачу» помогает значительно упростить программный код, но требует наличия поддержки в виртуальной машине, которая отсутствует в других системах. Давайте подробнее разберем основные достоинства и недостатки.

Основные сильные стороны

Язык Erlang изначально проектировался для организации параллельных вычислений и создания отказоустойчивых систем. Поскольку развитие процессорной техники пошло по пути наращивания числа ядер, в том же направлении должно развиваться и программирование. Сильные стороны языка Erlang как раз охватывают наиболее важные области, в которых предстоит работать текущему поколению программистов.

Динамичность и надежность

Первое и самое важное: Erlang обеспечивает высокую надежность. Базовые библиотеки хорошо протестированы, а приложения на Erlang отличаются высочайшей надежностью в сравнении с приложениями на других языках. Самое интересное, что разработчики языка достигли такой высокой надежности не в ущерб стратегиям динамической типизации, делающим Erlang таким продуктивным. Вместо подстраховки со стороны компилятора отказоустойчивость Erlang достигается за счет связывания параллельно выполняющихся процессов. Я был поражен простотой создания надежных мониторов без использования тех или иных хитростей, присущих операционным системам.

Я уверен, что вы найдете уникальным и привлекательным набор компромиссов, заложенных в язык Erlang. Язык и виртуальная машина Java не обладают нужным набором примитивов, чтобы хотя бы даже приблизиться к языку Erlang по таким показателям, как производительность и надежность. Библиотеки, основанные на виртуальной машине BEAM, также отражают эту философию, поэтому с их помощью так легко строить надежные распределенные системы.

Легковесные процессы, не имеющие разделяемых данных

Еще одной блистательной чертой Erlang является модель процессов, лежащая в основе всего и вся. Процессы в Erlang легковесны, поэтому программисты на Erlang используют их очень часто. Erlang опирается на философию принудительной неизменяемости, именно поэтому вероятность фатальных конфликтов в системах чрезвычайно низка. Парадигма взаимодействий на основе передачи сообщений с применением примитивов, встроенных в язык, упрощает реализацию приложений с высокой степенью изоляции, которую редко можно найти в объектно-ориентированных приложениях.

OTP, промышленные библиотеки

Поскольку язык Erlang родился и рос в недрах компании, занимающейся созданием телекоммуникационных систем с высочайшими требованиями к доступности и надежности, в него был заложен двадцатилетний опыт создания библиотек, поддерживающих такой же стиль разработки. Основной библиотекой является Open Telecom Platform (OTP). В ней вы найдете инструменты, которые помогут вам

создавать мониторы для поддержания жизнеспособности процессов, подключаться к базам данных и конструировать распределенные системы. В состав OTP входит полноценный веб-сервер и множество других средств, необходимых для создания телекоммуникационных приложений.

Главными достоинствами комплекта библиотек являются отказоустойчивость, масштабируемость, целостность и поддержка «горячей» замены кода. Вам не придется беспокоиться об этих особенностях, ваши серверные процессы просто получают их в свое распоряжение бесплатно.

Позвольте приложению потерпеть неудачу

Философия «позвольте приложению потерпеть неудачу» отлично подходит для организации систем в виде комплексов параллельных процессов – она позволяет не волноваться о фатальных ошибках в отдельно взятых процессах, потому что их всегда можно перезапустить. Модель функционального программирования только усиливает преимущества стратегии распределенных вычислений, реализованной в языке Erlang.

Как и другие языки, представленные в этой книге, Erlang имеет свои недостатки. Только природа недостатков иная. Ниже перечислены места, где агент Смит оказывается не так крут.

Основные недостатки

Главные проблемы языка Erlang проистекают из его происхождения и занимаемой им ниши. Синтаксис чужд большинству программистов. Кроме того, парадигма функционального программирования имеет массу специфических особенностей, вызывающих проблемы при переходе к ее использованию. Наконец, лучшая на сегодняшний день реализация опирается на виртуальную машину BEAM, а не Java. Давайте остановимся на недостатках чуть подробнее.

Синтаксис

Отмечу в который уже раз, что восприятие синтаксиса – во многом вопрос личных предпочтений. Тем не менее даже если оставить вопросы личных предпочтений, нужно признать, что Erlang имеет определенные проблемы. Рассмотрим две из них.

Интересно отметить, что Erlang обязан языку Prolog не только своими сильными сторонами, но и недостатками. Большинство про-

граммистов воспринимает Prolog как весьма туманный язык, с неочевидным и чуждым синтаксисом. Даже с добавлением синтаксического сахара переход на использование этого языка мог бы потребовать значительных усилий.

Выше в этой главе я упоминал о проблемах, связанных с конструкциями `if` и `case`. Синтаксические правила выглядят логичными – символ-разделитель должен вставляться между инструкциями – но не практичными, потому что не позволяют переупорядочивать утверждения `case`, `if` или блоки `receive` без изменения пунктуации. Все эти ограничения выглядят излишними. Имеются и другие причуды, такие как условное представление массивов чисел в виде строк. Избавление от них могло бы существенно помочь продвижению языка Erlang.

Интеграция

Наследие Prolog оказалось обоюдоострым оружием. Недавно появившаяся реализация виртуальной машины Erjang, основанная на JVM, добилась определенных успехов, но ей пока далеко до более удачных альтернатив. Виртуальная машина JVM тянет за собой багаж своих собственных проблем, таких как модель процессов и потоков выполнения, несовместимых с Erlang. Но JVM как платформа имеет и свои преимущества, включая богатство библиотек и сотни тысяч серверов, готовых принять новые приложения на базе JVM.

Заключительные замечания

Успех языка программирования – штука непостоянная. Erlang стоит перед серьезными рыночными препятствиями, и ему будет нелегко завлечь программистов на Java парадигмой программирования в стиле языка Lisp и синтаксисом, напоминающим язык Prolog. Erlang действительно начинает наращивать свою популярность благодаря своей способности решать проблемы, которые с трудом решаются в других языках. В этой борьбе между Андерсоном и агентом Смитом я даю им равные шансы на успех.

Глава 7

Clojure

Не надо пытаться. Надо либо делать, либо не делать.

Мастер Йода

Clojure – это реализация Lisp для виртуальной машины Java (JVM). Обескураживающий и мощный, Lisp является одним из первых языков программирования и одним из последних. Десятки диалектов были созданы с целью сделать Lisp массовым языком, но ни один из них не пользовался сколько-нибудь значимым успехом. Синтаксис и модель программирования были слишком далеки от обычных разработчиков. И все же в Lisp есть нечто притягательное, что заставляет разработчиков продолжать создавать новые диалекты. Некоторые из лучших университетов мира преподают Lisp, чтобы заложить его основы в молодые умы, пока они еще открыты для всего нового.

Во многих отношениях Clojure напоминает мудрого мастера кунгфу, оракула на холме, загадочного учителя джедаев – мастера Йоду. Этот положительный персонаж появился в фильме «Star War s Episode V: The Empire Strikes Back»[1]. Он использует весьма специфический стиль общения, произнося фразы в обратном направлении, чем-то напоминающий префиксную форму записи в языке Lisp (позднее меня поймете вы). Он кажется слишком маленьким, чтобы оказывать влияние на окружающих, как синтаксические правила Lisp, описывающие чуть больше, чем использование круглых скобок и символов. Но очень быстро становится очевидно, что мастер Йода имеет куда большее значение, чем могло показаться на первый взгляд. Он стар, как и Lisp, обладает мудростью, отточенной временем и испытанной огнем. Подобно макросам и конструкциям высшего порядка в Lisp, он обладает внутренней силой, которая неподвластна другим. Во многих отношениях можно сказать, что все началось с Lisp. Но, прежде чем

---

[1] «Star Wars Episode V: The Empire Strikes Back», режиссер Джодж Лукас (George Lucas), 1980; Беверли Хиллз (Калифорния): 20th Century Fox, 2004. (http://ru.wikipedia.org/wiki/Звездные_войны. – *Прим. перев.*)

погрузиться слишком глубоко, давайте договоримся, что мы сначала немного коснемся языка Lisp, а затем перейдем к языку Clojure.

7.1. Введение в Clojure

Так или иначе, Clojure – еще один диалект Lisp. Он имеет те же ограничения и те же самые сильные стороны. Чтобы понимать Clojure, нужно понимать Lisp.

О Lisp

Язык Lisp является старейшим языком программирования (после Fortran), имевшим коммерческий успех. Это функциональный язык программирования, но он не является чисто функциональным. Название языка происходит от английского «LISt Processing» (обработка списков), и в самом начале вы узнаете, что это значит. Язык Lisp обладает некоторыми интересными особенностями.

- Lisp – язык списков. Вызов функции выглядит как список, в котором первый элемент ссылается на вызываемую функцию, а остальные представляют ее аргументы.
- Lisp использует собственные структуры данных для представления программного кода. Последователи Lisp называют эту стратегию *данные как код* (data as code).

В результате объединения этих двух идей получился язык, идеально подходящий для метапрограммирования. На этом языке можно организовать код в виде именованных методов внутри классов. Получившиеся объекты можно заключить в дерево и получить простую объектную модель. Точно так же можно написать код, основывающийся на использовании прототипов, со слотами для данных и методов. А можно написать исключительно функциональную реализацию. Такая высокая гибкость позволяет превращать Lisp в язык, поддерживающий любую парадигму программирования по вашему выбору.

В своей книге «Hackers and Painters» [Gra04] Пол Грэхем (Paul Graham) описывает историю, как небольшая группа разработчиков использовала Lisp и его мощную модель программирования, чтобы выиграть у более крупных компаний. Они верили, что Lisp дает значительные преимущества. Фактически больше времени они потратили на размещение объявлений о предоставлении услуг по разработке программных продуктов на Lisp и других высокоуровневых языках.

Основными диалектами Lisp являются Common Lisp и Scheme. Языки Scheme и Clojure относятся к одному семейству диалектов,

носящему название lisp-1, а Common Lisp относится к семейству диалектов lisp-2. Главное отличие между семействами диалектов заключается в особенностях организации пространств имен. В Common Lisp используются разные пространства имен для переменных и функций, тогда как в Scheme используется одно общее пространство имен. Итак, определив одну сторону уравнения, где находится Lisp, перейдем к другой стороне, где находится Java.

На стороне JVM

Каждый диалект Lisp создавался для удовлетворения нужд своей аудитории пользователей. Одной из важнейших характеристик Clojure является его опора на JVM. Знакомясь с языком Scala, мы видели, насколько важное значение имеет наличие коммерчески успешной платформы. Вам не придется вынуждать пользователей ваших приложений на Clojure развертывать отдельный сервер. Несмотря на относительную новизну Clojure, выбрав его в качестве инструмента, вы получаете доступ к десяткам тысяч библиотек на Java, способных удовлетворить практически любые ваши потребности.

На протяжении всей главы вы будете видеть наглядные подтверждения присутствия JVM: в способе запуска программ, в используемых библиотеках и создаваемых нами артефактах. Но вы также будете испытывать раскрепощение. Clojure – функциональный язык, поэтому вы сможете использовать в своем коде самые передовые приемы. Clojure – язык с динамической типизацией, поэтому ваш код будет получаться более кратким и читаемым, а его создание будет приносить больше удовольствия. А еще Clojure обладает выразительностью языка Lisp.

Clojure и Java определенно нуждаются друг в друге. Языку Lisp нужен рынок, который может предложить виртуальная машина Java, а сообществу Java нужно серьезное обновление.

Готовность к встрече с миром параллельных вычислений

Последним элементом уравнения для этого языка является множество библиотек. Clojure – функциональный язык, делающий основной упор на функции без побочных эффектов. Но, когда возникает необходимость использовать изменяемое состояние, этот язык может предложить множество вспомогательных концепций. Транзакционная память (transactional memory) действует подобно транзакцион-

ным базам данных, обеспечивая безопасность и целостность данных при параллельном доступе к ней. Агенты (agents) позволяют инкапсулировать доступ к изменяемым ресурсам. Мы рассмотрим некоторые из этих концепций в третий день.

Вам не терпится поскорее познакомиться с Clojure? Тогда начнем!

7.2. День 1: Обучение Люка

В фильме «Звездные войны» ученик Люк был принят мастером Йода для продолжения обучения на пути становления джедая, начатого под руководством другого учителя. Как и Люк, вы уже начали свое обучение функциональному программированию. Вы использовали замыкания в Ruby и познакомились с функциями высшего порядка в Scala и Erlang. В этой главе мы будем учиться применять некоторые из этих концепций в Clojure.

Посетите домашнюю страницу проекта Clojure: http://dev.clojure.org/display/doc/Getting+Started. Следуя инструкциям, установите Clojure в своей операционной системе и в используемой среде разработки. При работе над книгой я пользовался версией Clojure 1.2. Возможно, вам сначала придется установить платформу Java, впрочем, в большинстве современных операционных систем уже имеется встроенная поддержка Java. Для управления проектами на Clojure и настройками Java я использовал инструмент Leiningen[1]. Этот инструмент позволяет создавать проекты и избавляет от необходимости заботиться о таких настройках Java, как пути к классам. После установки всего необходимого можно приступать к созданию нового проекта:

```
batate$ lein new seven-languages
Created new project in: seven-languages
batate$ cd seven-languages/
seven-languages batate$
```

Затем можно запустить интерактивный сеанс (консоль) Clojure, выполнив команду `repl`:

```
seven-languages batate$ lein repl
Copying 2 files to /Users/batate/lein/seven-languages/lib
user=>
```

Вместе с инструментом Leiningen устанавливаются некоторые библиотеки Java и архивы Clojure Java. В вашем случае может потре-

---

[1] https://github.com/technomancy/leiningen.

боваться запускать интерактивный сеанс Clojure как-то иначе. Поэтому, начиная с этого момента, я буду лишь просить вас запустить интерактивный сеанс, не указывая конкретную команду.

Итак, у нас имеется простая консоль. Далее, когда я буду просить выполнить какой-либо код, вы можете использовать для этого консоль, встроенные средства своей интегрированной среды разработки или текстовый редактор с поддержкой Clojure.

Давайте попробуем ввести какой-нибудь код:

```
user=> (println "Give me some Clojure!")
Give me some Clojure!
nil
```

Итак, консоль работает. В языке Clojure принято заключать вызовы функций в круглые скобки. Первый элемент – имя вызываемой функции, а остальные – аргументы. Скобки можно вкладывать друг в друга. Давайте посмотрим, как это делается, на примере простых математических вычислений.

Вызовы простых функций

```
user=> (- 1)
-1
user=> (+ 1 1)
2
user=> (* 10 10)
100
```

Это простые арифметические выражения. Деление выглядит чуть интереснее:

```
user=> (/ 1 3)
1/3
user=> (/ 2 4)
1/2
user=> (/ 2.0 4)
0.5
user=> (class (/ 1 3))
clojure.lang.Ratio
```

В Clojure имеется базовый тип данных, называемый *Ratio*. Это отличное средство реализации отложенных вычислений, позволяющее избежать потери точности. Однако вы по-прежнему можете использовать вещественные числа, если пожелаете. Есть возможность получить остаток от деления:

```
user=> (mod 5 4)
1
```

Это – оператор деления по модулю. Представленная в этих примерах форма записи называется *префиксной*. Все языки программирования, с которыми мы познакомились к настоящему моменту, поддерживают *инфиксную* форму записи, когда оператор располагается между операндами, например: `4 + 1 - 2`. Многие из нас предпочитают инфиксную форму записи, потому что привыкли к ней. Мы привыкли видеть арифметические выражения, записанные таким способом. Однако со временем вы легко сможете читать выражения, записанные в префиксной форме. Первое время вы будете испытывать неудобство, но потом это ощущение пройдет. Префиксная форма записи с круглыми скобками имеет свои преимущества. Взгляните на следующее выражение:

```
user=> (/ (/ 12 2) (/ 6 2))
2
```

Здесь нет никакой неоднозначности. Clojure вычислит эту инструкцию в соответствии с расстановкой скобок. А теперь взгляните на следующее выражение:

```
user=> (+ 2 2 2 2)
8
```

Вы легко можете добавлять все новые элементы в выражения. Такой прием можно даже использовать с операциями вычитания и деления:

```
user=> (- 8 1 2)
5
user=> (/ 8 2 2)
2
```

В более привычной инфиксной форме записи эти выражения выглядят как: `(8 - 1) - 2` и `(8 / 2) / 2`. Или если вам больше по душе, когда к одному оператору относятся только два операнда: `(- (- 8 1) 2)` и `(/ (/ 8 2) 2)`. Кроме того, с помощью простых операторов можно получать очень интересные результаты:

```
user=> (< 1 2 3)
true
user=> (< 1 3 2 4)
false
```

Отлично. Как видите, единственный оператор способен обрабатывать произвольные списки операндов.

Кроме префиксной формы записи и дополнительной возможности передавать операторам списки параметров, синтаксис языка Clojure

очень прост. Давайте немного поупражняемся, исследовав строгость системы типов и поддержку приведения типов:

```
user=> (+ 3.0 5)
8.0
user=> (+ 3 5.0)
8.0
```

Clojure выполняет приведение типов. Вообще говоря, далее вы убедитесь, что Clojure поддерживает строгую динамическую типизацию. А теперь давайте сосредоточимся на некоторых простейших строительных блоках Clojure, называемых *формами*.

Форма – это элемент синтаксиса. Когда Clojure выполняет синтаксический анализ кода, он сначала разбивает программу на фрагменты, называемые *формами*. После этого выполняется интерпретация или компиляция кода. Я не буду различать код и данные, потому что в Lisp это одно и то же. Логические значения, символы, строки, множества, ассоциативные массивы и векторы – все это формы, которые будут встречаться нам на протяжении всей главы.

Строки и символы

Вы уже знакомы со строками, но мы можем взглянуть на них с другой стороны. Строки можно заключать в двойные кавычки и использовать те же правила экранирования, что и в языке Ruby:

```
user=> (println "master yoda\nluke skywalker\ndarth vader")
master yoda
luke skywalker
darth vader
nil
```

Пока никаких неожиданностей. Обратите внимание, что до сих пор мы вызывали `println` с единственным аргументом, однако эта функция может вызываться без аргументов, и тогда она выводит пустую строку, и/или более одного аргумента, и тогда она просто объединит все аргументы в одну строку перед выводом.

Clojure позволяет преобразовывать любые значения в строки с помощью функции `str`:

```
user=> (str 1)
"1"
```

Если передать ей класс Java, она вызовет его метод `toString`. Эта функция также может принимать более одного аргумента:

```
user=> (str "yoda, " "luke, " "darth")
"yoda, luke, darth"
```

Поэтому многие разработчики на Clojure используют `str` для конкатенации строк. Как дополнительное удобство с помощью `str` можно объединять элементы, не являющиеся строками:

```
user=> (str "one: " 1 " two: " 2)
"one: 1 two: 2"
```

Допускается даже объединять элементы разных типов. Для представления единственного символа без применения двойных кавычек используется символ \, как показано ниже:

```
user=> \a
\a
```

И, как обычно, их можно объединить в строку с помощью все той же `str`:

```
user=> (str \f \o \r \c \e)
"force"
```

Давайте выполним некоторые сравнения:

```
user=> (= "a" \a)
false
```

Итак, символ – это не строка с длиной, равной 1.

```
user=> (= (str \a) "a")
true
```

Однкако символы легко можно преобразовать в строки. Но закончим знакомство со строками и перейдем к логическим выражениям.

Логические значения и выражения

Clojure – это язык, поддерживающий строгую динамическую типизацию. Напомню, что под динамической типизацией понимается возможность определения типов во время выполнения. Выше мы уже познакомились с некоторыми типами данных, тем не менее давайте немного сузим фокус. Следующие выражения возвращают логический результат:

```
user=> (= 1 1.0)
true
user=> (= 1 2)
false
user=> (< 1 2)
true
```

Как и в большинстве других языков, представленных в этой книге, значение `true` является символом. Но оно также является кое-чем

иным. Типы в языке Clojure унифицированы с системой типов Java. Определить класс объекта можно с помощью функции `class`. В следующих примерах определяется класс логического значения:

```
user=> (class true)
java.lang.Boolean
user=> (class (= 1 1))
java.lang.Boolean
```

Как видите, здесь явно торчат уши JVM. Такой подход к организации системы типов несет массу удобств. Логические результаты можно использовать во многих выражениях. Ниже приводится простой пример использования инструкции `if`:

```
user=> (if true (println "True it is."))
True it is.
nil
user=> (if (> 1 2) (println "True it is."))
nil
```

Как и в языке Io, выполняемый код передается инструкции `if` во втором аргументе. Язык Lisp дает очень удобную возможность интерпретировать данные как выполняемый код. Мы можем сделать условную конструкцию более читаемой, разбив ее на несколько строк:

```
user=> (if (< 1 2)
  (println "False it is not."))
False it is not.
nil
```

Мы можем передать код для ветки `else` в третьем аргументе:

```
user=> (if false (println "true") (println "false"))
false
nil
```

А теперь посмотрим, какие другие значения можно использовать в качестве логических. Во-первых, как получить `nil` в Clojure?

```
user=> (first ())
nil
```

Ага. Все просто. Символ называется `nil`.

```
user=> (if 0 (println "true"))
true
nil
user=> (if nil (println "true"))
nil
user=> (if "" (println "true"))
true
nil
```

0 and "" – это истинные значения, а nil – нет. Я буду знакомить вас с новыми логическими значениями по мере необходимости. А сейчас перейдем к знакомству с более сложными структурами данных.

Списки, ассоциативные массивы, множества и векторы

В любых функциональных языках основными структурами данных являются списки и кортежи. В Clojure к основным относятся три структуры: списки, ассоциативные массивы и векторы. В первую очередь рассмотрим тип коллекций, наиболее знакомый нам, – списки.

Списки

Список – это упорядоченная коллекция элементов. Элементами может быть все, что угодно, но в соответствии с идиомами языка Clojure списки используются для программного кода, а для хранения данных используются векторы. Впрочем, чтобы избежать путаницы, я покажу вам, как использовать списки для хранения данных. Поскольку списки интерпретируются как функции, их нельзя определять таким способом:

```
user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer
    cannot be cast to clojure.lang.IFn (NO_SOURCE_FILE:0)
```

Если вам действительно необходим список с элементами 1, 2 и 3, используйте один из следующих приемов:

```
user=> (list 1 2 3)
(1 2 3)
user=> '(1 2 3)
(1 2 3)
```

Создав список, им можно манипулировать, как обычно. Вторая форма определения списков называется *цитированием*, или *подавлением вычислений* (quoting). Списки поддерживают четыре основные операции: first (возвращает «голову» списка), rest (возвращает «хвост» списка), last (возвращает последний элемент) и cons (конструирует новый список из указанных «головы» и «хвоста»):

```
user=> (first '(:r2d2 :c3po))
:r2d2
user=> (last '(:r2d2 :c3po))
:c3po
user=> (rest '(:r2d2 :c3po))
(:c3po)
```

```
user=> (cons :battle-droid '(:r2d2 :c3po))
(:battle-droid :r2d2 :c3po)
```

Разумеется, эти операции можно комбинировать с функциями высшего порядка, но давайте отложим изучение данной темы, пока не познакомимся с последовательностями. А теперь рассмотрим близкого родственника списка – вектор.

Векторы

Как и списки, векторы являются упорядоченными коллекциями элементов. Векторы оптимизированы для произвольного доступа к их элементам. Определяются векторы с помощью квадратных скобок:

```
user=> [:hutt :wookie :ewok]
[:hutt :wookie :ewok]
```

Используйте списки для программного кода, а данные храните в векторах. Обращаться к элементам вектора можно следующим образом:

```
user=> (first [:hutt :wookie :ewok])
:hutt
user=> (nth [:hutt :wookie :ewok] 2)
:ewok
user=> (nth [:hutt :wookie :ewok] 0)
:hutt
user=> (last [:hutt :wookie :ewok])
:ewok
user=> ([:hutt :wookie :ewok] 2)
:ewok
```

Обратите внимание, что векторы сами являются функциями, принимающими индекс в качестве аргумента. Ниже показано, как можно объединить два вектора:

```
user=> (concat [:darth-vader] [:darth-maul])
(:darth-vader :darth-maul)
```

Возможно, вы заметили, что интерактивная оболочка Clojure вывела вектор как список. Многие функции, возвращающие коллекции, используют абстракцию Clojure, которая называется *последовательностью*. Мы поближе познакомимся с этой абстракцией во второй день. А пока просто знайте, что многие функции в Clojure возвращают последовательности, которые интерактивная оболочка выводит в виде списков.

Clojure позволяет получить «голову» и «хвост» вектора:

```
user=> (first [:hutt :wookie :ewok])
:hutt
user=> (rest [:hutt :wookie :ewok])
(:wookie :ewok)
```

Обе эти операции мы будем использовать при выполнении операций сопоставления с образцом. И списки, и векторы хранят свои элементы в определенном порядке. Однако в Clojure поддерживаются и неупорядоченные коллекции – множества и ассоциативные массивы.

Множества

Множество – это неупорядоченная коллекция элементов. Безусловно, коллекции этого типа тоже хранят элементы в некотором порядке, но этот порядок зависит от особенностей реализации, поэтому на него нельзя полагаться. Множества определяются с помощью конструкции `#{}`, как показано ниже:

```
user=> #{:x-wing :y-wing :tie-fighter}
#{:x-wing :y-wing :tie-fighter}
```

Мы можем присвоить это множество переменной `spacecraft` и выполнять операции над ней:

```
user=> (def spacecraft #{:x-wing :y-wing :tie-fighter})
#'user/spacecraft
user=> spacecraft
#{:x-wing :y-wing :tie-fighter}
user=> (count spacecraft)
3
user=> (sort spacecraft)
(:tie-fighter :x-wing :y-wing)
```

Имеется также возможность создавать сортированные множества, хранящие свои элементы и возвращающие их в определенном порядке:

```
user=> (sorted-set 2 3 1)
#{1 2 3}
```

Множества можно объединять:

```
user=> (clojure.set/union #{:skywalker} #{:vader})
#{:skywalker :vader}
```

Вычислять их разность:

```
(clojure.set/difference #{1 2 3} #{2})
```

Прежде чем продолжить движение вперед, хочу познакомить вас с еще одной удобной особенностью – множества тоже являются функциями. Множество `#{:jar-jar, :chewbacca}` одновременно является и коллекцией данных, и функцией. Проверить присутствие элемента в множестве можно следующим образом:

```
user=> (#{:jar-jar :chewbacca} :chewbacca)
:chewbacca
user=> (#{:jar-jar :chewbacca} :luke)
nil
```

Когда множество используется в качестве функции, оно возвращает первый аргумент, если этот аргумент является членом множества. Давайте на этом закончим знакомство с множествами и перейдем к ассоциативным массивам.

