Волшебство Kotlin

Язык программирования должен быть выразительным, безопасным, гибким и интуитивно понятным. Kotlin соответствует всем этим требованиям! Этот элегантный язык для JVM легко интегрируется с Java и позволяет легко переключаться между объектно-ориентированным и функциональным стилями программирования. Также он поддерживается компанией Google как один из основных языков программирования для Android. Овладев приемами, которые описываются в книге, вы сможете решать новые задачи уверенно и эффективно.

Эта книга научит вас писать на Kotlin выразительные, безопасные и простые в обслуживании программы. Опытный инженер Пьер-Ив Сомон научит вас подходить к решению общих задач с позиций функционального программирования. На множестве практических примеров вы увидите, как правильно обрабатывать ошибки и данные, управлять состоянием и использовать отложенные вычисления. Наглядные примеры и практические идеи автора помогут вам стать успешным разработчиком!

«Потрясающее введение во вселенную функционального программирования».
Алексей Слайковский, Oracle

«Отличные, простые и понятные примеры функционального программирования на Kotlin».
Эммануэль Медина, Global HITSS

«Прекрасный справочник для тех, кто хочет изучать Kotlin или функциональное программирование, но не знает, с чего начать. Описывает сложные задачи, которые часто недооцениваются, и объясняет их решение».
Бриджер Хоуэлл, SoFi

«Объединяя Kotlin и функциональное программирование, эта книга представляет только те теоретические выкладки, которые вам действительно пригодятся, и иллюстрирует их множеством практических примеров».
Жан-Франсуа Морен, Университет Лаваля

В книге рассматриваются:

- программирование с использованием функций;
- обработка необязательных данных;
- безопасная обработка ошибок и исключений;
- совместное использование изменяемого состояния.

Пьер-Ив Симон – программист, инженер-исследователь в Alcatel Submarine Networks. Автор книги Functional Programming in Java.

Издание адресовано разработчикам, использующим Java и Kotlin..

ISBN 978-5-97060-801-2

Волшебство Kotlin

Пьер-Ив Симон

Волшебство Kotlin

MANNING

Пьер-Ив Сомон

Волшебство Kotlin

The Joy of Kotlin

PIERRE-YVES SAUMONT

MANNING
Shelter Island

Волшебство Kotlin

ПЬЕР-ИВ СОМОН

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2020

УДК 004.43Kotlin
ББК 32.972
С61

Сомон. П.-И.
С61 Волшебство Kotlin / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2020. – 536 с.: ил.

ISBN 978-5-97060-801-2

Kotlin – один из самых новых языков в экосистеме Java, устраняющий многие ограничения Java и гораздо более универсальный. Среди его преимуществ: полная совместимость с Java и возможность интеграции на уровне исходного кода, широкая поддержка парадигмы функционального программирования, помогающая писать надежный и безопасный код, лаконичность синтаксиса, а также, что весьма немаловажно, гарантии поддержки со стороны IT-гиганта Google.

Пьер-Ив Сомон, опытный разработчик на Java, в своей книге подробно освещает нюансы программирования на Kotlin, переходя от общего описания языка к его характерным особенностям и возможностям, включая приемы функционального программирования.

Издание предназначено для разработчиков, знакомых с Java и стремящихся повысить безопасность своих программ, а также упростить их написание, тестирование и сопровождение.

УДК 004.43Kotlin
ББК 32.972

ISBN 978-1-617-29536-2 (англ.)
ISBN 978-5-97060-801-2 (рус.)

Оглавление

Содержание

Предисловие

Kotlin появился еще в 2011 году, но по-прежнему остается одним из самых новых языков в экосистеме Java. С тех пор появилась версия Kotlin, работающая на виртуальной машине JavaScript, а также версия, выполняющая компиляцию в машинный код. Это делает Kotlin гораздо более универсальным языком, чем Java, хотя между этими версиями есть большие различия, потому что версия для виртуальной машины Java опирается на стандартную библиотеку Java, недоступную в двух других версиях Kotlin. Сотрудники компании JetBrains, где создан язык Kotlin, прикладывают все силы, чтобы вывести все версии на один уровень, но, как вы понимаете, это требует времени.

Версия для JVM (Java Virtual Machine – виртуальной машины Java), безусловно, является наиболее широко используемой, и она получила дополнительный импульс к развитию, когда в Google объявили Kotlin одним из официальных языков разработки приложений для Android. Основная причина такого решения Google состоит в том, что последняя доступная в Android версия Java – это Java 6, тогда как Kotlin предлагает большинство особенностей Java 11 и многое другое. Kotlin также был объявлен официальным языком сценариев сборки в Gradle вместо Groovy, что позволяет использовать один и тот же язык и для сборки программ, и для самих собираемых программ.

В первую очередь язык Kotlin ориентирован на программистов на Java. Вполне возможно, что в будущем они будут изучать Kotlin как основной язык. Но пока основная масса программистов приходит в Kotlin из Java.

У каждого языка свой путь, определяемый некоторыми фундаментальными понятиями. Язык Java создавался на основе нескольких мощных идей. Он должен работать везде, т. е. в любой среде, где доступна JVM. Главный девиз: «Пиши один раз, запускай где угодно». Хотя некоторые могут утверждать обратное, но в целом Java соответствует этому девизу. Более того, теперь можно запускать почти везде не только программы на Java, но и программы на других языках, скомпилированные для JVM. Kotlin – один из таких языков.

Еще одна основополагающая идея Java – никакие нововведения никогда не станут причиной неработоспособности существующего кода. Хотя эта идея соблюдалась не всегда, но все же разработчики языка старались следовать ей. Конечно, это не всегда хорошо. Основным следствием является невозможность внесения в Java многих улучшений, имеющихся в других языках, потому что они могут привести к нарушению совместимости. Любая программа, скомпилированная с предыдущей версией Java, должна оставаться работоспособной в более новых версиях без перекомпиляции. Является ли это полезным или нет, это спорный вопрос, но в результате стремление максимально сохранить обратную совместимость постоянно играет против развития Java.

Также предполагалось, что Java сделает программы более безопасными за счет использования контролируемых исключений, что вынуждает программистов принимать эти исключения во внимание. Для многих это оказалось тяжким бременем, ведущим к практике постоянного преобразования контролируемых исключений в неконтролируемые.

Хотя Java является объектно-ориентированным языком, он должен быть настолько же быстрым, как и большинство языков, используемых для вычислительных задач. В свое время разработчики языка решили, что Java только выиграет, если, помимо объектов, представляющих числа и логические значения, в языке будут поддерживаться элементарные числовые типы, позволяющие выполнять вычисления намного быстрее. Из-за этого отсутствует возможность помещать значения примитивных типов в коллекции, такие как списки, множества и ассоциативные массивы. А когда были добавлены потоки данных (streams), разработчики языка решили создать специальные версии для примитивов, но не для всех, а только для наиболее часто используемых. Если вы пользуетесь некоторыми из неподдерживаемых примитивов, значит, вам не повезло.

То же самое произошло с функциями. В Java 8 появилась поддержка обобщенных функций, но обобщение возможно только для объектов. Поэтому для обработки целых, длинных целых, вещественных чисел двойной точности и логических значений были разработаны специализированные функции. (И снова, к сожалению, поддержка добавлена не для всех элементарных типов – для работы с однобайтовыми и короткими целыми, а также для вещественных чисел одинарной точности нет специализированных функций.) Что еще хуже, потребовались дополнительные функции для преобразования из одного элементарного типа в другой или из элементарных типов в объекты и обратно.

Язык Java был разработан более 20 лет назад. С тех пор многое изменилось, но большинство этих изменений нельзя было перенести в Java, потому что это нарушило бы совместимость. Некоторые изменения все же вносились, и совместимость сохранилась, но за счет удобства использования.

Для устранения этих ограничений было создано много новых языков, таких как Groovy, Scala и Clojure. Они в определенной степени совместимы с Java и позволяют применять существующие библиотеки Java,

а программисты на Java могут использовать библиотеки, написанные на этих языках.

Kotlin пошел другим путем. Он гораздо сильнее интегрирован с Java, что позволяет без проблем смешивать исходный код на Kotlin и Java в одном проекте! В отличие от других языков для JVM Kotlin не выглядит сильно отличающимся от Java (хотя разница, конечно же, имеется). Он больше похож на язык, которым должен стать язык Java. Некоторые даже говорят, что Kotlin – это Java, созданный правильно, потому что решает большинство проблем, характерных для Java. (Однако Kotlin пока не предлагает решений, связанных с ограничениями, свойственными JVM.)

Но, что еще более важно, Kotlin разрабатывался так, чтобы быть намного более восприимчивым ко многим новым приемам, появляющимся в функциональном программировании. В Kotlin есть изменяемые и неизменяемые ссылки, но предпочтение отдается неизменяемым. Kotlin также поддерживает большую часть абстракций функционального программирования, которые позволяют избегать управляющих структур (хотя он имеет традиционные управляющие структуры, помогающие упростить переход от традиционных языков).

Еще одна замечательная особенность Kotlin – он уменьшает потребность в шаблонном коде, позволяя сократить его до минимума. На Kotlin можно написать класс с необязательными свойствами (обладающий также функциями `equals`, `hashCode`, `toString` и `copy`) в одной строке кода, тогда как для объявления эквивалентного класса на Java потребуется около тридцати строк (включая методы свойств и перегруженные конструкторы).

Да, есть другие языки программирования, разработанные для преодоления ограничений Java в среде JVM, но Kotlin отличается тем, что прекрасно интегрируется с Java-программами на уровне исходного кода. Вы можете смешивать исходные файлы на Java и Kotlin в проектах и использовать единую цепочку сборки. Это меняет правила игры, особенно в отношении командного программирования, потому что использование Kotlin в среде Java – не более сложная задача, чем использование любой сторонней библиотеки. Это обеспечивает максимально плавный переход с Java на новый язык и позволяет писать программы, которые:

- безопаснее;
- проще в разработке, тестировании и сопровождении;
- более масштабируемые.

Я полагаю, что многие читатели – программисты на Java – рано или поздно осознают необходимость поиска новых решений своих повседневных проблем. Возможно, у вас уже возник вопрос, почему вы должны использовать Kotlin. Нет ли других языков в экосистеме Java, которые позволят легко применять безопасные методы программирования?

Конечно, есть, и одним из самых известных является Scala. Scala – очень хорошая альтернатива Java, но у Kotlin есть нечто большее. Scala может

взаимодействовать с Java на уровне библиотек, т. е. программы на Java могут использовать библиотеки Scala (объекты и функции), а программы на Scala могут использовать библиотеки Java (объекты и методы). Но программы на Scala и Java должны создаваться как отдельные проекты или хотя бы как отдельные модули, тогда как классы Kotlin и Java можно смешивать внутри одного модуля.

Прочитайте эту книгу, чтобы узнать больше о Kotlin.

Благодарности

Я хотел бы поблагодарить всех, кто участвовал в создании этой книги. Прежде всего, большое спасибо моему редактору Марине Майклз (Marina Michaels). Мне было очень приятно работать с Вами. Также спасибо моему научному редактору Александару Драгосавлевичу (Aleksandar Dragosavljević).

Большое спасибо также Джоэлю Котарски (Joel Kotarski), Джошуа Уайту (Joshua White) и Риккардо Терреллу (Riccardo Terrell) – техническим редакторам, Алессандро Кампейсу (Alessandro Campeis) и Бренту Уотсону (Brent Watson) – корректорам, которые помогли мне сделать эту книгу намного лучше. Спасибо всем рецензентам, читателям, участвующим в программе MEAP, и всем, кто прислал свои отзывы и комментарии, спасибо вам! Эта книга не получилась бы такой, какая она есть, без вашей помощи. Также я хотел бы поблагодарить людей, которые нашли время, чтобы просмотреть и прокомментировать книгу: Алексея Слайковского (Aleksei Slaikovskii), Алессандро Кампейса (Alessandro Campeis), Энди Кирша (Andy Kirsch), Бенджамина Голдберга (Benjamin Goldberg), Бриджера Хауэлла (Bridger Howell), Конора Редмонда (Conor Redmond), Дилана Макнейми (Dylan McNamee), Эммануэль Медину Лопес (Emmanuel Medina López), Фабио Фальси Родригес (Fabio Falci Rodrigues), Федерико Кирхейса (Federico Kircheis), Герго Михай Надя (Gergő Mihály Nagy), Грегора Раймана (Gregor Raýman), Джейсона Ли (Jason Lee), Жан-Франсуа Морина (Jean-François Morin), Кента Р. Спилнера (Kent R. Spillner), Линн Нортроп (Leanne Northrop), Марка Элстона (Mark Elston), Мэтью Халверсона (Matthew Halverson), Мэтью Проктора (Matthew Proctor), Нуно Александра (Nuno Alexandre), Рафаэля Вентальо (Raffaella Ventaglio), Рональда Харинга (Ronald Haring), Шило Морриса (Shiloh Morri), Винсента Терона (Vincent Theron) и Уильяма Э. Уилера (William E. Wheeler).

Хочу также поблагодарить сотрудников издательства Manning: Дейрдре Хиам (Deirdre Hiam), Фрэнсиса Бурана (Frances Buran), Кери Хейлза (Keri Hales), Дэвида Новака (David Novak), Мелоди Долаб (Melody Dolab) и Николь Берд (Nichole Beard).

О книге

Кому адресована эта книга

Цель этой книги – не просто помочь выучить язык Kotlin, но также научить писать гораздо более безопасные программы. Это не означает, что Kotlin следует использовать, только если вы решите писать более безопасные программы, и тем более не означает, что писать более безопасные программы можно только на Kotlin. Эта книга представляет примеры, написанные на Kotlin, потому что это один из самых дружественных языков для разработки безопасных программ в экосистеме JVM.

В книге рассказывается о методах, разработанных давно и в самых разных окружениях, хотя многие из них своими корнями уходят в функциональное программирование. Но эта книга не о фундаменталистском функциональном программировании, она о прагматичном безопасном программировании.

Все описанные здесь приемы годами использовались в экосистеме Java и доказали свою эффективность в разработке программ с гораздо меньшим количеством ошибок реализации, чем традиционные методы императивного программирования. Эти безопасные приемы можно реализовать на любом языке, и некоторые из них уже многие годы используются в Java, но часто для этого приходится преодолевать ограничения Java.

Эта книга не учит программированию с самого начала. Она писалась для профессиональных программистов, ищущих более простые и безопасные способы разработки безошибочных программ.

О чем вы узнаете

В этой книге вы познакомитесь с конкретными приемами, которые могут отличаться от уже знакомых вам, если прежде вы программировали на Java. Большинство из них покажутся вам незнакомыми или даже противоречащими приемам, которые программисты привыкли считать

оптимальными. Но многие (хотя и не все) оптимальные приемы были выработаны во времена, когда компьютеры имели 640 Кб памяти, 5 Мб дискового пространства и одноядерный процессор. Времена изменились. Сейчас простой смартфон – это настоящий компьютер с 3 Гб или более оперативной памяти, с твердотельным накопителем на 256 Гб и 8-ядерным процессором. Компьютеры тоже имеют много гигабайт оперативной памяти, терабайты дискового пространства и многоядерные процессоры.

В этой книге я расскажу о:

- дальнейшем увеличении абстракции;
- предпочтительном использовании неизменяемых объектов;
- ссылочной прозрачности;
- инкапсуляции общего изменяемого состояния;
- абстрагировании условных и управляющих структур;
- использовании правильных типов данных;
- использовании отложенных вычислений;
- и многом другом.

Дальнейшее увеличение абстракции

Один из наиболее важных методов, с которыми вы познакомитесь, – это дальнейшее увеличение абстракции (традиционные программисты привыкли считать преждевременную абстракцию злом, как и преждевременную оптимизацию). Но дальнейшее увеличение абстракции помогает лучше понять решаемую задачу, что в свою очередь гарантирует правильность решения.

Вы можете спросить, что в действительности подразумевается под дальнейшим увеличением абстракции. Здесь все просто – это умение распознавать типичные шаблоны в различных вычислениях и их абстрагирование, чтобы избежать повторяющегося кода.

Неизменяемость

Суть неизменяемости заключается в использовании только неизменяемых данных. Многим традиционным программистам трудно представить, как можно писать полезные программы, используя только неизменные данные. Разве программирование не основано в первую очередь на изменении данных? Если следовать такой логике, тогда бухгалтерский учет – это прежде всего изменение значений в бухгалтерской книге.

Переход от использования изменяемого к использованию неизменяемого учета произошел еще в XV веке, и с тех пор принцип неизменяемости был признан основным элементом безопасности для учета. Этот принцип также применим к программированию, как вы увидите в этой книге.

Ссылочная прозрачность

Ссылочная прозрачность позволяет писать детерминированные программы, т. е. программы, результаты работы которых можно предсказать

и осмыслить. Такие программы всегда дают одинаковые результаты для одних и тех же входных данных. Это не означает, что они всегда дают одинаковые результаты, но различия в результатах зависят только от изменений на входе, а не от внешних условий.

Такие программы не только безопаснее (потому что обладают предсказуемым поведением), но их также проще писать, сопровождать, обновлять и тестировать. А программы, которые легче тестировать, обычно проверяются более полно и, следовательно, получаются более надежными.

Инкапсуляции общего изменяемого состояния

Неизменяемые данные автоматически оказываются защищены от случайного изменения при совместном использовании, что часто вызывает много проблем при конкурентной и параллельной обработке данных, таких как взаимоблокировка, простои потоков выполнения и устаревание данных. Но отсутствие общего изменяемого состояния превращается в проблему, когда оно действительно необходимо. Это в первую очередь относится к конкурентному и параллельному программированию.