Ассоциативные массивы

Как вы уже знаете, ассоциативный массив – это коллекция пар ключ/значение. В языке Clojure ассоциативные массивы определяются с помощью фигурных скобок:

```
user=> {:chewie :wookie :lea :human}
{:chewie :wookie, :lea :human}
```

Это – пример объявления ассоциативного массива, но в таком виде они трудно читаются. Кроме того, легко ошибиться, указав нечетное число ключей и значений:

```
user=> {:jabba :hut :han}
java.lang.ArrayIndexOutOfBoundsException:3
```

В Clojure данная проблема решается заменой пробелов запятыми:

```
user=> {:darth-vader "obi wan", :luke "yoda"}
{:darth-vader "obi wan", :luke "yoda"}
```

Слово, следующее за двоеточием (`:`), – это ключевое слово, которое действует подобно символу в Ruby или атому в Prolog или Erlang. В языке Clojure имеются два типа форм, которые могут использоваться в качестве ключей, *ключевые слова* (keywords) и *символы* (symbols). Символы ссылаются на некоторые другие значения, а ключевые слова – на самих себя. Например, `true` и `map` – это символы. Используйте ключевые слова для именования сущностей предметной области, таких как свойства в ассоциативных массивах, на манер атомов в Erlang.

Давайте определим ассоциативный массив с именем `mentors`:

```
user=> (def mentors {:darth-vader "obi wan", :luke "yoda"})
#'user/mentors
user=> mentors
{:darth-vader "obi wan", :luke "yoda"}
```

Теперь из этого ассоциативного массива можно извлекать значения, передавая ключи в виде первого аргумента:

```
user=> (mentors :luke)
"yoda"
```

Ассоциативные массивы являются функциями. И ключевые слова тоже являются функциями:

```
user=> (:luke mentors)
"yoda"
```

Функция `:luke` ищет саму себя в ассоциативном массиве. Такой прием выглядит немного необычно, но иногда он может оказаться полезным. В качестве ключей и значений можно использовать данные любых типов.

Объединить два ассоциативных массива можно с помощью функции `merge`:

```
user=> (merge {:y-wing 2, :x-wing 4} {:tie-fighter 2})
{:tie-fighter 2, :y-wing 2, :x-wing 4}
```

Для объединения ассоциативных массивов, содержащих одинаковые ключи, можно также использовать оператор:

```
user=> (merge-with + {:y-wing 2, :x-wing 4} {:tie-fighter 2 :x-wing 3})
{:tie-fighter 2, :y-wing 2, :x-wing 7}
```

В данном случае оператор `+` сложил значения `4` и `3` ключей `x-wing`. На основе существующего ассоциативного массива можно создать новый ассоциативный массив:

```
user=>(assoc {:one 1} :two 2)
{:two 2, :one 1}
```

Можно создать сортированный ассоциативный массив, хранящий элементы в определенном порядке:

```
user=> (sorted-map 1 :one, 3 :three, 2 :two)
{1 :one, 2 :two, 3 :three}
```

Итак, мы познакомились с несколькими структурами данных и теперь можем переходить к знакомству с еще одной формой, позволяющей определять функции.

Определение функций

Функции занимают центральное место во всех диалектах Lisp. Определение новой функции производится с помощью формы `defn`.

```
user=> (defn force-it [] (str "Use the force," "Luke."))
#'user/force-it
```

В простейшем случае определение функции имеет вид `(defn [params] body)`. В примере выше мы определили функцию с именем `force-it` без параметров. Функция просто объединяет две строки в одну. Вызываются пользовательские функции точно так же, как и любые другие:

```
user=> (force-it)
"Use the force,Luke."
```

При желании в функцию можно добавить строку с описанием:

```
user=> (defn force-it
           "The first function a young Jedi needs"
           []
           (str "Use the force," "Luke"))
```

Это описание можно будет получить с помощью функции `doc`:

```
user=> (doc force-it)
-------------------------
user/force-it
([])
  The first function a young Jedi needs
nil
```

Давайте добавим параметр в функцию:

```
user=> (defn force-it
           "The first function a young Jedi needs"
           [jedi]
           (str "Use the force," jedi))
#'user/force-it
user=> (force-it "Luke")
"Use the force,Luke"
```

Функцию `doc` можно использовать для получения описания любой функции:

```
user=> (doc str)
-------------------------
clojure.core/str
([] [x] [x & ys])
  При вызове без аргументов вернет пустую строку. При вызове с одним
  аргументом x вернет x.toString(). (str nil) вернет пустую строку.
```

```
  При вызове с двумя и более аргументами вернет объединение строковых
  значений аргументов.
nil
```

Теперь, научившись определять простые функции, можно переходить к спискам параметров.

Связывание

Как и в большинстве других языков, рассматривавшихся выше, процесс назначения параметров на основе переданных аргументов называется *связыванием* (binding). Одна из замечательных особенностей Clojure заключается в возможности обратиться к любой части любого аргумента как к параметру. Например, допустим, что у нас имеется линия, представленная вектором точек, как показано ниже:

```
user=> (def line [[0 0] [10 20]])
#'user/line
user=> line
[[0 0] [10 20]]
```

Мы могли бы создать функцию для доступа к концу линии, как показано ниже:

```
user=> (defn line-end [ln] (last ln))
#'user/line-end
user=> (line-end line)
[10 20]
```

Но в действительности нам не нужна вся линия. Было бы намного лучше связать наш параметр со вторым элементом линии. В Clojure сделать это очень просто:

```
(defn line-end [[_ second]] second)
#'user/line-end
user=> (line-end line)
[10 20]
```

Данный прием называется *деструктуризация* (destructuring). Мы учли здесь структуру исходных данных и извлекли только нужный нам фрагмент. Давайте рассмотрим процедуру связывания поближе. В определении параметров функции присутствует конструкция `[[_ second]]`. Внешние квадратные скобки определяют вектор параметров. Внутренние квадратные скобки сообщают, что связываться будут только отдельные элементы списка или вектора. Символ подчеркивания (_) и `second` – это отдельные параметры, но в Clojure принято использовать символ подчеркивания для игнорируемых параметров. Выражаясь простым языком, это определение сообщает: «Параметра-

ми данной функции являются: _, который связывается с первым элементом первого аргумента, и second, который связывается со вторым элементом первого аргумента».

Привязки могут быть вложенными. Представьте, что у нас имеется определение игрового поля для игры в «крестики-нолики» и нам нужно получить значение клетки в центре. Представим игровое поле как матрицу с тремя строками и тремя столбцами:

```
user=> (def board [[:x :o :x] [:o :x :o] [:o :x :o]])
#'user/board
```

По условиям задачи нужно выбрать второй элемент второй строки:

```
user=> (defn center [[_ [_ c _] _]] c)
#'user/center
```

Отлично! Мы, по сути, использовали ту же самую концепцию. Давайте разберем это определение подробнее. Привязку осуществляет конструкция [[_ [_ c _] _]]. Мы связали один параметр с исходным аргументом [_ [_ c _] _]. Это определение гласит, что мы игнорируем первый и третий элементы (верхний и нижний ряд) игрового поля. А из второго элемента (среднего ряда) извлекли средний элемент [_ c _]. Справедливо было бы ожидать, что параметр будет связан с центральной клеткой игрового поля:

```
user=> (center board)
:x
```

Эту функцию можно немного упростить. Во-первых, совсем необязательно перечислять игнорируемые элементы, следующие за требуемым:

```
(defn center [[_ [_ c]]] c)
```

Во-вторых, деструктуризация может выполняться не только с применением списка аргументов, но и с помощью инструкции let. В любом диалекте Lisp связывание переменной со значением может выполняться с помощью let. Мы могли бы использовать let, чтобы спрятать прием деструктуризации от клиентов функции center:

```
(defn center [board]
    (let [[_ [_ c]] board] c))
```

let принимает два аргумента. Первый аргумент – вектор с символом для связывания ([[_ [_ c]]]), и второй аргумент – собственно значение (board). Далее должно следовать выражение, использующее это значение (здесь мы просто возвращаем c). Обе формы дают одинако-

вые результаты. Все зависит лишь от вашего желания – где должна произойти деструктуризация. Я покажу пару коротких примеров использования `let`, но имейте в виду, что те же самые приемы с успехом можно использовать внутри списка аргументов.

Вот как выполняется деструктуризация ассоциативного массива:

```
user=> (def person {:name "Jabba" :profession "Gangster"})
#'user/person
user=> (let [{name :name} person] (str "The person's name is " name))
"The person's name is Jabba"
```

А вот как можно объединить ассоциативные массивы и векторы:

```
user=> (def villains [{:name "Godzilla" :size "big"} {:name "Ebola"
:size "small"}])
#'user/villains
user=> (let [[_ {name :name}] villains] (str "Name of the second
villain: " name))
"Name of the second villain: Ebola"
```

Мы связали параметр с вектором, пропустили первый ассоциативный массив, а из второго выбрали значение ключа `:name`. Здесь можно наблюдать влияние языка Lisp на Prolog и, соответственно, на Erlang. Деструктуризация – это всего лишь разновидность сопоставления с образцом.

Анонимные функции

Функции в языке Lisp – это всего лишь данные. Функции высшего порядка в Lisp являются неотъемлемой его частью, потому код в нем интерпретируется как самые обычные данные. Поддержка анонимных функций позволяет создавать неименованные функции. Это фундаментальная особенность всех языков, рассматриваемых в данной книге. В Clojure функции высшего порядка определяются с помощью функции `fn`. Так как имя функции не указывается, синтаксис определения имеет вид: `(fn [parameters*] body)`. Рассмотрим простой пример.

Давайте задействуем функцию высшего порядка для создания списка с длинами слов, содержащихся в другом списке. Допустим, что имеется список имен людей:

```
user=> (def people ["Lea", "Han Solo"])
#'user/people
```

Определить длину одного слова можно следующим образом:

```
user=> (count "Lea")
3
```

А вот как можно создать список длин имен:

```
user=> (map count people)
(3 8)
```

Вы уже встречались с этими понятиями выше. В данном контексте `count` – это функция высшего порядка. В Clojure данная концепция чрезвычайно проста, благодаря тому что функция – это простой список, как любой другой элемент списка. Те же строительные блоки можно использовать для вычисления удвоенных длин слов:

```
user=> (defn twice-count [w] (* 2 (count w)))
#'user/twice-count
user=> (twice-count "Lando")
10
user=> (map twice-count people)
(6 16)
```

Благодаря простоте этой функции ее можно записать как анонимную функцию:

```
user=> (map (fn [w] (* 2 (count w))) people)
(6 16)
```

Можно также использовать более краткую форму:

```
user=> (map #(* 2 (count %)) people)
(6 16)
```

В этой краткой форме символ # определяет анонимную функцию, где `%` связывает каждый элемент последовательности. Символ # называют *макросом чтения* (reader macro).

Поддержка анонимных функций дает удобную возможность создавать безымянные функции *здесь и сейчас*. Вы уже видели их в действии в других языках.

Далее рассматривается несколько функций, выполняющих операции с коллекциями и использующих функции высшего порядка. Для демонстрации всех этих функций используется один общий вектор с именем `v`:

```
user=> (def v [3 1 2])
#'user/v
```

К нему будут применяться несколько анонимных функций, определяемых в примерах ниже.

apply

Функция `apply` применяет указанную ей функцию к списку аргументов. Вызов `(apply f '(x y))` действует подобно `(f x y)`:

```
user=> (apply + v)
6
user=> (apply max v)
3
```

filter

Функция `filter` напоминает функцию `find_all` в Ruby. Она принимает функцию, которая проверяет элементы последовательности и возвращает только те, что прошли проверку. Например, ниже показано, как извлечь все нечетные элементы или элементы меньше 3:

```
user=> (filter odd? v)
(3 1)
user=> (filter #(< % 3) v)
(1 2)
```

Мы еще вернемся к обсуждению анонимных функций, когда более детально будем рассматривать последовательности. А пока прервемся и посмотрим, что говорит о языке Clojure его создатель Рич Хикки (Rich Hickey).

Интервью с Ричем Хикки, создателем Clojure

Рич Хикки (Rich Hickey) ответил на несколько вопросов специально для читателей этой книги. Он особенно уверен в успехе данной версии Lisp в сравнении с остальными диалектами, поэтому интервью получилось немного более длинным, чем обычно. Я надеюсь, его ответы будут вам так же интересны, как и мне.

Брюс Тейт: Что побудило вас написать Clojure?

Рич Хикки: Я – простой практик, желавший получить преимущественно функциональный, расширяемый, динамичный язык программирования, с надежным механизмом поддержки параллельных вычислений, основанный на платформах, считающихся промышленным стандартом – JVM и CLR, – но я не нашел такой язык.

Брюс Тейт: Что больше всего вам нравится в этом языке?

Рич Хикки: Мне нравятся особый упор на абстракции в структурах данных, богатство библиотек и простота. Возможно, это разные вещи, но они тесно связаны между собой.

Брюс Тейт: Что бы вы изменили в языке, будь у вас возможность вернуться назад?

Рич Хикки: Я исследовал разные подходы к представлению чисел. Упакованные числовые типы – определенно не самое лучшее реше-

ние, реализованное в JVM. Это – область, над которой я активно продолжаю работать.

Брюс Тейт: Какое самое интересное применение Clojure вы видели?

Рич Хикки: Я думаю, приложение Flightcaster[1] (служба, предсказывающая задержки рейсов самолетов в реальном масштабе времени) максимально использовало многие возможности Clojure – от синтаксических абстракций макросов для определения предметно-ориентированного языка машинного обучения и производства статистических выкладок до взаимодействий с инфраструктурами Java, такими как Hadoop и Cascading.

Брюс Тейт: Почему вы так уверены в успехе Clojure, несмотря на неудачи других диалектов Lisp?

Рич Хикки: Это очень важный вопрос! Я не уверен, что такая характеристика, как «неудача», применима к основным диалектам Lisp (Scheme и Common Lisp). Scheme появился в результате попытки создать очень компактный язык для реализации базовых вычислений, тогда как Common Lisp явился следствием попытки стандартизировать различные диалекты Lisp, используемые в исследованиях. Да, они не стали практичными инструментами, которые могли бы использоваться в промышленном программировании, но надо понимать, что они и не создавались для этого.

Язык Clojure, напротив, создавался как многоцелевой, практичный инструмент разработки промышленных приложений и, соответственно, обладает дополнительными преимуществами, которых не имели прежние диалекты Lisp. Мы сотрудничаем с различными коллективами, мы используем опыт, накопленный другими языками, и мы решаем некоторые проблемы, традиционно свойственные языку Lisp.

Брюс Тейт: Насколько хорошо подходит язык Clojure для использования в коллективах разработчиков?

Рич Хикки: Существует мнение, что Lisp – это язык для разработчиков-одиночек, но мы понимаем, что разработка программных продуктов – это коллективный труд. Например, Clojure не поддерживает определяемые пользователем макросы чтения, которые могли бы читать код, написанный на маленьких, несовместимых микродиалектах.

Брюс Тейт: Почему вы предпочли использовать существующие виртуальные машины?

---

[1] http://www.infoq.com/articles/flightcaster-clojure-rails.

Рич Хикки: Наличие огромного количества библиотек, написанных на других языках, является немаловажным фактором в современной жизни. Возможность вызывать программный код на других языках и вызываться из этого кода является существенным преимуществом платформ JVM и CLR[1].

Идея стандартизации многоязычных платформ, независимых от операционной системы, только начала оформляться, когда более старые диалекты Lisp уже существовали. Бурный рост промышленности вызвал появление фактических стандартов. Как преимущество стала расцениваться поддержка повторного использования базовых технологий, таких как сложнейшие сборщики мусора и динамические компиляторы, подобные HotSpot. Поэтому Clojure родился как язык для платформы, а не как язык-платформа.

Брюс Тейт: Согласен. А что было сделано для увеличения дружелюбия Lisp?

Рич Хикки: Многое. Например, мы сумели избавиться от «проблемы» круглых скобок. Программисты на Lisp знают и ценят возможность интерпретации кода как данных, но было бы неправильно просто проигнорировать мнение тех, кого круглые скобки отталкивают. Я не думаю, что переход от `foo(bar, baz)` к `(foo bar baz)` вызовет у разработчиков какие-либо сложности. Я в свое время очень внимательно изучал особенности применения скобок в других диалектах Lisp, чтобы понять, можно ли как-то улучшить ситуацию. Как оказалось – можно! В старых диалектах Lisp круглые скобки использовались повсюду. Их было слишком много. В Clojure был предпринят иной подход – в нем были убраны группирующие скобки, что усложнило труд создателей макросов, но упростило труд прикладных программистов.

Благодаря уменьшению количества круглых скобок программный код на Clojure стал намного более читаемым, чем код на старых диалектах Lisp. Начальные двойные круглые скобки чаще можно встретить в программном коде на Java (например, жуткое выражение `((AType)athing).amethod()`), чем в коде на Clojure.

Что мы узнали в первый день

Clojure – это функциональный язык для виртуальной машины JVM. Этот диалект Lisp хотя и считается функциональным языком, но, так же как Scala и Erlang, не является исключительно функциональным.

---

[1] CLR – Common Language Runtime (общеязыковая среда выполнения) – виртуальная машина для платформы Microsoft .NET.

Он снижает вероятность появления побочных эффектов. В отличие от других диалектов Lisp, Clojure привнес не так много изменений в синтаксис. В числе основных можно назвать использование фигурных скобок для обозначения ассоциативных массивов и квадратных скобок для обозначения векторов. В некоторых местах допускается использовать запятые вместо пробелов и опускать ненужные круглые скобки.

Мы научились пользоваться простейшими формами Clojure. В их число входят логические значения, символы, числа, ключевые слова и строки. Мы также разобрались с некоторыми коллекциями. Например, мы выяснили, что списки и векторы являются упорядоченными коллекциями, при этом векторы оптимизированы для произвольного доступа к элементам, а списки оптимизированы для последовательного доступа. Мы также рассмотрели множества – неупорядоченные коллекции – и ассоциативные массивы, являющиеся коллекциями пар ключ/значение.

Мы определили несколько именованных функций, указав для каждой имя, список параметров и тело, с необязательной строкой описания. Затем мы использовали прием деструктуризации, чтобы связать параметр с произвольным элементом входного аргумента. Эта особенность напоминает сопоставление с образцом в Prolog и Erlang. Наконец, мы определили несколько анонимных функций и использовали их в комплексе с функциями обхода списков и ассоциативных массивов.

Во второй день мы рассмотрим прием рекурсии в языке Clojure, основной строительный блок в большинстве функциональных языков. Мы также познакомимся с последовательностями и отложенными вычислениями – краеугольными моделями в Clojure, наслаивающими мощные, обобщенные абстракции поверх коллекций.

А теперь сделаем перерыв, чтобы применить на практике полученные знания.

День 1: задания для самостоятельного решения

Clojure – относительно новый язык, но вокруг него уже сплотилось удивительно активное и постоянно растущее сообщество, одно из лучших, которые мне встречались при работе над этой книгой.

Найдите:

- примеры использования последовательностей Clojure;
- формальное определение функции в языке Clojure;
- сценарий для быстрого запуска интерактивной оболочки в вашей операционной системе.

Практические задания:

- напишите функцию `(big st n)`, которая возвращала бы `true` для строки `st`, содержащей больше `n` символов;
- напишите функцию `(collection-type col)`, возвращающую `:list`, `:map` или `:vector` в зависимости от типа коллекции `col`.

7.3. День 2: Йода и Сила

Как мастер-джедай, Йода учил своих подопечных быть преданными Силе, которая течет через все живое, и владеть ею. В этом разделе мы будем учиться владеть фундаментальными концепциями Clojure. Мы поговорим о последовательностях, об абстракциях, объединяющих все коллекции Clojure и связывающих их с коллекциями Java. Мы также познакомимся с отложенными вычислениями, стратегией вычисления элементов коллекций только в момент обращения к ним. А потом мы исследуем самую мистическую особенность языка, которую можно считать Силой всех диалектов Lisp, – макросы.

Рекурсивные вычисления с помощью loop и recur

Как вы уже знаете по опыту изучения других языков в этой книге, функциональные языки в большей степени опираются на рекурсию, чем на итерации. Ниже приводится пример рекурсивной программы, определяющей размер вектора:

```
(defn size [v]
    (if (empty? v)
            0
            (inc (size (rest v)))))

(size [1 2 3])
```

Здесь нет ничего сложного. Размер пустого списка равен нулю; а размер непустого списка на единицу больше размера его «хвоста». Мы уже видели подобные решения, когда знакомились с другими языками.

Вы также знаете, что при выполнении рекурсивных вызовов расходуется место на стеке, поэтому рекурсивные реализации могут продолжать потреблять память вплоть до полного ее исчерпания. Функциональные языки преодолевают это ограничение за счет оптимизации хвостовой рекурсии. Язык Clojure не поддерживает неявную оптимизацию хвостовой рекурсии из-за ограничений JVM. Эту опти-

мизацию следует выполнять явно, используя функции `loop` и `recur`. Представьте цикл в форме инструкции `let`.

```
(loop [x x-initial-value, y y-initial-value] (do-something-with x y))
```

Для данного вектора `loop` свяжет переменные в четных позициях со значениями в нечетных позициях[1]. Фактически, когда рекурсия отсутствует, функция `loop` действует в точности как инструкция `let`:

```
user=> (loop [x 1] x)
1
```

Функция `recur` повторно вызовет `loop`, но на этот раз передаст новые значения.

Давайте перепишем функцию `size`, добавив в нее вызов `recur`:

```
(defn size [v]
        (loop [l v, c 0]
        (if (empty? l)
                c
                (recur (rest l) (inc c)))))
```

Во второй версии `size` использована оптимизированная хвостовая рекурсия с применением `loop` и `recur`. Так как фактически нам не требуется возвращать значение, мы будем накапливать результат в *переменной-аккумуляторе*. В данном случае счетчик хранится в переменной `c`.

Данная версия оптимизирует хвостовую рекурсию, но для этого пришлось написать больше строк малопонятного кода. К сожалению, JVM – это обоюдоострое оружие. Если вы хотите получить сообщество, вам придется решать проблемы. Так как подобные функции присутствуют в API для работы с коллекциями, вам нечасто придется использовать `recur`. К тому же в арсенале Clojure имеются превосходные альтернативы рекурсии, включая «ленивые» (lazy) последовательности, которые рассматриваются ниже в этой главе.

На этом дурные вести второго дня закончились, и мы можем перейти к более приятным новостям. При исследовании последовательностей мы познакомимся с некоторыми особенностями языка Clojure, делающими его уникальным.

Последовательности

Последовательность – это абстракция, независимая от реализации, объемлющая все контейнерные типы, имеющиеся в экосистеме Clo-

---

[1] Здесь нулевая позиция считается четной, даже при том, что 0 не является ни четным, ни нечетным числом. – *Прим. перев.*

jure. Последовательностями являются коллекции (множества, ассоциативные массивы, векторы и др.), строки и даже структуры файловой системы (потоки и каталоги). Кроме того, последовательности служат обобщенной абстракцией для контейнеров Java, включая коллекции Java, массивы и строки. Вообще говоря, все, что поддерживает функции `first`, `rest` и `cons`, может использоваться как последовательность.

Выше, в примерах работы с векторами, интерактивная оболочка Clojure иногда выводила результат в виде списка:

```
user=> [1 2 3]
[1 2 3]
user=> (rest [1 2 3])
(2 3)
```

Обратите внимание, что мы начали с вектора. Результат его определения не является списком. Но в ответ на вызов функции `rest` интерактивная оболочка вернула последовательность. Это означает возможность работы с любыми коллекциями обобщенным способом. Давайте исследуем библиотеку функций для работы с последовательностями. Она слишком богата и разнообразна, чтобы ее можно было охватить целиком в единственном разделе главы, тем не менее я попробую дать вам представление о том, что в ней имеется. Я расскажу о функциях, которые изменяют, проверяют и создают последовательности, но я не буду углубляться в подробное их описание.

Проверки

Для проверки коллекций вы всегда будете использовать функции, которые называются *предикатами*. Они принимают последовательность и функцию, реализующую проверку, и возвращают логическое значение. Функция `every?` возвращает `true`, если функция проверки вернула `true` для каждого элемента последовательности:

```
user=> (every? number? [1 2 3 :four])
false
```

Как видите, один из элементов этой последовательности не является числом. Функция `some` возвращает `true`, если функция проверки вернула `true` хотя бы для одного элемента последовательности[1]:

```
(some nil? [1 2 nil])
true
```

---

[1] Точнее, `some` возвращает первое значение, отличное от `nil` и `false`. Например, `(some first [[ ] [1]])` вернет 1.

Один из элементов является значением nil. Функции not-every? и not-any? являются противоположностями функций, представленных выше:

```
user=> (not-every? odd? [1 3 5])
false
user=> (not-any? number? [:one :two :three])
true
```

Они действуют в полном соответствии с нашими ожиданиями. Давайте перейдем к функциям, изменяющим последовательности.

Изменение последовательностей

Библиотека функций для работы с последовательностями включает несколько инструментов, изменяющих последовательности разными способами. Вы уже знакомы с функцией filter. Выбрать только слова, длина которых больше четырех символов, можно следующим образом:

```
user=> (def words ["luke" "chewie" "han" "lando"])
#'user/words
user=> (filter (fn [word] (> (count word) 4)) words)
("chewie" "lando")
```

Вы также уже знакомы с функцией map, которая применяет переданную ей функцию ко всем элементам коллекции и возвращает результаты в виде новой последовательности. Ниже показано, как можно сконструировать последовательность квадратов чисел, хранящихся в исходном векторе:

```
user=> (map (fn [x] (* x x)) [1 1 2 3 5])
(1 1 4 9 25)
```

При изучении языков Scala и Erlang мы познакомились с генераторами списков, объединяющих функции отображения и фильтрации. Напомню, что генератор списков объединяет несколько списков и фильтров, получая допустимые комбинации из списков с помощью фильтров. Прежде всего рассмотрим самый простой случай. У нас имеется пара списков, в которых перечислены цвета и игрушки:

```
user=> (def colors ["red" "blue"])
#'user/colors
user=> (def toys ["block" "car"])
#'user/toys
```

С помощью генератора списков мы можем применить некоторую функцию ко всем цветам, как это делает функция map:

```
user=> (for [x colors] (str "I like " x))
("I like red" "I like blue")
```

Конструкция `[x colors]` связывает `x` с элементом из списка `colors`. Конструкция `(str "I like " x)` представляет здесь произвольную функцию, которая будет применяться ко всем `x` из `colors`. Гораздо больший интерес представляют ситуации, когда выполняется связывание с элементами сразу из нескольких списков:

```
user=> (for [x colors, y toys] (str "I like " x " " y "s"))
("I like red blocks" "I like red cars"
 "I like blue blocks" "I like blue cars")
```

Генератор списков создает все возможные комбинации из элементов двух списков. Внутри выражения можно также использовать ключевое слово `:when` для фильтрации значений:

```
user=> (defn small-word? [w] (< (count w) 4))
#'user/small-word?
user=> (for [x colors, y toys, :when (small-word? y)]
           (str "I like " x " " y "s"))
("I like red cars" "I like blue cars")
```

Здесь мы определили функцию-фильтр `small-word?`. Она считает коротким любое слово, содержащее меньше четырех символов. С помощью ключевого слова `:when` мы применили фильтр `small-word?` к значению `y` (`:when (small-word? y)`). И в результате получили все возможные комбинации `(x, y)`, где `x` – это элемент коллекции `colors`, `y` – элемент коллекции `toys` и `y` имеет размер меньше четырех символов. Код получился компактным, но достаточно выразительным. Это – идеальное сочетание. Поехали дальше.

Исследуя языки Erlang, Scala и Ruby, мы познакомились с функциями `foldl`, `foldleft` и `inject`. В языке Lisp существует эквивалентная функция `reduce`. С ее помощью, например, можно быстро вычислить факториал числа:

```
user=> (reduce + [1 2 3 4])
10
user=> (reduce * [1 2 3 4 5])
120
```

Существуют также функция для сортировки списков:

```
user=> (sort [3 1 2 4])
(1 2 3 4)
```

и функция для сортировки результатов, возвращаемых другими функциями:

```
user=> (defn abs [x] (if (< x 0) (- x) x))
#'user/abs
user=> (sort-by abs [-1 -4 3 2])
(-1 2 3 -4)
```

Здесь мы определили функцию abs, возвращающую абсолютное значение и используемую функцией, выполняющей сортировку. Это были самые основные функции преобразования последовательностей в Clojure. Далее мы перейдем к функциям, создающим последовательности, но для этого нам потребуется немного полениться.

Отложенные вычисления

Многие бесконечные последовательности довольно просто описываются на языке математики. В функциональных языках часто бывает желательно иметь аналогичную возможность и получать произвольные элементы подобных бесконечных последовательностей, не воссоздавая их полностью. Решить эту задачу можно с помощью отложенных вычислений (lazy evaluation). Используя эту стратегию, библиотека поддержки последовательностей в языке Clojure вычисляет элементы последовательностей только при непосредственном обращении к ним. Фактически большинство последовательностей вычисляется таким образом, отчего их называют «ленивыми». Давайте для начала создадим несколько конечных последовательностей, а затем перейдем к исследованию «ленивых» последовательностей.

Создание конечных последовательностей с функцией range

В отличие от Ruby, Clojure поддерживает диапазоны в виде функций. Функция range создает последовательность:

```
user=> (range 1 10)
(1 2 3 4 5 6 7 8 9)
```

Обратите внимание, что верхняя граница не включается – число 10 не вошло в созданную здесь последовательность. При необходимости можно указать шаг наращивания:

```
user=> (range 1 10 3)
(1 4 7)
```

Если шаг наращивания не указывается, можно опустить и нижнюю границу:

```
user=> (range 10)
(0 1 2 3 4 5 6 7 8 9)
```

По умолчанию за значение нижней границы принимается число ноль. Последовательности, создаваемые с помощью `range`, являются конечными. А как быть, если требуется определить последовательность, не имеющую верхней границы, чтобы в результате получилась бесконечная последовательность? Давайте посмотрим, как.

Бесконечные последовательности и take

Начнем с самой простой бесконечной последовательности – последовательности из единственного повторяющегося элемента. Такую последовательность можно определить как `(repeat 1)`. Если попытаться обратиться к этой конструкции в интерактивной оболочке, она будет выводить единицы, пока вы не прервете этот процесс. Очевидно, что должен быть некоторый способ получить конечное подмножество элементов. И такой способ есть – функция `take`:

```
user=> (take 3 (repeat "Use the Force, Luke"))
("Use the Force, Luke" "Use the Force, Luke" "Use the Force, Luke")
```

Здесь мы создали бесконечную последовательность из повторяющихся строк «Use the Force, Luke» (используй Силу, Люк) и извлекли из нее три первых элемента. С помощью функции `cycle` можно организовать повторение элементов списка:

```
user=> (take 5 (cycle [:lather :rinse :repeat]))
(:lather :rinse :repeat :lather :rinse)
```

Здесь мы извлекли пять первых элементов из последовательности повторяющихся элементов вектора `[:lather :rinse :repeat]`. Имеется также возможность отбросить несколько первых элементов последовательности:

```
user=> (take 5 (drop 2 (cycle [:lather :rinse :repeat])))
(:repeat :lather :rinse :repeat :lather)
```

Следите за операциями изнутри наружу: с помощью `cycle` мы создали бесконечную последовательность, вызовом `drop` отбросили два первых элемента и затем с помощью `take` извлекли пять первых элементов. Однако совсем необязательно составлять подобные выражения, которые требуется читать изнутри наружу. Для применения каждой функции к результату можно воспользоваться новым оператором `->>`:

```
user=> (->> [:lather :rinse :repeat] (cycle) (drop 2) (take 5))
(:repeat :lather :rinse :repeat :lather)
```

Итак, на основе исходного вектора создается последовательность вызовом `cycle`, затем вызовом `drop 2` отбрасываются два первых элемента, и, наконец, вызовом `take 5` извлекаются пять первых элементов. Иногда код, выполняющийся слева направо, легче читается. А как быть, если потребуется добавить разделитель между словами? Для этого можно воспользоваться функцией `interpose`:

```
user=> (take 5 (interpose :and (cycle [:lather :rinse :repeat])))
(:lather :and :rinse :and :repeat)
```

Здесь вызов функции `interpose` вставляет ключевое слово `:and` между всеми элементами бесконечной последовательности. Эту функцию можно рассматривать как обобщенную версию функции `join` в языке Ruby. А что, если потребуется составить последовательность из перемежающихся членов других последовательностей? Для этого можно задействовать функцию `interleave`:

```
user=> (take 20 (interleave (cycle (range 2)) (cycle (range 3))))
(0 0 1 1 0 2 1 0 0 1 1 2 0 0 1 1 0 2 1 0)
```

Здесь мы потребовали чередовать элементы двух бесконечных последовательностей, `(cycle (range 2))` и `(cycle (range 3))`. А затем извлекли 20 первых элементов. Если выделить числа в четных и нечетных позициях, получатся две последовательности: `(0 1 0 1 0 1 0 1 0 1)` и `(0 1 2 0 1 2 0 1 2 0)` соответственно. Отлично.

Еще один способ создания последовательностей предоставляет функция `iterate`. Взгляните на следующие примеры:

```
user=> (take 5 (iterate inc 1))
(1 2 3 4 5)
user=> (take 5 (iterate dec 0))
(0 -1 -2 -3 -4)
```

`iterate` принимает функцию и начальное значение. Затем она последовательно применяет указанную функцию к предыдущему найденному значению, начиная с начального. В этих двух примерах были использованы функции `inc` и `dec`.

Следующий пример находит два соседних числа в последовательности Фибоначчи. Напомню, что каждое число в этой последовательности является суммой двух предыдущих. Для исходной пары `[a b]` функция `fib-pair` определяет следующую пару как `[b, a + b]`. Определение следующей пары на основе текущей можно реализовать в виде анонимной функции, как показано ниже:

```
user=> (defn fib-pair [[a b]] [b (+ a b)])
#'user/fib-pair
```

```
user=> (fib-pair [3 5])
[5 8]
```

Следующий пример использует функцию `iterate` для построения бесконечной последовательности. Не запускайте пока этот пример:

```
(iterate fib-pair [1 1])
```

Извлечь первые элементы из всех пар можно с помощью `map`:

```
(map
first
(iterate fib-pair [1 1]))
```

В результате получается бесконечная последовательность. Теперь можно извлечь из нее пять первых элементов:

```
user=> (take 5
       (map
       first
       (iterate fib-pair [1 1])))
(1 1 2 3 5)
```

или извлечь элемент с индексом 500:

```
(nth (map first (iterate fib-pair [1 1])) 500)
(225... more numbers ...626)
```

Производительность данного решения действительно выше всяких похвал. С помощью «ленивых» последовательностей часто можно описать решение рекурсивных задач, таких как поиск чисел Фибоначчи. Как еще один пример, ниже приводится функция вычисления факториала:

```
user=> (defn factorial [n] (apply * (take n (iterate inc 1))))
#'user/factorial
user=> (factorial 5)
120
```

Здесь мы получаем `n` элементов из бесконечной последовательности `(iterate inc 1)`, затем находим их произведение с помощью `apply *`. Решение получилось тривиально простым. Теперь, когда мы познакомились с «ленивыми» последовательностями, можно заняться исследованием новых функций Clojure – `defrecord` и `defprotocol`.

defrecord и defprotocol

Выше уже упоминалась возможность интеграции с Java, но пока мы не имели возможности убедиться, насколько тесно Clojure интегрируется с JVM. В конечном счете JVM определяет все типы и интерфейсы.

(Для тех, кто не знаком с Java, замечу, что типы в Java в большинстве своем являются классами, а интерфейсы – это классы, не имеющие реализации.) Чтобы обеспечить бесшовную интеграцию с JVM, значительная часть языка Clojure реализована на Java.

С ростом эффективности Clojure как языка JVM в его реализации все больше стало появляться кода, написанного на самом языке Clojure. Чтобы обеспечить такой рост, разработчикам Clojure необходим был некоторый способ конструировать быстрые, открытые расширения, обеспечивающие возможность выражать свои намерения с применением абстракций. В результате появились макрос `defrecord` для определения типов и форма `defprotocol`, позволяющая объединять функции с типами. С точки зрения языка Clojure, типы и протоколы (такие как интерфейсы) являются лучшими чертами объектно-ориентированного стиля программирования, а наследование реализации – худшими. Конструкции `defrecord` и `defprotocol` оставляют в Clojure лучшие черты ООП и отбрасывают худшие.

На момент написания данной книги эти две особенности языка играли особенно важную роль, и они продолжают развиваться. Далее я буду опираться на опыт Стюарта Хэллоуэя (Stuart Halloway), сооснователя компании Relevance и автора книги «Programming Clojure» [Hal09], который поможет нам в практической реализации. Мы также еще раз вернемся к другому функциональному языку на основе JVM – Scala – и перепишем программу Compass на языке Clojure. Итак, приступим.

Прежде всего определим протокол. Протокол в языке Clojure – это своего рода контракт. Типы, поддерживающие протокол, должны включать определенный набор функций, полей и аргументов. Ниже приводится определение протокола, описывающего работу компаса:

clojure/compass.clj
http://media.pragprog.com/titles/btlang/code/clojure/compass.clj

```
(defprotocol Compass
  (direction [c])
  (left [c])
  (right [c]))
```

Этот протокол определяет абстракцию с именем `Compass` и перечисляет функции, которые должна поддерживать эта абстракция, – `direction`, `left` и `right`. Теперь можно приступать к реализации протокола с помощью `defrecord`. Для начала определим четыре направления:

```
(def directions [:north :east :south :west])
```

Далее нам нужна будет функция, обрабатывающая поворот. Напомню, что направление определяется целыми числами `0`, `1`, `2` и `3`, которые в новой реализации представлены ключевыми словами: `:north`, `:east`, `:south` и `:west` соответственно. Прибавление `1` к направлению соответствует повороту компаса на 90 градусов вправо. Чтобы обеспечить переход от `:west` к `:north`, мы будем брать остаток от деления `base/4` (точнее, `base/число-направлений`):

```
(defn turn
  [base amount]
  (rem (+ base amount) (count directions)))
```

Функция `turn` действует в соответствии с нашими ожиданиями. Я загрузил файл с реализацией компаса и опробовал функцию:

```
user=> (turn 1 1)
2
user=> (turn 3 1)
0
user=> (turn 2 3)
1
```

Иначе говоря, поворот вправо, при следовании в направлении `:east`, дает нам направление `:south`, поворот вправо, при следовании в направлении `:west`, дает нам направление `:north`, а три поворота вправо, при следовании в направлении `:south`, дают нам направление `:east`.

Пришло время реализовать протокол. Эту реализацию мы включим в определение `defrecord`. Будем добавлять новые реализации постепенно. Сначала укажем, что `defrecord` реализует требуемый протокол:

```
(defrecord SimpleCompass [bearing]
  Compass
```

Здесь определяется простая запись с именем `SimpleCompass`. Она имеет единственное поле `bearing`. Теперь приступим к реализации протокола `Compass`, начав с функции `direction`:

```
(direction [_] (directions bearing))
```

Функция `direction` возвращает элемент вектора `directions` с индексом `bearing`. Например, `(directions 3)` вернет `:west`. Список аргументов каждой функции включает ссылку на экземпляр (аналог `self` в Ruby или `this` в Java), но мы не используем его, поэтому просто добавили в список аргументов символ подчеркивания (`_`). Перейдем к функциям `left` и `right`:

```
(left [_] (SimpleCompass. (turn bearing 3)))
(right [_] (SimpleCompass. (turn bearing 1)))
```

Напомню, что в Clojure используются неизменяемые значения. Это означает, что в результате поворота будет возвращаться новый, измененный экземпляр компаса, а прежний экземпляр останется неизменным. Обе функции, `left` и `right`, используют синтаксис, который мы еще не видели. Конструкция `(SimpleCompass. arg)` – это вызов конструктора записи `SimpleCompass`, связывающий аргумент `arg` с первым параметром. Чтобы убедиться в этом, попробуйте выполнить `(String. "new string")` в интерактивной оболочке. В ответ она должна вернуть новую строку `"new string"`.

Функции `left` и `right` получились очень простыми. Каждая возвращает новый экземпляр компаса с соответствующим направлением движения `bearing`. Функция `right` выполняет поворот вправо один раз, а функция `left` выполняет поворот вправо три раза. К настоящему моменту у нас имеется тип `SimpleCompass`, реализующий протокол `Compass`. Нам осталось написать функцию, возвращающую строковое представление объекта, но метод `toString` уже определен в классе `java.lang.Object`. Нам не составит труда добавить его в наш тип.

```
Object
(toString [this] (str "[" (direction this) "]" )))
```

Здесь мы реализовали часть протокола `Object`, определив метод `toString`, возвращающий строку вида: `SimpleCompass [:north]`.

Теперь реализацию типа можно считать законченной. Создадим новый компас:

```
user=> (def c (SimpleCompass. 0))
#'user/c
```

Функции поворота возвращают новые экземпляры компаса:

```
user=> (left c) ; вернет новый компас
#:SimpleCompass{:bearing 3}
```

```
user=> c ; первоначальный компас не изменился
#:SimpleCompass{:bearing 0}
```

Обратите внимание, что первоначальный компас не изменился. Так как фактически мы определили новый тип данных в JVM, обращаться к любым его полям можно как к полям Java. Кроме того, к полям типа можно обращаться как к элементам ассоциативного массива Clojure:

```
user=> (:bearing c)
0
```

Благодаря тому что типы действуют подобно ассоциативным массивам, мы легко можем разрабатывать прототипы новых типов в виде ассоциативных массивов и по окончании преобразовывать их в типы. Кроме того, в тестах типы можно замещать аналогичными им ассоциативными массивами. Эта особенность дает еще некоторые преимущества:

- типы прекрасно уживаются с конструкциями параллельных вычислений в языке Clojure. В третий день мы узнаем, как создавать изменяемые ссылки на объекты Clojure, поддерживающие транзакционную целостность на манер реляционных баз данных;
- в этом примере мы реализовали протокол, но это не единственный способ создания новых типов. Так как в конечном итоге создаются типы JVM, они способны взаимодействовать с классами и интерфейсами Java.

В виде `defrecord` и `defprotocol` Clojure предлагает возможность писать «родной» для JVM код без применения Java. Такой код без всяких ограничений может взаимодействовать с другими типами JVM, включая классы и интерфейсы Java. Типы на языке Clojure могут наследовать типы или реализовывать интерфейсы Java. Классы Java также могут наследовать типы Clojure. Разумеется, весь спектр взаимодействий с Java намного шире, но эта его часть является наиболее важной. Теперь, когда вы научились расширять возможности языка Java, давайте посмотрим, как можно расширять сам язык Clojure с помощью макросов.

Макросы

В этом разделе мы будем ссылаться на главу с описанием языка Io. В разделе 3.3, в подразделе «Сообщения», мы реализовали оператор `unless`. Он имеет форму: `(unless test form1)`. То есть конструкция `unless` должна вызвать `form1`, если `test` имеет ложное значение. В Clojure нельзя реализовать эту конструкцию в виде обычной функции, потому что каждый ее параметр будет выполняться автоматически в момент вызова:

```
user=> ; Неправильная реализация unless
user=> (defn unless [test body] (if (not test) body))
#'user/unless
user=> (unless true (println "Danger, danger Will Robinson"))
Danger, danger Will Robinson
nil
```

Мы обсуждали эту проблему во время знакомства с языком Io. В большинстве языков при вызове функции сначала выполняются параметры, а затем результаты их выполнения помещаются на стек вызовов. В данном случае нам требуется, чтобы блок кода не выполнялся, если условие имеет ложное значение. В Io эта проблема решается за счет задержки выполнения сообщения `unless`. В Lisp можно использовать макросы. Когда мы вводим код `(unless test body)`, нам нужно, чтобы Lisp преобразовал его в `(if (not test) body)`. В этом нам помогут макросы.

Программы на языке Clojure выполняются в два этапа. На первом этапе выполняется трансляция всех макросов. Увидеть, что происходит на этом этапе, можно с помощью команды `macroexpand`. Мы уже использовали выше пару макросов, которые называются *макросами чтения* (reader macros). Точка с запятой (`;`) – это комментарий, знак одиночной кавычки (`'`) – сама кавычка, а знак решетки (`#`) – анонимная функция. Чтобы предотвратить преждевременное выполнение выражения, которое требуется развернуть, поместите кавычку в начало:

```
user=> (macroexpand ''something-we-do-not-want-interpreted)
(quote something-we-do-not-want-interpreted)
```

```
user=> (macroexpand '#(count %))
(fn* [p1__97] (count p1__97))
```

Это – макросы. В общем случае на этапе развертывания макросов программный код можно интерпретировать как списки. Если нежелательно, чтобы функция была выполнена немедленно, добавьте перед ней кавычку. Clojure оставит аргументы в неприкосновенности. Наша реализация `unless` выглядит, как показано ниже:

```
user=> (defmacro unless [test body]
         (list 'if (list 'not test) body))
#'user/unless
```

Обратите внимание, что Clojure подставил `test` и `body`, не вычисляя (не выполняя) их, но нам пришлось добавить кавычки перед `if` и `not`. Нам также пришлось упаковать программный код в списки. Таким образом, мы сконструировали список кода в том виде, в каком он должен быть выполнен. Теперь можно применить `macroexpand` к нашей реализации:

```
user=> (macroexpand '(unless condition body))
(if (not condition) body)
```

и попробовать выполнить ее:

```
user=> (unless true (println "No more danger, Will."))
nil
user=> (unless false (println "Now, THIS is The FORCE."))
Now, THIS is The FORCE.
nil
```

Сейчас мы с вами изменили базовое определение языка! Мы добавили собственную управляющую структуру, не вынуждая проектировщиков языка определять ее. Макросы являются, пожалуй, самой мощной особенностью Lisp. Лишь немногие языки способны на это. Секрет заключается в возможности представлять данные как код, а не как строки. То есть код уже является высокоуровневой структурой данных.

Закончим второй день на этом. Сегодня мы узнали много нового, и следует сделать паузу, чтобы закрепить приобретенные знания.

Что мы узнали во второй день

Закончился еще один насыщенный день. Вы добавили в свой арсенал огромное множество новых абстракций. Давайте вспомним еще раз, о чем сегодня рассказывалось.

Сначала мы исследовали прием рекурсии. Так как JVM не поддерживает оптимизацию хвостовой рекурсии, нам пришлось явно сделать это с помощью `loop` и `recur`. Эти конструкции циклов позволяют реализовать самые разные алгоритмы, которые обычно основываются на рекурсивных вызовах функций, однако синтаксис в этом случае выглядит немного пугающим.

Мы также опробовали последовательности. Clojure инкапсулирует в эту абстракцию операции доступа ко всем коллекциям. С помощью стандартной библиотеки мы сумели применить обобщенные стратегии обработки коллекций. Мы опробовали разные функции, изменяющие коллекции и выполняющие поиск элементов в них. Применение функций высшего порядка еще больше расширило возможности библиотеки.

Поддержка «ленивых» коллекций позволила нам еще шире раздвинуть рамки возможного. Применение «ленивых» коллекций может упрощать алгоритмы и откладывать вычисления, что, в свою очередь, может приводить к увеличению производительности и ослаблению связей.

Затем мы потратили некоторое время на реализацию типов. С помощью `defrecord` и `defprotocol` мы реализовали типы, прекрасно уживающиеся с виртуальной машиной JVM.

Наконец, мы познакомились с макросами и с их помощью добавили в язык новую конструкцию. Мы узнали, что программы на языке Clojure выполняются в два этапа, на первом из которых производится *развертывание макросов*, а на втором – интерпретация и выполнение программного кода. Мы реализовали управляющую структуру `unless`, выполнив подстановку вызова функции `if` на этапе развертывания макроса.

Не торопитесь, потратьте немного времени, чтобы опробовать на практике новые умения.

День 2: задания для самостоятельного решения

Сегодняшний день был наполнен исследованиями некоторых сложных и мощных особенностей языка Clojure. Потратьте немного времени на эксперименты с ними, чтобы лучше понять, как они действуют.

Найдите:

- реализацию каких-нибудь часто используемых макросов на языке Clojure;
- пример определения «ленивой» последовательности;
- описание текущего состояния `defrecord` и `defprotocol` (на момент написания этих строк они все еще находились в разработке).

Практические задания:

- добавьте в управляющую конструкцию `unless` условие `else` с использованием макроса;
- определите с помощью `defrecord` свой тип, реализующий некоторый протокол.

7.4. День 3: Глаз дьявола

В фильме «Звездные войны» магистр Йода первым распознал зло в Дарте Вейдере. Работая над языком Clojure, Рич Хикки (Rich Hickey) смог определить основные проблемы, усложняющие разработку многозадачных объектно-ориентированных систем. Мы много раз повторяли, что изменяемое состояние – зло, скрытое в недрах объектно-ориентированных программ. Мы видели несколько разных подходов к решению проблем с изменяемым состоянием. В Io и Scala используется модель на основе акторов и предоставляются неизменяемые конструкции, позволяющие программистам решать задачи без использования изменяемого состояния. В Erlang имеются акторы с легковесными процессами и виртуальная машина, эффективно

поддерживающая мониторинг и взаимодействия, обеспечивающие непревзойденную надежность. В Clojure используется иной подход к реализации параллельных вычислений. Он использует *программную транзакционную память* (Software Transactional Memory, STM). В этом разделе мы познакомимся с особенностями работы STM и несколькими дополнительными инструментами, позволяющими использовать общие данные в многопоточных приложениях.

Ссылки и транзакционная память

Базы данных используют механизм транзакций для поддержки целостности и непротиворечивости данных. Современные базы данных используют как минимум два механизма управления параллельными вычислениями. Блокировки предотвращают одновременный доступ к одной и той же записи из двух конкурирующих транзакций. Механизм контроля совпадений использует множество версий одних и тех же данных, чтобы позволить каждой транзакции получить собственную копию. Если какая-либо транзакция вступит в конфликт с другой транзакцией, механизм базы данных просто перезапустит эту транзакцию.

В некоторых языках программирования, таких как Java, для защиты ресурсов от одновременного доступа из разных потоков выполнения используются блокировки. Механизмы блокировок, как правило, перекладывают основное бремя управления ими на плечи программиста. Приступив к их использованию, быстро понимаешь, насколько тяжкое это бремя.

В других языках, таких как Clojure, используется *программная транзакционная память* (Software Transactional Memory, STM). Этот механизм основан на использовании нескольких версий данных для поддержания их целостности и непротиворечивости. В отличие от Scala, Ruby или Io, когда в программе на Clojure требуется изменить состояние ссылки, это необходимо делать в рамках транзакции. Давайте посмотрим, как все это работает.

Ссылки

Ссылки в языке Clojure – это всего лишь контейнеры, хранящие значения. Доступ к ссылкам должен осуществляться с соблюдением определенных правил. В данном случае таким правилом является поддержание STM – запрещается изменять ссылки за рамками транзакции. Чтобы увидеть, как действует весь этот механизм, создадим ссылку:

```
user=> (ref "Attack of the Clones")
#<Ref@ffdadcd: "Attack of the Clones">
```

Пока ничего интересного. Мы можем присвоить ссылку, например:

```
user=> (def movie (ref "Star Wars"))
#'user/movie
```

и получить ссылку обратно:

```
user=> movie
#<Ref@579d75ee: "Star Wars">
```

Но чтобы получить значение, хранящееся по ссылке, следует использовать `deref`:

```
user=> (deref movie)
"Star Wars"
```

Можно также использовать сокращенную форму `deref`:

```
user=> @movie
"Star Wars"
```

Да, так лучше. Теперь мы легко можем получить значение, хранящееся по ссылке. Но мы пока не пытались изменить состояние ссылки. Давайте попробуем сделать это. В Clojure для этого нужно передать функцию, которая изменит значение. В первом аргументе этой функции будет передана разыменованная ссылка:

```
user=> (alter movie str ": The Empire Strikes Back")
java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)
```

Как видите, состояние по ссылке можно изменить только в рамках транзакции – в вызове функции `dosync`. Предпочтительный способ изменения значения по ссылке – передать некоторую функцию, осуществляющую необходимые операции:

```
user=> (dosync (alter movie str ": The Empire Strikes Back"))
"Star Wars: The Empire Strikes Back"
```

Установить начальное значение можно с помощью `ref-set`:

```
user=> (dosync (ref-set movie "Star Wars: The Revenge of the Sith"))
"Star Wars: The Revenge of the Sith"
```

Посмотрим, изменилось ли значение по ссылке:

```
user=> @movie
"Star Wars: The Revenge of the Sith"
```

Именно этого мы и ожидали. Значение действительно изменилось. Кому-то может показаться слишком утомительным изменять значе-

ния переменных таким способом, однако такой политикой Clojure избавляет нас от еще большей головной боли в будущем. Мы точно знаем, что программы, действующие подобным образом, будут выполняться абсолютно корректно, без опаски попасть в состояние гонки за ресурсами или взаимоблокировки. Большая часть нашего кода будет использовать парадигмы функционального программирования, а STM мы припасем для задач, которые удобнее решать с применением изменяемого состояния.