Устранением изменяемого состояния исключается возможность случайного обмена ошибочными данными, благодаря чему программы становятся безопаснее. Но конкурентное и параллельное программирование подразумевает совместное изменение общего состояния. Без этого конкурирующие и параллельные потоки выполнения не смогут сотрудничать друг с другом. Этот конкретный вариант использования изменяемого общего состояния можно абстрагировать и инкапсулировать так, чтобы получить возможность повторного использования без риска, потому что будет иметься одна полностью протестированная универсальная реализация.

В этой книге вы узнаете, как абстрагировать и инкапсулировать общее изменяемое состояние, чтобы реализовать его только один раз и потом использовать везде, где оно необходимо.

Абстрагирование условных и управляющих структур

Вторым распространенным источником ошибок в программах после общего изменяемого состояния являются управляющие структуры. Традиционные программы состоят из таких управляющих структур, как циклы и проверки условий. С этими структурами так легко ошибиться, что разработчики языка постарались максимально абстрагировать детали. Одним из лучших примеров является цикл `for-each`, который ныне присутствует в большинстве языков (хотя в Java он все еще называется `for`).

Другая распространенная проблема – правильное использование `while` и `do while` (или `repeat until`) и, в частности, выбор места, где должно проверяться условие. Еще одна проблема – конкурентное изменение элементов коллекций во время их обхода в цикле, когда проблема общего изменяемого состояния возникает даже при использовании единственного потока выполнения! Абстрагирование управляющих структур позволяет полностью устранить подобные проблемы.

Использование правильных типов данных

В традиционном программировании такие универсальные типы, как `int` и `String`, используются для представления величин без учета единиц измерения. Как следствие, очень легко допустить ошибку, сложив мили с галлонами или доллары с минутами. Использование типов значений может полностью устранить проблемы такого рода при очень низких затратах, даже если используемый язык не предлагает истинных типов значений.

Отложенные вычисления

Большинство распространенных языков называют *строгими*, в том смысле, что аргументы, передаваемые методу или функции, вычисляются заранее, перед их обработкой. На первый взгляд, такое поведение вполне оправданно, хотя часто это не так. Отложенные, или «ленивые», вычисления – это прием, заключающийся в откладывании вычисления элементов до момента, когда они действительно становятся необходимы. Программирование почти целиком основано на отложенных вычислениях.

Например, условие в конструкции `if...else` вычисляется немедленно, перед его проверкой, но выполнение ветвей откладывается, в том смысле, что выполняется только ветвь, соответствующая условию. Эта отложенность скрыта, и программист не контролирует ее. Явное использование отложенных вычислений помогает писать гораздо более эффективные программы.

Читатели

Эта книга адресована читателям, уже имеющим опыт программирования на Java. Необходимо также некоторое понимание параметризованных (обобщенных) типов. В этой книге широко используются параметризованные функции, или варианты, которые редко применяются в Java (хотя это мощный инструмент). Не пугайтесь, если вы еще не знакомы с этими приемами; я расскажу о них и объясню, зачем они нужны, когда подойдет время.

Структура книги

Книга предполагает последовательное чтение, потому что каждая следующая глава основана на понятиях, рассматриваемых в предыдущих главах. Я использую слово «чтение», но эта книга предназначена не только для чтения. Очень немногие разделы содержат одну лишь теорию.

Чтобы извлечь максимум пользы из книги, выполняйте все упражнения, встречающиеся в процессе чтения. Каждая глава содержит ряд упражнений с необходимыми инструкциями и подсказками, которые помогут вам прийти к решению. Каждое упражнение сопровождается предлагаемым решением и тестом, который вы можете использовать для проверки своего решения.

ПРИМЕЧАНИЕ Программный код примеров доступен бесплатно на GitHub (http://github.com/pysaumont/fpinkotlin). В него включены все элементы, необходимые для импортирования проекта в IntelliJ (рекомендуется) или для компиляции и выполнения с использованием Gradle 4. Пользующиеся Gradle могут редактировать код в любом текстовом редакторе. Предполагается, что Kotlin можно использовать в Eclipse, но я не могу этого гарантировать. IntelliJ – намного более удобная интегрированная среда разработки (IDE), которую можно бесплатно получить на сайте Jetbrains (https://www.jetbrains.com/idea/download).

Выполнение упражнений

Упражнения играют важную роль в обучении и помогают понять то, о чем рассказывает эта книга. В процессе чтения теоретических разделов какие-то идеи и понятия могут остаться для вас непонятными, и в этом нет ничего страшного. Но выполнение упражнений играет особенно важную роль в процессе обучения, поэтому я призываю вас не пропускать никаких упражнений.

Некоторые могут показаться довольно сложными, и у вас может возникнуть соблазн заглянуть в предложенные решения. Это нормально, но после этого вернитесь к упражнению и выполните его, не заглядывая в решение. Если вы ограничитесь простым просмотром решения, у вас почти наверняка возникнут проблемы при попытке выполнить более сложные упражнения в последующих главах.

Вам не придется вводить много кода, потому что большинство упражнений состоит в написании реализаций функций, для которых вы получаете готовые окружение и сигнатуру функции. Ни одно упражнение не потребует ввода больше десятка строк кода; а в большинстве случаев решения состоят из четырех или пяти строк. Закончив упражнение (т. е. добившись безошибочной компиляции вашей реализации), просто запустите соответствующий тест, чтобы убедиться в правильности решения.

Важно отметить, что каждое упражнение является самодостаточным и независящим от остальной части главы, поэтому код, представленный внутри главы, повторяется от упражнения к упражнению. Это необходимо, потому что часто следующее упражнение основано на предыдущем. По этой причине реализации могут отличаться даже при использовании одного и того же класса. Как следствие, старайтесь не заглядывать в последующие упражнения, не завершив предыдущие, потому что вы увидите решения для еще не решенных упражнений.

Обучение методикам в этой книге

Методики, описанные в этой книге, не сложнее традиционных. Они просто разные. Те же задачи можно решать с использованием традиционных методик, но перевод решения с одной методики на другую иногда может сопровождаться потерей эффективности.

Изучение новых методик сродни изучению иностранного языка. Как нельзя эффективно мыслить на одном языке и переводить на другой, так же нельзя рассуждать терминами традиционного программирования, основанного на изменении состояния и потоке управления, и преобразовывать свой код в функции, обрабатывающие неизменяемые данные. И точно так же, как вы должны научиться думать на новом языке, вы должны научиться думать по-другому. Этого нельзя добиться простым чтением; это приходит с написанием кода. Поэтому обязательно практикуйтесь!

Вот почему я не ожидаю, что вы придете к полному пониманию написанного в этой книге, просто прочитав ее, и даю так много упражнений; *старайтесь* выполнять все упражнения, чтобы полностью понять представленные идеи. Сами по себе темы несложные, но, чтобы понять их, одного только чтения недостаточно. Если бы вы могли понять суть без выполнения упражнений, тогда вам, вероятно, не понадобилась бы эта книга.

Решение упражнений – это ключ к успешному освоению методик, описываемых в этой книге. Я советую стараться решать все встречающиеся упражнения по порядку, ничего не пропуская, прежде чем продолжать чтение. Если у вас не получится найти решение, попробуйте еще раз, и, только если у вас ничего не получится со второй попытки, переходите к предлагаемым решениям.

Если вам сложно понять что-то, задайте вопрос на форуме (ссылка приводится в следующем разделе). Задавая вопросы и получая ответы на форуме, вы не только помогаете себе, но также человеку, отвечающему на вопрос (и многим другим, у кого может возникнуть та же проблема). Мы в большей мере учимся, когда отвечаем на вопросы (в основном на свои собственные), а не когда задаем их.

Исходный программный код

Книга содержит много примеров исходного кода в виде листингов и фрагментов в обычном тексте. В обоих случаях исходный код оформляется моноширинным шрифтом, чтобы его можно было отличить от обычного текста.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили переносы строк и изменили ширину отступов, чтобы уместить строки кода по ширине книжной страницы. Многие листинги сопровождаются дополнительными аннотациями, подчеркивающими наиболее важные идеи.

Исходный код можно загрузить в виде файла архива или клонировать с помощью Git. Код для упражнений организован в модули с именами, отражающими названия глав, а не их номера. По этой причине IntelliJ отсортирует их в алфавитном порядке, а не в порядке следования в книге.

Чтобы помочь вам понять, какой модуль к какой главе относится, я подготовил список глав с соответствующими именами модулей в фай-

ле *README* (http://github.com/pysaumont/fpinkotlin).

Исходный код из всех листингов в книге доступен также для загрузки на веб-сайте издательства Manning: https://www.manning.com/books/the-joy-ofkotlin.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста,

сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Пьер-Ив Сомон (Pierre-Yves Saumont) – опытный разработчик на Java с тридцатилетней практикой проектирования и создания корпоративного программного обеспечения. Работает инженером-исследователем в ASN (Alcatel Submarine Networks).

Об иллюстрации на обложке

На обложке «Волшебства Kotlin» изображен рисунок, озаглавленный как «Традиционный женский костюм в Тартарии, 1700». Рисунок взят из книги «Collection of the Dresses of Different Nations, Ancient and Modern» (Коллекция костюмов разных народов, античных и современных) Томаса Джефферис (Thomas Jefferys), опубликованной в Лондоне между 1757 и 1772 годами. На титульной странице указано, что это выполненная вручную каллиграфическая цветная гравюра, обработанная гуммиарабиком.

Томас Джефферис (1719–1771) носил звание «географа короля Георга III». Английский картограф, он был ведущим поставщиком карт того времени. Он выгравировал и напечатал множество карт для нужд правительства и других официальных органов, широкий спектр коммерческих карт и атласов, в частности, Северной Америки. Будучи картографом, интересовался местной одеждой народов, населяющих разные земли, и собрал блестящую коллекцию различных платьев в четырех томах. Очарование далекими землями и дальние путешествия для удовольствия были относительно новым явлением в конце XVIII века, и коллекции, такие как эта, были весьма популярны, так как знакомили с внешним видом жителей других стран.

Разнообразие рисунков, собранных Джефферисом, свидетельствует о проявлении народами мира около 200 лет яркой индивидуальности и уникальности. С тех пор стиль одежды сильно изменился и исчезло разнообразие, характеризующее различные области и страны. Теперь трудно отличить по одежде даже жителей разных континентов. Если взглянуть на это с оптимистической точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду разнообразию личной жизни или в угоду более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну техническую книгу от другой, издательство Manning проявило инициативу и деловую сметку, украшая обложки книг изображениями, основанными на богатом разнообразии жизненного уклада народов двухвековой давности, придав новую жизнь рисункам Джефферис.

1 Создание безопасных программ

Эта глава охватывает следующие темы:

- выявление программных ловушек;
- проблемы с эффектами;
- как ссылочная прозрачность увеличивает безопасность программ;
- использование подстановочной модели в рассуждениях о программе;
- максимальное использование абстракций.

Программирование – опасная работа. Если вы программист-любитель, вас наверняка удивит такое утверждение. Вы, вероятно, думали, что находитесь в безопасности, сидя перед экраном и клавиатурой. Вы можете думать, что рискуете не более чем заработать боль в спине, если сидеть слишком долго, некоторые проблемы со зрением при чтении мелких символов на экране или даже тендинит запястья, если вам случается слишком много печатать. Но если вы профессиональный программист (или хотите им стать), реальность намного хуже.

Главная опасность – ошибки, скрывающиеся в ваших программах. Ошибки могут стоить дорого, если появятся не вовремя. Помните ошибку Y2K? Во многих программах, написанных между 1960 и 1990 годами, для обозначения года в датах использовались только две цифры, поскольку программисты не ожидали, что их программы доработают до следующего столетия. Многие из этих программ, все еще использовавшихся

в 1990-х годах, рассматривали бы 2000 год как 1900. Ориентировочная стоимость этой ошибки в долларах США в 2017 году составила 417 млрд[1].

Но иногда ошибка, возникающая в одной программе, может стоить намного выше. 4 июня 1996 года первый полет французской ракеты Ariane 5 закончился аварией через 36 с после старта. Сбой произошел из-за ошибки в системе навигации. Одно целочисленное арифметическое переполнение привело к потере 370 млн долл[2].

Как бы вы себя чувствовали, если бы были привлечены к ответственности за такую катастрофу? Как бы вы себя чувствовали, если бы постоянно писали такие программы и не были уверены, что программа, работающая сегодня, будет работать и завтра? Именно так работает большинство программистов: они пишут недетерминированные программы, которые возвращают разные результаты при запуске с одними и теми же входными данными. Пользователи знают об этом, и когда программа возвращает результаты, отличающиеся от ожидаемых, они повторяют попытку, надеясь, что недетерминированность приведет к другому результату. И иногда это случается, потому что никто не знает, от чего зависит результат, возвращаемый программой.

С развитием искусственного интеллекта (ИИ) проблема надежности программного обеспечения стала еще более актуальной. Если программы, принимающие решения, такие как автопилоты в самолетах или в беспилотных автомобилях, могут поставить под угрозу человеческую жизнь, мы должны быть уверены, что они работают в точности, как задумано.

Что нужно, чтобы сделать программы более безопасными? Некоторые ответят, что нужны лучшие программисты. Но хорошие программисты подобны хорошим водителям. 90 % программистов согласны с тем, что только 10 % достаточно хороши, но при этом 90 % программистов считают, что они являются частью 10 %!

Самое необходимое качество для программистов – умение признавать собственные ограничения. Посмотрим правде в глаза: мы – всего лишь средние программисты. Мы тратим 20 % времени на написание программ с ошибками, а затем 40 % времени – на рефакторинг кода, чтобы получить программы без видимых ошибок. А позже мы тратим еще 40 % на отладку кода, который уже находится в эксплуатации, потому что ошибки делятся на две категории: очевидные и неочевидные. Вне всяких сомнений, неочевидные ошибки станут очевидными – это всего лишь вопрос времени. Остается вопрос: как долго и насколько большой будет наноситься ущерб, прежде чем ошибки станут очевидными.

Как можно решить эту проблему? Никакой инструмент, методика или дисциплина не могут гарантировать отсутствие ошибок в наших программах. Но есть методики, которые могут устранить отдельные катего-

---

[1] Согласно оценке потребительских цен федерального резервного банка: «Consumer Price Index (estimate) 1800–» (https://www.minneapolisfed.org/community/teaching-aids/cpi-calculator-information/consumer-price-index-1800).

[2] Отчет комиссии по расследованию аварии Ariane 501 (http://www.astrosurf.com/luxorion/astronautique-accident-ariane-v501.htm).

рии ошибок и гарантировать, что оставшиеся ошибки будут присутствовать только в ограниченных (небезопасных) областях наших программ. Это имеет огромное значение, потому что делает поиск ошибок намного проще и эффективнее. Среди таких методик находится рекомендация писать простые программы, в которых отсутствие ошибок очевидно, вместо сложных программ, где не видно очевидных ошибок[1].

В оставшейся части главы я кратко познакомлю вас с такими понятиями, как неизменяемость, ссылочная прозрачность и подстановочная модель, а также дам несколько рекомендаций, которые помогут вам сделать свои программы намного безопаснее. В следующих главах мы будем применять эти идеи снова и снова.

1.1 Программные ловушки

Программирование часто рассматривают как способ описания выполнения некоторого процесса. Такое описание обычно включает перечисление действий, изменяющих состояние программной модели с целью выполнения поставленной задачи и принятия решения о результатах таких изменений. Так видит программирование большинство, даже если это не программисты.

Если перед вами поставили слишком сложную задачу, вы делите ее на шаги. Затем выполняете первый шаг, проверяете результат и по результатам проверки выбираете следующий шаг для выполнения. Например, программу сложения двух положительных значений `a` и `b` можно представить следующим псевдокодом:

- если b = 0, вернуть a;
- иначе увеличить a и уменьшить b;
- начать с начала с новыми значениями a и b.

В этом псевдокоде можно заметить инструкции, традиционные для большинства языков программирования: проверка условия, изменение переменной, ветвление и возврат значения. Этот код можно представить графически, в виде блок-схемы, как показано на рис. 1.1.

Легко заметить, где в этой программе могут возникнуть проблемы. Измените любые данные на блок-схеме или начальный или конечный пункт любой стрелки, и вы получите программу с ошибкой. Если повезет, вы получите программу, которая вообще не запускается или работает вечно и никогда не останавливается. Это можно считать удачей, потому что вы сразу увидите, что в программе есть проблема, которую нужно исправить. На рис. 1.2 показаны три такие проблемы.

---

[1] «…есть два способа конструирования программного обеспечения: один из них заключается в том, чтобы сделать его настолько простым, чтобы в нем явно не было недостатков, а другой – сделать его настолько сложным, чтобы в нем не было явных недостатков. Первый способ гораздо сложнее». *С.А.Р. Хоар* (C.A.R. Hoare). The Emperor's Old Clothes // Communications of the ACM 24 (февраль 1981): с. 75–83.