Атомы

Если требуется обеспечить безопасность для единственной ссылки, нескоординированной с другими действиями, можно использовать атомы. Эти элементы данных можно изменять вне контекста транзакций. Как и ссылки, атомы в языке Clojure инкапсулируют доступ к изменяемым ресурсам. Давайте опробуем их. Сначала создадим атом:

```
user=> (atom "Split at your own risk.")
#<Atom@53f64158: "Split at your own risk.">
```

Теперь свяжем атом с переменной `danger`:

```
user=> (def danger (atom "Split at your own risk."))
#'user/danger
user=> danger
#<Atom@3a56860b: "Split at your own risk.">
user=> @danger
"Split at your own risk."
```

Связать переменную `danger` с другим атомом можно с помощью `reset!`:

```
user=> (reset! danger "Split with impunity")
"Split with impunity"
user=> danger
#<Atom@455fc40c: "Split with impunity">
user=> @danger
"Split with impunity"
```

Функция `reset!` замещает атом целиком, но предпочтительнее изменять значение атома с помощью дополнительной функции. Если требуется изменить большой вектор, изменить атом на месте можно с помощью функции `swap!`, как показано ниже:

```
user=> (def top-sellers (atom []))
#'user/top-sellers
user=> (swap! top-sellers conj {:title "Seven Languages", :author "Tate"})
```

```
[{:title "Seven Languages in Seven Weeks", :author "Tate"}]
user=> (swap! top-sellers conj {:title "Programming Clojure" :author
"Halloway"})
[{:title "Seven Languages in Seven Weeks", :author "Tate"}
 {:title "Programming Clojure", :author "Halloway"}]
```

Как и при работе со ссылками, вы можете создать значение один раз и затем изменять его с помощью `swap!`. А теперь рассмотрим практический пример.

Кэш в атоме

Теперь вы знакомы со ссылками и атомами. С более обобщенной философией их использования мы познакомимся, когда будем исследовать язык Haskell. Суть состоит в том, чтобы заключить некоторый набор данных в пакет, который позднее можно будет изменять с помощью функций. Операции со ссылками должны выполняться под управлением транзакций, а операции с атомами – нет. Давайте сконструируем простой кэш в атоме. Это отличная задача, как раз для атомов. Кэш будет представлять собой ассоциативный массив, связывающий имена со значениями. Данный пример любезно предоставлен Стюартом Хллоуэеем (Stuart Halloway), сооснователем компании Relevance[1], оказывающей услуги по обучению языку Clojure.

Итак, нам нужно создать кэш и определить функции, реализующие добавление новых элементов в кэш и удаление элементов из кэша. Сначала опишем создание кэша:

clojure/atomcache.clj
http://media.pragprog.com/titles/btlang/code/clojure/atomcache.clj

```
(defn create
  []
  (atom {}))
```

Данная функция просто создает атом. Мы позволим клиенту этого класса связывать его. Далее реализуем возможность получать элемент кэша по ключу:

```
(defn get
  [cache key]
  (@cache key))
```

---

[1] http://www.thinkrelevance.com.

Функция принимает кэш и ключ. Кэш – это атом, поэтому его нужно разыменовать и вернуть элемент, ассоциированный с указанным ключом. Наконец, реализуем добавление элемента в кэш:

```
(defn put
  ([cache value-map]
    (swap! cache merge value-map))
  ([cache key value]
    (swap! cache assoc key value)))
```

Мы определили функцию `put`, имеющую два тела. Первое с помощью `merge` добавляет в кэш все элементы ассоциативного массива `value-map`, переданного в виде аргумента. Второе с помощью `assoc` добавляет указанные ключ и значение. Ниже демонстрируется порядок использования нашего кэша. В этом примере мы сначала добавляем в кэш новое значение, а затем возвращаем его:

```
(def ac (create))
(put ac :quote "I'm your father, Luke." )
(println (str "Cached item: " (get ac :quote)))
```

В результате этот фрагмент выведет:

```
Cached item: I'm your father, Luke.
```

Атомы и ссылки дают простой и надежный способ выполнения синхронных операций с изменяемым состоянием. В следующих нескольких разделах мы рассмотрим пару примеров асинхронных операций.

Агенты

Агенты, подобно атомам, являются обертками вокруг некоторых данных. Как и отложенные задания (futures) в Io, агент блокирует выполнение до момента, когда данные не станут доступны. Применяя агенты, пользователи могут изменять данные асинхронно с помощью функций, причем функции будут вызываться из других потоков выполнения. Изменить состояние агента в каждый конкретный момент времени сможет только одна функция.

Опробуем их в деле. Определим функцию с именем `twice`, удваивающую любое значение, переданное ей:

```
user=> (defn twice [x] (* 2 x))
#'user/twice
```

Теперь определим агента с именем `tribbles` с начальным значением 1:

```
user=> (def tribbles (agent 1))
#'user/tribbles
```

Изменить значение агента `tribbles` можно, послав ему новое значение:

```
user=> (send tribbles twice)
#<Agent@554d7745: 1>
```

Эта функция будет вызвана в другом потоке выполнения. Прочитаем значение агента:

```
user=> @tribbles
2
```

Операция чтения значения из ссылки, агента или атома никогда не блокируется. Чтение должно выполняться быстро. Посмотрим, как изменяются значения, извлекаемые из каждого агента:

```
user=> (defn slow-twice [x]
         (do
             (Thread/sleep 5000)
             (* 2 x)))
#'user/slow-twice
user=> @tribbles
2
user=> (send tribbles slow-twice)
#<Agent@554d7745: 16>
user=> @tribbles
2
user=> ; выждать пять секунд
user=> @tribbles
4
```

Не старайтесь вникнуть в тонкости синтаксиса – конструкция `(Thread/sleep 5000)` просто вызывает Java-метод `sleep` класса `Thread`, – сосредоточьтесь на значении агента.

Мы определили версию `twice`, действующую с задержкой в пять секунд. Этого времени вполне достаточно, чтобы в интерактивной оболочке увидеть, как изменяется результат `@tribbles` с течением времени.

Итак, мы извлекли значение `tribbles`. Но мы не можем быть уверены, что извлеченное значение получено в результате изменений, *инициированных в текущем потоке выполнения*. Если такая уверенность необходима, можно вызвать `(await tribbles)` или `(awaitfor timeout`

tribbles), где timeout – предельное время ожидания в миллисекундах. Имейте в виду, что функции await и await-for блокируют вызывающий поток, пока агент не выполнит все задания, но применение этих функций не гарантирует, что после возврата из них вы не прочтете значение, полученное в результате изменений, инициированных в других потоках. Если вы строите логику выполнения исходя из того, что получили последнее значение, – вы уже допустили грубую ошибку. Описываемые здесь инструменты Clojure работают с мгновенными снимками данных, которые могут устаревать немедленно. Собственно, так и действует механизм контроля совпадений в базах данных.

Отложенные задания

В Java имеется возможность запускать параллельные потоки выполнения для решения каких-либо задач. Конечно, вы можете использовать интеграцию с Java и запускать потоки таким способом, но есть лучшее решение. Допустим, вам требуется запустить поток для выполнения сложных вычислений. С этой целью можно было бы использовать агента. Или, скажем, вам нужно запустить вычисления, но вы не желаете ждать их окончания. Тогда помощь в этом вам могут оказать отложенные задания (futures). Давайте посмотрим, как.

Прежде всего создадим отложенное задание. Ссылка на него возвращается немедленно:

```
user=> (def finer-things (future (Thread/sleep 5000) "take time"))
#'user/finer-things
user=> @finer-things
"take time"
```

В зависимости от того, как быстро вы печатаете на клавиатуре, вам *может* потребоваться подождать, пока результат будет вычислен. Конструкция future принимает одно или более выражений и возвращает результат последнего из них. Сами вычисления выполняются в другом потоке. Если попытаться разыменовать ссылку на отложенное задание, операция чтения будет заблокирована, пока результат не станет доступен.

Итак, отложенное задание – это конструкция для выполнения параллельных вычислений, возвращающая ссылку асинхронно, до того, как результат будет вычислен. Отложенные задания можно использовать для выполнения нескольких продолжительных вычислений в параллельных потоках.

Что мы пропустили

Clojure – это диалект языка Lisp, самого по себе богатого всякими возможностями. Он основан на виртуальной машине JVM, разработка которой ведется уже больше десяти лет. Clojure также реализует ряд новых, мощных концепций. Охватить все возможности этого языка просто невозможно в одной главе. Поэтому я лишь вкратце упомяну некоторые его особенности, о существовании которых следует знать.

Метаданные

Иногда бывает желательно связать тип с некоторыми метаданными. Clojure позволяет определять такие метаданные для символов и коллекций и извлекать их программно. Конструкция `(with-meta value metadata)` вернет новое значение `value` и связанные с ним метаданные `metadata`, обычно в виде ассоциативного массива.

Интеграция с Java

Clojure прекрасно интегрируется с Java. Мы лишь слегка затронули тему интеграции в этой главе, когда создали свой тип на основе JVM. Мы вообще не использовали библиотеки Java. Мы также не касались вопросов форм совместимости с Java. Например, конструкция `(.toUpperCase "Fred")` вызовет функцию-член `.toUpperCase` строки `"Fred"`.

Мультиметоды

Объектно-ориентированные языки поддерживают лишь один стиль объединения данных и операций над ними. Clojure позволяет выстраивать свою организацию кода с применением мультиметодов. С их помощью можно связать библиотеку функций с типом, обеспечить поддержку полиморфизма на основе типов, метаданных, аргументов и даже атрибутов. Это очень мощная и гибкая концепция. Она позволяет, например, организовать наследование в стиле языка Java, наследование на основе прототипов или вообще что-то совершенно другое.

Данные потоков

Clojure поддерживает различные модели параллельных вычислений, предлагая атомы, ссылки и агенты. Но иногда возникает потреб-

ность сохранять данные отдельно для каждого потока выполнения. В Clojure этой цели служат ссылки на переменные (vars). Например, конструкция `(binding [name "value"] ...)` свяжет имя `name` со значением `"value"` *только в текущем потоке выполнения*.

Что мы узнали в третий день

Сегодня мы прошлись по инструментам организации параллельных вычислений. Мы встретились с несколькими интересными конструкциями.

Ссылки позволили нам организовать параллельный доступ к изменяемому состоянию из нескольких потоков выполнения. Мы использовали программную транзакционную память (STM). От нас потребовалось лишь обеспечить выполнение всех изменений в рамках транзакций, то есть с применением функции `dosync`.

Затем мы опробовали атомы — легковесные конструкции с меньшей степенью защиты, но более простые в использовании. Атомы могут изменяться за пределами транзакций.

В заключение мы попробовали использовать агента для реализации пула заданий, которого можно было бы задействовать для выполнения продолжительных вычислений. Агенты отличаются от акторов в языке Io, позволяя изменять значение с помощью произвольных функций. Кроме того, агенты возвращают мгновенный снимок данных, которые могут измениться в любой момент.

День 3: задания для самостоятельного решения

Во второй день мы сосредоточились на освоении дополнительных абстракций. В третий день мы познакомились с инструментами параллельных вычислений. В упражнениях, перечисленных ниже, вам будет предложено объединить знания, полученные в эти два дня.

Найдите:

- реализацию очереди, блокирующую выполнение потока, если она пуста, до появления в ней нового элемента.

Практические задания:

- на основе ссылок создайте вектор банковских счетов. Напишите функции `debit` и `credit`, изменяющие сумму на счете.

В следующем задании я хочу предложить вам решить одну интересную задачу, получившую название «Проблема спящего парикма-

хера». Впервые эту задачу сформулировал Эдсгер Дейкстра (Edsger Dijkstra) в 1965 году[1]. Вот краткое изложение условий:

- имеется парикмахерская;
- клиенты приходят в парикмахерскую через разные интервалы времени, от 10 до 30 миллисекунд;
- в зале ожидания имеются три стула;
- в парикмахерской работает один парикмахер и имеется единственное кресло для обслуживаемого клиента;
- когда кресло освобождается, в него садится следующий клиент, парикмахер просыпается и приступает к работе;
- если кресло и все стулья в зале ожидания заняты, клиент уходит;
- стрижка занимает 20 миллисекунд;
- после стрижки клиент встает и уходит.

Напишите многопоточную программу, определяющую число клиентов, которых может принять парикмахер за 10 секунд.

7.5. В заключение о Clojure

Clojure сочетает в себе мощь языка Lisp и удобство JVM. От JVM язык Clojure получил обширное сообщество, платформу для развертывания приложений и богатую библиотеку кода. От Lisp он унаследовал соответствующие сильные и слабые стороны.

Парадокс языка Lisp

Язык Clojure является, пожалуй, самым мощным и гибким из всех, представленных в этой книге. Мультиметоды позволяют писать мультипарадигмальный код, макросы дают возможность переопределять синтаксические конструкции что называется «на лету». Никакой другой язык в этой книге не способен предложить подобное сочетание. Такая гибкость дает программистам невероятную силу. В своей книге «Hackers and Painters» Пол Грэхем (Paul Graham) приводит хронологию одного проекта, когда программисты, вооруженные языком Lisp, смогли оставить далеко позади всех своих конкурентов. Некоторые консалтинговые компании даже готовы биться об заклад, что Clojure сможет обеспечить продуктивность

---

[1] http://ru.wikipedia.org/wiki/Проблема_спящего_парикмахера. – *Прим. перев.*

и качество работы, просто недоступные при использовании других языков.

Гибкость Lisp одновременно является и его недостатком. Макросы – мощное оружие в руках опытного специалиста, но оно же может оказаться разрушительным при бездумном и неумелом использовании. Та же самая возможность почти без усилий выразить решение задачи всего в нескольких строчках при использовании множества мощных абстракций делает Lisp особенно сложным языком для всех, кроме опытных программистов.

Чтобы по достоинству оценить Clojure, необходимо не только знать Lisp, но и быть знакомым с другими уникальными аспектами экосистемы Java и самого языка Clojure. Давайте подробнее рассмотрим наиболее сильные стороны Clojure.

Основные сильные стороны

Clojure – один из немногих языков, соперничающих за право стать вторым популярным языком на виртуальной машине Java. Тому есть несколько веских предпосылок.

Дружелюбный Lisp

Тим Брэй (Tim Bray), эксперт в области языков программирования и известный блоггер, назвал Clojure дружелюбным Lisp в статье «Eleven Theses on Clojure»[1]. Фактически он назвал Clojure «лучшим диалектом Lisp из когда-либо существовавших». Я не могу не согласиться, что Clojure – очень дружелюбный Lisp.

В этой главе вы узнали от самого Рича Хикки (Rich Hickey), что делает язык Clojure дружелюбным диалектом Lisp.

- *Уменьшение количества скобок.* Синтаксис Clojure стал более читаемым благодаря использованию квадратных скобок для обозначения векторов, фигурных скобок – для обозначения ассоциативных массивов и комбинаций символов – для обозначения множеств.
- *Экосистема.* Разнообразие диалектов Lisp распыляет усилия разработчиков, в результате объем поддержки и набор библиотек, которые можно получить для каждого конкретного диалекта, оказываются невелики. Как ни странно, но появление еще одного диалекта может помочь решить эту проблему.

---

[1] http://www.tbray.org/ongoing/When/200x/2009/12/01/Clojure-Theses.

Опираясь на JVM, язык Clojure может воспользоваться этим и дать программистам на Java еще более богатый выбор.

- *Сдержанность*. Проявив сдержанность и ограничив синтаксис языка Clojure, сведя к минимуму использование макросов чтения, Хикки не только ограничил мощь Clojure, но и снизил вероятность появления вредных его диалектов.

Кто-то может высоко ценить язык Lisp как самостоятельный язык программирования. С этой точки зрения Clojure можно рассматривать как обновленный Lisp. И в этом отношении он очень неплох.

Параллельные вычисления

Подходы к параллельным вычислениям, реализованные в Clojure, имеют все шансы в корне изменить наши взгляды на проектирование многозадачных систем. Транзакционная память STM действительно ложится дополнительным бременем на плечи разработчика, но она и защищает их же, определяя ситуации изменения состояния в защищенных функциях, – вы не сможете изменить состояние вне контекста транзакции.

Интеграция с Java

Clojure великолепно интегрируется с Java. Он использует некоторые «родные» типы Java, такие как строки и числа, и дает возможность добавлять указания на типы для повышения производительности. Но еще большим достоинством Clojure является тесная интеграция с JVM, благодаря которой типы, реализованные на Clojure, могут без всяких ограничений использоваться в приложениях на Java. Начав заниматься языком Clojure, вы вскоре увидите, что он сам реализован в JVM.

Отложенные вычисления

Clojure включает мощные механизмы отложенных вычислений. Отложенные вычисления способны упростить решение многих задач. Вы лишь мельком видели, как с помощью «ленивых» последовательностей можно организовать решение проблем. «Ленивые» последовательности способны уменьшить объем ненужных вычислений, откладывая их до момента, когда они действительно потребуются. Наконец, отложенные вычисления – это всего лишь один из инструментов решения сложных задач. Вы часто будете использовать «ленивые» последовательности взамен рекурсии, итераций или полностью реализованных коллекций.

Данные как код

Программы – это списки. В любом диалекте Lisp данные можно представить как код. Знание языка Ruby помогло мне оценить возможность писать программы в программах. На мой взгляд, такая возможность является наиболее ценной для любого языка программирования. Функциональные программы дают возможность метапрограммирования посредством функций высшего порядка. Lisp еще дальше развил эту идею, давая возможность интерпретировать данные как код.

Основные недостатки

Clojure изначально создавался как универсальный язык программирования. Насколько он будет пользоваться популярностью среди прочих языков JVM, еще только предстоит узнать. Clojure реализует замечательные абстракции, но их слишком много. Чтобы по-настоящему овладеть всеми возможностями Clojure, программист должен быть высокообразованным и очень талантливым человеком. Ниже перечислены некоторые особенности языка, которые я считаю его недостатками.

Префиксная запись

Представление кода в форме списка – одна из наиболее сильных особенностей любого диалекта Lisp, но она обходится дорогой ценой – необходимостью использовать префиксную (или польскую) форму записи[1]. Типичные объектно-ориентированные языки имеют совершенно иной синтаксис. Привыкнуть к префиксной форме записи нелегко. Для этого требуется хорошая память, а кроме того, разработчик должен научиться читать выражения изнутри наружу. Иногда я ловил себя на том, что при чтении кода на Clojure вынужден слишком много времени уделять деталям. Синтаксис Lisp не запоминается мною надолго. Говорят, что с опытом ситуация улучшается, но я пока не достиг того уровня.

Читаемость

Еще одна плата за возможность интерпретации данных как кода – угнетающе большое число круглых скобок. Удобство для компьютеров и человека – не одно и то же. Расстановка и число круглых скобок

---

[1] В Clojure имеются макросы ->> и ->, несколько смягчающие эту проблему.

в программах все еще являются проблемой. Программисты на Lisp в большей степени, чем другие, полагаются на текстовые редакторы в проверке парности круглых скобок, но никакой, даже самый совершенный инструмент не способен решить проблему плохой читаемости кода. Почет и уважение Ричу, что немного ослабил эту проблему, но она все еще остается проблемой.

Скорость обучения

Clojure – очень богатый язык, из-за чего кривая изучения его имеет очень пологий вид. Ваша команда должна состоять из по-настоящему талантливых разработчиков, чтобы реализовать проект на Lisp. «Ленивые» последовательности, функциональное программирование, макросы, транзакционная память и изощренность подходов к решению некоторых задач – все это весьма мощные концепции, для овладения которыми требуется время.

Ограниченный Lisp

За любые компромиссы приходится платить. Будучи языком, опирающимся на JVM, Clojure ограничивает оптимизацию хвостовой рекурсии. Программисты на Clojure вынуждены использовать неуклюжий синтаксис применения функции `recur`. Попробуйте реализовать функцию `(size x)`, вычисляющую размер последовательности `x` с помощью рекурсии и пары инструкций `loop/recur`.

Устранение макросов чтения, определяемых пользователем, также является важным шагом. Преимущества его очевидны. Неумелое использование макросов чтения может приводить к разрушению языка. Цена такого шага тоже очевидна – вы теряете еще один инструмент метапрограммирования.

Доступность

Одной из лучших черт Ruby или ранних версий Java является доступность каждого, как языка программирования. Оба этих языка относительно просты в освоении. Clojure, напротив, предъявляет слишком большие требования к разработчикам. Количество инструментов, абстракций и понятий в нем для многих может оказаться ошеломляющим.

Заключительные замечания

Основные достоинства и недостатки языка Clojure связаны с его гибкостью и богатством возможностей. Вам придется проявить не-

дюжинное упорство, чтобы освоить Clojure. В действительности если вы программируете на Java, то вам упорства не занимать. Просто сейчас вы тратите свое время на изучение абстракций Java прикладного уровня. Вы ищете возможность ослабления связей, применяя Spring или аспектно-ориентированное программирование. Но не получаете дополнительной гибкости на уровне самого языка. Многие из вас пока соглашаются на эти компромиссы. Однако, по моему скромному мнению, новые требования к распараллеливанию вычислений и увеличение сложности решаемых задач постепенно будут делать платформу Java все менее жизнеспособной.

Если вы желаете использовать модель экстремального программирования и готовы потратить время и силы на изучение языка, Clojure будет отличным выбором. Я уверен, что это замечательный язык для коллективов дисциплинированных и образованных разработчиков. Благодаря Clojure вы сможете писать качественный код намного быстрее.

Глава 8

Haskell

Логика – это маленькая птичка,
щебечущая в лугах.

Спок

Для многих пуристов функционального программирования язык Haskell является воплощением чистоты и свободы. Это мощный и богатый язык, но всякое богатство имеет свою цену. Вы не сможете откусить только маленький кусочек. Haskell заставит вас съесть функциональный пирог целиком, до последней крошки. Вспомните Спока из сериала «Звездный путь». Цитата в заголовке главы[1] является типичной для персонажа. Его характерная черта – целеустремленная чистота мышления, выработанная поколениями предков. В отличие от Scala, Erlang и Clojure, допускающих использование императивных понятий в небольших количествах, Haskell не оставляет пространства для маневра. Этот чисто функциональный язык заставит вас попыхтеть, когда возникнет потребность выполнить ввод/вывод или обеспечить накопление некоторой информации.

8.1. Введение в Haskell

Как обычно, чтобы понять, почему создатели языка пошли на те или иные компромиссы, следует обратиться к его истории. В начале и середине 1980-х появилось сразу несколько чисто функциональных языков программирования. Основной задачей новых исследований стал поиск путей реализации отложенных вычислений, с которыми мы познакомились во время знакомства с Clojure, и приемов исключительно функционального программирования. На конференции

---

[1] «StarTrek: The Original Series», Episodes 41 and 42: «I, Mudd»/«The Trouble with Tribbles». Режиссер Марк Дэниелс (Marc Daniels). 1967; Бербанк (Калифорния): 20th CBS Paramount International Television, 2001. («Звездный путь: Оригинальный сериал», серии 41 и 42: «Я, Мадд»/«Проблема с трибблами». – *Прим. перев.*)

по функциональным языкам программирования и компьютерным архитектурам (Functional Programming Languages and Computer Architecture, FPCA) в 1987 году была сформирована группа для разработки открытого стандарта исключительно функционального языка программирования. В недрах этой группы в 1990 году родился язык Haskell. Этот стандарт был пересмотрен в 1998 году и получил название Haskell 98. В последующие годы он пересматривался еще несколько раз, и теперь определение новой версии Haskell носит название Haskell Prime.

Итак, Haskell изначально создавался как исключительно функциональный язык программирования, сочетающий в себе идеи лучших функциональных языков, с особым вниманием к механизму отложенных вычислений.

Haskell имеет статическую систему типов со строгим контролем за ними, как язык Scala. Модель типов главным образом построена на их автоматическом определении и многими обозревателями считается одной из самых эффективных среди функциональных языков. Далее вы увидите, что система типов поддерживает полиморфизм и имеет очень ясную архитектуру.

Язык Haskell поддерживает также другие концепции, встречавшиеся нам в этой книге. Он позволяет использовать сопоставление с образцом и ограничители в стиле Erlang. Вы также увидите сходство механизмов отложенных вычислений в Haskell и Clojure и поддержку генераторов списков, аналогичных подобным механизмам в Clojure и Erlang.

Как исключительно функциональный язык Haskell не оставляет места в программах для побочных эффектов. Функции не производят побочных эффектов, но они могут возвращать их для выполнения в будущем. Пример этого вы увидите в третий день, тогда же вы познакомитесь еще и с примером сохранения состояния с использованием концепции *монад*.

В первые два дня мы пройдемся по наиболее типичным концепциям функционального программирования, таким как выражения, функции, функции высшего порядка и др. Мы также познакомимся с моделью типов в языке Haskell, заключающей в себе несколько новых для нас концепций. Третий день заставит вас попыхтеть. В этот день мы познакомимся с поддержкой параметризованных типов и монадами – довольно сложными в освоении концепциями. Итак, приступим.

8.2. День 1: логический

Как и Спок, вы увидите, что базовые понятия языка Haskell довольно просты в освоении. Вы познакомитесь с особенностями определения функций. В Haskell функции *всегда* возвращают одни и те же значения для одних и тех же аргументов. Во время работы над этой главой я использовал компилятор GHC – Glasgow Haskell Compiler версии 6.12.1. Он доступен для всех основных платформ, но вы можете также использовать другие реализации. Как обычно, начнем с запуска интерактивного сеанса (консоли). Введите команду `ghci`:

```
GHCi, version 6.12.1: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
```

Как видите, Haskell загружает несколько пакетов, после чего выводит приглашение к вводу команд.

Выражения и простые типы

Давайте пока отложим обсуждение системы типов в языке Haskell, а в этом разделе сфокусируемся на использовании простых типов. Как и при обсуждении других языков, начнем с чисел и некоторых простых выражений. Затем мы быстро перейдем к более сложным типам, таким как функции.

Числа

Вы уже знаете, с чего начать. Введите несколько выражений:

```
Prelude> 4
4
Prelude> 4 + 1
5
Prelude> 4 + 1.0
5.0
Prelude> 4 + 2.0 *5
14.0
```

Операции выполняются в логичном и ожидаемом порядке:

```
Prelude> 4 * 5 + 1
21
Prelude> 4 * (5 + 1)
24
```

Обратите внимание на возможность группировать операции с помощью круглых скобок. Итак, мы познакомились с парой числовых типов, теперь перейдем к символьным данным.

Символьные данные

Строки определяются как последовательности символов, заключенные в двойные кавычки:

```
Prelude> "hello"
"hello"
Prelude> "hello" + " world"

<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of '+' at <interactive>:1:0-17
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: "hello" + " world"
    In the definition of 'it': it = "hello" + " world"

Prelude> "hello" ++ " world"
"hello world"
```

Обратите внимание, что конкатенация строк выполняется с помощью оператора `++`, а не `+`. Одиночные символы определяются, как показано ниже:

```
Prelude> 'a'
'a'
Prelude> ['a', 'b']
"ab"
```

Имейте в виду, что строки – это всего лишь последовательности символов. А теперь перейдем к логическим значениям.

Логические значения

Логический тип является еще одним простым типом, поведение которого совпадает с поведением логических типов в большинстве других языков в этой книге, где используется инфиксная форма записи выражений. Операторы «равно» и «не равно» в Haskell возвращают логические выражения:

```
Prelude> (4 + 5) ==9
True
Prelude> (5 + 5) /= 10
False
```

Опробуем инструкцию `if/then`:

```
Prelude> if (5 == 5) then "true"

<interactive>:1:23: parse error (possibly incorrect indentation)
```

Вот оно, первое большое отличие от других языков в этой книге! Отступы в Haskell, как оказывается, имеют значение. Компилятор Haskell предположил, что, возможно, имеется следующая строка, имеющая некорректный отступ. Некоторые особенности оформления отступов будут показаны ниже. Мы не будем углубляться в обсуждение оформления отступов в управляющих структурах, поэтому просто следуйте примерам, которые будут демонстрироваться ниже, и все у вас будет в порядке. Давайте попробуем выполнить полную инструкцию `if/then/else`:

```
Prelude> if (5 == 5) then "true" else "false"
"true"
```

В языке Haskell инструкция `if` в действительности является функцией, а не управляющей синтаксической конструкцией, то есть она возвращает значение, как любая другая функция. Давайте попробуем проверить истинность/ложность некоторых значений:

```
Prelude> if 1 then "true" else "false"

<interactive>:1:3:
    No instance for (Num Bool)
      arising from the literal '1' at <interactive>:1:3
    ...
```

Haskell имеет строгую систему контроля типов. Функция `if` может принимать только логические значения. Попробуем вызвать другой конфликт типов:

```
Prelude> "one" +1

<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of '+' at <interactive>:1:0-8
    ...
```

Это сообщение об ошибке дает нам первую подсказку о строении системы типов в Haskell. Оно говорит: «Нет такой функции `+`, которая принимала бы аргумент типа `Num`, за которым следовал бы аргумент типа `[Char]`, то есть список символов». Обратите внимание, что мы пока ничего не говорили компилятору о типах. Он сам определяет типы, исходя из фактического синтаксиса. Мы в любой момент мо-

жем узнать, что делает механизм вывода типов в Haskell. Для этого достаточно включить флаг :t, как показано ниже:

```
Prelude> :set +t
Prelude> 5
5
it :: Integer
Prelude> 5.0
5.0
it :: Double
Prelude> "hello"
"hello"
it :: [Char]
Prelude> (5 == (2 + 3))
True
it :: Bool
```

Теперь после ввода каждого выражения вы сможете увидеть тип значения, возвращаемого этим выражением. Хочу сразу предупредить, что применительно к числам ключ :t может давать неожиданные результаты. Особенно это относится к интерпретации чисел интерактивной оболочкой. Попробуйте воспользоваться функцией :t:

```
Prelude> :t 5
5 :: (Num t) =>t
```

Как видите, результат отличается от того, что мы видели выше (it :: Integer). Интерактивная оболочка пытается интерпретировать значения максимально обобщенно, если не была выполнена команда :set +t. Вместо простого типа мы получили класс, который описывает группу родственных типов. Подробнее об этом мы поговорим в разделе 8.4, в подразделе «Классы».

Функции

Функции являются краеугольным камнем всей парадигмы программирования на языке Haskell. Так как в Haskell используется статическая система типов со строгим контролем, определение каждой функции должно состоять из двух частей: необязательного описания типов и реализации. Далее мы галопом пройдемся по концепциям, которые уже видели в других языках, поэтому держитесь крепче.

Определение простых функций

Функции в языке Haskell традиционно делятся на две части: объявление типа и объявление функции.

Для начала познакомимся с определением функций в интерактивной оболочке. Для связывания аргументов с параметрами мы будем использовать функцию `let`. Прежде чем приступать к определению своих функций, опробуем функцию `let`. Как и в языке Lisp, функция `let` в Haskell связывает переменную с функцией в локальной области видимости.

```
Prelude> let x = 10
Prelude> x
10
```

При создании модулей на языке Haskell функции объявляются так:

```
double x = x * 2
```

Однако в интерактивной оболочке связывание функции с переменной мы будем выполнять с помощью `let`, чтобы потом ее можно было использовать. Ниже приводится пример определения простой функции `double`:

```
Prelude> let double x = x * 2
Prelude> double 2
4
```

Теперь перейдем к использованию файлов программ. Благодаря этому мы сможем записывать многострочные определения. При использовании компилятора GHC полное определение функции `double` выглядит так:

haskell/double.hs
http://media.pragprog.com/titles/btlang/code/haskell/double.hs

```
module Main where

    double x = x + x
```

Обратите внимание, что здесь мы добавили *модуль* с именем `Main`. В Haskell модули позволяют собрать программный код в одно пространство имен. Модуль `Main` играет особую роль. Это – модуль верхнего уровня. А теперь сосредоточимся на функции `double`. Загрузите модуль `Main` в интерактивную оболочку и вызовите функцию `double`, как показано ниже:

```
Prelude> :load double.hs
[1 of 1] Compiling Main         ( double.hs, interpreted )
Ok, modules loaded: Main.
*Main> double 5
10
```

Пока что мы нигде не указывали типов. Haskell прощает такую «оплошность» и определяет типы автоматически. Каждое определение функции можно снабдить объявлением типа возвращаемого значения, как показано ниже:

haskell/double_with_type.hs
http://media.pragprog.com/titles/btlang/code/haskell/double_with_type.hs

```
module Main where

    double :: Integer -> Integer
    double x = x +x
```

Загрузите и используйте этот модуль, как было показано выше:

```
[1 of 1] Compiling Main         ( double_with_type.hs, interpreted )
Ok, modules loaded: Main.
*Main> double 5
10
```

Увидеть тип, ассоциированный с новой функцией, можно так:

```
*Main> :t double
double :: Integer -> Integer
```

Это определение означает, что функция `double` принимает аргумент типа `Integer` (первый `Integer`) и возвращает значение типа `Integer`.

Такое определение типа накладывает некоторые ограничения. Если вернуться к версии функции `double` без объявлений типов, можно увидеть совсем иное:

```
*Main> :t double
double :: (Num a) => a ->a
```

Они различны! В данном случае `a` – это переменный тип. Данное определение означает: «Функция `double` принимает единственный аргумент некоторого типа `a` и возвращает значение этого же типа `a`». Такое определение функции позволяет применять ее к значениям любых типов, поддерживающих функцию `+`. А теперь пойдем по нарастающей и попробуем реализовать что-нибудь более интересное, например функцию вычисления факториала.

Рекурсия

Начнем с простой рекурсии. Ниже приводится однострочная реализация вычисления факториала в интерактивной оболочке:

```
Prelude> let fact x = if x == 0 then 1 else fact (x - 1) * x
Prelude> fact 3
6
```

Начнем по порядку. Факториал `x` – это `1`, если `x` имеет значение `0`, и `fact (x - 1) * x` в противном случае. Мы можем немного улучшить реализацию, задействовав сопоставление с образцом. Фактически сопоставление с образцом выглядит и действует так же, как в языке Erlang:

haskell/factorial.hs
http://media.pragprog.com/titles/btlang/code/haskell/factorial.hs

```
module Main where
    factorial :: Integer -> Integer
    factorial 0 = 1
    factorial x = x * factorial (x - 1)
```

Определение состоит из трех строк. В первой строке определяются типы аргумента и возвращаемого значения. В следующих двух строках определяются два тела функции, которые вызываются в зависимости от результатов сопоставления со значением аргумента. Факториал 0 равен 1, а факториал произвольного числа `x` равен `x * factorial(x - 1)`. Это определение в точности соответствует математическому определению факториала. В данном случае порядок следования образцов играет важную роль. Haskell выберет реализацию, соответствующую первому совпадению. Если вы пожелаете изменить порядок на обратный, используйте ограничители. Ограничителями в Haskell являются условные выражения, ограничивающие значение аргумента:

haskell/fact_with_guard.hs
http://media.pragprog.com/titles/btlang/code/haskell/fact_with_guard.hs

```
module Main where
    factorial :: Integer -> Integer
    factorial x
        | x > 1 = x * factorial (x - 1)
        | otherwise =1
```

В этом примере ограничители, возвращающие логические значения, находятся слева, а применяемые функции – справа. Когда ограничение удовлетворяется, Haskell вызывает соответствующую функцию. Ограничители часто используются взамен сопоставления с образцом, и мы использовали их здесь, чтобы описать базовый случай рекурсии.

Кортежи и списки

Haskell, как и другие языки, поддерживает оптимизацию хвостовой рекурсии для повышения эффективности работы рекурсивных алго-

ритмов. Давайте рассмотрим несколько версий вычисления последовательности чисел Фибоначчи на языке Haskell. Сначала рассмотрим самую простую реализацию:

haskell/fib.hs

http://media.pragprog.com/titles/btlang/code/haskell/fib.hs

```
module Main where
    fib :: Integer -> Integer
    fib 0 = 1
    fib 1 = 1
    fib x = fib (x - 1) + fib (x - 2)
```

Здесь все достаточно очевидно. Вызов `fib 0` или `fib 1` возвращает `1`, а вызов `fib x` возвращает `fib (x - 1) + fib (x - 2)`. Но это решение неэффективно. Давайте рассмотрим более эффективное решение.

Программирование с использованием кортежей

Кортежи позволяют получить более эффективную реализацию. Кортеж – это коллекция с фиксированным количеством элементов. Кортежи в языке Haskell определяются как последовательности элементов через запятую, заключенные в круглые скобки. Следующая реализация создает кортеж соседних чисел Фибоначчи и использует счетчик в помощь рекурсии. Ниже приводится простое решение:

```
fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
fibTuple (x, y, 0) = (x, y, 0)
fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)
```

Функция `fibTuple` принимает кортеж с тремя элементами и возвращает также кортеж с тремя элементами. Но будьте внимательны. Один параметр в виде кортежа с тремя элементами – это далеко не то же самое, что три параметра. Вызовем эту функцию, передав ей два начальных числа, `0` и `1`. Дополнительно передадим ей счетчик. По мере уменьшения счетчика будут выполняться рекурсивные вызовы, и первые два аргумента будут изменяться, последовательно получая следующую пару чисел последовательности. Вызов `fibTuple (0, 1, 4)` приведет к следующей последовательности вызовов:

- `fibTuple (0, 1, 4)`
- `fibTuple (1, 1, 3)`
- `fibTuple (1, 2, 2)`
- `fibTuple (2, 3, 1)`
- `fibTuple (3, 5, 0)`

Запустить программу можно так:

```
Prelude> :load fib_tuple.hs
[1 of 1] Compiling Main             ( fib_tuple.hs, interpreted )
Ok, modules loaded: Main.
*Main> fibTuple(0, 1, 4)
(3, 5, 0)
```

Ответ находится в первой позиции. Его можно извлечь так:

```
fibResult :: (Integer, Integer, Integer) -> Integer
fibResult (x, y, z) = x
```

Для извлечения первого элемента мы просто использовали сопоставление с образцом. Модель использования можно упростить:

```
fib :: Integer -> Integer
fib x = fibResult (fibTuple (0, 1, x))
```

Следующая функция реализует очень быстрый генератор чисел Фибоначчи с помощью двух вспомогательных функций. Ниже приводится вся программа целиком:

haskell/fib_tuple.hs

http://media.pragprog.com/titles/btlang/code/haskell/fib_tuple.hs

```
module Main where
    fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer,
Integer)
    fibTuple (x, y, 0) = (x, y, 0)
    fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)

    fibResult :: (Integer, Integer, Integer) -> Integer
    fibResult (x, y, z) =x

    fib :: Integer -> Integer
    fib x = fibResult (fibTuple (0, 1, x))
```

А вот результаты (которые появляются мгновенно):

```
*Main> fib 100
354224848179261915075
*Main> fib 1000
    43466557686937456435688527675040625802564660517371780
    40248172908953655541794905189040387984007925516929592
    25930803226347752096896232398733224711616429964409065
    33187938298969649928516003704476137795166849228875
```

Давайте попробуем другой подход, основанный на композиции функций.

Использование кортежей и приема композиции

Иногда бывает необходимо объединить вызовы функций в единую цепочку, передавая результат выполнения одной функции в вызов другой. Ниже приводится пример, вычисляющий второй элемент списка как «голову» «хвоста» этого списка:

```
*Main> let second = head . tail
*Main> second [1, 2]
2
*Main> second [3, 4, 5]
4
```

Здесь мы объявили функцию в интерактивной оболочке. Объявление `second = head . tail` эквивалентно объявлению `second lst = head (tail lst)`. То есть мы передаем результат одной функции в вызов другой. Давайте задействуем эту особенность в еще одной реализации вычисления чисел Фибоначчи. Мы будем вычислять пару, как и прежде, но без счетчика:

```
fibNextPair :: (Integer, Integer) -> (Integer, Integer)
fibNextPair (x, y) = (y, x + y)
```

Имея последовательность из двух чисел, мы всегда можем вычислить следующее число. Функция ниже должна рекурсивно вычислять следующий элемент в последовательности:

```
fibNthPair :: Integer -> (Integer, Integer)
fibNthPair 1 = (1, 1)
fibNthPair n = fibNextPair (fibNthPair (n - 1))
```

Базовым случаем является значение `(1, 1)` для значения `n`, равного `1`, а каждый следующий элемент последовательности вычисляется на основе предыдущего. Таким образом, мы можем получить любую пару в последовательности:

```
*Main> fibNthPair(8)
(21,34)
*Main> fibNthPair(9)
(34,55)
*Main> fibNthPair(10)
(55,89)
```

Теперь осталось извлечь первые элементы из каждой пары и объединить их в последовательность. С этой целью мы составим композицию из вспомогательной функции `fst`, извлекающей первый элемент, и функции `fibNthPair`, конструирующей пары чисел:

haskell/fib_pair.hs
http://media.pragprog.com/titles/btlang/code/haskell/fib_pair.hs

```
module Main where
    fibNextPair :: (Integer, Integer) -> (Integer, Integer)
    fibNextPair (x, y) = (y, x + y)

    fibNthPair :: Integer -> (Integer, Integer)
    fibNthPair 1 = (1, 1)
    fibNthPair n = fibNextPair (fibNthPair (n - 1))

    fib :: Integer -> Integer
    fib = fst . fibNthPair
```

Выражаясь простым языком, функция fib извлекает первый элемент из *n*-го кортежа. Вот и все. А теперь перейдем от кортежей к спискам.

Обход списков

Мы уже знакомы со списками по другим языкам, поэтому я не буду повторно рассказывать базовые сведения о них. Для начала мы рассмотрим небольшой пример рекурсии с применением списка, а потом познакомимся с несколькими функциями, с которыми вы пока незнакомы. Разбиение списка на «голову» и «хвост» можно использовать в любых операциях связывания, таких как инструкция `let` или сопоставление с образцом:

```
*Main> let (h:t) = [1, 2, 3, 4]
*Main> h
1
*Main> t
[2,3,4]
```

Здесь мы связали список `[1, 2, 3, 4]` с конструкцией `(h:t)`. Эту конструкцию можно считать аналогом конструкций `head|tail` в языках Prolog, Erlang и Scala. С помощью этого инструмента можно создавать простые рекурсивные определения. Например, функции `size` и `prod` для списков:

haskell/lists.hs
http://media.pragprog.com/titles/btlang/code/haskell/lists.hs

```
module Main where
    size [] = 0
    size (h:t) = 1 + sizet

    prod [] = 1
    prod (h:t) = h * prodt
```

Определяя эти функции, я положился на механизм вывода типов в языке Haskell, но сама идея должна быть понятна. Размер списка (`size`) равен размеру его «хвоста» плюс 1.

```
Prelude> :load lists.hs
[1 of 1] Compiling Main
    ( lists.hs, interpreted )
Ok, modules loaded: Main.
*Main> size "Fascinating."
12
```

Функция `zip` – это мощный инструмент конструирования списков. Ниже демонстрируется применение данной функции:

```
*Main> zip "kirk" "spock"
[('kirk','spock')]
```

Итак, мы создали кортеж с двумя элементами. Аналогичным способом можно объединить два списка:

```
Prelude> zip ["kirk", "spock"] ["enterprise", "reliant"]
[("kirk","enterprise"),("spock","reliant")]
```

Как видите, функция `zip` дает эффективный способ объединения двух списков.

До сих пор рассматривавшиеся особенности языка Haskell в значительной степени были похожи на аналогичные особенности других языков. Теперь перейдем к исследованию некоторых более сложных конструкций, включая диапазоны и генераторы списков.

Создание списков

Мы уже видели несколько способов рекурсивной обработки списков. В этом разделе мы рассмотрим несколько приемов создания списков. В частности, мы познакомимся с рекурсивным способом, с диапазонами и генераторами списков.

Рекурсия

Основным строительным блоком, используемым для конструирования списков, является оператор двоеточия (`:`), создающий список из «головы» и «хвоста». Мы уже имели возможность видеть этот оператор в обратной операции, когда выполняли сопоставление с образцом в рекурсивном вызове функции. Здесь оператор двоеточия (`:`) находится в левой части инструкции `let`:

```
Prelude> let h:t = [1, 2, 3]
Prelude> h
```

```
1
Prelude> t
[2,3]
```

Оператор двоеточия (:) можно использовать не только для разбиения списков, но и для их создания. Вот как может выглядеть такая операция:

```
Prelude> 1:[2, 3]
[1,2,3]
```

Напомню, что списки в языке Haskell могут хранить только данные одного типа. Нельзя добавить список в список целых чисел, например:

```
Prelude> [1]:[2, 3]

<interactive>:1:8:
    No instance for (Num [t])
      arising from the literal '3' at <interactive>:1:8
```

Однако в список списков можно добавить список и можно даже добавить пустой список:

```
Prelude> [1]:[[2], [3, 4]]
[[1],[2],[3,4]]
Prelude> [1]:[]
[[1]]
```

Итак, мы увидели, как конструировать списки. Допустим теперь, что нам нужно создать функцию, возвращающую четные числа из списка. Одно из решений заключается в том, чтобы сконструировать новый список:

haskell/all_even.hs
http://media.pragprog.com/titles/btlang/code/haskell/all_even.hs

```
module Main where
    allEven :: [Integer] -> [Integer]
    allEven [] = []
    allEven (h:t) = if even h then h:allEven t else allEvent
```

Эта функция принимает список целых чисел и возвращает список четных целых чисел. Для пустого списка `allEven` вернет пустой список. Если исходный список не пуст и его «голова» является четным числом, она добавляется в результат применения `allEven` к «хвосту» исходного списка. Если «голова» является нечетным числом, она отбрасывается и функция `allEven` рекурсивно применяется к «хвосту». Все просто! Перейдем к другим способам создания списков.

Диапазоны и композиция функций

Подобно языкам Ruby и Scala, в языке Haskell имеются диапазоны и дополнительный синтаксический сахар для их поддержки. Диапазоны в Haskell определяются значениями границ:

```
Prelude> [1..2]
[1,2]
Prelude> [1..4]
[1,2,3,4]
```

Вы указываете конечные точки, а Haskell вычисляет все значения диапазона. По умолчанию шаг наращивания равен 1. А что случится, если Haskell не сможет достичь верхней границы диапазона с шагом наращивания по умолчанию?

```
Prelude> [10..4]
[]
```

Вы получите пустой список. Шаг наращивания можно определить, указав следующий элемент в списке:

```
Prelude> [10, 8 .. 4]
[10,8,6,4]
```

Допускается также использовать дробные числа:

```
Prelude> [10, 9.5 .. 4]
[10.0,9.5,9.0,8.5,8.0,7.5,7.0,6.5,6.0,5.5,5.0,4.5,4.0]
```

Диапазоны – это синтаксический сахар, помогающий создавать последовательности. Последовательности необязательно должны быть конечными. Как и в языке Clojure, в Haskell поддерживается возможность получить несколько элементов последовательности:

```
Prelude> take 5 [ 1 ..]
[1,2,3,4,5]
Prelude> take 5 [0, 2 ..]
[0,2,4,6,8]
```

Подробнее о «ленивых» последовательностях мы поговорим во второй день. А пока познакомимся еще с одним способом автоматического создания списков – с генераторами списков.

Генераторы списков

Впервые с генераторами списков мы столкнулись при знакомстве с языком Erlang. В Haskell генераторы списков действуют точно так же: с левой стороны находится выражение, генерирующее очередной

элемент списка, а с правой – генераторы и фильтры, в точности как в языке Erlang. Давайте рассмотрим несколько примеров. Удвоить значения элементов списка можно так:

```
Prelude> [x * 2 | x <- [1, 2, 3]]
[2,4,6]
```

Выражаясь на простом языке, этот генератор списков говорит: «Собрать все результаты выражения x * 2, где каждое значение x извлекается из списка [1, 2, 3]».

Как и в Erlang, в генераторах списков можно также использовать сопоставление с образцом. Допустим, что у нас имеется список координат вершин многоугольника и нам требуется найти его зеркальное отражение относительно диагонали. Для этого достаточно поменять значения координат x и y местами, как показано ниже:

```
Prelude> [ (y, x) | (x, y) <- [(1, 2), (2, 3), (3, 1)]]
[(2,1),(3,2),(1,3)]
```

Или, чтобы найти симметричное отражение по горизонтали, можно вычесть значение координаты x каждой вершины из 4:

```
Prelude> [ (4 - x, y) | (x, y) <- [(1, 2), (2, 3), (3, 1)]]
[(3,2),(2,3),(1,1)]
```

Кроме того, с помощью генератора списков можно также находить все возможные комбинации. Допустим, что нам требуется найти все возможные комбинации из десантных отрядов Кирка, Спока и Маккоя:

```
Prelude> let crew = ["Kirk", "Spock", "McCoy"]
Prelude> [(a, b) | a <- crew, b <- crew]
[("Kirk","Kirk"),("Kirk","Spock"),("Kirk","McCoy"),
("Spock","Kirk"),("Spock","Spock"),("Spock","McCoy"),
("McCoy","Kirk"),("McCoy","Spock"),("McCoy","McCoy")]
```

У нас почти получилось, но данная реализация не проверяет повторяющихся имен. Добавим условие в фильтр генератора списков:

```
Prelude> [(a, b) | a <- crew, b <- crew, a /= b]
[("Kirk","Spock"),("Kirk","McCoy"),("Spock","Kirk"),
("Spock","McCoy"),("McCoy","Kirk"),("McCoy","Spock")]
```

Так лучше, но порядок следования имен не имеет для нас значения. Давайте изменим условие в фильтре, потребовав, чтобы имена следовали в алфавитном порядке:

```
Prelude> [(a, b) | a <- crew, b <- crew, a < b]
[("Kirk","Spock"),("Kirk","McCoy"),("McCoy","Spock")]
```

Написав простой и компактный генератор списков, мы ответили на вопрос. Генераторы списков отлично подходят, когда нужно быстро создать или преобразовать список.

Интервью с Филиппом Уодлером (Philip Wadler)

Теперь, когда вы познакомились с некоторыми основными возможностями языка Haskell, давайте узнаем, что говорит один из членов комитета, занимавшегося проектированием Haskell. Профессор кафедры теоретической информатики Эдинбургского университета Филипп Уодлер (Philip Wadler) принимал активное участие в разработке не только языка Haskell, но также Java и XQuery. Ранее он работал в Avaya Labs, Bell Labs, Glasgow, Chalmers, Oxford, CMU, Xerox Parc и Stanford.

Брюс Тейт: Как вашей команде пришла идея создать Haskell?

Филипп Уодлер: В конце 80-х годов существовало большое число групп, занимавшихся проектированием и реализацией функциональных языков, и мы поняли, что сможем добиться гораздо большего, работая вместе. Мы сразу поставили амбициозную цель: создать язык, который мог бы служить основой для дальнейших исследований, пригодный для обучения студентов и для использования в промышленности. Вся хронология подробно описывается в статье, которую мы написали для конференции по истории развития языков программирования (History of Programming Languages conference, HOPL), которую можно найти в Сети[1].

Брюс Тейт: Что больше всего нравится вам в этом языке?

Филипп Уодлер: Я по-настоящему наслаждаюсь возможностью использования генераторов списков. Приятно видеть, что они наконец-то смогли пробиться в другие языки, такие как Python.

Мне нравятся классы типов, обеспечивающие простую форму обобщенного программирования. Одним лишь добавлением ключевого слова `derived` в определение своего типа данных вы получаете подпрограммы для сравнения значений, преобразования их в строки и обратно и другие. Я считаю эту возможность очень удобной, и мне не хватает ее, когда я пишу на других языках.

Любой хороший язык программирования включает средства расширения самого себя, позволяющие создавать дополнительные синтаксические конструкции, специализированные для ре-

---

[1] http://www.haskell.org/haskellwiki/History_of_Haskell.

шения конкретных задач. Haskell особенно хорош в этом отношении. Отложенные вычисления, lambda-выражения, монады и стрелочная нотация, классы типов, выразительная система типов и шаблоны Haskell – все они поддерживают возможность расширения языка.

Брюс Тейт: Что бы вы изменили в языке, будь у вас возможность вернуться назад?

Филипп Уодлер: В мире распределенных вычислений нам все чаще приходится заниматься разработкой программ, которые могут выполняться на множестве компьютеров и передавать данные друг другу. Когда вы посылаете некоторое значение, вы, вероятно, желаете отправить само значение (уже вычисленное), а не программу (со всеми значениями ее переменных), которую можно выполнить, чтобы получить значение. Поэтому в распределенном мире, как мне кажется, было бы лучше по умолчанию использовать немедленные вычисления (неотложенные), но дать возможность применять модель отложенных вычислений, когда это желательно.

Брюс Тейт: С каким из самых интересных применений Haskell вам приходилось сталкиваться?

Филипп Уодлер: Я всегда поражаюсь, какие задачи некоторые решают с помощью Haskell. Помню, много лет тому назад я был изумлен, встретив систему обработки текстов на естественных языках, реализованную на Haskell, и еще раз спустя несколько лет, когда столкнулся с приложением на Haskell, применявшимся в исследовании сворачиваемости белка, связанных с поиском лекарства против СПИДа. Стоило мне только заглянуть на страницу сообщества Haskell, и я сразу увидел список из сорока промышленных приложений на Haskell. Многие пользователи в настоящее время работают в финансовой сфере: ABN Amro, Credit Suisse, Deutsche Bank и Standard Chartered. Проект Facebook использует Haskell в качестве внутреннего инструмента для обновления сценариев на PHP. Но самое интересное применение Haskell, которое я видел, – сборка мусора. Нет, не та «сборка мусора» в программных системах, о которой мы наслышаны, а самая настоящая – я говорю о программируемых механизмах, используемых в мусоровозах!

Что мы узнали в первый день

Haskell – это функциональный язык программирования. Особо нужно отметить, что это чисто функциональный язык. Функции в этом языке для одного и того же набора значений аргументов всегда воз-

вращают одни и те же результаты. Они не имеют побочных эффектов. В первый день мы в основном рассматривали особенности, с которыми уже встречались в других языках.

Сначала мы познакомились с выражениями и простыми типами данных. Из-за отсутствия поддержки изменяемых переменных мы создали несколько математических функций, используя рекурсивные алгоритмы и списки. Мы рассмотрели простые выражения на языке Haskell и преобразовали их в функции. Мы также применяли приемы сопоставления с образцом и ограничители, известные нам по языкам Erlang и Scala. В качестве простых коллекций мы использовали списки и кортежи.

В заключение мы занялись рассмотрением темы создания списков, которая привела нас к генераторам списков, диапазонов и даже «ленивых» последовательностей. А теперь попробуйте применить вновь полученные знания на практике.