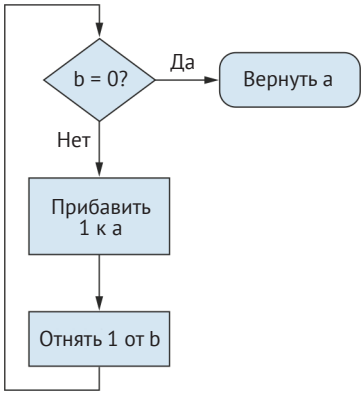


Рис. 1.1 Блок-схема, представляющая программу как процесс, протекающий во времени. В ходе выполнения программы выполняются разные преобразования и изменение состояния, пока наконец не будет получен результат

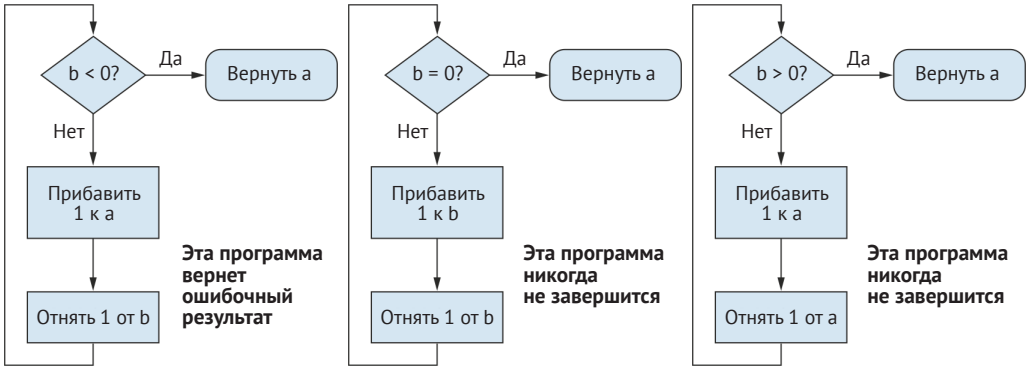


Рис. 1.2 Три версии программы с ошибками

Первая версия возвращает неверный результат, а вторая и третья никогда не завершатся. Однако обратите внимание, что отдельные языки программирования могут не позволить написать некоторые из этих версий. Ни одну из них нельзя написать на языке, который не допускает изменения значения по ссылке, и ни одну из них нельзя написать на языке, который не поддерживает ветвление или циклы. Вы можете подумать, что для устранения проблемы достаточно лишь использовать такой язык. Конечно, можно пойти этим путем. Но вы будете ограничены небольшим количеством языков, и может так получиться, что ни один из них не будет разрешен для использования в вашей профессиональной сфере.

Есть другое решение? Да, есть: отказаться от использования ссылок, допускающих возможность изменения, ветвления (если ваш язык это позволяет) и циклов. То есть нужно просто выработать определенную дисциплину программирования.

Не используйте потенциально опасные возможности, такие как изменение значений переменных и циклы. Это так просто! А если обнаружится, что без изменяемых ссылок или циклов не обойтись, абстрагируйте их. Напишите какой-нибудь компонент, который раз и навсегда скроет

в своих недрах изменение состояния, и вы навсегда избавитесь от проблемы. (Некоторые более или менее экзотические языки предлагают такие компоненты «из коробки», но, вероятно, это тоже не те языки, которые вы сможете использовать в своей сфере.) То же относится к циклам. Большинство современных языков наряду с традиционными циклами предлагают абстракции циклов. И снова использовать или не использовать циклы – это вопрос дисциплины. Применяйте только качественные детали! Подробнее об этом в главах 4 и 5.

Другой распространенный источник ошибок – пустые (`null`) ссылки. Как будет показано в главе 6, Kotlin позволяет ясно отделить код, который допускает пустые ссылки, от кода, который запрещает их. Но вообще желательно полностью искоренить использование пустых ссылок в программах.

Многие ошибки вызваны зависимостями программ от внешнего мира. Но зависимости от внешнего мира необходимы почти во всех программах. Ограничение влияния этих зависимостей конкретными областями в ваших программах упростит поиск и устранение проблем, хотя и не избавит полностью от появления ошибок такого типа.

В этой книге вы познакомитесь с несколькими правилами, помогающими сделать программы намного безопаснее. Вот их список:

- не используйте изменяемые ссылки (переменные) и абстрагируйте единичные случаи, когда отказаться от изменения состояния не получается;
- не используйте управляющие конструкции;
- ограничьте эффекты (взаимодействия с внешним миром) определенными областями в коде. То есть не нужно выводить что-либо в консоль или на любое другое устройство, записывать данные в файлы, базы данных, сети или куда-либо еще за пределами этих ограниченных областей;
- не возбуждайте исключений. Возбуждение исключения – это современная форма ветвления (`GOTO`), из-за которого код начинает напоминать *тарелку со спагетти,* – трудно понять, где и что начинается, и невозможно проследить, куда движется выполнение. В главе 7 вы узнаете, как избежать возбуждения исключений.

1.1.1 Безопасная обработка эффектов

Как я уже отмечал, под «эффектами» подразумеваются взаимодействия с внешним миром, такие как вывод в консоль, запись в файл, в базу данных или в сеть, а также изменение чего-то, находящегося вне области видимости компонента. Программы обычно пишутся небольшими блоками, имеющими свои области видимости. В некоторых языках эти блоки называются *процедурами*; в других (например, Java) – *методами*. В Kotlin они называются *функциями*, хотя это название несет несколько иной смысл, чем понятие функции в математике.

Функции в Kotlin фактически являются методами, как в Java и многих других современных языках. Эти блоки кода имеют *область видимости*,

т. е. область программы, видимую только этими блоками. Блоки имеют не только свою, огражденную от других область видимости, но и доступ к внешним областям видимости и – посредством их – к внешнему миру. Следовательно, любое изменение внешнего мира, вызванное функцией или методом (например, изменение в области видимости класса, в котором определен метод), является эффектом.

Некоторые методы (функции) возвращают значение. Одни изменяют внешний мир, а другие делают и то, и другое. Когда метод или функция возвращает значение и вызывает эффект, такой эффект называется *побочным*. Появление побочных эффектов в программировании всегда считалось неправильным. В медицине термин «побочные эффекты» часто используется для описания нежелательных результатов лечения. В программировании побочный эффект – это явление, наблюдаемое за пределами программы, *дополняющее* результат, возвращаемый программой.

Если программа не возвращает результата, нельзя сказать, что наблюдаемый эффект является побочным; это главный эффект. Однако он может сопровождаться побочными (вторичными) эффектами, которые также часто считаются нежелательными с точки зрения принципа «единственной ответственности».

Безопасные программы конструируются из функций, которые принимают аргумент и возвращают значение, и все. Нас не волнует, что происходит *внутри* функций, потому что теоретически там никогда ничего не происходит. Некоторые языки предлагают только такие функции без эффектов: программы, написанные на этих языках, не имеют видимых эффектов, кроме возвращаемого значения. Но в действительности это значение может быть новой программой, которую можно запустить и получить эффект. Такой подход можно использовать в любом языке, но часто он считается неэффективным (что спорно). Безопасной альтернативой является ясное отделение кода, производящего эффект, от остальной части программы и даже, насколько это возможно, его абстрагирование. С несколькими приемами, позволяющими сделать это, вы познакомитесь в главах 7, 11 и 12.

1.1.2 Увеличение безопасности программ за счет ссылочной прозрачности

Отсутствия побочных эффектов (изменений во внешнем мире) недостаточно, чтобы сделать программу безопасной и детерминированной. Программы также не должны подвергаться влиянию внешнего мира – результат их работы должен зависеть только от аргументов. То есть программы не должны читать данные из консоли, файла, сети, базы данных или даже из переменных системы.

Код, который ничего не изменяет и не зависит от внешнего мира, называется *ссылочно-прозрачным* (referentially transparent). Ссылочно-прозрачный код обладает рядом интересных свойств:

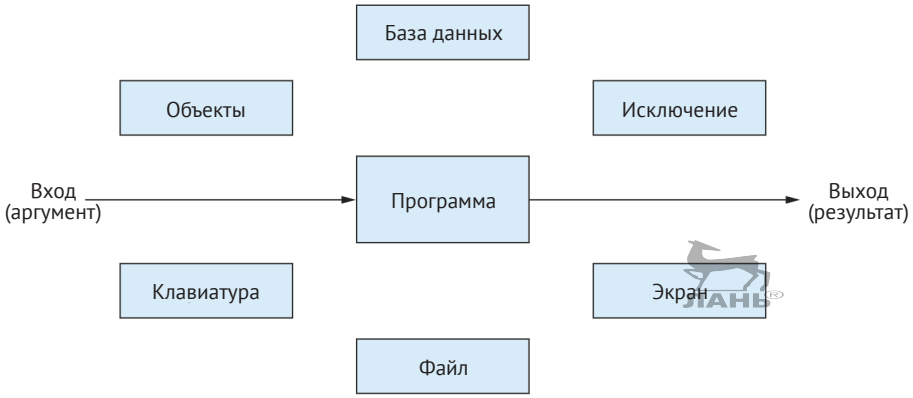
- *самодостаточность* – его можно использовать в любом контексте, достаточно лишь передать ему допустимые аргументы;
- *детерминированность (определенность)* – для одних тех же аргументов он всегда возвращает одно и то же значение; он может возвращать ошибочный результат, но, по крайней мере, результат всегда будет одинаковым для одного и того же аргумента;
- *никогда не возбуждает исключений* – он может вызывать ошибки, такие как нехватка памяти или переполнение стека, но эти ошибки явно указывают на ошибки в коде. Это не та ситуация, которую вы, как программист, или пользователи вашего API должны обрабатывать (но важно предусмотреть защиту от аварийного завершения приложения, которая часто не дается автоматически, и в конечном итоге исправить ошибку);
- *не создает условий, вызывающих неожиданный сбой другого кода,* – например, он не изменяет аргументы и любые другие внешние данные, из-за чего вызывающая сторона могла бы столкнуться с проблемой неактуальных данных или с исключениями одновременного доступа;
- *не зависит от какого-либо внешнего устройства* – он не зависнет из-за того, что какое-то внешнее устройство (база данных, файловая система или сеть) недоступно, работает слишком медленно или вышло из строя.

На рис. 1.3 показаны различия между ссылочно-прозрачными и непрозрачными программами.

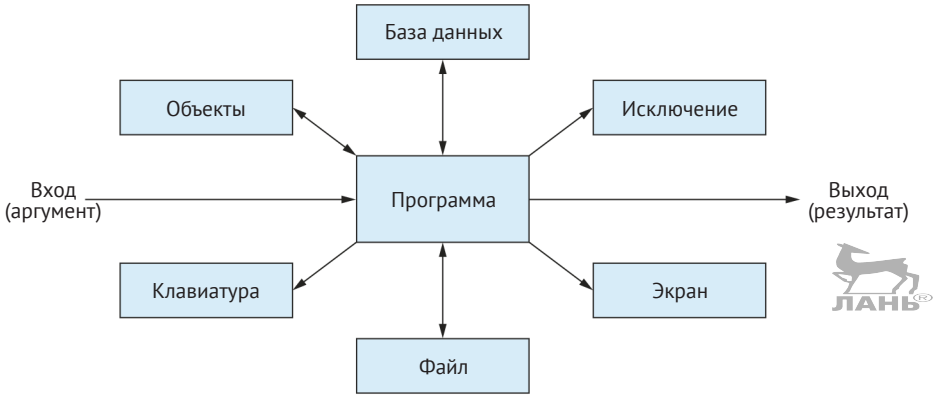
1.2 Выгоды безопасного программирования

По моему описанию выше вы наверняка догадаетесь, какие выгоды несет ссылочная прозрачность:

1. *Ссылочно-прозрачные программы проще анализировать благодаря их детерминированности*. Конкретные входные аргументы всегда будут приводить к одному и тому же результату. Во многих случаях правильность таких программ можно доказать, не проводя тщательного тестирования и не опасаясь, что она выполнится как-то не так при необычных условиях.
2. *Ссылочно-прозрачные программы проще тестировать*. Поскольку такие программы не имеют побочных эффектов, вам не понадобятся имитации, которые часто требуются при тестировании для изоляции компонентов программы от внешнего мира.
3. *Ссылочно-прозрачные программы имеют более модульную организацию*. Причина в том, что такие программы строятся из функций, имеющих только вход и выход, – нет никаких побочных эффектов, никаких исключений и изменений контекста, которые нужно обрабатывать, нет общего изменяемого состояния и никаких операций, одновременно изменяющих одно и то же состояние.



Ссылочно-прозрачная программа не влияет на внешний мир – она лишь принимает аргумент на входе и возвращает результат на выходе. Результат зависит только от аргумента



Программа, не являющаяся ссылочно-прозрачной, может читать данные из внешнего мира или записывать их во внешние элементы или в файлы, изменять внешние объекты, читать с клавиатуры, выводить на экран и т. д. Результат работы такой программы непредсказуем

Рис. 1.3 Сравнение ссылочно-прозрачной и ссылочно-непрозрачной программ

4 *Составлять и реорганизовывать ссылочно-прозрачные программы намного проще*. Создание такой программы начинается с разработки различных базовых функций, а затем эти функции объединяются в функции более высокого уровня. Этот процесс повторяется, пока не будет получена единственная функция, представляющая всю программу. А так как все эти функции являются ссылочно-прозрачными, их можно использовать для создания других программ без каких-либо изменений.

5 *Ссылочно-прозрачные программы по своей природе поддерживают многопоточное выполнение, потому что не изменяют общего состояния*. Это не означает, что все данные должны быть неизменными, – неизменными должны быть только общие данные. Программисты, применяющие это правило, вскоре понимают, что неизменяемые

данные всегда безопаснее, даже если изменения не видны снаружи. Одна из причин состоит в том, что данные, не являющиеся общими, не станут случайно общедоступными после рефакторинга. Повсеместное использование неизменяемых данных гарантирует, что такого рода проблемы никогда не возникнут.

В оставшейся части этой главы я представлю несколько примеров, как использование ссылочной прозрачности помогает писать безопасные программы.

1.2.1 Использование подстановочной модели в рассуждениях о программе

Основным преимуществом использования функций, которые возвращают значение без любого другого наблюдаемого эффекта, является их эквивалентность возвращаемому значению. Такая функция ничего не делает. Она имеет значение, зависящее только от ее аргументов. Как следствие, вызов функции или любое ссылочно-прозрачное выражение всегда можно заменить его значением, как показано на рис. 1.4.

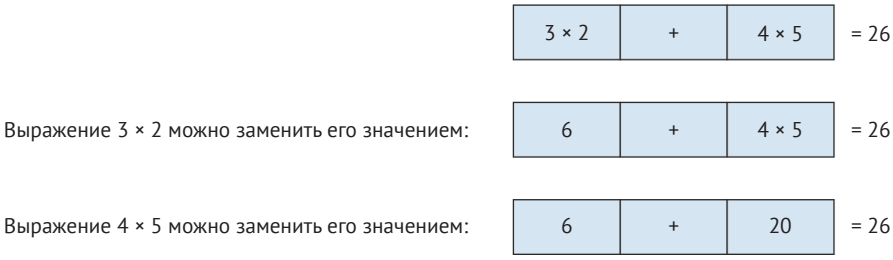


Рис. 1.4 Замена ссылочно-прозрачных выражений их значениями не меняет сути дела

Применительно к функциям прием подстановочного моделирования позволяет заменить вызов любой функции ее возвращаемым значением. Взгляните на следующий код:

```
fun main(args: Array<String>) {
    val x = add(mult(2, 3), mult(4, 5))
    println(x)
}

fun add(a: Int, b: Int): Int {
    log(String.format("Returning ${a + b} as the result of $a + $b"))
    return a + b
}

fun mult(a: Int, b: Int) = a * b

fun log(m: String) {
    println(m)
}
```

Замена вызовов `mult(2, 3)` и `mult(4, 5)` соответствующими возвращаемыми значениями не влияет на результат работы программы, которая упрощается до выражения

```
val x = add(6, 20)
```

Замена вызова функции `add` ее возвращаемым значением, напротив, существенно влияет на результат программы, потому что в этом случае функция `log` не будет вызвана и регистрация результата не произойдет. Это, может быть, и не важно, но, как бы то ни было, такая подстановка меняет результат программы.

1.2.2 Применение принципов соблюдения безопасности на простом примере

Чтобы увидеть, как преобразовать небезопасную программу в более безопасную, рассмотрим простой пример реализации покупки пончика с помощью кредитной карты.

Листинг 1.1. Программа на Kotlin с побочными эффектами

```
fun buyDonut(creditCard: CreditCard): Donut {
  val donut = Donut()
  creditCard.charge(donut.price) ①
  return donut ②
}
```

① **Взимает плату с кредитной карты – это побочный эффект**
② **Возвращает пончик**

В этом коде взимание платы с кредитной карты является побочным эффектом. Взимание платы, вероятно, включает обращение в банк, проверку действительности кредитной карты, авторизацию и регистрации транзакции. Функция возвращает пончик.

Проблема с подобным кодом в том, что его сложно тестировать. Тестовый прогон программы должен включать обращение в банк и регистрацию транзакции с использованием некоторого фиктивного счета. Или вам придется создать фиктивную кредитную карту, чтобы зарегистрировать эффект вызова функции `charge` и проверить ее состояние после тестирования.

Чтобы иметь возможность проверить программу без обращения в банк или без создания фиктивной карты, необходимо устранить побочный эффект. Но так как программа все же должна снимать средства с кредитной карты, единственное решение – добавить представление этой операции в возвращаемое значение. В данном случае функция `buyDonut` должна возвращать не только пончик, но и представление платежа. Для представления платежа можно использовать класс `Payment`, как показано в листинге 1.2.

Листинг 1.2. Класс Payment

```
class Payment(val creditCard: CreditCard, val amount: Int)
```

Этот класс содержит все данные, необходимые для представления платежа, включая сведения о кредитной карте и сумму для списания. Поскольку функция `buyDonut` должна возвращать `Donut` и `Payment`, с этой целью можно создать отдельный класс, например `Purchase`.

```
class Purchase(val donut: Donut, val payment: Payment)
```

Вам часто будут нужны такие классы, предназначенные для хранения двух (или более) значений разного типа, потому что, чтобы сделать программы безопаснее, нужно заменить побочные эффекты возвратом представления этих эффектов.