День 1: задания для самостоятельного решения

К настоящему времени создание функциональных программ не должно казаться вам чересчур сложным занятием, если, конечно же, вы внимательно читали предыдущие главы с описанием функциональных языков. В этом разделе я предлагаю вам решить чуть более сложные задачи.

Найдите:

- вики-страницу с описанием языка Haskell;
- сообщество пользователей Haskell, где бы вы могли найти поддержку для вашего компилятора.

Практические задания:

- подсчитайте, сколькими разными способами можно написать `allEven`;
- напишите функцию, принимающую список и возвращающую список с теми же элементами, но расположенными в обратном порядке;
- напишите функцию, создающую двухэлементные кортежи со всеми возможными комбинациями цветов: черного, белого, синего, желтого и красного; имейте в виду, что комбинации, такие как `(черный, синий)` и `(синий, черный)`, включающие одни и те же цвета, но в разном порядке, считаются одинаковыми;
- напишите генератор списков для создания таблицы умножения; таблица может быть представлена, например, списком трехэлементных кортежей, где первые два элемента являются

целыми значениями от 1 до 9, а третий представляет произведение первых двух;

- решите задачу раскрашивания карты (раздел 4.2, подраздел «Раскрашивание карты») на языке Haskell.

8.3. День 2: Самая сильная черта характера Спока

Часто так бывает, что мы первое время не замечаем самых главных черт характера персонажей. В случае со Споком все гораздо проще. Он всегда логичен и полностью предсказуем. Самой яркой чертой языка Haskell также являются его предсказуемость и логичность. Во многих университетах преподают Haskell в контексте рассуждений о программах. Если программа написана на Haskell, доказать правильность ее работы намного проще, чем если бы она была написана на любом другом императивном языке. В этом разделе мы углубимся в практические концепции, обеспечивающие более высокую предсказуемость. Сначала мы рассмотрим функции высшего порядка. Затем поговорим о стратегии их комбинирования в языке Haskell. Попутно мы познакомимся с частично примененными (partially applied) функциями и каррированием. И в заключение коснемся темы отложенных вычислений. Нас ждет весьма напряженный день, поэтому приступим не мешкая.

Функции высшего порядка

Все языки, представленные в этой книге, поддерживают идею программирования с применением функций высшего порядка. В языке Haskell эта концепция используется очень широко. Далее мы быстро пробежимся по анонимным функциями и посмотрим, как применять их в комплексе со стандартными функциями обработки списков. Я буду двигаться вперед намного быстрее, чем при описании других языков, потому что вы уже знакомы со многими понятиями, рассматривавшимися в предыдущих главах. Итак, анонимные функции.

Анонимные функции

Как и следовало ожидать, синтаксис определения анонимных функций в языке Haskell удивительно прост. Он имеет вид: `(\param1 .. paramn -> function_body)`. Взгляните на следующий пример:

```
Prelude> (\x -> x) "Logical."
"Logical."
Prelude> (\x -> x ++ " captain.") "Logical,"
"Logical, captain."
```

Сами по себе они не представляют ничего особенного. Но в комбинации с другими функциями они являются весьма мощным инструментом. Здесь сначала мы написали анонимную функцию, просто возвращающую первый параметр. Затем написали другую анонимную функцию, добавляющую строку.

map и where

Как мы уже имели возможность убедиться, анонимные функции играют важную роль при обработке списков. В языке Haskell имеется функция `map`:

```
map (\x -> x * x) [1, 2, 3]
```

Функция `map` применяется к анонимной функции и списку. `map` вызывает анонимную функцию для каждого элемента списка и собирает результаты в новый список. Пока ничего необычного, но форма вызова `map` может иметь более наглядный вид. Мы можем оформить этот вызов в виде отдельной функции и определить анонимную функцию как функцию, доступную только в локальной области видимости:

haskell/map.hs
http://media.pragprog.com/titles/btlang/code/haskell/map.hs

```
module Main where
    squareAll list = map square list
        where square x = x *x
```

Здесь объявляется функция `squareAll`, принимающая параметр с именем `list`. Далее используется функция `map`, которая применяет функцию `square` ко всем элементам списка `list`. И здесь задействуется новая особенность – ключевое слово `where`, объявляющее локальную версию функции `square`. В предложении `where` допускается связывать не только функции, но и переменные. Дополнительные примеры использования `where` мы увидим далее в этой главе. А пока взгляните на получившийся результат:

```
*Main> :load map.hs
[1 of 1] Compiling Main             ( map.hs, interpreted )
Ok, modules loaded: Main.
*Main> squareAll [1, 2, 3]
[1,4,9]
```

Функции `map` могут также передаваться части функций, которые называют *секциями* (sections):

```
Prelude> map (+ 1) [1, 2, 3]
[2,3,4]
```

Фактически `(+ 1)` – это частично примененная функция. Функция `+` принимает два параметра, а мы указали только один. В результате мы получили функцию вида `(x + 1)` с единственным параметром `x`.

filter, foldl, foldr

Следующей часто используемой функцией является `filter`, которая выполняет указанную проверку для каждого элемента списка:

```
Prelude> odd 5
True
Prelude> filter odd [1, 2, 3, 4, 5]
[1,3,5]
```

Поддерживаются также функции, выполняющие свертку влево и вправо, действующие подобно аналогичным функциям в Clojure и Scala. Вы часто будете использовать функции, являющиеся вариациями функций `foldl` и `foldr`:

```
Prelude> foldl (\x carryOver -> carryOver + x) 0 [1 .. 10]
55
```

Функция `foldl` принимает начальное значение `0` аккумулятора, применяет переданную ей функцию к каждому элементу заданного списка, используя предыдущий результат в виде первого аргумента `carryOver` и очередной элемент списка – в виде второго аргумента. Другая разновидность функции `fold` с успехом может использоваться для выполнения свертки с помощью оператора:

```
Prelude> foldl1 (+) [1 .. 3]
6
```

Здесь оператор `+` используется в роли чистой функции, принимающей два параметра и возвращающей целое число. Результат получается тот же, как в следующем выражении:

```
Prelude> 1 + 2 + 3
6
```

Существует также функция `foldr1`, выполняющая свертку справа налево.

Как вы уже наверняка поняли, Haskell предлагает множество других стандартных функций для работы со списками, и многие из них

являются функциями высшего порядка. Но, вместо того чтобы тратить место в этой главе на их описание, я позволю вам познакомиться с ними самостоятельно. А теперь я хочу перейти к обсуждению способов комбинирования функций в языке Haskell.

Частично примененные функции и карринг

Мы уже знакомы с приемами композиции и частичного применения функций. Эти понятия занимают достаточно важное место в языке Haskell, чтобы мы потратили некоторое время на них.

Все функции в языке Haskell принимают один параметр. Вы можете спросить: «Как же тогда пишутся функции, такие как `+`, складывающая два числа?»

В действительности здесь нет никакого противоречия. Все функции на самом деле принимают единственный параметр. Продемонстрируем это на примере простой функции с именем `prod`:

```
Prelude> let prod x y = x * y
Prelude> prod 3 4
12
```

Мы создали действующую функцию, и вы можете убедиться в этом. Давайте взглянем на тип функции:

```
Prelude> :t prod
prod :: (Num a) => a -> a -> a
```

Часть `(Num a) =>` означает: «В следующем определении типа элемент `a` представляет тип `Num`». Остальное вы уже видели прежде, и тогда я не говорил всей правды, чтобы не усложнять объяснения. Теперь пришло время узнать все до конца. В языке Haskell используется концепция разбиения одной функции с несколькими аргументами на несколько функций, каждая из которых принимает единственный аргумент. Достигается это за счет поддержки частичного применения.

Не дайте этим хитросплетениям терминов запутать себя. Прием частичного применения заключается в связывании *некоторых* аргументов, но не всех. Например, за счет частичного применения функции `prod` мы можем создать другие функции:

```
Prelude> let double = prod 2
Prelude> let triple = prod 3
```

Взгляните сначала на левую сторону этих функций. Мы определили функцию `prod` как принимающую два параметра, но применили только один – первый. Соответственно, `prod 2` вычисляется довольно

просто, достаточно взять оригинальную функцию `prod x y = x * y`, подставить `2` вместо `x`, и в результате получится `prod y = 2 * y`. Новая функция действует в полном соответствии с нашими ожиданиями:

```
Prelude> double 3
6
Prelude> triple 4
12
```

Как видите, здесь нет никакой мистики. Когда Haskell встречает вызов `prod 2 4`, он в действительности выполняет `(prod 2) 4`, как описывается ниже.

- Сначала применяется `prod 2`. В результате получается функция `(\y -> 2 * y)`.
- Затем применяется `(\y -> 2 * y) 4`, или `2 * 4`, что дает в результате 8.

Этот процесс называется *карринг* (currying), и он выполняется для всех функций с несколькими аргументами. В результате гибкость возрастает, а синтаксис упрощается. В большинстве случаев нет необходимости задумываться об этом, потому что значения каррированной и некаррированной функций совпадают.

Отложенные вычисления

Так же широко, как функции для работы с последовательностями в Clojure, в Haskell используется механизм отложенных вычислений. С помощью этого механизма можно создавать функции, возвращающие бесконечные списки. Взгляните на следующий пример, где определяется бесконечный диапазон, начинающийся со значения `x` и имеющий шаг приращения `y`:

haskell/my_range.hs
http://media.pragprog.com/titles/btlang/code/haskell/my_range.hs

```
module Main where
  myRange start step = start:(myRange (start + step) step)
```

Синтаксис может показаться непривычным, но общий эффект получается удивительным. Здесь мы определили функцию с именем `myRange`, принимающую начальное значение и шаг приращения для нашего диапазона. Мы использовали прием композиции для создания списка из значения `start`, играющего роль «головы», и результата вызова `(myRange (start + step) step)`, играющего роль «хвоста». Для вызова `myRange 1 1` будут выполнены следующие действия:

- `1:myRange (2 1)`
- `1:2:myRange (3 1)`
- `1:2:3:myRange (4 1)`

...и так далее.

Эта рекурсия будет продолжаться до бесконечности, поэтому такие функции обычно используются в комбинации с другими, ограничивающими число рекурсивных вызовов. Загрузите модуль `my_range.hs` и выполните:

```
*Main> take 10 (myRange 10 1)
[10,11,12,13,14,15,16,17,18,19]
*Main> take 5 (myRange 0 5)
[0,5,10,15,20]
```

Некоторые рекурсивные функции работают более эффективно, когда в них используется прием конструирования списка. Ниже приводится пример вычисления последовательности чисел Фибоначчи с применением отложенных вычислений и приема конструирования списков:

haskell/lazy_fib.hs
http://media.pragprog.com/titles/btlang/code/haskell/lazy_fib.hs

```
module Main where
    lazyFib x y = x:(lazyFib y (x + y))

    fib = lazyFib 11

    fibNth x = head (drop (x - 1) (take (x) fib))
```

Первая функция создает последовательность, каждый элемент в которой является суммой двух предыдущих. В виде этой функции мы уже получаем последовательность, но реализацию можно еще улучшить. Чтобы получить корректную последовательность чисел Фибоначчи, необходимо начать с двух чисел, 1 и 1, поэтому функция `fib` вызывает `lazyFib`, передавая ей два первых числа. В заключение определяется еще одна вспомогательная функция, дающая пользователю возможность извлечь единственное число, вызывая `drop` и `take`. Ниже показано, как действует эта функция:

```
*Main> take 5 (lazyFib 0 1)
[1,1,2,3,5]
*Main> take 5 (fib)
[1,1,2,3,5]
*Main> take 5 (drop 20 (lazyFib 0 1))
[10946,17711,28657,46368,75025]
```

```
*Main> fibNth 3
2
*Main> fibNth 6
8
```

Три функции дали нам красивое и компактное решение. Мы определяем бесконечную последовательность, а Haskell вычисляет только ту ее часть, которая действительно необходима. Еще более интересные результаты можно получить, комбинируя бесконечные последовательности. Для начала давайте соединим две последовательности чисел Фибоначчи с некоторым смещением относительно друг друга:

```
*Main> take 5 (zipWith (+) fib (drop 1 fib))
[2,3,5,8,13]
```

Неожиданно мы получили последовательность чисел Фибоначчи. Эти функции высшего порядка прекрасно взаимодействуют друг с другом. Здесь мы вызываем `zipWith`, которая объединяет элементы из двух бесконечных списков по их индексам, и передаем ей функцию `+`.

Аналогично можно реализовать удвоение элементов диапазона:

```
*Main> take 5 (map (*2) [1 ..])
[2,4,6,8,10]
```

Здесь мы использовали функцию `map`, чтобы применить частично примененную функцию `* 2` к бесконечному диапазону `[1 ..]`, и затем извлекли из получившейся бесконечной последовательности пять первых элементов.

Вся прелесть функциональных языков состоит в том, что они позволяют составлять самые немыслимые композиции. Например, мы можем использовать прием композиции функций в сочетании с частично примененными функциями и «ленивыми» последовательностями:

```
*Main> take 5 (map ((* 2) . (* 5)) fib)
[10,10,20,30,50]
```

Это довольно любопытное решение, поэтому давайте разберем его. Вычисления здесь выполняются изнутри наружу: сначала выполняется `(* 5)`. Это – частично примененная функция. Любое значение, которое получит эта функция, будет умножено на пять. Результат вызова этой функции передается другой частично примененной функции, `(* 2)`. Данная композиция функций передается функции `map` и применяется к каждому элементу бесконечной последовательности `fib`. Полученная бесконечная последовательность передается функ-

ции `take 5`, которая извлекает из последовательности пять первых чисел Фибоначчи, умноженных сначала на 5, а потом еще раз на 2.

Прием композиции упрощает решение задач – используя его, вы просто передаете одну функцию следующей. Выражение `f . g x` в языке Haskell является более краткой формой записи выражения `f(g x)`. Составляя вызовы функций таким способом, может потребоваться применить их от первой до последней. Сделать это проще с помощью оператора точки (`.`). Например, инвертирование изображения, поворот его по вертикали, а затем по горизонтали можно было бы реализовать так: `(flipHorizontally . flipVertically . invert) image`.

Интервью с Саймоном Пейтоном-Джонсом

Давайте сделаем короткий перерыв и узнаем, что говорит еще один член комитета, занимавшегося разработкой языка Haskell. Саймон Пейтон-Джонс (Simon Peyton Jones) семь лет читал лекции в колледже Лондонского университета и девять лет работал профессором в университете города Глазго, перед тем как перешел в Microsoft Research (Кембридж) в 1998 году, где сосредоточился на исследованиях в области реализации и применения функциональных языков программирования на одно- и многопроцессорных компьютерах. Он является ведущим проектировщиком компилятора, использовавшегося в этой книге.

Брюс Тейт: Расскажите об истории создания Haskell.

Саймон Пейтон-Джонс: Самое необычное в языке Haskell – он является успешным языком, созданным комитетом. Чаще успешные языки первоначально создаются одним человеком или очень маленькой группой. Haskell в этом смысле отличается от других языков: он изначально создавался международной группой из двадцати исследователей. Мы нашли множество точек соприкосновения в области основных принципов построения языка – а Haskell очень принципиальный язык – и тем самым обеспечения его согласованности.

Кроме того, рост популярности Haskell начался спустя почти 20 лет после того, как он был спроектирован. Обычно языки переживают взлет или падение (что случается гораздо чаще) в первые несколько лет своего существования. Что же произошло с Haskell? Я полагаю, что это является следствием принципиальной приверженности Haskell чистоте, отсутствию побочных эффектов – незнакомой дисциплины, препятствовавшей распространению Haskell. Эти долгосрочные выгоды становятся все более очевидными. Я думаю, что

любые языки, которые появятся в будущем, независимо от того, будут они похожи на Haskell или нет, будут иметь мощные механизмы управления побочными эффектами.

Брюс Тейт: Что больше всего нравится вам в этом языке?

Саймон Пейтон-Джонс: Помимо чистоты, самой интересной особенностью Haskell является, пожалуй, его система типов. Проверка статических типов является самым широко используемым приемом проверки программ из доступных на сегодняшний день: миллионы программистов ежедневно создают собственные типы (которые являются лишь частичными спецификациями), а компиляторы проверяют их при каждой компиляции. Типы – это унифицированный язык моделирования (Unified Modeling Language, UML) в функциональном программировании: язык проектирования, формирующий постоянную часть программы.

С первого дня система типов в языке Haskell отличалась необычной выразительностью, в основном благодаря классам типов и переменных типов высшего порядка (higher-kinded type). С тех пор Haskell превратился в лабораторию по исследованию новых систем типов, чем мне особенно нравится заниматься. Мультипараметрические классы типов, параметрический полиморфизм высшего ранга, полиморфизм первого рода, неявные параметры, GADT и семейства типов... все это мне очень нравится! Что еще более важно, мы расширили набор свойств, которые могут быть статически проверены системой типов.

Брюс Тейт: Что бы вы изменили в языке, будь у вас возможность вернуться назад?

Саймон Пейтон-Джонс: Я бы улучшил систему записей. Существуют определенные причины, объясняющие ее простоту, но все же она остается слабым местом языка.

Также я усовершенствовал бы систему модулей. В частности, я бы предпочел иметь возможность доставки пакета P на языке Haskell кому бы то ни было, просто сказав: «P должен импортировать интерфейсы I и J: вам следует предоставить их, а он, в свою очередь, предоставит вам интерфейс K». В Haskell отсутствует формальный способ выразить это.

Брюс Тейт: С каким из самых интересных применений Haskell вам приходилось сталкиваться?

Саймон Пейтон-Джонс: Haskell – по-настоящему универсальный язык программирования, что является его силой и слабостью одновременно, потому что он не имеет какой-то определенной области

применения, где его возможности проявлялись бы особенно ярко. И вместе с тем Haskell – это среда, в которой многие сумели найти изящные и необычные решения своих задач. Взгляните хотя бы на работу Конал Элиот (Conal Elliot) в области создания анимационных эффектов, которая заставила меня взглянуть на «значения, изменяющиеся во времени» с другой стороны как на единственно доступные для управления в функциональной программе[1]. Существует также масса библиотек парсеров и комбинаторов форматированного вывода, каждый из которых заключает любопытные решения, скрывая их за простыми интерфейсами. И наконец, Жан-Марк Эбер (Jean-Marc Eber) продемонстрировал мне библиотеку комбинаторов для описания финансовых задач, с которыми я никогда не сталкивался в своей практике.

В каждом из упомянутых случаев среда программирования (Haskell) обеспечила более высокий уровень выразительности, которого трудно было бы достичь в обычных, императивных языках.

Итак, вы получили достаточно знаний, чтобы с их помощью решать довольно сложные задачи на языке Haskell, но вы пока не знакомы с некоторыми простыми особенностями, такими как ввод/вывод, управление состоянием и обработка ошибок. Для их освоения нам потребуются дополнительные теоретические изыскания. Поэтому в третий день мы займемся изучением монад.

Что мы узнали во второй день

Сегодня мы рассмотрели функции высшего порядка. Мы начали с тех же стандартных функций для работы со списками, которые уже знакомы нам по другим языкам, представленным в этой книге. Мы увидели функцию `map`, несколько версий функции `fold`, а также некоторые дополнительные функции, такие как `zip` и `zipWith`. После опробования их с фиксированными списками мы перешли к исследованию «ленивых» последовательностей, подобных тем, с которыми мы встречались в языке Clojure.

В процессе знакомства с функциями мы узнали, как можно определять некоторые из их параметров. Этот прием называется *частичным применением функций*. Затем мы использовали частично примененные функции для преобразования функций, которые выглядят как

---

[1] http://conal.net/papers/icfp97/. – *Прим. перев.*

функции с несколькими параметрами (`f (x, y)`), в функции, принимающие аргументы по одному (`f(x)(y)`). Мы узнали также, что все функции в языке Haskell являются каррированными, что, в свою очередь, объясняет, почему так необычно выглядят в языке Haskell сигнатуры функций с несколькими параметрами. Например, функция `f x y = x + y` имеет сигнатуру `f :: (Num a) => a -> a -> a`.

Мы также познакомились с композицией функций – приемом, используемым для передачи возвращаемого значения одной функции на вход другой функции. Благодаря ему мы смогли эффективно составить цепочку функций.

Наконец, мы исследовали отложенные вычисления. Мы смогли определить функции, создающие бесконечные списки, и извлекать из этих списков только нужные нам фрагменты. Мы определили последовательность чисел Фибоначчи и использовали механизмы композиции и доступа к «ленивым» последовательностям для создания новой «ленивой» последовательности.

День 2: задания для самостоятельного решения

Найдите:

- функции для работы со списками, строками и кортежами;
- способ сортировки списков.

Практические задания:

- напишите функцию сортировки, принимающую исходный список и возвращающую новый, отсортированный список;
- напишите функцию сортировки, возвращающую новый, отсортированный список, которая принимала бы исходный список и функцию для сравнения двух элементов списка;
- напишите функцию преобразования строки в число. Функция должна принимать строки в формате `$2,345,678.99`, которые могут содержать ведущие нули;
- напишите функцию, принимающую аргумент `x` и возвращающую «ленивую» последовательность, в которой каждое третье число начинается с `x`. Затем напишите функцию, возвращающую «ленивую» последовательность, в которой каждое пятое число начинается с `y`. Объедините эти функции с помощью приема композиции, чтобы получить «ленивую» последовательность, в которой каждое восьмое число начинается с `x + y`;
- с помощью частично примененной функции определите одну функцию, возвращающую половину числа, и другую, добавляющую `\n` в конец произвольной строки.

Для более амбициозных читателей ниже предлагаются более сложные задания:

- напишите функцию, определяющую наибольший общий делитель для двух целых чисел;
- создайте «ленивую» последовательность простых чисел;
- напишите функцию, разбивающую длинный текст на отдельные строки (вставкой символа `\n`) по границам слов;
- добавьте в функцию из предыдущего упражнения нумерацию строк;
- в предыдущее упражнение добавьте функции для выравнивания текста по левому краю, по правому краю, по центру, добавлением пробелов.

8.4. День 3: Слияние разумов

В фильме «Звездный путь» Спок обладал особым даром, который он называл *слиянием разумов*. Поклонники Haskell часто утверждают о похожей связи со своим языком. У многих из них особое уважение вызывает система типов в языке. Потратив достаточно много времени на эксперименты с языком, я понял, в чем это выражается. Система типов обладает достаточной гибкостью и богатством возможностей, чтобы понять мои намерения и избавить меня от лишних хлопот. Как приятное дополнение я получаю в награду автоматическую проверку типов при создании своих функций, что особенно важно при использовании абстракций.

Классы и типы

Система типов является одной из самых привлекательных особенностей Haskell. Она обеспечивает автоматический вывод типов, избавляя программистов от тяжелых обязанностей. Она также является достаточно надежной, чтобы обнаруживать даже самые тонкие ошибки. Она поддерживает полиморфизм, то есть позволяет рассматривать различные формы одного типа как одно и то же. В этом разделе мы рассмотрим несколько примеров типов и определим несколько собственных типов.

Простые типы

Давайте обернемся назад и вспомним, что мы уже знаем о простых типах. Прежде всего включите флаг вывода типов в интерактивной оболочке:

```
Prelude> :set +t
```

Теперь мы будем видеть, какие типы возвращают вводимые нами инструкции. Попробуем ввести несколько символов и строк:

```
Prelude> 'c'
'c'
it :: Char
Prelude> "abc"
"abc"
it :: [Char]
Prelude> ['a', 'b', 'c']
"abc"
it :: [Char]
```

Слово `it` здесь обозначает введенное выражение, его можно читать как «это», а пару двоеточий (`::`) следует читать как «имеет тип». Символ в языке Haskell является значением простого типа. Строка – это массив символов, при этом не важно, как будет представлена строка в программе, в виде массива символов в квадратных скобках или как последовательность в двойных кавычках. В обоих случаях получается одно и то же значение:

```
Prelude> "abc" == ['a', 'b', 'c']
True
```

Существуют и другие простые типы:

```
Prelude> True
True
it :: Bool
Prelude> False
False
it :: Bool
```

По мере углубления в систему типов эти результаты помогут нам увидеть, что получается в действительности. Давайте определим несколько собственных типов.

Пользовательские типы

Пользовательские типы данных определяются с помощью ключевого слова `data`. В простейшем случае объявление типа содержит конечное число значений. Например, тип `Boolean` определяется примерно так:

```
data Boolean = True | False
```

Это означает, что элемент данных типа `Boolean` может иметь единственное значение, `True` или `False`. Пользовательские типы объявляются

точно так же. Представьте уменьшенную колоду игральных карт, содержащую карты двух мастей и пяти достоинств:

haskell/cards.hs
http://media.pragprog.com/titles/btlang/code/haskell/cards.hs

```
module Main where
    data Suit = Spades | Hearts
    data Rank = Ten | Jack | Queen | King | Ace
```

В этом примере `Suit` и `Rank` – это *конструкторы типов*. Здесь мы определили новые пользовательские типы с помощью ключевого слова `data`. Этот модуль можно загрузить, как показано ниже:

```
*Main> :load cards.hs
[1 of 1] Compiling Main             ( cards.hs, interpreted )
Ok, modules loaded: Main.
*Main> Hearts

<interactive>:1:0:
    No instance for (Show Suit)
      arising from a use of 'print' at <interactive>:1:0-5
```

Упс! Что не так? В сообщении говорится, что интерактивная оболочка не знает, как вывести запрошенное значение. Самое простое решение этой проблемы – унаследовать функцию `show` в объявлениях своих типов:

haskell/cards-with-show.hs
http://media.pragprog.com/titles/btlang/code/haskell/cards-with-show.hs

```
module Main where
    data Suit = Spades | Hearts deriving (Show)
    data Rank = Ten | Jack | Queen | King | Ace deriving (Show)
    type Card = (Rank, Suit)
    type Hand = [Card]
```

Обратите, что здесь мы добавили несколько псевдонимов типов. Тип `Card` – это кортеж со значениями типа `Rank` и `Suit`, а тип `Hand` – это список кортежей типа `Card`. Эти типы можно теперь использовать для создания новых функций:

```
value :: Rank -> Integer
value Ten = 1
value Jack = 2
value Queen = 3
value King = 4
value Ace = 5

cardValue :: Card -> Integer
cardValue (rank, suit) = value rank
```

Для реализации любой игры в карты необходима возможность присваивать достоинства картам. Это легко. Масть (`Suit`) в действительности не играет роли. Мы просто определили функцию, вычисляющую значение достоинства (`Rank`) карты, и еще одну – вычисляющую `cardValue`. Следующий пример демонстрирует применение этих функций:

```
*Main> :load cards-with-show.hs
[1 of 1] Compiling Main         ( cards-with-show.hs, interpreted )
Ok, modules loaded: Main.
*Main> cardValue (Ten, Hearts)
1
```

Здесь мы работаем со сложным кортежем пользовательских типов. Система типов понимает, что мы хотим выразить, что упрощает наши рассуждения.

Функции и полиморфизм

Выше вы имели возможность видеть несколько типов функций. Давайте теперь рассмотрим простую функцию:

```
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

Мы могли бы добавить в нее тип, чтобы она выглядела так:

```
backwards :: Hand -> Hand
...
```

Однако такая функция `backwards` сможет работать только со списком одного типа – списком карт. В действительности же нам необходимо следующее:

```
backwards :: [a] -> [a]
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

В данном случае мы получили полиморфную функцию. Конструкция `[a]` означает, что мы можем использовать списки любых типов. Это означает также, что, получая список значений некоторого типа `a`, функция возвращает список значений того же типа `a`. С помощью конструкции `[a] -> [a]` мы определили шаблон типов входных и выходных значений для нашей функции. То есть мы сообщили компилятору, что если на вход функции передать, например, список целых чисел, она должна вернуть список целых чисел. Теперь Haskell имеет достаточно информации, чтобы понять наши намерения.

Давайте сконструируем полиморфный тип данных. Ниже приводится определение типа трехэлементного кортежа, содержащего элементы одного типа:

haskell/triplet.hs
http://media.pragprog.com/titles/btlang/code/haskell/triplet.hs

```
module Main where
    data Triplet a = Trio a a a deriving (Show)
```

Слева мы имеем `data Triplet a`. В данном случае `a` – это переменный тип. Итак, теперь любой трехэлементный кортеж, содержащий элементы одного типа, будет интерпретироваться как значение типа `Triplet a`. Взгляните:

```
*Main> :load triplet.hs
[1 of 1] Compiling Main             ( triplet.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t Trio 'a' 'b' 'c'
Trio 'a' 'b' 'c' :: Triplet Char
```

Для создания кортежа я использовал *конструктор данных* `Trio`. Подробнее об этих конструкторах мы поговорим в следующем разделе. Опираясь на наше объявление типа, Haskell определит, что кортеж имеет тип `Triplet a`, точнее `Triplet Char`, сможет передать его любой функции, ожидающей получить аргумент типа `Triplet a`. Мы фактически сконструировали шаблон типа, описывающий любые трехэлементные кортежи, содержащие значения одного типа.

Рекурсивные типы

Существует также возможность определять рекурсивные типы. Например, представьте дерево. Вообще говоря, существует множество разновидностей деревьев, но в нашем дереве значения могут храниться только в листах. Узел в таком дереве может быть либо листом, либо списком деревьев. Описать подобное дерево можно следующим образом:

haskell/tree.hs
http://media.pragprog.com/titles/btlang/code/haskell/tree.hs

```
module Main where
    data Tree a = Children [Tree a] | Leaf a deriving (Show)
```

Итак, у нас есть один конструктор типа, `Tree`. У нас также имеется два конструктора данных, `Children` и `Leaf`. С их помощью мы можем сконструировать дерево:

```
Prelude> :load tree.hs
[1 of 1] Compiling Main             ( tree.hs, interpreted )
Ok, modules loaded: Main.
*Main> let leaf = Leaf 1
*Main> leaf
Leaf 1
```

Для начала мы создали дерево, состоящее из единственного листа. Новый лист мы присвоили переменной. Единственная задача конструктора данных `Leaf` состоит в том, чтобы сохранить значение вместе с типом. Доступ к каждой части сохраненной информации можно получить с помощью сопоставления с образцом:

```
*Main> let (Leaf value) = leaf
*Main> value
1
```

Сконструируем дерево посложнее.

```
*Main> Children[Leaf 1, Leaf 2]
Children [Leaf 1,Leaf 2]
*Main> let tree = Children[Leaf 1, Children [Leaf 2, Leaf 3]]
*Main> tree
Children [Leaf 1,Children [Leaf 2,Leaf 3]]
```

Мы построили дерево с двумя дочерними узлами, каждый из которых является листом. Затем мы построили дерево с двумя узлами – листом и правым деревом. И снова для получения информации о каждом узле можно использовать сопоставление с образцом. Мы можем построить еще более сложные деревья. Определение является рекурсивным, поэтому мы можем погружаться сколь угодно глубоко с помощью `let` и сопоставления с образцом.

```
*Main> let (Children ch) = tree
*Main> ch
[Leaf 1,Children [Leaf 2,Leaf 3]]
*Main> let (fst:tail) = ch
*Main> fst
Leaf 1
```

Намерения архитектора системы типов очевидны, и мы имеем возможность извлекать информацию о любых фрагментах. Вне всяких сомнений, такой подход влечет за собой накладные расходы, но по мере погружения в абстракции иногда дополнительные накладные расходы обходятся дешевле борьбы со все возрастающей сложностью.

Система типов в языке Haskell позволяет присоединять функции к каждому конкретному конструктору типа. Давайте рассмотрим функцию, определяющую глубину дерева (или высоту, если хотите):

```
depth (Leaf _) = 1
depth (Children c) = 1 + maximum (map depth c)
```

Первый образец в нашей функции прост. Если это лист, глубина дерева равна единице, независимо от содержимого листа.

Следующий образец немного сложнее. Если `depth` вызывается для дочернего узла типа `Children`, мы добавляем единицу к выражению `maximum (map depth c)`. Функция `maximum` находит максимальный элемент в массиве, а вызов `map depth c` возвращает список глубин всех дочерних узлов. В данном случае можно наблюдать, как используются конструкторы данных в помощь сопоставления с элементами структуры данных.

Классы

К настоящему моменту мы рассмотрели пару примеров использования системы типов. Мы создали конструкторы пользовательских типов и получили шаблоны, позволяющие определять типы данных и объявлять функции для работы с ними. Однако в языке Haskell имеется еще одно очень важное понятие, связанное с системой типов. Это понятие называется *класс*, но будьте внимательны. Это не объектно-ориентированный класс – классы вообще никак не связаны с данными. Классы в языке Haskell дают возможность контролировать полиморфизм и перегрузку.

Например, в Haskell нельзя сложить два логических значения, но можно сложить два числа. Подобные ограничения реализуются с помощью классов. В частности, *класс определяет, какие операции могут применяться к исходным данным*. Классы в Haskell чем-то сродни протоколам в Clojure.

Вот как они действуют. Класс определяет сигнатуры некоторых функций. Тип считается экземпляром класса, если поддерживает все его функции. Например, в стандартной библиотеке Haskell существует класс с именем `Eq`.

Вот как выглядит его определение:

```
class Eq a where
   (==), (/=) :: a -> a -> Bool

       -- Minimal complete definition:
       -- (==) or (/=)
   x /= y = not (x == y)
   x == y = not (x /= y)
```

Итак, тип является экземпляром класса `Eq`, если он поддерживает функции == и /=. Вы можете также определить шаблонные реали-

зации. Кроме того, если экземпляр определяет одну из упомянутых функций, другая будет добавлена автоматически.

Классы поддерживают наследование, которое действует в соответствии с с нашими представлениями. Например, класс `Num` имеет подклассы `Fractional` и `Real`. Иерархия наиболее важных классов в Haskell 98 представлена на рис. 8.1. Запомните, экземплярами классов являются типы, а не объекты данных!

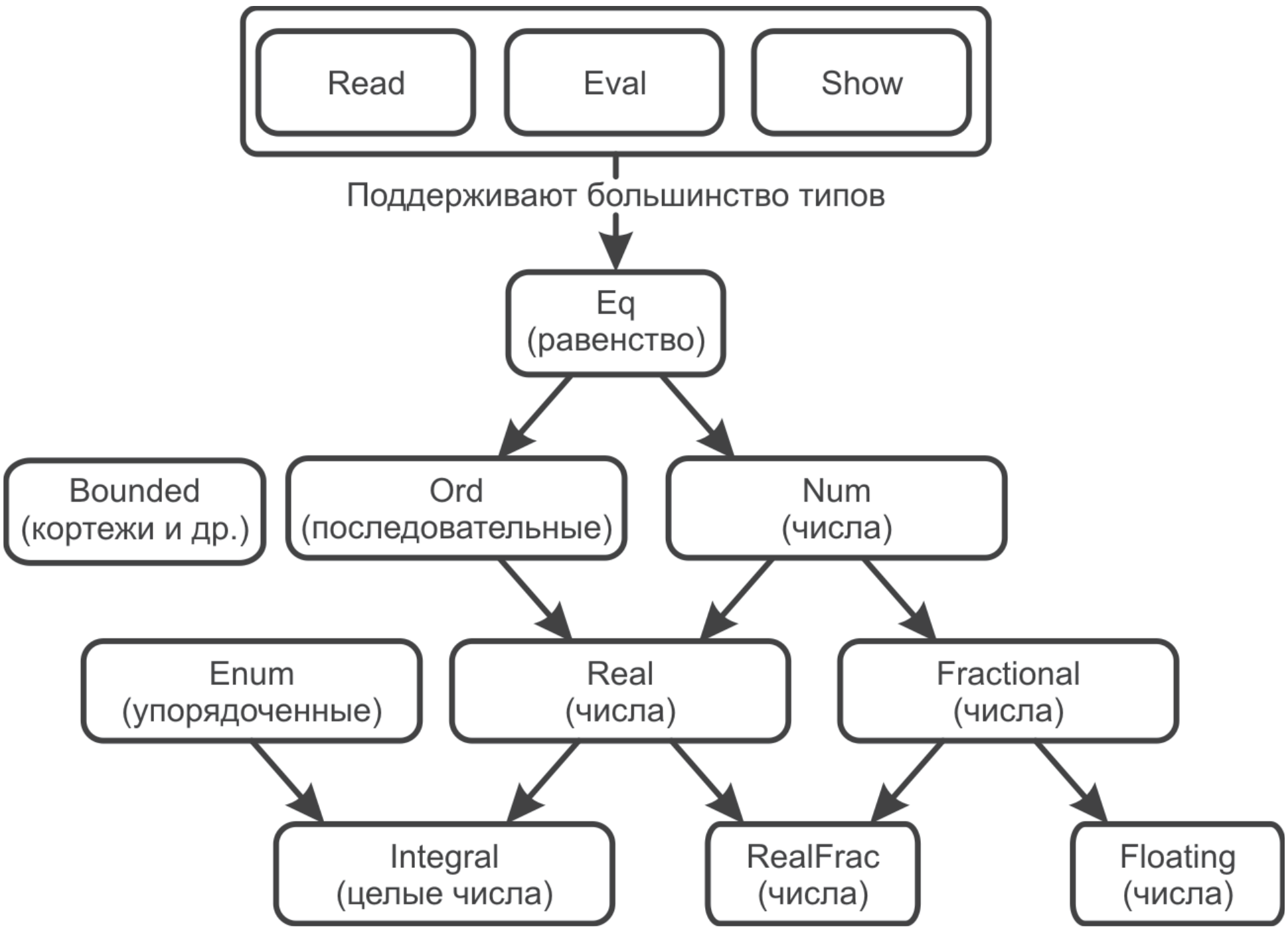

Рис. 8.1 ❖ Важнейшие классы в языке Haskell

Монады

В то время когда я решил написать эту книгу, я страшился раздела о монадах. Однако спустя некоторое время, потраченное на изучение этой темы, я понял, что в ней нет ничего особенно сложного. Сначала я дам вам простое и понятное описание задач, при решении которых могут пригодиться монады. А затем мы познакомимся с высокоуровневым описанием принципов создания монад. В заключение я покажу вам некоторый синтаксический сахар, который поможет вам понять, как они работают.

Прийти к пониманию рассматриваемого предмета мне помогла пара руководств: вики-страница, посвященная языку Haskell[1], содержащая несколько замечательных примеров, а также страница «Understanding Monads»[2], где приводится несколько отличных практических примеров. Но, возможно, вам понадобится разобрать еще множество примеров из других источников, чтобы прийти к полному пониманию монад.

Задача: пьяный пират

Допустим, что некоторый пират ищет сокровища по имеющейся у него карте. Пират пьян, он становится в известную ему точку, выбирает требуемое направление и начинает двигаться к сокровищу, шатаясь и падая. Он делает два шатающихся шага, падает, преодолевает один шаг ползком и опять поднимается. В императивном языке вам пришлось бы выполнить последовательность инструкций, как показано ниже, где `v` – значение, в котором хранится расстояние, пройденное от начальной точки:

```
def treasure_map(v)
    v = stagger(v)
    v = crawl(v)
    return( v)
end
```

Внутри `treasure_map` нам пришлось последовательно вызвать несколько функций, изменяющих состояние – пройденное расстояние. Проблема в том, что здесь состояние является изменяемым. Ту же задачу можно было бы решить функциональным способом:

haskell/drunken-pirate.hs
http://media.pragprog.com/titles/btlang/code/haskell/drunken-pirate.hs

```
module Main where

    stagger :: (Num t) => t -> t
    stagger d = d + 2
    crawl d = d +1

    treasureMap d =
```

---

1 http://www.haskell.org/tutorial/monads.html.
2 http://en.wikibooks.org/wiki/Haskell/Understanding_monads (похожее руководство на русском языке можно найти по адресу: http://ru.wikibooks.org/wiki/Haskell/Модули_и_монады – *Прим. перев.*).

```
crawl (
stagger (
stagger d))
```

Как видите, функциональное определение воспринимается сложнее. Вместо привычного `stagger`, `stagger`, `crawl` (шаг, шаг, перемещение ползком) нам приходится читать `crawl`, `stagger`, `stagger`, да и положение аргументов кажется неудобным. Вместо этого хотелось бы получить возможность составлять цепочки из последовательных вызовов функций. Мы могли бы использовать выражение `let`:

```
letTreasureMap (v, d) = let d1 = stagger d
                            d2 = stagger d1
                            d3 = crawl d2
                        in d3
```

Язык Haskell позволяет объединять выражения `let` и возвращать окончательное значение с помощью инструкции `in`. Однако эта версия выглядит почти так же неудовлетворительно, как и первая. Входные и выходные данные остаются теми же самыми, поэтому должен быть более простой путь композиции функций такого рода. Итак, нам требуется преобразовать `stagger(crawl(x))` в `stagger(x) • crawl(x)`, где символ • является функцией композиции. Это и есть монада.

Проще говоря, монады позволяют составлять композиции из функций, обладающих определенными свойствами. В языке Haskell монады используются в нескольких случаях. Во-первых, для осуществления ввода/вывода, который довольно сложно выразить в чисто функциональных языках, – функция всегда должна возвращать один и тот же результат при одних и тех же входных значениях, но в случае с вводом/выводом функция должна изменять некоторое состояние, например содержимое файла.

Кроме того, реализация решения задачи о пьяном пирате, представленная выше, опирается на сохранение промежуточных состояний. Монады дают возможность имитировать состояние программы. В языке Haskell имеется специальный синтаксис, который называется *do-синтаксис*, позволяющий программам действовать в императивном стиле. Do-синтаксис опирается на монады.

Наконец, обработка таких, казалось бы, простых вещей, как ошибки, оказывается весьма сложной в реализации, потому что тип возвращаемого значения во многих случаях оказывается в зависимости от успеха функции. Для этой цели в языке Haskell предусмотрена монада `Maybe`. Давайте заглянем немного глубже.

Компоненты монад

На самом простом уровне монада состоит из трех компонентов:

- конструктор типа, основанный на некотором типе контейнера. Роль контейнера может выполнять простая переменная, список или что-то еще, способное хранить значение. Мы будем использовать контейнер для хранения функции. Выбор контейнера будет отличаться в зависимости от того, что должна делать монада;
- функция с именем `return`, обертывающая другую функцию и помещающая ее в контейнер. Смысл такого имени станет вам понятен чуть ниже, когда мы перейдем к знакомству с формой записи `do`. А пока запомните, что `return` заворачивает функцию в монаду;
- функция связывания с именем `>>=`, которая разворачивает обернутую функцию. Мы будем использовать функцию связывания для составления цепочек из вызовов функций.

Все монады должны соответствовать трем правилам, перечисленным ниже. Для некоторой монады `m`, некоторой функции `f` и некоторого значения `x`:

- должна иметься возможность использовать конструктор типа для создания монады, которая способна хранить значение этого типа;
- должна иметься возможность заворачивать и разворачивать значения без потери информации (`monad >>= return = monad`);
- вложенные функции связывания должны выполняться в том же порядке, как и при последовательном вызове (`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`).

Мы не будем тратить много времени на обсуждение этих правил, однако отмечу, что причины, стоящие за ними, достаточно просты. Они обеспечивают множество полезных трансформаций без потери информации. Желающие вникнуть в их суть могут воспользоваться указанными выше ссылками.

Но достаточно теории. Давайте создадим простую монаду. Мы создадим одну монаду с самого начала, а затем я завершу главу знакомством с некоторыми полезными монадами.

Создание монады с нуля

Первое, что нам нужно, – это конструктор типа. Наша монада будет хранить функцию и значение:

haskell/drunken-monad.hs
http://media.pragprog.com/titles/btlang/code/haskell/drunken-monad.hs

```
module Main where
    data Position t = Position t deriving (Show)

    stagger (Position d) = Position (d + 2)
    crawl (Position d) = Position (d + 1)

    rtn x = x
    x >>== f = fx
```

Здесь присутствуют все три обязательных компонента монады: контейнер типа, функция `return` и функция связывания. Наша монада – простейшая из возможных. Контейнер типа – это простой конструктор типа, который выглядит как `data Position t = Position t`. Он всего лишь определяет базовый тип, основанный на произвольном шаблоне типа. Далее, нам нужна функция `return`, заворачивающая функцию в значение. Поскольку наша монада самая простая из возможных, нам достаточно вернуть значение самой монады, обернув его соответственно: `(rtn x = x)`. Наконец, нам нужна функция связывания, обеспечивающая композицию функций. У нас она называется `>>==`, и мы определили ее как вызов связанной функции со значением монады (`x >>== f = f x`). Вместо `>>=` и `return` мы использовали имена `>>==` и `rtn`, чтобы предотвратить конфликты со встроенными функциями монад в языке Haskell.

Обратите внимание, что мы также переписали функции `stagger` и `crawl`, и теперь они используют нашу доморощенную монаду вместо обычных целых чисел. А теперь проведем тест-драйв монады. Напомню, что мы остановились на синтаксисе преобразования вложенных вызовов в композицию. Преобразованное передвижение к сокровищам теперь выглядит так:

```
treasureMap pos = pos >>==
                  stagger >>==
                  stagger >>==
                  crawl >>==
                  rtn
```

Оно действует в точности так, как ожидалось:

```
*Main> treasureMap (Position 0)
Position5
```

Монады и форма записи do

Этот синтаксис выглядит намного лучше, но, как вы уже наверняка догадались, с добавлением небольшого количества синтаксического

сахара он может выглядеть еще лучше. Таким синтаксическим сахаром в языке Haskell является синтаксис `do`. Этот синтаксис особенно удобен для решения таких задач, как организация ввода/вывода. Следующий фрагмент читает строку с консоли и выводит в перевернутом виде с использованием `do`-синтаксиса:

haskell/io.hs
http://media.pragprog.com/titles/btlang/code/haskell/io.hs

```
module Main where
    tryIo = do putStr "Enter your name: " ;
               line <- getLine ;
               let { backwards = reverse line } ;
               return ("Hello. Your name backwards is " ++ backwards)
```

Обратите внимание, что эта программа начинается с объявления функции. Затем используется простая форма записи `do`, которая является синтаксическим сахаром для монад. Это придало программе императивный вид, но в действительности она основана на использовании монады. А теперь остановимся на некоторых синтаксических правилах, знание которых вам обязательно пригодится.

Присваивание выполняется с помощью оператора `<-`. При использовании интерпретатора GHCI строки должны отделяться точками с запятой, а выражения `let` в пределах выражений `do` должны заключаться в фигурные скобки. Если имеется множество строк, их следует заключить в скобки `:{` и `}:`, каждая из которых должна находиться в отдельной строке. Теперь вы можете наконец понять, почему мы дали обертывающей функции имя `return`. Она упаковывает возвращаемое значение в компактную форму, которую может воспринять инструкция `do`. Этот код действует, как если бы он был написан на императивном языке, поддерживающем изменяемые состояния, но в нем для поддержки состояния используется монада. Все операции ввода/вывода должны выполняться с использованием одной из соответствующих монад в блоке `do`.

Различные вычислительные стратегии

Каждая монада имеет ассоциированную с ней вычислительную стратегию. Монада идентичности, использованная в примере с пьяным пиратом, просто возвращает все, что было помещено в нее. Мы использовали ее для преобразования вложенной структуры программы в последовательную. Давайте рассмотрим другой пример. Как это ни странно, список тоже является монадой с функцией `return` и функцией связывания (`>>=`), которые определены, как показано ниже:

```
instance Monad [] where
    m >>= f = concatMap f m
    return x = [x]
```

Напомню, что обязательными компонентами любой монады является некоторый контейнер, конструктор типа, метод `return`, заворачивающий функцию, и метод связывания, разворачивающий ее. `Monad` – это класс, а конструкция `[]` создает ее экземпляр, давая нам наш конструктор типа. Далее следует функция, обертывающая возвращаемый результат.

В случае со списками функция заворачивается в список. Чтобы развернуть ее, наша функция связывания вызывает указанную функцию для каждого элемента списка с помощью `map` и затем объединяет результаты. Последовательность вызовов функций `concat` и `map` используется настолько часто, что была создана функция `concatMap`, вызывающая обе эти функции, однако с тем же успехом мы могли бы использовать конструкцию `concat (map f m)`.

Чтобы вы могли получить представление, как действует монада списка, ниже приводится пример ее использования с применением формы записи `do`:

```
Main> let cartesian (xs,ys) = do x <- xs; y <- ys; return (x,y)
Main> cartesian ([1..2], [3..4])
[(1,3),(1,4),(2,3),(2,4)]
```

Здесь определяется простая функция с применением `do`-синтаксиса и монад. Она извлекает `x` из списка `xs` и `y` из списка `xy` и возвращает все комбинации `x` и `y`. Используя такой прием, легко можно реализовать подбор пароля.

haskell/password.hs

http://media.pragprog.com/titles/btlang/code/haskell/password.hs

```
module Main where
    crack = do x <- ['a'..'c' ] ; y <- ['a'..'c' ] ; z <- ['a'..'c' ];
               let { password = [x, y, z] } ;
               if attempt password
                    then return (password, True)
                    else return (password, False)

attempt pw = if pw == "cab" then True else False
```

Здесь мы использовали монаду списка для поиска всех возможных комбинаций. Обратите внимание, что в этом контексте конструкция `x <- [lst]` означает: «для каждого `x` из списка `[lst]`». Всю тяжелую

работу мы переложили на Haskell, а нам остается только опробовать каждый пароль. Искомый пароль в этом примере жестко «зашит» в функцию `attempt`. Эту задачу можно решить с применением самых разных вычислительных стратегий, таких как применение генераторов списков, но данный пример демонстрирует применение вычислительной стратегии на основе монад списков.

Монада Maybe

Итак, мы с вами увидели две разновидности монад: монады идентичности и монады списков. На примере последней мы узнали, что монады поддерживают вычислительную стратегию. В этом разделе мы займемся исследованием монады `Maybe` и задействуем ее для решения типичной задачи программирования: обработки ошибок, возникающих в функциях. Кому-то может показаться, что речь пойдет о царстве баз данных и удаленных взаимодействий, однако в реальной жизни существуют куда более простые API, нуждающиеся в поддержке ошибочных ситуаций. Представьте функцию поиска подстроки в строке, возвращающую индекс найденного совпадения. Если искомая подстрока присутствует в строке, функция возвращает значение типа `Integer`. В противном случае – значение типа `Nothing`.