Вместо создания специализированного класса `Purchase` можно использовать обобщенный класс `Pair`. Этот класс параметризуется двумя типами, в данном случае `Donut` и `Payment`. Кроме `Pair`, Kotlin предлагает также класс `Triple`, позволяющий представить три значения. Такой класс мог бы очень пригодиться в языке, подобном Java, потому что определение класса `Purchase` в Java подразумевает объявление конструктора, методов чтения и, возможно, методов `equals` и `hashCode`, а также `toString`. Но его ценность в Kotlin несколько ниже, потому что тот же результат можно получить всего одной строкой кода:

```
data class Purchase(val donut: Donut, val payment: Payment)
```

Класс `Purchase` уже не нуждается в явном конструкторе и методах чтения. А если добавить ключевое слово `data` перед определением класса, Kotlin автоматически включит в класс реализации по умолчанию `equals`, `hashCode`, `toString` и `copy`. Два экземпляра класса данных будут считаться равными, только если все их свойства будут равны. Если понятие «равно» в вашем случае имеет другой смысл, вам придется переопределить эти функции своими реализациями.

```
fun buyDonut(creditCard: CreditCard): Purchase {
    val donut = Donut()
    val payment = Payment(creditCard, Donut.price)
    return Purchase(donut, payment)
}
```

На данном этапе вас больше не волнует проблема взимания платы с кредитной карты. Это добавляет некоторую свободу выбора при создании приложения. Можно обработать платеж немедленно или сохранить его для дальнейшей обработки. Можно даже объединить сохраненные платежи с одной и той же карты и обработать их в одной операции. Это поможет сэкономить деньги за счет уменьшения банковских сборов за обслуживание кредитной карты.

В листинге 1.3 показана функция `combine`, объединяющая платежи. Если кредитные карты не совпадают, она возбуждает исключение. Это не противоречит тому, что я говорил о безопасных программах, не генерирующих исключений. Здесь попытка объединить два платежа с двумя разными кредитными картами считается ошибкой, поэтому она должна вызывать сбой приложения. (Впрочем, это не самый лучший выход. Но подождите до главы 7, где я расскажу, как справляться с такими ситуациями без создания исключений.)

Листинг 1.3. Объединение нескольких платежей

```
package com.fpinkotlin.introduction.listing03
class Payment(val creditCard: CreditCard, val amount: Int) {
    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")
}
```

В этом сценарии функция `combine` будет не самым эффективным решением при покупке сразу нескольких пончиков. Более оптимальное решение: заменить функцию `buyDonut` функцией `buyDonuts(n: Int, creditCard: CreditCard)`, которая показана в листинге 1.4, но в этом случае необходимо определить новый класс `Purchase`. Как вариант, если перед этим вы решили взять `Pair<Donut, Payment>`, вы должны будете использовать `Pair <List<Donut>, Payment>`.

Листинг 1.4. Покупка сразу нескольких пончиков

```
package com.fpinkotlin.introduction.listing05
data class Purchase(val donuts: List<Donut>, val payment: Payment)
fun buyDonuts(quantity: Int = 1, creditCard: CreditCard): Purchase =
    Purchase(List(quantity) {
        Donut()
    }, Payment(creditCard, Donut.price * quantity))
```

Здесь `List(quantity) { Donut() }` создает список с `quantity` элементами и последовательно применяет функцию `{ Donut() }` к значениям от `0` до `quantity - 1`. Функция `{ Donut() }` эквивалентна

```
{ index -> Donut{} }
```

или

```
{ _ -> Donut{} }
```

В случае с единственным параметром часть `parameter ->` можно опустить и ссылаться на параметр как `it`. Поскольку в данном случае он не используется, код сокращается до `{ Donut () }`. Если вам что-то пока не понятно, не беспокойтесь: я расскажу об этом подробнее в следующей главе.

Также обратите внимание, что параметр `quantity` имеет значение по умолчанию `1`. Это позволяет вызвать функцию `buyDonuts`, используя следующий синтаксис без параметра `quantity`:

```
buyDonuts(creditCard = cc)
```

Чтобы реализовать подобную возможность в языке Java, вам пришлось бы написать перегруженную версию метода, например:

```
public static Purchase buyDonuts(CreditCard creditCard) {
    return buyDonuts(1, creditCard);
```

```
}
public static Purchase buyDonuts(int quantity,
                                 CreditCard creditCard) {
    return new Purchase(Collections.nCopies(quantity, new Donut()),
                        new Payment(creditCard, Donut.price * quantity));
}
```

Теперь программу можно протестировать без использования имитаций. Например, вот как мог бы выглядеть тест для метода `buyDonuts`:

```
import org.junit.Assert.assertEquals
import org.junit.Test
class DonutShopKtTest {
    @Test
    fun testBuyDonuts() {
        val creditCard = CreditCard()
        val purchase = buyDonuts(5, creditCard)
        assertEquals(Donut.price * 5, purchase.payment.amount)
        assertEquals(creditCard, purchase.payment.creditCard)
    }
}
```

Еще одно преимущество такой организации кода: программа упрощает комбинирование ее элементов. При совершении сразу нескольких покупок первая версия программы обратится в банк и оплатит каждую покупку отдельно (что повлечет списание банком соответствующей комиссии за каждую покупку). Новая версия может сгруппировать все платежи, сделанные с помощью одной и той же карты, и списать деньги только один раз – сразу за все покупки. Для группировки платежей вам понадобятся дополнительные функции из класса `List` в языке Kotlin:

- `groupBy(f: (A) -> B): Map<B, List<A>>` – принимает функцию, получающую значение типа `A` и возвращающую значение типа `B`, и в свою очередь возвращает ассоциативный массив с ключами типа `B` и значениями типа `List<A>`. Эту функции можно использовать для группировки платежей по кредитной карте.
- `values: List<A>` – функция экземпляра типа `Map`, которая возвращает список всех значений в ассоциативном массиве.
- `map(f: (A) -> B): List<B>` – функция экземпляра типа `List`, которая принимает функцию, получающую значение типа `A` и возвращающую значение типа `B`, применяет ее ко всем элементам в списке `A` и возвращает список элементов типа `B`.
- `reduce(f: (A, A) -> A): A` – функция экземпляра типа `List`, которая использует операцию (представленную функцией `f: (A, A) -> A`) для преобразования (свертки) списка в единственное значение. Операцией может быть, например, сложение. В таком случае можно было бы передать функцию `f(a, b) = a + b`.

Используя эти функции, можно создать новую, которая группирует платежи кредитной картой, как показано в листинге 1.5.

Листинг 1.5. Группировка платежей кредитной картой

```
package com.fpinkotlin.introduction.listing05;
class Payment(val creditCard: CreditCard, val amount: Int) {
    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")
    companion object {
        fun groupByCard(payments: List<Payment>): List<Payment> =
            payments.groupBy { it.creditCard } ①
                    .values ②
                    .map { it.reduce(Payment::combine) } ③
    }
}
```

① Преобразует `List<Payment>` в `Map<CreditCard, List<Payment>>`, где каждый список `List<Payment>` содержит все платежи для конкретной кредитной карты
② Преобразует `Map<CreditCard, List<Payment>>` в `List<List<Payment>>`
③ Преобразует список `List<Payment>` в единственный объект `Payment`, содержащий общую сумму всех платежей из `List<Payment>`

Обратите внимание, что в последней строке в функции `groupByCard` используется ссылка на функцию `Payment::combine`. Ссылки на функции подобны ссылкам на методы в Java. Если этот пример не совсем понятен вам, не волнуйтесь – эта книга как раз и предназначена для разъяснения подобных особенностей! Когда дойдете до конца, вы станете экспертом в написании подобного кода.

1.2.3 Максимальное использование абстракций

Как было показано выше, используя *чистые функции*, т. е. функции без побочных эффектов, можно писать более безопасные программы, которые проще тестировать. Объявлять такие функции можно с помощью ключевого слова `fun` или как функции-значения, подобно аргументам методов `groupBy`, `map` или `lower` в листинге 1.5. *Функции-значения* – это такие функции, которые, в отличие от функций `fun`, могут передаваться из одной части программы в другую. Их можно предавать в аргументах другим функциям или возвращать из функций. Как это сделать, вы узнаете в следующих главах.

Но самая важная идея – *абстракция*. Посмотрите на функцию `reduce`. Она принимает аргумент, описывающий операцию, и использует эту операцию, чтобы сократить список до одного значения. Это может быть любая операция, при условии, что она принимает два операнда одного типа.

Рассмотрим список целых чисел. Вы можете написать функцию `sum`, вычисляющую сумму элементов, функцию `product`, вычисляющую произведение элементов, или функцию `min` или `max` для вычисления минимального или максимального элемента списка. Но есть другой путь – использовать для всех этих вычислений функцию `reduce`. Это и есть абстракция. Вы абстрагируете часть, которая является общей для всех операций

в функции reduce, и определяете переменную часть (операцию) в виде аргумента.

Можно пойти еще дальше. Функция reduce является частным случаем более общей функции, которая может давать результат другого типа, отличного от типа элементов списка. Например, ее можно применить к списку символов и создать строку String. Вычисления начинаются с некоторого начального значения (возможно, с пустой строки). В главах 3 и 5 вы узнаете, как использовать эту функцию, называемую fold.

Функция reduce не будет работать с пустым списком. Представьте список целых чисел: чтобы вычислить сумму, нужно иметь хотя бы один элемент в списке. Но если список пустой, что вернуть в этом случае? Логично предположить, что результатом должен быть 0, но этот подход работает только для вычисления суммы. Он не годится для вычисления произведения.

Вернемся к нашей функции groupByCard. Она похожа на бизнес-функцию, которую можно использовать только для группировки платежей по кредитным картам. Но на самом деле это не так! Эту функцию можно использовать для группировки элементов любого списка по любому из их свойств. Эту функцию следовало бы абстрагировать и поместить в класс List, чтобы ее можно было использовать повторно. (Она уже определена в Kotlin-классе List.)

Максимальное использование абстракций позволяет делать программы еще более безопасными, потому что абстрагированная часть будет написана только один раз. Как следствие, после ее полного тестирования исчезнет риск появления новых ошибок в этой части.

В оставшейся части этой книги вы узнаете множество приемов абстрагирования. Например, как абстрагировать циклы, чтобы избавиться от необходимости писать их снова и снова, как абстрагировать параллельные вычисления, чтобы иметь возможность легко переключаться с последовательной обработки на параллельную, просто выбирая функцию в классе List.

Итоги

- Программы можно сделать более безопасными, явно отделяя функции, которые возвращают значения, от эффектов, связанных с взаимодействием с внешним миром.
- Функции легче анализировать и тестировать, потому что их результат предопределен и не зависит от внешнего состояния.
- Максимальное использование абстракций повышает безопасность, удобство сопровождения и простоту тестирования, а также упрощает возможность повторного использования.
- Применение принципов безопасного программирования, таких как неизменяемость и ссылочная прозрачность, защищает программы от случайного использования общего изменяемого состояния, которое часто является источником ошибок в многопоточных окружениях.

2 Функциональное программирование на Kotlin: обзор

Эта глава охватывает следующие темы:

- объявление и инициализация полей и переменных;
- классы и интерфейсы в Kotlin;
- два типа коллекций в Kotlin;
- функции (и управляющие структуры);
- обработка пустых ссылок `null`.

В этой главе я дам краткий обзор языка Kotlin. Я буду предполагать, что вы уже знаете (хотя бы немного) язык Java, поэтому подчеркну различия между этими двумя языками. Цель не в том, чтобы обучить вас языку Kotlin. Для этого есть другие прекрасные книги. Если вам нужен подробный рассказ о Kotlin, я советую прочитать книгу *Kotlin in Action* Дмитрия Жемерова (Dmitry Jemerov) и Светланы Исаковой (Svetlana Isakova), выпущенную издательством Manning (2017)[1].

Здесь вы получите первое представление о том, чего ожидать от Kotlin. Не старайтесь запомнить все, о чем здесь рассказывается. Просто оцените некоторые поразительные особенности языка Kotlin и посмотрите, чем он отличается от Java. В следующих главах я повторно рассмотрю каждое правило безопасного программирования. В оставшейся части этой главы я дам обзор наиболее важных преимуществ Kotlin. Конечно, этот обзор не является исчерпывающим, и далее в книге вы найдете описание дополнительных преимуществ, не упомянутых здесь.

[1] *Жемеров Д., Исакова С.* Kotlin в действии. М.: ДМК Пресс, 2018. ISBN: 978-1-61729-329-0, 978-5-97060-497-7. – *Прим. перев.*

2.1 Поля и переменные в Kotlin

Поля в Kotlin объявляются и инициализируются с использованием следующего синтаксиса:

```
val name: String = "Mickey"
```

Обратите внимание на отличия от языка Java:

- сначала следует ключевое слово `val`, означающее, что ссылка `name` является неизменяемой (соответствует ключевому слову `final` в Java);
- тип (`String`) следует за именем поля и отделяется от него двоеточием (`:`);
- в конце строки отсутствует символ точки с запятой (`;`). Вы можете добавлять точки с запятой, но они не являются обязательными, потому что конец строки имеет тот же смысл. Точки с запятой нужно использовать, только когда несколько инструкций помещается в одну строку, что, впрочем, делать не рекомендуется.

2.1.1 Тип можно опустить для простоты

Предыдущий пример можно упростить до:

```
val name = "Mickey"
```

Kotlin сам может определить тип, заглянув вперед и проверив значение, присваиваемое полю. Эта особенность называется *автоматическим определением типа* и во многих случаях позволяет опустить объявление типа. Но иногда автоматическое определение типа невозможно, например когда тип значения нельзя определить однозначно или когда поле не инициализируется. В таких ситуациях требуется явно объявлять тип.

Однако в общем случае рекомендуется явно указывать тип. Так вы сможете гарантировать соответствие типа, определяемого компилятором Kotlin, ожидаемому типу. Поверьте мне, это не всегда так!

2.1.2 Изменяемые поля

В начале раздела 2.1 я сказал, что ключевое слово `val` определяет неизменяемую ссылку. Значит ли это, что все ссылки всегда неизменяемы? Нет, но в большинстве случаев предпочтение следует отдавать ссылкам `val`. Если ссылку нельзя изменить, ее невозможно будет испортить после инициализации. По этой же причине ссылки должны инициализироваться как можно раньше. (Хотя, как вы увидите далее, Kotlin обычно запрещает использование неинициализированных ссылок. Этим он отличается от Java, где неинициализированным ссылкам автоматически присваивается `null` и допускается их использование.)

Чтобы получить изменяемую ссылку, нужно заменить `val` на `var`, как в следующем примере. Это даст возможность изменить ссылку позже:

```
var name = "Mickey"

...
name = "Donald"
```

Но не забывайте, что желательно избегать использования `var`, если известно, что ссылки не могут измениться. Это поможет сделать программу более безопасной.

2.1.3 Отложенная инициализация

Иногда ключевое слово `var` используется только потому, что требуется отложить инициализацию ссылки, которая не изменяется после инициализации. Причины задержки инициализации могут быть самыми разными. Например, инициализация может потребовать продолжительных вычислений, которые желательно не выполнять, если их результат никогда не будет использован.

Обычно в таких случаях объявляется ссылка `var`, которая получает значение `null`, пока не будет инициализирована действительным значением, которое потом никогда не меняется. Это раздражает, потому что Kotlin различает типы, поддерживающие и не поддерживающие пустое значение `null`. Типы, не поддерживающие значение `null`, намного безопаснее, поскольку устраняют риск исключения `NullPointerException`. Когда значение неизвестно во время объявления и не изменяется после инициализации, необходимость использовать `var` вызывает досаду, потому что вынуждает использовать тип с поддержкой `null`, например:

```
var name: String? = null

...
name = getName()
```

Здесь ссылка имеет тип `String?`, который поддерживает значение `null`, хотя могла бы иметь тип `String`, не поддерживающий `null`. Вы можете использовать определенное значение для представления неинициализированной ссылки, например:

```
var name: String = "NOT_INITIALIZED_YET"

...
name = getValue()
```

Или, если `name` никогда не должно содержать пустого значения, для обозначения неинициализированной ссылки можно использовать пустую строку. Но как бы то ни было, вы вынуждены использовать `var`, даже притом что после инициализации значение ссылки никогда не изменяется. Но Kotlin предлагает более удачное решение:

```
val name: String by lazy { getName() }
```

В этом случае функция `getName()` будет вызвана только один раз, при первом обращении к `name`. Вместо лямбда-выражения можно также использовать ссылку на функцию:

```
val name: String by lazy(::getName)
```

Под словами «при первом обращении к `name`» подразумевается первая операция разыменования ссылки для получения значения, на которое она указывает. Взгляните на следующий пример:

```
fun main(args: Array<String>) {

    val name: String by lazy { getName() }
    println("hey1")
    val name2: String by lazy { name }
    println("hey2")

    println(name)
    println(name2)
    println(name)
    println(name2)
}

fun getName(): String {
    println("computing name...")
    return "Mickey"
}
```

После запуска эта программа выведет:

```
hey1
hey2
computing name...
Mickey
Mickey
Mickey
Mickey
```

Отложенную инициализацию нельзя использовать для изменяемых ссылок. Если вам совершенно необходима изменяемая ссылка с отложенной инициализацией, используйте ключевое слово `lateinit`, которое имеет тот же эффект, но без автоматической инициализации по требованию:

```
lateinit var name: String
...
name = getName()
```

Эта конструкция помогает избежать использования типа с поддержкой `null`. Но не дает никаких преимуществ по сравнению с `lazy`, кроме случаев, когда инициализация должна выполняться извне, например инфраструктурой внедрения зависимостей. Обратите внимание, что всегда следует отдавать предпочтение внедрению зависимостей через аргументы конструктора, потому что этот подход позволяет использовать неизменяемые свойства. Как вы увидите в главе 9, об отложенной инициализации можно говорить очень долго.

2.2 Классы и интерфейсы в Kotlin

Классы в Kotlin объявляются с использованием немного другого синтаксиса, отличного от используемого в Java. Класс `Person` со свойством `name` типа `String` можно объявить в Kotlin так:

```
class Person constructor(name: String) {
    val name: String
```

```
    init {
        this.name = name
    }
}
```

Это объявление эквивалентно следующему коду на Java:

```
public final class Person {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Как видите, версия на Kotlin выглядит компактнее. Обратите внимание на следующие особенности:

- в Kotlin классы являются общедоступными по умолчанию, поэтому нет необходимости использовать слово `public`. Чтобы сделать класс необщедоступным, можно использовать модификаторы `private`, `protected` и `internal`. Модификатор `internal` означает, что класс доступен только внутри модуля, в котором определен. В Java нет эквивалента понятию «закрытый для пакета» в Kotlin (соответствует отсутствию какого-либо модификатора). В отличие от Java действие модификатора `protected` ограничивается расширением (наследованием) классов и не включает классы в том же пакете;
- в Kotlin классы по умолчанию являются конечными (`final`), поэтому в Java эквивалентный класс должен объявляться с модификатором `final`. В Java большинство классов должно объявляться как `final`, но программисты часто забывают это сделать. Kotlin решает эту проблему, делая классы конечными по умолчанию. Чтобы сделать класс Kotlin не конечным, используйте модификатор `open`. Это гораздо безопаснее, потому что классы, открытые для расширения (наследования), должны предусматривать такую возможность;
- конструктор объявляется сразу после имени класса, а его реализация находится в блоке `init`. Этот блок имеет доступ к параметрам конструктора;
- объявлять методы доступа к свойствам не требуется. Они генерируются автоматически во время компиляции;
- в отличие от Java общедоступные (`public`) классы необязательно должны определяться в файле с именем, совпадающим с именем класса. Вы можете назвать файл как угодно. Более того, в одном файле можно определить несколько общедоступных классов. Но это не значит, что вы должны это делать. Определение каждого общедоступного класса в своем файле с именем, совпадающим с именем класса, упростит поиск в будущем.

2.2.1 Еще большее сокращение кода

Код на Kotlin можно сократить еще больше. Во-первых, так как блок `init` в примере выше содержит единственную строку, его можно объединить с объявлением свойства `name`:

```
class Person constructor(name: String) {

    val name: String = name

}
```

Можно сделать еще шаг и объединить объявление конструктора, объявление свойства и инициализацию свойства:

```
class Person constructor(val name: String) {

}
```

Так как блок определения класса получился пустым, его вообще можно удалить. А также можно убрать слово *constructor* (независимо от того, пустой блок или нет):

```
class Person (val name: String)
```

Кроме того, в одном классе можно объявить несколько свойств:

```
class Person(val name: String, val registered: Instant)
```

Как видите, Kotlin избавляет от необходимости писать шаблонный код, благодаря чему листинги получаются более компактными и удобочитаемыми. Имейте в виду, что код пишется один раз, но читается многократно. Когда код легко читается, его также легче поддерживать.

2.2.2 Реализация интерфейса или расширение класса

Если необходимо, чтобы класс реализовал один или несколько интерфейсов или расширял какой-то другой класс, эти интерфейсы или класс нужно перечислить после объявления класса:

```
class Person(val name: String, val registered: Instant) : Serializable,
        Comparable<Person> {

    override fun compareTo(other: Person): Int {
        ...
    }
}
```

При расширении (наследовании) класса используется тот же синтаксис. Разница лишь в том, что за именем наследуемого класса следуют имена параметров в круглых скобках:

```
class Member(name: String, registered: Instant) : Person(name, registered)
```

Но не забывайте, что классы являются конечными (`final`) по умолчанию. Чтобы этот пример скомпилировался, расширяемый класс должен быть объявлен открытым (`open`), т. е. *открытым для расширения*:

```
open class Person(val name: String, val registered: Instant)
```

Хорошей практикой считается разрешать расширение только для классов, специально предназначенных для него. Как видите, Kotlin, в отличие от Java, пытается реализовать этот принцип, запрещая расширять классы, явно не предназначенные для расширения.

2.2.3 Создание экземпляра класса

При создании экземпляра класса Kotlin избавляет от необходимости повторно указывать тип, хотя и в меньшей степени. Например, вместо

```
final Person person = new Person("Bob", Instant.now());
```

можно вызвать конструктор как обычную функцию (которой он, впрочем, и является):

```
val person = Person("Bob", Instant.now())
```

В этом есть определенный смысл, потому что конструктор `Person` является функцией, преобразующей множество всех возможных пар, состоящих из строки и даты регистрации, в множество всех возможных персон. Теперь посмотрим, как Kotlin справляется с перегрузкой этих конструкторов.

2.2.4 Перегрузка конструкторов

Иногда свойства могут быть необязательными и иметь значения по умолчанию. В предыдущем примере можно считать, что датой регистрации по умолчанию является дата создания экземпляра. В Java для этого нужно написать два конструктора, как показано в листинге 2.1.

Листинг 2.1. Типичный Java-объект с необязательным свойством

```
public final class Person {

    private final String name;

    private final Instant registered;

    public Person(String name, Instant registered) {
        this.name = name;
        this.registered = registered;
    }

    public Person(String name) {
        this(name, Instant.now());
    }

    public String getName() {
        return name;
    }

    public Instant getRegistered() {
        return registered;
    }
}
```

В Kotlin тот же результат можно получить, добавив значение по умолчанию после имени свойства:

```
class Person(val name: String, val registered: Instant = Instant.now())
```

Также поддерживается более традиционный способ переопределения конструкторов:

```
class Person(val name: String, val registered: Instant = Instant.now())
{
    constructor(name: Name) : this(name.toString()) {
        // сюда можно добавить реализацию дополнительного конструктора
    }
}
```


Так же как в Java, если явно не объявить конструктор без аргументов, он будет создан автоматически.

Приватные конструкторы и свойства

Так же как в Java, конструктор можно объявить приватным и запретить создавать экземпляры во внешнем коде:

```
class Person private constructor(val name: String)
```

Но в отличие от Java приватные конструкторы не нужны для предотвращения создания экземпляров служебных классов, содержащих только статические члены. Kotlin помещает статические члены на уровне пакета вне любого класса.

Методы доступа и свойства

В Java считается плохой практикой открывать прямой доступ к свойствам объекта. Вместо этого определяются методы доступа к значениям свойств. Эти методы доступа обычно называются методами *чтения* и *записи*. В следующем примере вызывается метод чтения имени человека:

```
val person = Person("Bob")
...
println(person.name) // Вызов метода чтения
```

Даже притом что код выглядит так, будто он обращается к полю `name` напрямую, на самом деле используется сгенерированный метод чтения. Он имеет то же имя, что и поле, и его вызов не должен сопровождаться круглыми скобками.

Обратите внимание, что вызов метода `println` экземпляра `System.out` выглядит гораздо проще, чем в Java. Не то чтобы это имело большое значение, потому что ваши программы, вероятно, никогда не будут выводить информацию в консоль, но эту особенность стоит отметить.

2.2.5 Создание методов equals и hashCode

Если класс `Person` представляет данные, ему, вероятно, понадобятся методы `hashCode` и `equals`. Определение этих методов в Java – утомительное занятие, чреватое ошибками. К счастью, хорошая интегрированная сре-

да разработки на Java (например, IntelliJ) сгенерирует их для вас. В листинге 2.2 показан код, который генерирует IntelliJ.

Листинг 2.2. Объект данных в Java, сгенерированный в IntelliJ

```
public final class Person {

    private final String name;

    private final Instant registered;

    public Person(String name, Instant registered) {
        this.name = name;
        this.registered = registered;
    }

    public Person(String name) {
        this(name, Instant.now());
    }

    public String getName() {
        return name;
    }

    public Instant getRegistered() {
        return registered;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(name, person.name) &&
                Objects.equals(registered, person.registered);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, registered);
    }
}
```

Наличие возможности генерировать такой код в среде разработки избавляет от утомительной и рутинной работы, но, так или иначе, вам придется жить с этим ужасным фрагментом кода, не отличающимся удобочитаемостью. Хуже того – вам придется поддерживать его! Если позже вы добавите новое свойство, которое должно учитываться методами `hashCode` и `equals`, вам придется удалить эти два метода и создать их заново. Kotlin позволяет решить эту задачу намного проще:

```
data class Person(val name: String, val registered: Instant = Instant.now())
```

Да, достаточно добавить ключевое слово `data` перед объявлением класса. Функции `hashCode` и `equals` генерируются не этапе компиляции. Вы их никогда не увидите, хотя можете пользоваться ими как обычными функ-

циями. Кроме того, Kotlin генерирует функцию `toString`, преобразующую экземпляр класса в удобочитаемый для человека вид, и функцию `copy`, позволяющую копировать объект со всеми его свойствами. Kotlin также генерирует дополнительные функции `componentN`, позволяющие получить доступ к каждому свойству класса, как будет показано в следующем разделе.

2.2.6 Деструктуризация объектов данных

В каждом классе данных с *n* свойствами автоматически генерируются функции `component1` – `componentN`, позволяющие обращаться к свойствам в том порядке, в каком они определены в классе. Основное назначение этих функций – деструктуризация объектов для более простого доступа к их свойствам:

```
data class Person(val name: String, val registered: Instant = Instant.now())

fun show(persons: List<Person>) {
    for ((name, date) in persons)
        println(name + "'s registration date: " + date)
}

fun main(args: Array<String>) {
    val persons = listOf(Person("Mike"), Person("Paul"))
    show(persons)
}
```

Функцию `show` можно переписать иначе:

```
fun show(persons: List<Person>) {
    for (person in persons)
        println(person.component1()
            + "'s registration date: " + person.component2())
}
```

Как видите, поддержка деструктуризации делает код чище и короче за счет отсутствия необходимости разыменовывать ссылки на свойства объекта при каждом обращении к ним.

2.2.7 Реализация статических членов в Kotlin

Классы в Kotlin не имеют статических членов. Чтобы сымитировать их, нужно создать специальную конструкцию, называемую *объектом-компаньоном*, или *сопутствующим объектом*:

```
data class Person(val name: String,
                  val registered: Instant = Instant.now()) {
    companion object {
        fun create(xml: String): Person {
            TODO("Реализовать создание " +
                 "экземпляра Person из строки с кодом XML")
        }
    }
}
```

Функцию `create` можно вызвать относительно вмещающего класса, подобно статическим методам в Java:

```
Person.create(someXmlString)
```

Ее также можно вызвать относительно объекта-компаньона, но в этом нет необходимости:

```
Person.Companion.create(someXmlString)
```

С другой стороны, чтобы вызвать эту функцию из кода на Java, обязательно нужно сослаться на объект-компаньон. Чтобы получить возможность вызывать его как статический метод класса, достаточно добавить аннотацию `@JvmStatic`. Дополнительную информацию о вызове функций Kotlin из кода на Java (и наоборот) вы найдете в приложении А.

Кстати, обратите внимание, что Kotlin предлагает функцию `TODO`, которая делает код намного более согласованным. Эта функция генерирует исключение во время выполнения, напоминая вам о работе, которую вы должны были сделать!

2.2.8 Синглтоны

Часто бывает необходимо создать единственный экземпляр данного класса. Такой экземпляр называется *синглтоном* (одиночкой). Шаблон проектирования «Синглтон» – это метод, гарантирующий невозможность создания более одного экземпляра класса. В Java этот шаблон имеет сложную реализацию, потому что трудно гарантировать создание не более одного экземпляра. В Kotlin синглтон легко создать, заменив слово `class` словом `object`:

```
object MyWindowAdapter: WindowAdapter() {
    override fun windowClosed(e: WindowEvent?) {
        TODO("не реализовано")
    }
}
```

Объект `object` не имеет конструкторов. Если в объекте есть свойства, они должны быть инициализированы или объявлены абстрактными.

2.2.9 Предотвращение создания экземпляров служебных классов

В Java служебные классы обычно содержат только статические методы. В таких случаях желательно запретить создание экземпляров служебного класса. В Java для этого конструкторы объявляются приватными (`private`). В Kotlin тоже можно объявить конструкторы приватными, но в этом нет особого смысла, потому что есть возможность создавать функции вне классов на уровне пакета. Для этого создайте файл с любым именем и объявите в нем пакет. Затем определите функции, не включая их в классы:

```
package com.acme.util

fun create(xml: String): Person {
    ...
}
```

Вызвать такую функцию можно по ее полному имени:

```
val person = com.acme.util.create(someXmlString)
```

А можно импортировать пакет и использовать только имя функции:

```
import com.acme.util.*
val person = create(someXmlString)
```

Поскольку код на Kotlin выполняется в JVM, должен иметься некоторый способ вызова функций уровня пакета из кода на Java. Этот способ описан в приложении А.

2.3 В Kotlin нет элементарных типов

В Kotlin отсутствуют свои элементарные типы данных, по крайней мере, на уровне программиста. Вместо этого для ускорения вычислений используются элементарные типы Java. Но вы, как программист, будете манипулировать только объектами. Класс объекта для целых чисел отличается от представления объекта целых чисел в Java. Вместо `Integer` вы будете использовать класс `Int`. Другие числовые и логические типы имеют те же имена, что и в Java. Как и в Java, вы можете использовать подчеркивания в числах, а также:

- длинные целые обозначать завершающим символом `L`, а числа с плавающей точкой – символом `F`;
- числа с плавающей точкой двойной точности обозначать десятичной точкой, например `2.0` или `.9`;
- шестнадцатеричные числа обозначать префиксом `0x`, например `0xBE_24_1C_D3`;
- двоичные литералы обозначать префиксом `0b`: `0b01101101_11001010_10010011_11110100`.

Отсутствие элементарных типов упрощает программирование, так как не требуется использовать специальные функции, как в Java, и поддерживаются коллекции числовых и логических значений, не требующие преобразовывать элементарные значения в экземпляры классов и обратно.

2.4 Два типа коллекций в Kotlin

Коллекции в Kotlin основаны на коллекциях Java, но Kotlin добавляет в них дополнительные возможности. Особенно важно отметить, что в Kotlin всего два типа коллекций: изменяемые и неизменяемые. Обычно первое, с чем вы начинаете экспериментировать, – это создание кол-

лекций с использованием специализированных функций. Следующий код создает неизменяемый список, содержащий целые числа 1, 2 и 3:

```
val list = listOf(1, 2, 3)
```

По умолчанию коллекции в Kotlin являются неизменяемыми.

ПРИМЕЧАНИЕ На самом деле неизменяемые коллекции в Kotlin не являются по-настоящему неизменяемыми. Это просто коллекции, которые нельзя изменить. Как следствие, некоторые предпочитают называть их *коллекциями только для чтения*, что тоже не совсем верно, потому что они доступны не только для чтения. Но давайте пока не будем об этом беспокоиться. В главе 5 вы научитесь создавать настоящие неизменяемые коллекции.

Функция `listOf` – это функция уровня пакета, т. е. она не является частью класса или интерфейса. Она определена в пакете `kotlin.collections`, и ее можно импортировать, используя следующий синтаксис:

```
import kotlin.collections.listOf
```

Однако ее не требуется импортировать явно, потому что все функции из этого пакета импортируются неявно, как если бы вы добавили инструкцию:

```
import kotlin.collections.*
```

Аналогичным образом импортируются многие другие пакеты. Это напоминает автоматическое импортирование пакета `java.lang` в Java.

Имейте в виду, что неизменяемость не запрещает выполнять операции со списками, например:

```
val list1 = listOf(1, 2, 3)
val list2 = list1 + 4
val list3 = list1 + list2
println(list1)
println(list2)
println(list3)
```

Этот код создает список с целыми числами 1, 2 и 3. Затем создает новый список, добавляя элемент в первый список. Наконец, он создает еще один список, объединяя два имеющихся. Как показывают результаты, ни один из списков не изменился:

```
[1, 2, 3] [1, 2, 3, 4] [1, 2, 3, 1, 2, 3, 4]
```

Если вам понадобится изменяемая коллекция, вы должны явно потребовать это:

```
val list1 = mutableListOf(1, 2, 3)
val list2 = list1.add(4)
val list3 = list1.addAll(list1)
println(list1)
println(list2)
println(list3)
```

Как показано ниже, результат получился совсем другим:

```
[1, 2, 3, 4, 1, 2, 3, 4]
true
true
```

Здесь все операции применяются к первому списку и имеют кумулятивный эффект. Благодаря механизму автоматического определения типа здесь не возникло ошибки при присваивании результата операции (с типом `Boolean`) ссылкам. Kotlin автоматически назначил этим ссылкам тип `Boolean`. Это хорошая причина явно определять ожидаемый тип. Это предотвратит компиляцию следующего кода:

```
val list1: List<Int> = mutableListOf(1, 2, 3)
val list2: List<Int> = list1.add(1) // <-- Ошибка компиляции
val list3: List<Int> = list1.addAll(list2) // <-- Ошибка компиляции
println(list1)
println(list2)
println(list3)
```

Оператор `+` – это *инфиксная* функция-расширение с именем `plus`. Она объявлена в интерфейсе `Collection`, который реализует класс `List`. Она транслируется в статическую функцию, создающую новый список из своих аргументов. Оператор `+` можно использовать с изменяемыми списками и получать тот же результат, как если бы списки были неизменяемыми, т. е. исходные списки останутся без изменений.

Те из вас, кто знает, что хранение неизменяемых структур данных обычно реализуется с использованием общих данных, могут быть разочарованы, узнав, что неизменяемые списки в Kotlin не имеют общих данных. (Они могут иметь общие элементы, но *не* данные в списке.) Добавление элемента в список означает создание совершенно нового списка. В главе 5 вы узнаете, как создать свой неизменяемый список, использующий прием с общими данными для экономии памяти и повышения производительности при выполнении некоторых операций.

2.5 Пакеты в Kotlin

Вы уже видели, что функции можно объявлять на уровне пакетов, – это важное отличие пакетов в Kotlin от пакетов в Java. Еще одна отличительная черта пакетов в Kotlin – они не должны соответствовать структуре каталогов.

По аналогии с классами, которые не должны определяться в одноименных файлах, организация пакетов необязательна в структуре каталогов. Пакеты являются лишь идентификаторами. Отсутствует понятие *подпакета* (т. е. пакета, вложенного в другой пакет). Имена файлов не имеют значения для компилятора (достаточно только, чтобы они имели расширение *.kt*). Однако я все же рекомендую придерживаться соглашений Java в отношении соответствий между пакетами и каталогами по двум причинам:

1. При использовании в программе файлов с кодом на Java и Kotlin вам придется поместить файлы Java в каталоги с именами, соответствующими именам пакетов. То же придется проделать с файлами Kotlin.
2. Даже если программа состоит только из файлов с кодом на Kotlin, найти исходный файл будет намного проще, если следовать соглашениям Java: достаточно лишь взглянуть на имя пакета (например, в инструкции импортирования) и преобразовать его в путь к файлу.

2.6 Видимость в Kotlin

Правила видимости в Kotlin отличаются от правил в Java. Функции и свойства можно определять не только в классах, но и на уровне пакета. Все элементы, определяемые на уровне пакета, являются общедоступными по умолчанию. Если элемент объявлен приватным, он будет доступен только в этом файле.

Также элемент можно объявить *внутренним* (`internal`), в этом случае он будет доступен только внутри модуля. *Модуль* – это набор файлов, компилируемых вместе, такой как:

- проект Maven;
- набор исходных файлов Gradle;
- модуль IntelliJ;
- проект Eclipse;
- группа файлов, компилируемых одним заданием Ant.

Модуль организуется для упаковки кода в один JAR-файл. Модуль может включать несколько пакетов, а один пакет может распространяться на несколько модулей.

Классы и интерфейсы по умолчанию общедоступны. К ним также можно применить один из следующих модификаторов видимости:

- `private`;
- `protected`;
- `internal`;
- `public`.

Приватный (`private`) элемент доступен только внутри класса, в котором он определен. В Kotlin, в отличие от Java, приватные члены *внутреннего* класса (определенного внутри другого класса) недоступны для внешнего (вмещающего) класса.

Конструкторы классов общедоступны по умолчанию, но к ним тоже можно применять модификаторы области видимости, как показано здесь:

```
class Person private constructor (val name: String,
                                  val registered: Instant)
```

В отличие от Java в Kotlin не поддерживается видимость, *приватная для пакета* (в Java используется по умолчанию). С другой стороны, в Kotlin имеется специфическая внутренняя (`internal`) область видимо-

сти. Элемент с модификатором `internal` доступен из любой точки в том же модуле. (Обратите внимание, что в Gradle код в тестовом наборе имеет доступ к `internal`-элементам в соответствующем основном наборе исходного кода.)

2.7 Функции в Kotlin

Функция в языке Kotlin эквивалентна методу в Java. Это название не имеет ничего общего с понятием функции в математике или в функциональном программировании.

> **ПРИМЕЧАНИЕ** Слово *функция* в Kotlin используется даже для обозначения элементов, которые не являются истинными функциями или вообще не являются функциями. В следующей главе вы узнаете, что такое истинная функция. В оставшейся части этой главы я буду использовать слово «функция» в том смысле, в каком оно используется в языке Kotlin (т. е. эквивалент Java-метода).

2.7.1 Объявление функций

Подобно свойствам, в языке Kotlin функции можно объявлять на уровне пакета или внутри классов и объектов. Объявление функции начинается с ключевого слова `fun`:

```
fun add(a: Int, b: Int): Int {
    return a + b
}
```

Если тело функции можно уместить в одну строку, тогда блок, ограничиваемый фигурными скобками, можно заменить следующим синтаксисом:

```
fun add(a: Int, b: Int): Int = a + b
```

Такой синтаксис называют *синтаксисом выражений*. Он позволяет опустить имя типа возвращаемого значения в теле:

```
fun add(a: Int, b: Int) = a + b
```

Имейте в виду, что смешивание двух синтаксисов может приводить к неожиданным результатам. Например, следующий код вполне допустим:

```
fun add(a: Int, b: Int) = {
    a + b
}
```

но тип возвращаемого значения почти наверняка будет отличаться от того, что вы могли бы ожидать. Подробнее об этом я расскажу в следующей главе. В таких случаях явное объявление типа помогает компилятору предупредить вас об ошибке (даже притом что `return` отсутствует в этом примере).

2.7.2 Локальные функции

Как известно, функции можно определять внутри классов и объектов. Но кроме того, их можно также объявлять внутри других функций. Ниже приводится пример сложной функции, возвращающей число делителей целого числа. На данный момент вам не нужно понимать, как работает эта функция. Мы поговорим об этом в следующих главах. Но обратите внимание, что функция `sumOfPrimes` содержит определение функции `isPrime`:

```
fun sumOfPrimes(limit: Int): Long {

    val seq: Sequence<Long> = sequenceOf(2L) +
            generateSequence(3L, {
                it + 2
            }).takeWhile{
                it < limit
            }

    fun isPrime(n: Long): Boolean =
            seq.takeWhile {
                it * it <= n
            }.all {
                n % it != 0L
            }

    return seq.filter(::isPrime).sum()
}
```

Функцию `isPrime` нельзя определить вне функции `sumOfPrimes`, потому что она *замкнута на* переменной `seq`. Такая конструкция называется *замыканием*. Как вы узнаете в следующей главе, замыкания превращают замыкающие переменные в аргументы функции. Этот код эквивалентен следующему:

```
fun sumOfPrimes(limit: Int): Long {
    val seq: Sequence<Long> = sequenceOf(2L) +
            generateSequence(3L, {
                it + 2
            }).takeWhile{
                it < limit
            }

    return seq.filter {
        x -> isPrime(x, seq)
    }.sum()
}

fun isPrime(n: Long, seq: Sequence<Long>): Boolean =
        seq.takeWhile {
            it * it <= n
        }.all {
            n % it != 0L
        }
```

Замыкание позволяет для вызова `isPrime` использовать ссылку на функцию (эквивалентную ссылке на метод в Java) вместо лямбда-выражения. С другой стороны, это делает функцию `isPrime` непригодной для использования вне функции `sumOfPrimes`.

Здесь функция `isPrime`, которая принимает параметры (`n: Long, seq: Sequence<Long>`), вероятно, будет бесполезна вне `sumOfPrimes`. Поэтому `isPrime` лучше объявить как локальную функцию, позволяющую использовать замыкание и ссылку на функцию.

2.7.3 Переопределение функций

При расширении класса или реализации интерфейса часто требуется переопределять функции. В отличие от Java переопределение должно явно обозначаться ключевым словом `override`:

```
override fun toString() = ...
```

Это одна из редких ситуаций, когда синтаксис Kotlin оказывается более многословным, чем синтаксис Java. Но такое многословие обеспечивает большую безопасность программ, предотвращая случайное переопределение функций. (В Java тот же эффект можно получить с помощью аннотации `@Override`.)

2.7.4 Функции-расширения

Функции-расширения – это функции, которые можно применять к объектам, как если бы они были функциями экземпляра соответствующего класса. Этот механизм широко используется в Kotlin. Представьте, что вы решили написать функцию `length`, возвращающую длину списка:

```
fun <T> length(list: List<T>) = list.size
```

Да, смысла в такой функции немного, но это всего лишь пример. В Kotlin такую функцию можно объявить так:

```
fun <T> List<T>.length() = this.size
```

Функция `length` в этом примере – это функция-расширение, добавляющая в интерфейс `List` дополнительную функцию, которую можно использовать, как если бы она была функцией экземпляра:

```
fun <T> List<T>.length() = this.size

val ints = listOf(1, 2, 3, 4, 5, 6, 7)

val listLength = ints.length()
```

В отличие от функции `size`, которую можно вызывать, используя синтаксис обращения к свойству, при вызове `length()` обязательно следует использовать круглые скобки. Кроме того, такие функции-расширения нельзя вызывать как методы экземпляра из кода на Java – они должны вызываться как статические методы. В Kotlin функции можно добавлять даже в параметризованные классы, например:

```
fun List<Int>.product(): Int = this.fold(1) { a, b -> a * b }

val ints = listOf(1, 2, 3, 4, 5, 6, 7)

val product = ints.product()
```

Как видите, функции-расширения похожи на статические функции в Java. Но их можно применять к экземплярам, а не к классам, что позволяет включать их в цепочки вызовов.

2.7.5 Лямбда-выражения

Как и в Java, *лямбда-выражения* – это анонимные функции, т. е. реализации функций, на которые нельзя сослаться по имени. Однако синтаксис лямбда-выражений в Kotlin несколько отличается от синтаксиса в Java. Лямбда-выражения должны заключаться в фигурные скобки, как в следующем примере:

```
fun triple(list: List<Int>): List<Int> = list.map({ a -> a * 3 })
```

Когда лямбда-выражение является последним аргументом функции, его можно вынести за круглые скобки:

```
fun triple(list: List<Int>): List<Int> = list.map { a -> a * 3 }

fun product(list: List<Int>): Int = list.fold(1) { a, b -> a * b }
```

В примере с функцией `map` лямбда-выражение является не только последним, но и единственным аргументом, поэтому круглые скобки можно совсем убрать. Как видно во втором примере, аргументы лямбда-выражения не требуется заключать в круглые скобки. Фактически их нельзя заключить в круглые скобки, потому что иначе изменится их смысл. Например, следующий код не скомпилируется:

```
fun List<Int>.product(): Int = this.fold(1) { (a, b) -> a * b }
```

Типы параметров в лямбда-выражениях

Компилятор Kotlin автоматически определяет типы параметров в лямбда-выражениях, но часто ошибается в своих выводах. Это обусловлено стремлением ускорить компиляцию. Когда для правильного определения типа может потребоваться слишком много времени, компилятор может ошибиться, и тогда желательно, чтобы вы сами указали типы параметров. Вот как это делается:

```
fun List<Int>.product(): Int = this.fold(1) { a: Int, b: Int -> a * b }
```

Несмотря на то что часто можно не указывать типы параметров и положиться на автоматическое определение, скоро вы заметите еще одну выгоду. Если вы указали типы вручную и ваш код не компилируется, компилятор (или IDE) сообщит вам, как отличаются указанные вами типы от определенных автоматически.

Многострочные лямбда-выражения

Код лямбда-выражения можно расположить на нескольких строках, как в следующем примере:

```
fun List<Int>.product(): Int = this.fold(1) { a, b ->
        val result = a * b
        result
}
```

Это лямбда-выражение возвращает значение последнего выражения в последней строке. Конечно, можно было бы использовать ключевое слово `return`, но его смело можно опустить.

Упрощенный синтаксис лямбда-выражений

Для лямбда-выражений с единственным параметром Kotlin предлагает упрощенный синтаксис. Этот параметр неявно получает имя `it`. Предыдущий пример можно переписать:

```
fun triple(list: List<Int>): List<Int> = list.map { it * 3 }
```

Но не всегда разумно использовать этот синтаксис. Обычно это упрощение лучше использовать, когда лямбда-выражения не вложены; иначе вам будет трудно догадаться, что есть что! В любом случае не пренебрегайте возможностью записывать лямбда-выражения в несколько строк:

```
fun triple(list: List<Int>): List<Int> = list.map {
    it * 3
}

fun triple(list: List<Int>): List<Int> = list.map { a ->
    a * 3
}
```

Добавляя разрывы строк после стрелки (`->`), вы упростите определение используемого синтаксиса.

Лямбда-выражения в замыканиях

Так же как Java, Kotlin позволяет замыкать лямбда-выражения на переменных в объемлющей области видимости:

```
val multiplier = 3

fun multiplyAll(list: List<Int>): List<Int> = list.map {
    it * multiplier
}
```

Помните, что замыкания обычно следует заменять аргументами функций, что сделает ваш код более безопасным. Например:

```
fun multiplyAll(list: List<Int>, multiplier: int): List<Int> = list.map {
    it * multiplier
}
```

Используйте замыкания, только когда они охватывают узкую область (например, в функциях, определяемых внутри других функций). В таких случаях вполне безопасно замкнуть аргументы или временный результат вмещающей функции. В отличие от Java Kotlin позволяет замыкать изменяемые переменные. Но если вы хотите писать более безопасные программы, избегайте использования изменяемых ссылок.

2.8 Пустое значение null в Kotlin

Kotlin обрабатывает пустые ссылки `null` особым образом. Как будет показано в главе 6, пустые ссылки чаще других оказываются источником ошибок в компьютерных программах. Kotlin пытается решить эту проблему, заставляя явно обрабатывать пустые ссылки.

Kotlin различает типы с поддержкой и без поддержки значения `null`. Возьмем, например, целые числа. Для представления целых чисел в диапазоне [–2 147 483 648 ... 2 147 483 647] в Kotlin используется тип `Int`. Ссылка этого типа может иметь любое значение из данного диапазона, но не более того. В частности, она не может иметь значения `null`, потому что `null` не входит в этот диапазон. С другой стороны, в Kotlin есть тип `Int?`, который может принимать любое значение из диапазона, указанного выше, плюс значение `null`.

Тип `Int` называют типом, не поддерживающим `null`, а тип `Int?` – поддерживающим. Kotlin использует такое разделение для всех типов: на поддерживающие `null` (с окончанием `?`) и не поддерживающие `null`. Самое интересное, что любой тип, не поддерживающий `null`, является дочерним типом соответствующего типа, поддерживающего `null`, поэтому следующий код допустим:

```
val x: Int = 3
val y: Int? = x
```

А этот – нет:

```
val x: Int? = 3
val y: Int = x
```

2.8.1 Приемы работы с типами, поддерживающими null

При использовании только типов, не поддерживающих `null`, программа никогда не возбудит исключения `NullPointerException`. Напротив, при использовании типов с поддержкой `null` исключение `NullPointException` не кажется невероятным. Kotlin заставляет явно обработать исключение или принять на себя полную ответственность. Следующий код не будет компилироваться:

```
val s: String? = someFunctionReturningAStringThatCanBeNull()
val l = s.length
```

Оператор точки (`.`), который также называют оператором *разыменования*, нельзя использовать здесь, потому что он может вызвать NPE (`NullPointerException`). Вместо этого вы должны написать такой код:

```
val s: String? = someFunctionReturningAStringThatCanBeNull()
val l = if (s != null) s.length else null
```

Kotlin упрощает такие варианты использования, предлагая оператор *безопасного вызова*, `?.`:

```
val s: String? = someFunctionReturningAStringThatCanBeNull()
val l = s?.length
```

Обратите внимание, что Kotlin определит тип значения `l` как `Int?`. Этот синтаксис, однако, практичнее использовать в цепочках вызовов:

```
val city: City? = map[companyName]?.manager?.address?.city
```

В таких ситуациях `companyName` может отсутствовать в ассоциативном массиве `map` или у нее может не быть управляющего `manager`, или у управляющего отсутствует адрес `address`, или в адресе не указан город `city`. Защиту от пустых ссылок можно также организовать с помощью вложенных конструкций `if...else`, но синтаксис Kotlin намного удобнее. Он также намного компактнее эквивалентного решения на Java:

```
City city = Optional.ofNullable(map.get(companyName))
                    .flatMap(Company::getManager)
                    .flatMap(Employee::getAddress)
                    .flatMap(Address::getCity)
                    .getOrElse(null);
```

Как я уже говорил, Kotlin позволяет также принять на себя всю ответственность за происходящее, т. е. за возможные исключения NPE:

```
val city: City? = map[companyName]!!.manager!!.address!!.city
```

В этом фрагменте, если какой-то элемент будет иметь значение `null` (кроме `city`), будет возбуждено исключение NPE.

2.8.2 Оператор Элвис и значение по умолчанию

Иногда вместо `null` желательно использовать определенное значение по умолчанию. В Java такую возможность дает метод `Optional.getOrElse()`. Но в Kotlin можно использовать оператор Элвис:

```
val city: City = map[company]?.manager?.address?.city ?: City.UNKOWN
```

Здесь, если какой-либо элемент имеет значение `null`, вместо него будет использовано специальное значение по умолчанию, определяемое оператором `?:`, который иногда называют оператором Элвис. Если вам интересно, откуда взялось такое название, поверните оператор на 90° вправо. (Если вы достаточно молоды, то можете повернуть голову на 90° влево.)

2.9 Поток выполнения программы и управляющие структуры

Управляющие структуры – это элементы программы, управляющие потоком выполнения программы. Они составляют основу императивного программирования, выражая, как должны выполняться вычисления. Как вы увидите в следующих главах, управляющие структуры являются основным источником ошибок в компьютерных программах, поэтому их следует избегать по мере возможности. Вы увидите, что программы получаются гораздо более безопасными, если из них полностью исключить структуры управления.

Прежде всего, можно полностью отказаться от управления потоком выполнения и заменить управляющие структуры выражениями и функциями. В отличие от некоторых языков, специально разработанных для безопасного программирования, в Kotlin есть и управляющие структуры (такие же, как в Java), и функции, которые можно использовать для замены этих структур. Но некоторые из структур управления на самом деле отличаются от своих версий в языке Java.

2.9.1 Условная конструкция

В Java конструкция `if...else` является управляющей структурой. Она проверяет условие и направляет поток выполнения программы в один из двух блоков инструкций в зависимости от выполнения условия. Вот простой пример на Java:

```
int a = ...
int b = ...

if (a < b) {
    System.println("a is smaller than b");
} else {
    System.println("a is not smaller than b");
}
```

В Kotlin конструкция `if...else` является выражением, возвращающим значение. Она имеет ту же форму, что и в Java, но возвращает результат первого блока, если условие выполняется. В противном случае возвращает результат второго блока:

```
val a: Int = ...
val b: Int = ...

val s = if (a < b) {
    "a is smaller than b"
} else {
    "a is not smaller than b"
}
println(s)
```

Так же как в Java, фигурные скобки можно опустить, если соответствующий блок состоит из единственной инструкции:

```
val a: Int = ...
val b: Int = ...

val s = if (a < b)
          "a is smaller than b"
        else
          "a is not smaller than b"

println(s)
```

В Java этот прием считается плохой практикой, но не в Kotlin, где добавление строки в ветвь без добавления фигурных скобок приведет к ошибке компиляции. Однако, когда ветви в выражении `if...else` со-

держат несколько инструкций, они должны заключаться в блоки (так же, как в Java). При этом ключевое слово `return` не должно использоваться:

```
val a: Int = 6
val b: Int = 5

val percent = if (b != 0) {
                  val temp = a / b
                  temp * 100
              } else {
                  0
              }
```

Здесь первая ветвь включает две инструкции, поэтому она заключена в блок. Значением блока служит значение, возвращаемое последней инструкцией в блоке. Фигурные скобки, ограничивающие второй блок, можно опустить, потому что этот блок содержит только одну инструкцию. Для единообразия и удобочитаемости кода программисты обычно заключают в фигурные скобки обе ветви, даже если это необходимо только в одной из них.

В блок `if` можно добавить эффекты и сделать эту функцию более похожей на управляющую структуру в Java. Но старайтесь избегать этой практики. В предыдущей главе я говорил, что эффекты следует использовать только в ограниченных «небезопасных» частях программы. За их пределами `if...else` следует использовать только как выражение, без побочных эффектов.

2.9.2 Использование конструкций с несколькими условиями

Когда требуется организовать выполнение более двух условий ветвей, в Java используется структура `switch...case`. Она позволяет сравнить переменную с целочисленными значениями, перечислениями или строками:

```
// Код на Java
String country = ...
String capital;

switch(country) {
  case "Australia":
    capital = "Canberra";
    break;
  case "Bolivia":
    capital = "Sucre";
    break;
  case "Brazil":
    capital = "Brasilia";
    break;
  default:
    capital = "Unknown";
}
```

В Kotlin для той же цели используется конструкция `when`, которая является выражением, а не управляющей структурой:

```
val country = ...

val capital = when (country) {
    "Australia" -> "Canberra"
    "Bolivia"   -> "Sucre"
    "Brazil"    -> "Brasilia"
    else        -> "Unknown"
}
```

Вместо единственного значения справа от каждой стрелки можно использовать многострочный блок; в этом случае возвращаемым значением будет результат последней инструкции в блоке. В `when` не требуется использовать `break` в конце каждого варианта. Как и в случае с выражением `if...then`, в блоках не должны производиться эффекты.

Важно отметить, что Kotlin не позволит использовать неисчерпывающее выражение `when`. В нем должны быть перечислены все возможные варианты. При использовании перечисления для каждого возможного значения можно определить свой вариант. Если позднее в перечисление добавится новое значение и вы забудете добавить его обработку, код перестанет компилироваться. (Вариант `else` – самый простой способ обработки всех вариантов, которые не определены явно.) Вы также можете использовать конструкцию `when` с другим синтаксисом:

```
val country = ...

val capital = when {
    tired                  -> "Check for yourself"
    country == "Australia" -> "Canberra"
    country == "Bolivia"   -> "Sucre"
    country == "Brazil"    -> "Brasília"
    else                   -> "Unknown"
}
```

Такой способ использования конструкции `when` может пригодиться, если не все условия зависят от одного и того же аргумента. В предыдущем примере `tired` – это логическое значение, инициализированное где-то еще. Если условие выполняется, возвращается значение справа от соответствующей стрелки. Условия проверяются в порядке их перечисления. Первое условие, которое выполняется, определяет значение всего выражения `when`.

2.9.3 Циклы

В Java используется несколько видов циклов:

- с индексами для перебора диапазонов числовых значений;
- для перебора значений в коллекциях;
- для повторения итераций, пока не выполнится некоторое условие.

В Kotlin тоже есть аналогичные управляющие структуры. Цикл `while` в Kotlin похож на свой аналог в Java. Эта управляющая структура используется, чтобы описать, *как* должна работать программа. Но если ваше описание окажется неточным, программа будет работать неправильно,

даже если намерение было правильным. Вот почему безопаснее заменить управляющие структуры функциями, которые говорят, *что* нужно делать (а не как это делать).

Циклы с индексами тоже имеются в Kotlin, хотя на самом деле цикл с индексами выполняет перебор индексов из коллекции:

```
for(i in 0 until 10 step 2) {
    println(i)
}
```

Этот цикл эквивалентен следующему:

```
val range = 0 until 10 step 2
for (i in range) println(i)
```

Эта управляющая структура выполняет итерации по диапазону индексов. Kotlin предлагает три основные функции для создания диапазонов. `until` конструирует восходящий диапазон с шагом `step`, равным `1` по умолчанию, если он не задан явно. Начальное значение включается в диапазон, а конечное значение – нет. `until` и `step` – это функции аргумента `Int`, и они могут использоваться, как показано ниже:

```
for (i in 0.until(10).step(2)) println(i)
```

Подобно многим другим функциям, их можно использовать в инфиксной нотации:

```
for (i in 0 until 10 step 2) println(i)
```

Двумя другими полезными функциями для создания диапазонов являются оператор `..` (две точки), похожий на `until`, но включающий верхнюю границу, и `downTo`, генерирующий нисходящий диапазон. Для большей безопасности не следует использовать диапазоны с циклами `for` – только со специальными функциями, абстрагирующими итерации, такими как `fold`. Больше об этом рассказывается в главе 4.

Неконтролируемые исключения в Kotlin

В отличие от Java в Kotlin нет контролируемых исключений. Все исключения неконтролируемые. Это соответствует тому, что делает большинство Java-программистов в настоящее время: они упаковывают контролируемые исключения в неконтролируемые. Еще одно отличие между Kotlin и Java заключается в том, что конструкция `try..catch..finally` является выражением, возвращающим значение. Ее можно использовать так:

```
val num: Int = try {
    args[0].toInt()
} catch (e: Exception) {
    0
} finally {
    // Код в этом блоке выполняется всегда.
}
```

Как вы уже видели на примере `if...else`, возвращаемое значение является результатом выполнения последней инструкции в блоке. Разница

в том, что в `try..catch..finally` фигурные скобки являются *обязательными*. Возвращаемое значение является результатом либо последней инструкции в блоке `try`, если нет исключений, либо последней инструкции в блоке `catch` в противном случае.

Автоматическое освобождение ресурсов

Kotlin может автоматически освобождать ресурсы, как это делает Java в конструкции *try with resource*, если ресурс реализует интерфейс `Closable` или `AutoClosable`. Основное отличие в том, что Kotlin предлагает функцию `use`:

```
File("myFile.txt").inputStream()
                  .use {
                      it.bufferedReader()
                        .lineSequence()
                        .forEach (::println)
        }
```

Этот конкретный код приводится только как пример автоматической обработки закрываемых ресурсов. Ни в чем другом он не представляет большого интереса, потому что функция `lineSequence` возвращает `Sequence`, которая является ленивой конструкцией. *Ленивая* означает, что операция (чтение строк из файла) происходит только при обращении к этим строкам.

За пределами блока функции `use` входной поток автоматически закроется, и обращаться к последовательности будет бессмысленно. Именно поэтому мы применили эффект (`println`), что противоречит принципам безопасного программирования. Следующий код скомпилируется, но возбудит исключение `IOException: Stream Closed` во время выполнения:

```
val lines: Sequence<String> = File("myFile.txt")
        .inputStream()
        .use {
            it.bufferedReader()
              .lineSequence()
        }

lines.forEach(::println)
```

Решение заключается в том, чтобы прочитать поток до завершения блока:

```
val lines: List<String> = File("myFile.txt")
        .inputStream()
        .use {
          it.bufferedReader()
            .lineSequence()
            .toList()
        }

lines.forEach(::println)
```

Главная проблема в том, что этот прием требует прочитать весь файл в память. Кстати, для построчной обработки файлов Kotlin предлагает намного более простое решение – функцию `forEachLine`:

```
File("myFile.txt").forEachLine { println(it) }
```

Также можно использовать `useLines`, возвращающую `Sequence`. Например, предыдущий пример можно переписать так:

```
File("myFile.txt").useLines{ it.forEach(::println) }
```

Интеллектуальное приведение типов в Kotlin

В Java иногда бывает необходимо привести ссылку к некоторому типу. Так как эта операция может вызвать исключение `ClassCastException`, если объект окажется не того типа, приходится предварительно проверять его тип с помощью оператора `instanceof`:

```
Object payload = message.getPayload();
int length = -1;
if (payload intanceof String) {
    String stringPayload = (String) payload;
    length = stringPayload.length();
}
```

Это довольно неуклюжий код. В настоящем объектно-ориентированном программировании проверка и приведение типа часто считается плохой практикой. Они могут свидетельствовать, что было приложено недостаточно усилий для использования автоматического приведения типов. Но в Kotlin есть специальный механизм так называемого интеллектуального приведения типов (smart cast). Вот как можно использовать этот механизм в предыдущем примере:

```
val payload: Any = message.payload

val length: Int = if (payload is String)
    payload.length
else
    -1
```

Интеллектуальным такое приведение типа называется потому, что в первой ветви функции `if` Kotlin знает, что `payload` имеет тип `String`, поэтому выполняет приведение автоматически. Интеллектуальное приведение также можно использовать с функцией `when`:

```
val result: Int = when (payload) {
        is String -> payload.length
        is Int -> payload
        else -> -1
    }
```

Приведение типа можно также выполнить обычным, *небезопасным* способом – с помощью оператора `as`:

```
val result: String = payload as String
```

Если объект окажется не того типа, эта строка возбудит исключение `ClassCastException`. Для подобных операций приведения типов Kotlin предлагает другой, безопасный синтаксис:

```
val result: String? = payload as? String
```

Если попытка приведения типа завершится неудачей, `result` получит значение `null` и исключения не возникнет. Но имейте в виду, что если вы стремитесь писать безопасные программы на Kotlin, избегайте оператора `as?`, так же как и типов с поддержкой `null`.

Равенство и идентичность

Одна из классических ловушек в Java – путаница между равенством и идентичностью. Эта проблема еще более усложняется из-за наличия примитивов, факта интернирования строк и особенностей обработки целых чисел:

```
// Код на Java.
int a = 2;
System.out.println(a == 2); // true
Integer b = Integer.valueOf(1);
System.out.println(b == Integer.valueOf(1)); // true
System.out.println(b == new Integer(1)); // false
System.out.println(b.equals(new Integer(1))); // true
Integer c = Integer.valueOf(512);
System.out.println(c == Integer.valueOf(512)); // false
System.out.println(c.equals(Integer.valueOf(512))); // true
String s = "Hello";
System.out.println(s == "Hello"); // true
String s2 = "Hello, World!".substring(0, 5);
System.out.println(s2 == "Hello"); // false
System.out.println(s2.equals("Hello")); // true
```

Какая каша! (Целые числа `b` и `c` ведут себя по-разному, потому что для маленьких целых чисел в Java возвращается общая версия объекта, поэтому `Integer.valueOf(1)` всегда возвращает один и тот же объект, а `Integer.valueOf(512)` – новые отдельные объекты.) Для безопасности в Java всегда проверяйте равенство объектов с помощью метода `equals`, а примитивов – с помощью оператора `==`. Оператор `==` проверяет равенство для примитивов и идентичность для объектов.

В Kotlin все намного проще. Идентичность (или *ссылочное равенство*) проверяется с помощью оператора `===`. Равенство (или *структурное равенство*) проверяется с помощью оператора `==`, который фактически является более краткой формой записи вызова функции `equals`. Тот простой факт, что проверка равенства (`==`) проще проверки идентичности (`===`), предотвращает множество ошибок. Оба оператора, `==` и `===`, можно инвертировать, заменив первый символ `=` восклицательным знаком `!`.

Интерполяция строк

Kotlin предлагает гораздо более простой, в сравнении с Java, синтаксис смешивания значений со строками. В предыдущем примере я исполь-

зовал оператор + для создания параметризованной строки, что несколько непрактично. В Java предпочтительнее было бы использовать метод `String.format()`:

```
// Код на Java
System.out.println(String.format("%s's registration date: %s", name, date));
```

Kotlin позволяет выразить то же самое намного проще:

```
println("$name's registration date: $date")
```

Допускается даже использовать целые выражения, нужно лишь заключить их в фигурные скобки:

```
println("$name's registration date: ${date.atZone(
                                    ZoneId.of("America/Los_Angeles"))}")
```

Этот прием, который называется *интерполяцией строк*, значительно упрощает работу со строками и делает код более простым для понимания.

Многострочные строки

Kotlin поддерживает возможность определения значений с многострочным текстом с использованием тройных кавычек (`"""`) и функции `trimMargin`:

```
println("""This is the first line
         |and this is the second one.""".trimMargin())
```

Этот код выведет:

```
This is the first line
and this is the second one.
```

Функция `trimMargin` принимает необязательный параметр, строку, которая используется как ограничитель отступов. Как можно догадаться по предыдущему примеру, по умолчанию используется ограничитель `"|"`.

Вариантность: параметризованные типы и подтипы

Вариантность описывает, как параметризованные типы ведут себя по отношению к подтипам. *Ковариантность* означает, что `Matcher<Red>` считается подтипом `Matcher<Color>`, если `Red` является подтипом `Color`. В таких случаях `Matcher<T>` называется ковариантным для `T`. Если, напротив, `Matcher<Color>` считается подтипом `Matcher<Red>`, тогда `Matcher<T>` называется *контрвариантным* для `T`.

В Kotlin можно явно указать вариантность с помощью ключевых слов `in` и `out`, которые короче и, надеюсь, проще для понимания, чем слова «контрвариантный» и «ковариантный». (Отсутствие ключевого слова означает *инвариантность*.)

Чтобы было понятнее, вообразите список `List<String>`. Так как `String` является подтипом `Any`, очевидно, что `List<String>` также можно рассматривать как `List<Any>`. В Java, где не поддерживается понятие вариантности, для этого нужно использовать подстановочные знаки.

2.9.4 Какие проблемы может вызывать поддержка вариантности?

Экземпляр `String` – это экземпляр `Any`, поэтому можно записать:

```
val s = "A String"
val a: Any = s
```

Это допустимо, потому что `Any` является родителем для `String`. Если бы `MutableList<Any>` был родителем `MutableList<String>`, вы могли бы записать:

```
val ls = mutableListOf("A String")
val la: MutableList<Any> = ls // <- Ошибка компиляции.
la.add(42)
```

Если бы такой код компилировался, вы смогли бы вставить `Int` в список строк. Это не проблема при использовании неизменяемых списков. Добавление элемента `Int` в неизменяемый список строк приводит к созданию нового списка типа `List<Any>` без изменения типа исходного списка:

```
val ls = listOf("A String")
val la = ls + 42 // <- Kotlin определяет тип `la` как `List<Any>`
```

В Java все типы являются *инвариантными*, т. е., даже если `A` является родительским типом для `B`, тип `List<A>` не является ни родительским типом, ни дочерним типом для `List<B>`. `List<A>` и `List<B>` – это два разных типа на этапе компиляции (и один и тот же на этапе выполнения). Проблема с инвариантными типами заключается в невозможности написать такой код:

```
fun <T> addAll(list1: MutableList<T>,
               list2: MutableList<T>) {
  for (elem in list2) list1.add(elem)
}

val ls = mutableListOf("A String")
val la: MutableList<Any> = mutableListOf()
addAll(la, ls) // <-- ошибка компиляции!
```

Даже притом что все элементы типа `String` будут добавлены в `List<Any>`, что абсолютно правильно, Kotlin не сможет объединить `MutableList<Any>` и `MutableList<String>` в `MutableList<T>`. Чтобы выполнить такое объединение, нужно явно сообщить компилятору, что `MutableList<Any>` можно использовать так, как если бы он был супертипом для `MutableList<String>`, потому что он будет использоваться только для чтения (`out`) и никогда для записи (`in`). Сделать это можно с помощью квалификатора `out`, как показано в следующем коде:

```
fun <T> addAll(list1: MutableList<T>,
               list2: MutableList<out T>) { // <-- Объявить T ковариантным.
  for (elem in list2) list1.add(elem)
}
```

```
val ls = mutableListOf("A String")
val la: MutableList<Any> = mutableListOf()
addAll(la, ls) // <-- Компилируется без ошибок.
```

Здесь ключевое слово `out` указывает, что `list2` является ковариантом для `T`. Как нетрудно догадаться, контрвариантность объявляется с помощью ключевого слова `in`. Поэтому ту же задачу можно решить другим способом – объявив `list1` типом `in` (потребляемым, но никогда не создаваемым).

2.9.5 Когда использовать объявления ковариантности и контрвариантности

Для обозначения ковариантности и контрвариантности в Kotlin используются слова `out` и `in` соответственно. Допустим, у вас есть следующий интерфейс:

```
interface Bag<T> {
  fun get(): T
}
```

Так как этот интерфейс определяет единственную функцию, возвращающую `T` (и не определяет функции, принимающей `T`), результат `Bag<T>` можно безопасно присвоить любой ссылке `Bag<V>`, если `V` является супертипом для `T`. Но вы должны явно обозначить свое намерение, объявив параметр типа ковариантным с помощью ключевого слова `out`:

```
open class MyClassParent

class MyClass: MyClassParent()

interface Bag<out T> {
  fun get(): T
}

class BagImpl : Bag<MyClass> {
  override fun get(): MyClass = MyClass()
}

val bag: Bag<MyClassParent> = BagImpl()
```

> **ПРИМЕЧАНИЕ** Если параметр типа имеет вариантность `out` и вы не указали это явно, хорошая среда разработки (такая как IntelliJ) предупредит, что его можно сделать ковариантным.

И наоборот, если интерфейс имеет только функции, принимающие аргументы типа `T` и никогда не возвращающие `T`, параметр типа можно объявить контрвариантным, использовав ключевое слово `in`:

```
open class MyClassParent

class MyClass: MyClassParent()

interface Bag<in T> {
  fun use(t: T): Boolean
```

```
}

class BagImpl : Bag<MyClassParent> {
  override fun use(t: MyClassParent): Boolean = true
}

val bag: Bag<MyClass> = BagImpl()
```

Если ни одно из ключевых слов, `in` или `out`, не указано, тип считается инвариантным. Выбор между `out` и `in` прост нужно лишь запомнить, что ковариантность используется, когда тип только выводится (возвращаемые значения), а контрвариантность используется, когда тип только вводится (аргументы).

2.9.6 Объявление вариантности в точке определения и точке использования

Какой бы удобной ни была возможность объявлять вариантность в точке определения типа, вы встретитесь со многими ситуациями, когда это невозможно. Если интерфейс `Bag` определяет функции, принимающие и возвращающие тип `T`, его вариантность нельзя объявить в точке определения интерфейса:

```
interface Bag<T> {
  fun get(): T
  fun use(t: T): Boolean
}
```

Вариантность параметра `T` нельзя объявить, потому что в методе `get` он должен быть ковариантным, а в методе `use` – контрвариантным. В данном случае объявить вариантность в точке определения невозможно. Но сохраняется возможность объявить вариантность в точке использования.

Взгляните на следующую функцию `useBag`:

```
open class MyClassParent

class MyClass: MyClassParent()

interface Bag<T> {
  fun get(): T
  fun use(t: T): Boolean
}

class BagImpl : Bag<MyClassParent> {
  override fun get(): MyClassParent = MyClassParent()

moverride fun use(t: MyClassParent): Boolean = true
}

fun useBag(bag: Bag<MyClass>): Boolean {
  // обработать bag
  return true
}

val bag3 = useBag(BagImpl()) // <-- Ошибка компиляции!
```

Здесь компилятор сообщит об ошибке, потому что `useBag` принимает аргумент типа `Bag<MyClass>`, а все, что у вас есть, – это `Bag<MyClassParent>`. Чтобы этот код скомпилировался, нужно объявить контрвариантность для `T`. Но этого нельзя сделать в объявлении интерфейса `Bag<T>`, потому что возникнет конфликт с функцией `get(t: T)`, где `T` находится в позиции `out`. Решение заключается в ограничении типа в точке использования:

```
fun useBag(bag: Bag<in MyClass>): Boolean {
  // обработать bag
  return true
}
```

Обратная операция тоже возможна:

```
fun createBag(): Bag<out MyClassParent> = BagImpl2()

class BagImpl2 : Bag<MyClass> {
  override fun use(t: MyClass): Boolean = true
  override fun get(): MyClass = MyClass()
}
```

Как видите, `in MyClass` и `out MyClassParent` – это ограниченные типы: `in MyClass` – это подтип `MyClass`, который может использоваться только в позиции `in`, тогда как `out MyClassParent` – это подтип `MyClassParent`, который может использоваться только в позиции `out`. Эти ограничения проверяются компилятором. Можно сказать, что `in MyClass` и `out MyClassParent` являются *проекциями* типов `MyClass` и `MyClassParent`.

Как я уже говорил, эта глава является лишь кратким обзором языка Kotlin. В следующих главах я расскажу о других его особенностях, используемых для создания безопасных программ. Но, если вам нужно более глубокое знакомство с Kotlin, я бы посоветовал прочитать книгу *Kotlin in Action* Дмитрия Жемерова (Dmitry Jemerov) и Светланы Исаковой (Svetlana Isakova), выпущенную издательством Manning (2017)[1] и написанную членами команды разработчиков Kotlin.

Итоги

- Поля и переменные имеют разный синтаксис и разные области видимости.
- Новый синтаксис объявления классов и интерфейсов позволяет избавиться от значительной части шаблонного кода, особенно в классах данных. Он позволяет создать эквивалент Java-объекта со свойствами, методами чтения и записи, конструкторами, с функциями `equals`, `hashCode`, `toString` и `copy` в одной строке.
- Функции можно определять на уровне пакетов (аналог статических методов в Java), в классах, объектах и даже внутри других функций.

[1] *Жемеров Д., Исакова С.* Kotlin в действии. М.: ДМК Пресс, 2018. ISBN: 978-1-61729-329-0, 978-5-97060-497-7. – *Прим. перев.*

- Поддержка функций-расширений позволяет добавлять функции в существующие классы и вызывать их, как если бы они были функциями экземпляра.
- В Kotlin условные структуры управления заменяются выражениями, возвращающими значение.
- В Kotlin есть структуры управления циклическим выполнением, но их почти всегда можно заменить функциями.
- Kotlin различает типы с поддержкой и без поддержки `null` и предлагает операторы для безопасной обработки значений `null`.

3 Программирование с функциями

Эта глава охватывает следующие темы:

- основы функций и их представление;
- использование лямбда-выражений;
- использование функций высшего порядка и каррированных функций;
- использование правильных типов.

В главе 1 мы узнали, что одним из наиболее важных методов безопасного программирования является четкое отделение частей программ, которые не зависят ни от чего, кроме входных данных от части, зависящей от состояния внешнего мира. Для программ, состоящих из *подпрограмм* (которые часто называют процедурами, методами или функциями), это деление транзитивно применяется и к подпрограммам. В Java они называются методами, но в Kotlin их принято называть функциями, что, вероятно, более уместно, потому что функция – это математический термин с точным определением (обычное дело для математики). Функции в Kotlin можно сравнить с математическими функциями, когда они не оказывают никакого эффекта, кроме возврата значения, которое зависит только от аргумента. Такие функции программисты часто называют *чистыми функциями*. Поэтому, чтобы писать безопасные программы, нужно:

- использовать чистые функции для вычислений;
- использовать чистые эффекты для передачи результатов вычислений во внешний мир.

Чистые функции необходимы, чтобы гарантировать получение одного и того же результата для одного и того же аргумента. В противном случае программа будет *недетерминированной*, а значит, вы никогда не сможете проверить правильность программы. Чистые эффекты могут показаться менее важными, но, если подумать, *нечистые эффекты* – это эффекты, включающие вычисления, которые нелегко проверить, поэтому их следует помещать в отдельные (чистые) функции.

Для некоторых программистов, ищущих способы написания безопасных программ, четкое отделение чистых функций от чистых эффектов не является конечной целью. Чистые функции легко проверить, но чистые эффекты – нет. Можно ли это изменить? Да, если рассматривать эффекты с другой точки зрения. Именно это дает чистое функциональное программирование. В этой парадигме все является функцией. Значения – это функции, функции – это функции, и эффекты – это тоже функции. Вместо применения эффектов функциональные программисты используют функции, возвращающие данные, которые представляют невыполненный требуемый эффект. В такой парадигме программирования все является функцией и все является данными, потому что нет никакой разницы между данными и функциями.

Почему бы просто не использовать чисто функциональное программирование? Это возможно, но порой очень трудно, если не применять язык, специально предназначенный для этого. Такие языки, как Java и Kotlin, предлагают множество инструментов функционального программирования, но поддержка функциональных эффектов ограничена. В главе 12 я покажу, как обрабатывать эффекты в функциональной манере, потому что этот метод используется во всех видах программирования (часто без понимания программистом). Хотя это не только возможно, но и необходимо, вы, вероятно, не всегда будете интерпретировать эффекты таким способом.

В этой главе вы узнаете, как для вычислений использовать чистые и каррированные функции. У вас могут возникнуть проблемы с пониманием некоторых фрагментов кода, представленных в этой главе. Это вполне ожидаемо, потому что сложно вводить функции без использования других функциональных конструкций, таких как `List`, `Option` и др. Наберитесь терпения. Все эти конструкции и компоненты мы обсудим в следующих главах.

3.1 Что такое функция?

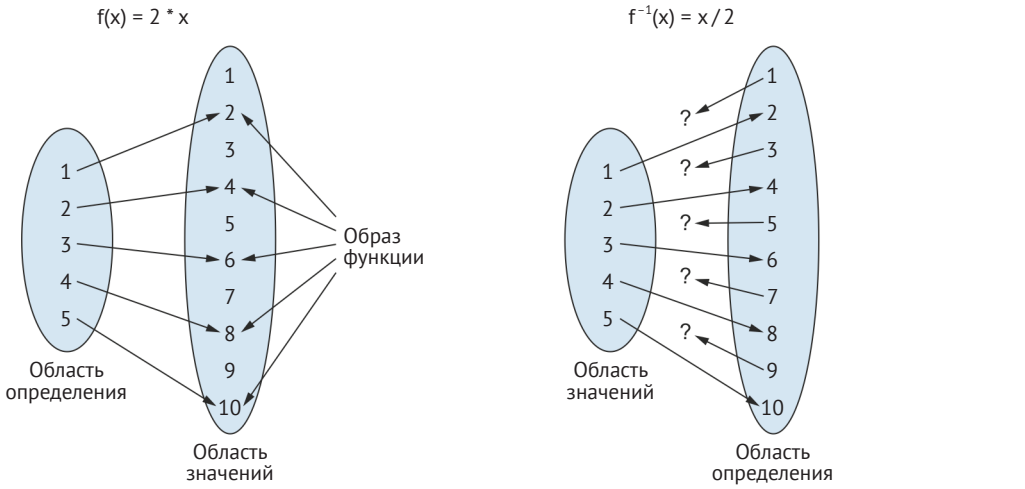
В этом разделе мы поближе познакомимся с функциями. *Функция* – это прежде всего математическое понятие. Она описывает связь между исходным (называется *областью определения* функции) и конечным множеством данных (называется *областью значений* функции). Области определения и значений не обязательно должны быть разными. Например, область определения и область значений функции могут совпадать и содержать множество целых чисел.

3.1.1 Отношения между двумя областями функций

Функции всегда выполняют одно важное условие: любому элементу в области определения соответствует один и только один элемент в области значений, как показано на рис. 3.1. Из этого условия вытекают некоторые интересные следствия:

- в области определения нет элементов, для которых не имелось бы соответствующих элементов в области значений;
- в области значений нет двух элементов, соответствующих одному элементу в области определения;
- в области значений нет элементов, не соответствующих ни одному элементу в области определения;
- в области значений может быть элемент, соответствующий нескольким значениям в области определения.

ПРИМЕЧАНИЕ Множество элементов области значений, для которых имеются соответствующие элементы в области определения, называют *образом функции*.



f(x) – это функция натурального аргумента, дающая в результате натуральное число

$f^{-1}(x)$ не является функцией с областью определения, охватывающей все натуральные числа. Областью определения этой функции является множество четных чисел (образ функции f)

Рис. 3.1 Любому элементу в области определения соответствует один и только один элемент в области значений

Это свойство функций иллюстрирует рис. 3.1. Можно, например, определить функцию

```
f(x) = x + 1
```

где x – положительное целое число. Эта функция представляет отношение между каждым положительным целым числом и числом, следую-

щим за ним. Этой функции можно дать любое имя, например которое поможет вспомнить смысл функции:

```
successor(x) = x + 1
```

Это может показаться отличной идеей, но никогда не доверяйте слепо названиям функций. Например, ту же функцию можно определить так:

```
predecessor(x) = x + 1
```

С точки зрения компилятора здесь нет никакой ошибки, потому что имя функции не обязано в точности соответствовать ее определению. Но выбор таких имен, не соответствующих смыслу функций, считается дурной практикой. На самом деле имя функции не является ее частью, это всего лишь удобный способ обратиться к ней.

Обратите внимание, что я говорю о том, *чем является* функция (ее определение), а не *что она делает*. Функция ничего не делает. Функция `successor` не добавляет `1` к своему аргументу `x`. Вы можете добавить `1` к целому числу, чтобы вычислить следующее за ним значение, но функция `successor` не делает этого. Она просто описывает связь между целым числом и числом, следующим за ним в порядке возрастания:

```
successor(x)
```

Выражение `successor(x)` – это всего лишь эквивалент выражения `x + 1`, т. е. каждый раз, встретив выражение `successor(x)`, вы можете его заменить на `(x + 1)`. Обратите внимание на круглые скобки, которые используются для изоляции выражения. Они не нужны, когда выражение используется отдельно, но в некоторых случаях они могут быть необходимы.

3.1.2 Обратные функции в Kotlin

Функция может иметь или не иметь обратную функцию. Если `f(x)` является функцией, принимающей `A` и возвращающей `B` (`A` – это область определения, а `B` – область значений), тогда обратная ей функция обозначается как $f^{-1}(x)$ и имеет область определения `B` и область значений `A`. Если представить тип функции как `A -> B`, тогда обратная функция (если она существует) будет иметь тип `B -> A`.

> **ПРИМЕЧАНИЕ** В Kotlin используется немного другой синтаксис для представления типов функций, с типом аргумента в скобках: `(A) -> B` и `(B) -> A`. С этого момента я буду использовать синтаксис, принятый в языке Kotlin.

Обратное отношение для функции – это функция, если она удовлетворяет тем же требованиям, что и любая функция: любому элементу в области определения соответствует один и только один элемент в области значений. То есть отношение, обратное для `successor(x)`, которое назовем `predecessor(x)` (хотя его точно так же можете было бы назвать `xyz`), не является функцией на множестве натуральных чисел *N*, включающем

`0`, потому что `0` не имеет предшествующего значения в множестве натуральных чисел. Но если рассматривать `successor(x)` как функцию на множестве целых чисел (включающем положительные и отрицательные числа), тогда `predecessor(x)` будет считаться обратной функцией для `successor(x)`.

Некоторые простые функции не имеют обратных функций. Например, функция

```
f(x) = (2 * x)
```

не имеет обратной функции на множестве натуральных чисел *N*, потому что обратным для `f` отношением является `f'(x) = x/2`, значение которого не принадлежит *N*, если `x` – нечетное число. Но оно может считаться обратной функцией, если определить это отношение как функцию на множестве четных целых чисел.

3.1.3 Частичные функции

Чтобы считаться функцией, отношение должно удовлетворять двум требованиям:

1. должно быть определено для всех элементов в области определения;
2. в области определения не должно быть элементов, которым соответствовало бы более одного элемента в области значений.

Отношение, которое определено не для всех элементов в области определения, но удовлетворяет остальным требованиям, часто называют *частичной функцией*. И по аналогии нечастичные функции иногда называют *полными функциями*. Строго говоря, истинная функция всегда является полной, а частичная не является функцией. Но вам будет полезно знать словарь, используемый многими программистами.

Отношение `predecessor(x)` – это частичная функция на множестве натуральных чисел *N* (множестве положительных целых чисел плюс `0`), но полная функция на *N** – множестве натуральных чисел без `0`, а его область значений совпадет с *N*. Частичные функции играют важную роль в программировании, потому что многие ошибки являются результатом использования частичной функции, как если бы она была полной. Например, отношение `f(x) = 1/x` является частичной функцией на *N* с областью значений *Q* (множеством рациональных чисел), потому что оно не определено для `0`. Но оно является полной функцией на *N** с областью значений *Q*, а также полной функцией на *N* с областью значений (*Q* плюс ошибка). Добавляя элемент в область значений (ошибка), можете частичную функцию превратить в полную. Но для этого функция должна иметь возможность вернуть ошибку. Заметили аналогию с компьютерными программами? Далее в этой книге вы увидите, что превращение частичных функций в полные является важной частью безопасного программирования.

3.1.4 Композиция функций

Функции – это строительные блоки, из которых можно составлять другие функции. Композиция (сочетание) функций `f` и `g` обозначается как `f ° g` и читается как *f от g*. Если `f(x) = x * 2` и `g(x) = x + 1`, тогда

```
f ° g (x) = f(g(x)) = f(x + 1) = (x + 1) * 2
```

Две формы записи – `f ° g(x)` и `f(g(x))` – эквивалентны. Но запись композиции как `f(g(x))` подразумевает использование представления аргумента `x`. С помощью записи `f ° g` можно выразить композицию функций без использования представления аргумента. Применив эту функцию к числу `5`, вы получите:

```
f ° g(5) = f(g(5)) = f(5 + 1) = 6 * 2 = 12
```

Интересно отметить, что в общем случае `f ° g` и `g ° f` дают разные результаты, хотя иногда могут быть эквивалентными. Например:

```
g ° f(5) = g(f(5)) = g(5 * 2) = 10 + 1 = 11
```

Функции применяются в порядке, обратном порядку их следования в записи. Например, в случае `f ° g` сначала применяется функция `g`, а затем `f`.

3.1.5 Функции нескольких аргументов

До сих пор я говорил только о функциях одного аргумента. А существуют ли функции нескольких аргументов? Вообще говоря, функций нескольких аргументов не существует. Помните определение функции? Функция описывает связь между исходным (областью определения) и конечным множеством данных (областью значений). Это не связь между двумя или более исходными множествами и конечным множеством. Функция не может иметь несколько аргументов. Но произведение двух множеств само по себе является множеством. Функция от такого произведения множеств может выглядеть как функция нескольких аргументов. Давайте рассмотрим следующую функцию:

```
f(x, y) = x + y
```

Между $N \times N$ и N может быть определенное отношение; в таком случае это функция. Но она имеет только один аргумент, который является элементом $N \times N$, где $N \times N$ – множество всех возможных пар целых чисел. Элементом этого множества является пара целых чисел, а пара является частным случаем более общего понятия *кортеж*, которое используется для представления комбинаций из нескольких элементов. *Пара* – это кортеж с двумя элементами.

Кортежи обычно записываются в круглых скобках, т. е. `(3, 5)` – это