Связать такие результаты бывает очень непросто. Допустим, что у нас имеется функция, выполняющая парсинг веб-страницы. Нам требуется извлечь раздел `body` из этой страницы, а затем – первый абзац из этого раздела. Типичное решение заключается в том, чтобы определить функции со следующими сигнатурами:

```
paragraph XmlDoc -> XmlDoc
...
body XmlDoc -> XmlDoc
...
html XmlDoc -> XmlDoc
...
```

и использовать их следующим образом:

```
paragraph body (html doc)
```

Проблема состоит в том, что функции `paragraph`, `body` и `html` могут терпеть неудачу, поэтому нам нужно предусмотреть поддержку типа `Nothing`. В языке Haskell имеется такой тип, который называется `Just`. Конструкция `Just x` может служить оберткой для типа `Nothing` или любого другого:

```
Prelude> Just "some string"
Just "some string"
Prelude> Just Nothing
Just Nothing
```

Получить фактический тип Just можно с помощью сопоставления с образцом. Итак, вернемся к нашему примеру. Функции paragraph, body и html могли бы возвращать свои результаты в виде Just XmlDoc. В этом случае мы сможем воспользоваться инструкцией case (которая работает точно так же, как инструкция case в Erlang) и реализовать нечто подобное:

```
case (html doc) of
  Nothing -> Nothing
  Just x -> case body x of
              Nothing -> Nothing
              Just y -> paragraph 2 y
```

Но такая реализация весьма далека от идеала, особенно если сравнить ее с желаемой paragraph 2 body (html doc). Здесь нам пригодится монада Maybe, определение которой приводится ниже:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return          = Just
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x

    ...
```

Тип, который здесь обертывается, – это тип Maybe a. Данный тип может обертывать Nothing или Just a.

Функция return имеет очень простой вид. Она всего лишь заворачивает результат в Just. Функция связывания также имеет простую реализацию. Для значения типа Nothing она возвращает функцию, которая, в свою очередь, возвращает Nothing. Для Just x она возвращает функцию, которая, в свою очередь, возвращает x. Функция return сможет завернуть все, что угодно. Теперь вы легко сможете построить цепочку операций:

```
Just someWebPage >>= html >>= body >>= paragraph >>= return
```

Как видите, мы смогли добиться безупречной компоновки элементов. Нам удалось это благодаря тому, что монада принимает решение с помощью наших функций.

Что мы узнали в третий день

Сегодня мы познакомились с тремя важными концепциями: типы, классы и монады в языке Haskell. Мы начали с типов, посмотрев, как выводятся типы существующих функций, чисел, логических значений и символов. Затем мы познакомились с возможностью определения пользовательских типов. Для примера мы определили тип, представляющий игральные карты, имеющие масть и достоинство. Мы узнали, как параметризовать типы, и даже использовали рекурсивные определения типов.

Обсуждение языка мы завершили знакомством с монадами. Так как Haskell является исключительно функциональным языком, на нем сложно выразить решение задач в императивном стиле или с использованием приема накопления состояния. Обе эти проблемы создатели Haskell предлагают решать с помощью монад. Монада – это конструктор типа с несколькими функциями, осуществляющими обертывание функций и составление цепочек из них. Монады можно комбинировать с контейнерами разных типов, чтобы обеспечить различные виды вычислительных стратегий. Мы использовали монады, чтобы получить возможность записывать код в более естественном, императивном стиле, и для обработки различных допустимых вариантов.

День 3: задания для самостоятельного решения

Найдите:

- несколько руководств с описанием монад;
- список монад в языке Haskell.

Практические задания:

- напишите функцию поиска значения в хэше с использованием монады `Maybe`. Напишите хэш, хранящий другие хэши, с несколькими уровнями вложенности. Используйте монаду `Maybe` для извлечения элемента из хэша на произвольном уровне вложенности;
- реализуйте лабиринт на языке Haskell. Вам потребуется определить типы `Maze` (лабиринт) и `Node` (узел), а также функцию, возвращающую узел по заданным координатам. Узел должен содержать список выходов в другие узлы;
- реализуйте функцию поиска выхода из лабиринта, использующую монаду списка;

- реализуйте тип `Monade` на нефункциональном языке. (См. серию статей о реализации монад на языке Ruby[1].)

8.5. В заключение о Haskell

Из всех языков, представленных в этой книге, Haskell является единственным, который был создан целым комитетом. После появления исключительно функциональных языков, использующих семантику отложенных вычислений, был сформирован комитет для создания открытого стандарта, целью которого было объединение существующих наработок и будущих исследований. Первая версия Haskell вышла в 1990 году. С тех пор язык и его сообщество заметно выросли.

Haskell поддерживает широкий диапазон особенностей функциональной парадигмы, включая генераторы списков, отложенные вычисления, частично примененные функции и карринг. В действительности все функции в языке Haskell принимают единственный параметр, а для поддержки функций с несколькими аргументами используется карринг.

Система типов Haskell обеспечивает великолепный баланс между безопасностью и гибкостью типов. Полностью система полиморфных шаблонов дает возможность создавать весьма сложные пользовательские типы и даже классы типов, поддерживающие наследование интерфейсов. Обычно программисты на Haskell не отягощают себя тонкостями, связанными с типами, за исключением объявления функций, тем не менее система типов надежно защищает их от любых ошибок, связанных с неправильным использованием типов.

Разработчики программ на Haskell, как и на любом другом исключительно функциональном языке, вынуждены идти на определенные хитрости, когда требуется писать код в императивном стиле или обслуживать накапливаемое состояние. Ввод/вывод также может вызывать сложности. К счастью, на эти случаи в Haskell имеются монады. Монада – это конструктор типа и контейнер, а также пара функций для завертывания и развертывания функций. Контейнеры разных типов поддерживают разные вычислительные стратегии. Упомянутые функции дают программистам возможность компоновать монады

---

[1] http://moonbase.rydia.net/mental/writings/programming/monads-in-ruby/00introduction.html.

самыми необычными способами, применяя *do-синтаксис*. Это – синтаксический сахар, обеспечивающий поддержку императивного стиля программирования, хотя и с некоторыми ограничениями.

Основные сильные стороны

Поскольку в языке Haskell предпринят бескомпромиссный подход к определению функций без побочных эффектов, преимущества и недостатки языка часто могут иметь весьма гротескный вид. Давайте перечислим их.

Система типов

Если вам нравится строгий контроль типов (да даже если и не нравится), вы полюбите систему типов в Haskell. Она всегда оказывается там, где нужна, и не мешает там, где в ней нет необходимости. Система типов способна добавлять дополнительный уровень защиты от распространенных ошибок. Вы сможете выловить их уже на этапе компиляции, а не во время выполнения. Но дополнительная защита – это только часть возможностей.

Наиболее интересной, пожалуй, особенностью системы типов в Haskell является простота связывания новых типов с новым поведением. Вы можете создавать сложнейшие типы, что называется, «с нуля». С помощью конструкторов типов и классов можно изменять до неузнаваемости даже существующие типы и классы, такие как монады, практически не прилагая усилий. Благодаря классам ваши собственные типы с успехом могут использовать существующие библиотеки для Haskell.

Выразительность

Язык Haskell имеет фантастические возможности. Он обладает всем, что требуется для краткого выражения идей, включая богатую библиотеку и мощный синтаксис. Он позволяет создавать собственные типы и даже допускает рекурсивные определения типов, давая возможность связывать функции с данными без применения чрезмерного количества синтаксических конструкций. В академической среде нет более сильного языка, чем Haskell, пригодного для обучения функциональному программированию. Он содержит все, что вам необходимо.

Чистота модели программирования

Чистые, беспримесные модели программирования способны радикально изменить подходы к решению задач. Они вынуждают оста-

вить устаревшие парадигмы программирования и использовать иные пути для достижения целей. Исключительно функциональные языки дают вам надежную опору. Функции, написанные на этих языках, для одних и тех же входных данных всегда будут возвращать один и тот же результат. Эта их особенность упрощает рассуждения о программах. Опираясь на нее, порой можно уверенно доказать, что программа работает правильно. Она также избавляет от множества проблем, порождаемых побочными эффектами, такими как чрезмерное усложнение алгоритмов или снижение производительности параллельных вычислений в некоторых ситуациях.

Семантика отложенных вычислений

Когда-то давно использование функциональных языков означало использование рекурсии. Стратегии отложенных вычислений предлагают совершенно новые подходы к обработке данных. Часто с применением отложенных вычислений можно создавать более короткие и производительные программы, чем с использованием других стратегий.

Академическая поддержка

Некоторые из известных и влиятельных языков, таких как Pascal, появились и выросли в академической среде, пользуясь преимуществами исследований, проводившихся в этой среде. Как основной язык обучения приемам функционального программирования, Haskell продолжает улучшаться с каждым годом. И хотя он не получил повсеместного распространения, вы всегда сможете найти группы программистов, решающих с его помощью важные задачи.

Основные недостатки

Теперь вы знаете, что не существует идеальных языков. Сила языка Haskell имеет обратную сторону.

Негибкость модели программирования

Поддержка исключительно функциональной парадигмы программирования имеет свои преимущества, но она же влечет за собой множество проблем. Возможно, вы обратили внимание, что программирование с привлечением монад обсуждалось в последней части последней главы книги о программировании, и это не случайно. Освоение данной концепции требует определенного напряжения ума. Но обратите внимание, что мы использовали монады для выполнения та-

ких задач, как написание программ в императивном стиле, обработка ввода/вывода и анализ результатов выполнения функций, способных потерпеть неудачу, которые легче легкого решаются на других языках. Я уже говорил это при описании других языков, но я не могу не повторить свои слова здесь. Хотя Haskell и делает кое-что сложное простым, он также может усложнять простое.

Некоторые стили программирования свойственны только определенным парадигмам. Для реализации алгоритмов, решающих задачи шаг за шагом, отлично подходят императивные языки. Функциональные языки явно не подходят для создания сценариев и программ с большим объемом ввода/вывода. Чистота в глазах одного может выглядеть как неудачный компромисс в глазах другого.

Сообщество

С точки зрения компромиссов, вы сами можете видеть разницу в подходах между языками Scala и Haskell. Да, оба языка являются строго типизированными, но они проповедуют в корне отличающиеся философии. Язык Scala полон компромиссов, а Haskell, наоборот, является исключительно функциональным языком. Благодаря компромиссам Scala изначально привлек к себе гораздо более широкое внимание, чем Haskell. Разумеется, было бы неправильно измерять успех численностью сообщества, тем не менее численность должна быть достаточно высокой, чтобы обеспечить успех, и чем шире сообщество, тем больше возможностей и ресурсов.

Сложность изучения

Монады – не единственная концепция в языке Haskell, требующая умственного напряжения для ее освоения. Другой такой же сложной концепцией является карринг – прием, используемый в каждой функции, принимающей более одного аргумента. Большинство простых функций имеют параметризованные типы, а функции для работы с числами часто используют классы типов. Хотя эта цена в конечном итоге может быть и оправдана, вы должны быть сильным программистом с хорошей теоретической подготовкой, чтобы иметь шанс преуспеть в освоении Haskell.

Заключительные замечания

Из функциональных языков программирования, представленных в этой книге, Haskell является наиболее сложным в изучении. Акцент на применение монад и система типов делают дорогу изучения

крутой и тернистой. Однако, когда я овладел основными ключевыми концепциями, многое для меня упростилось, и Haskell стал одним из самых полезных языков, которые я знаю. Опираясь на систему типов и изящество применения монад, однажды вы оглянетесь на этот язык как один из наиболее важных в данной книге.

Язык Haskell играет еще одну роль. Чистота подходов и поддержка в академических кругах положительно влияют на наше понимание искусства программирования. Лучшие из будущих поколений функциональных программистов будут оттачивать свои навыки на языке Haskell.

Глава 9

Послесловие

Поздравляю, вы закончили знакомство с семью языками программирования. Не ждите, что в этой главе я назову победителей и проигравших, эта книга совсем не о том. Она просто раскрывает перед вами новые идеи. Возможно вы, как и я на раннем этапе моей карьеры, глубоко зарылись в коммерческие или промышленные проекты, разрабатываемые большими командами с небольшой свободой выбора. В то время я был весьма ограничен в выборе языков программирования, как любитель кино в 1970-е, живущий в небольшом городке с единственным кинотеатром, заказывавшим только кассовые блокбастеры.

Когда я начал писать программное обеспечение для себя самого, я чувствовал себя как киноман, обнаруживший, что существует и другое, независимое кино. У меня получилось зарабатывать себе на жизнь программированием на языке Ruby, но я не настолько наивен, чтобы считать Ruby всемогущим языком. Так же как независимое кино служит одной из движущих сил в искусстве, познание других языков программирования оказывает влияние на представления об организации программ. Давайте освежим в памяти некоторые моменты, освещавшиеся в этой книге.

9.1. Модели программирования

Модели программирования меняются очень медленно. Вы уже увидели, что новые модели появляются примерно через каждые двадцать лет, или что-то около того. Я начинал учиться программированию еще на процедурных языках, таких как Basic и Fortran. В колледже я освоил более структурированный подход, изучив язык Pascal. Поступив на работу в компанию IBM, я начал писать коммерческие программы на C и C++ и впервые познакомился с Java. Тогда же я начал писать объектно-ориентированный код. Мой опыт программирования насчитывает уже более 30 лет, но я видел только две основные парадигмы программирования. Вы можете спросить, почему я с таким воодушевлением рассказывал в этой книге о других парадигмах? Хороший вопрос!

Хотя парадигмы программирования изменяются крайне медленно, они все-таки изменяются. Подобно урагану, они могут разрушать все на своем пути, ломая карьеры и приводя к упадку компании, не задумывающиеся об инвестициях в будущее. Если вы обнаруживаете, что начинаете бороться с парадигмой программирования – это повод задуматься. Потребность в поддержке параллельных вычислений и высокой надежности кода начинает подталкивать нас в направлении высокоуровневых языков программирования. Как минимум мы начинаем замечать более специализированные языки, лучше подходящие для решения некоторых задач и представляющие иные модели программирования.

Объектно-ориентированное программирование (Ruby, Scala)

В настоящее время господствующей является объектно-ориентированная парадигма программирования, как правило, на языке Java. Эта парадигма опирается на три основные идеи: инкапсуляция, наследование и полиморфизм. Знакомясь с языком Ruby, мы исследовали динамическую типизацию. Вместо того чтобы вводить ограничения на основе определений классов и объектов, Ruby вводит ограничения, опираясь на имена методов, поддерживаемых объектами. Мы узнали также, что Ruby поддерживает некоторые функциональные концепции в виде блоков кода.

Язык Scala тоже поддерживает объектно-ориентированную парадигму. И хотя в нем используется статическая типизация, его система типов менее обременительна для программиста, в сравнении с системой типов в языке Java, из-за возможности автоматического определения типов, упрощающей синтаксис. Благодаря этой особенности Scala автоматически выводит типы переменных, опираясь на синтаксические подсказки и используемые операции. В плане поддержки функциональных концепций язык Scala ушел намного дальше языка Ruby.

Оба этих языка широко используются для создания современных промышленных приложений, и оба имеют куда более радужные перспективы, чем многие основные языки, такие как Java. Существует множество вариаций объектно-ориентированных языков, к числу которых можно также отнести языки, основанные на прототипах.

Программирование на основе прототипов (Io)

В действительности можно смело утверждать, что языки программирования на основе прототипов являются разновидностью объектно-

ориентированных языков, но они имеют ряд существенных отличий, поэтому я описывал их как представителей другой модели программирования. В этих языках вместо классов используются экземпляры объектов, являющиеся прототипами. Некоторые экземпляры играют роль прототипов для других экземпляров. К семейству этих языков относятся JavaScript и Io. Простые и выразительные языки программирования на основе прототипов обычно поддерживают динамическую типизацию, они прекрасно подходят для разработки сценариев и приложений, особенно пользовательских интерфейсов.

Как вы уже знаете, язык Io реализует простую модель программирования с компактным, непротиворечивым синтаксисом, дающим возможность составлять мощные комбинации. Мы использовали Io в самых разных контекстах, от создания параллельных программ до реализации собственного предметно-ориентированного языка (DSL). Но программирование на основе прототипов не является самой специфической парадигмой среди тех, что нам встретились в этой книге.

Логическое программирование (Prolog)

Prolog появился из семейства языков, созданных для логического программирования. С помощью Prolog мы написали несколько приложений, решающих довольно узкий круг проблем, но результаты оказались весьма любопытными. Мы определяли логические ограничения, известные нам, а Prolog находил решение.

Когда модель программирования укладывалась в эту парадигму, мы оказывались в состоянии получить результаты, написав всего несколько строк кода. Семейство языков логического программирования играет важную роль в создании приложений в таких областях, как управление движением воздушных судов и гражданское строительство. Ограниченные механизмы обработки логических правил можно найти и в других языках, таких как C и Java. В свое время язык Prolog послужил основой для разработки языка Erlang, принадлежащего другому семейству языков.

Функциональное программирование (Scala, Erlang, Clojure, Haskell)

Пожалуй, самой интересной парадигмой программирования из числа представленных в этой книге является парадигма функционального программирования. Степень чистоты функциональных языков отличается, но концепции остаются неизменными. Функциональные про-

граммы на этих языках конструируются из математических функций. Вызов одной и той же функции с одними и теми же значениями аргументов в таких языках всегда будет возвращать один и тот же результат, а побочные эффекты либо не приветствуются, либо вообще запрещены на уровне языка. Вы можете компоновать эти функции самыми разными способами.

Вы уже успели убедиться, что функциональные языки обычно более выразительны, в сравнении с объектно-ориентированными языками. Программный код на этих языках часто оказывается короче и проще, чем код на объектно-ориентированных языках, благодаря более обширному арсеналу инструментов компоновки программ. Мы познакомились с функциями высшего порядка и такими сложными концепциями, как каррирование, которые не всегда можно найти в объектно-ориентированных языках. Как вы узнали во время знакомства с языком Haskell, разные уровни чистоты языка влекут за собой разные преимущества и недостатки. Одной из важнейших побед функциональных языков является устранение побочных эффектов, что существенно упрощает программирование параллельных вычислений. С устранением изменчивого состояния исчезли и многие проблемы, традиционно сопутствующие параллельным вычислениям.

Смена парадигмы

Чтобы заняться функциональным программированием, можно пойти несколькими путями. Вы можете сразу и полностью отказаться от объектно-ориентированных приемов или выбрать менее крутой путь эволюционного развития.

В процессе знакомства с семью языками вы видели, что они охватывают этап длиной в четыре десятилетия и, как минимум, столько же парадигм программирования. Я надеюсь, что вы сможете дать свою оценку эволюции языков программирования. Вы имели возможность наблюдать три совершенно разных подхода к поддержке парадигм. В языке Scala был выбран путь мирного сосуществования. Программисты на Scala могут писать объектно-ориентированные программы, попутно используя функциональные приемы. Обе парадигмы вполне естественно уживаются в языке. В языке Clojure был избран путь совместимости. Этот язык основан на JVM, что дает возможность использовать Java-объекты в программах на Clojure, но суть философии Clojure заключается в полном отказе от некоторых приемов объектно-ориентированного программирования. В отличие от Scala, поддержка взаимодействий Clojure-Java в языке Clojure направле-

на на усиление существующих возможностей виртуальной машины Java, а не на расширение языка. Haskell и Erlang в значительной мере являются самостоятельными языками. Они вообще не поддерживают объектно-ориентированное программирование ни в какой форме. Учитывая вышесказанное, у вас есть возможность одновременно использовать обе парадигмы, полностью порвать все связи с объектно-ориентированной парадигмой или использовать приемы функционального программирования, сохранив возможность применения объектно-ориентированных библиотек. Выбирать вам.

Какой бы выбор вы не сделали, вы уже поднялись на одну ступень выше, познакомившись с разными языками. Как Java-разработчик я целых десять лет жил в замкнутом мире, в основном из-за того, что я, как и многие другие, оставался непосвященным и не слышал негромких криков. Между тем господствующие фреймворки, такие как Spring, застряли на анонимных вложенных классах из-за того, что продолжали существовать в тесном, замкнутом пространстве. Мои пальцы страдали от огромного объема кода, который им приходилось набивать, мои глаза слезились от того, что я должен был читать весь этот код. Современные Java-разработчики знают намного больше, отчасти потому, что такие, как Мартин Одерски (Martin Odersky) и Рич Хикки (Rich Hickey), дали нам альтернативы, которые выдвигают программирование на новый уровень и заставляют Java подвинуться.

9.2. Параллельные вычисления

Тема выбора более удачных языковых конструкций и моделей программирования для организации параллельных вычислений неоднократно поднималась в этой книге. Подходы в разных языках часто оказывались разительно отличными, но весьма эффективными. Давайте еще раз пройдемся по некоторым из них.

Управляемое изменение состояния

Безусловно, самой обсуждаемой темой, касающейся параллельных вычислений, в этой книге была модель программирования. Объектно-ориентированное программирование допускает наличие побочных эффектов и изменяемого состояния. В совокупности эти две особенности делают программы намного более сложными, чем они могли бы быть. Когда в этот коктейль добавляется поддержка потоков выполнения и процессов, сложность программ возрастает просто неимоверно.

Функциональные языки программирования вводят ряд важных правил. Вызовы одной и той же функции с одними и теми же значениями входных аргументов всегда будут давать один и тот же результат. Переменные могут получить свое значение только один раз. Вместе с исчезновением изменяемого состояния исчезает проблема «гонки за ресурсами» и все сопутствующие ей сложности. Однако мы видели ряд приемов, идущих дальше основной модели программирования. Давайте рассмотрим их поближе.

Акторы в Io, Erlang и Scala

Суть акторов остается неизменной, независимо от того, основаны ли они на объектах или процессах. Она заключается в трансформации неструктурированных межпроцессных взаимодействий в структурированные сообщения, передаваемые между разными компонентами посредством поддерживаемых ими очередей сообщений. Для обработки сообщений в Erlang и Scala используется механизм сопоставления с образцом. В главе 6 «Erlang» мы рассмотрели пример реализации игры в русскую рулетку, имитирующей выстрел из револьвера. Напомню, что патрон находится в третьем гнезде барабана:

erlang/roulette.erl
http://media.pragprog.com/titles/btlang/code/erlang/roulette.erl

```
-module(roulette).
-export([loop/0]).

% send a number, 1-6
loop() ->
    receive
        3 -> io:format("bang.~n" ), exit({roulette,die,at,erlang:ti
me()});
        _ -> io:format("click~n" ), loop()
end.
```

Затем мы запускали процесс, присваивая его идентификатор переменной `Gun`. Мы могли «убить» процесс, послав сообщение `Gun ! 3`. Виртуальная машина и язык Erlang поддерживают возможность мониторинга выполняющихся процессов, позволяя получать извещения и даже перезапускать процессы при первых же признаках проблем.

Отложенные задания

К модели акторов язык Io добавил две дополнительные конструкции: сопрограммы и отложенные задания. Сопрограммы позволяют двум

объектам действовать параллельно, передавая управление друг другу в подходящие моменты времени. Отложенные задания были добавлены с целью дать возможность параллельно выполнять длительные вычисления.

Мы использовали инструкцию `futureResult := URL with("http://google.com/") @fetch`. Несмотря на то что результат невозможно было получить немедленно, она сразу же возвращает управление программе, блокируя ее выполнение, только при попытке извлечь результат. Отложенные задания в Io фактически преобразуются в результат, когда он становится доступен.

Транзакционная память

Во время знакомства с языком Clojure мы видели несколько интересных способов организации параллельных вычислений. Программная транзакционная память (Software Transactional Memory, STM) заворачивает каждую попытку доступа к совместно используемому ресурсу в транзакцию. Аналогичный подход используется в базах данных, где он обеспечивает целостность данных при одновременном выполнении конкурирующих запросов. Каждую попытку доступа мы заключали в вызов функции `dosync`. Благодаря этому приему разработчики на Clojure могут преодолеть ограничения функциональной парадигмы, когда это действительно необходимо, и при этом избежать конфликтов между потоками выполнения и процессами.

STM является относительно новой идеей, которая только начинает проникать в более популярные языки. Будучи разновидностью Lisp, язык Clojure является идеальной площадкой для внедрения таких приемов, потому что Lisp является мультипарадигмальным языком. Программисты могут использовать разные парадигмы программирования, когда это имеет смысл, пребывая в уверенности, что приложение будет обеспечивать целостность и высокую производительность, даже при большом количестве одновременных попыток доступа к данным.

Следующее поколение программистов сможет получить больше от своих языков. Простейшие инструменты, позволяющие запустить поток выполнения и ждать на семафоре, не удовлетворяют современным требованиям. Новейшие языки должны будут иметь согласованную философию поддержки параллельных вычислений и соответствующие инструменты. Может случиться так, что потребность

в параллельных вычислениях сделает устаревшими целые парадигмы программирования или, напротив, старые языки будут дополнены новыми средствами управления изменяемостью переменных и более интеллектуальными конструкциями параллельного выполнения, такими как акторы и отложенные задания.

9.3. Конструкции программирования

Одним из самых захватывающих во время работы над этой книгой было исследование основных строительных блоков в различных языках. С каждым новым языком я представлял вам новые концепции и конструкции программирования, которые могут встретиться вам в других языках на вашем пути. Вот они, мои самые восхитительные открытия.

Генераторы списков

Как вы уже видели, в Erlang, Clojure и Haskell[1], генераторы списков представляют собой компактные структуры, объединяющие несколько идей в одну мощную конструкцию. Генераторы списков состоят из двух частей: фильтра и отображения.

Первым языком, при исследовании которого мы познакомились с генераторами списков, был Erlang. Мы начали с корзины покупателя, такой как `Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}]`. Чтобы добавить налог, мы сконструировали единственный генератор списков, решающий задачу одним махом:

```
8>  WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
8>  {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

Некоторые создатели языка отмечают генераторы списков как одну из их самых любимых особенностей. В этом смысле у меня с ними много общего.

Монады

Самый большой интеллектуальный мой рост вызвал, пожалуй, раздел о монадах. В исключительно функциональных языках отсутствует возможность создавать программы с изменяемым состоянием. Поэтому нам пришлось прибегнуть к услугам монад, дающим воз-

---

[1] В языке Scala тоже есть генераторы списков, но мы не использовали их.

можность компоновать из вызовов функций конструкции, помогающие структурировать решение задач, как если бы изменяемое состояние было допустимым. Язык Haskell поддерживает `do`-синтаксис для монад, облегчающий их применение.

Мы также узнали, что монады способны упрощать сложные вычисления. Каждая монада поддерживает определенную вычислительную стратегию. Мы использовали монаду `Maybe` для обработки ошибочных состояний, таких как ошибка поиска в списке, когда функция может вернуть `Nothing`. Мы использовали монаду списка для вычисления декартова произведения и поиска всех возможных комбинаций.

Сопоставление

Одной из еще более широко используемых конструкций программирования, которые мы видели, является сопоставление с образцом. Впервые мы встретились с ней, когда знакомились с языком Prolog, но мы видели ее также в языках Scala, Erlang, Clojure и Haskell. Каждый из этих языков позволяет значительно упростить код с помощью конструкции сопоставления. В числе областей применения этой особенности можно назвать синтаксический анализ, передачу распределенных сообщений, деструктуризацию, унификацию, обработку документов XML и многое другое.

Вспомним службу перевода с одного языка на другой, написанную на языке Erlang, которая может служить типичным примером применения конструкции сопоставления с образцом:

erlang/translate_service.erl
http://media.pragprog.com/titles/btlang/code/erlang/translate_service.erl

```
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
        {From, "casa" } ->
            From ! "house" ,
            loop();

        {From, "blanca" } ->
            From ! "white" ,
            loop();

        {From, _} ->
            From ! "I don't understand." ,
```

```
            loop()
end.

translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

Функция `loop` сопоставляет идентификатор процесса (`From`) и слово, требующее перевода, со словом `casa` или `blanca` или с шаблонным символом. Конструкция сопоставления дает программисту возможность быстро выбрать важные части сообщения, не требуя от него выполнения какого-либо парсинга.

Унификация

В языке Prolog используется унификация – конструкция, родственная сопоставлению с образцом. Вы узнали, что Prolog будет подставлять возможные значения в правило, пока не добьется соответствия левой и правой частей. Попытки будут продолжаться до исчерпания всех возможностей. Действие унификации мы рассматривали на примере программы с именем `concatenate`:

prolog/concat.pl
http://media.pragprog.com/titles/btlang/code/prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
    concatenate(Tail1, List, Tail2).
```

Мы узнали, что унификация делает программу такой мощной потому, что она может действовать в трех режимах: проверка истинности, сопоставление левой стороны и сопоставление правой стороны.

9.4. Найдите свой стиль

На протяжении всей книги упоминались фильмы и персонажи из них. Съемка фильма заключается в комбинировании вашего опыта, умений актеров, мест и истории, которую нужно рассказать. Все, что вы делаете, должно радовать вашу аудиторию. Чем больше вы знаете, тем лучше будут получаться ваши фильмы.

С теми же мерками можно подходить и к программированию. У нас тоже есть своя аудитория. Я не говорю о пользователях наших приложений – я говорю о тех, кто будет читать наш код. Чтобы стать великим программистом, вы должны писать для своей аудитории и

найти свой стиль, который понравится ей. У вас будет много возможностей обрести этот стиль, и ему будет только на пользу, если вы будете знать, что могут предложить вам другие языки. Ваш стиль – ваш уникальный способ самовыражения в коде. Он не сможет улучшаться, если вы не будете наращивать свой опыт. Я надеюсь, что эта книга помогла вам в поисках своего стиля. Но еще больше я надеюсь, что вы получили удовольствие.

Приложение А

Список литературы

[Arm07] Joe Armstrong. «Programming Erlang: Software for a Concurrent World». The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2007.

[Gra04] Paul Graham. «Hackers and Painters: Big Ideas from the Computer Age». O'Reilly & Associates, Inc, Sebastopol, CA, 2004.

[Hal09] Stuart Halloway. «Programming Clojure». The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. «Programming in Scala». Artima, Inc., Mountain View, CA, 2008.

[TFH08] David Thomas, Chad Fowler, and Andrew Hunt. «Programming Ruby: The Pragmatic Programmers' Guide». The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. +7 (499) 782-38-89; электронный адрес **books@alians-kniga.ru**.

Брюс Тейт

Семь языков за семь недель

Практическое руководство по изучению языков программирования

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 32. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф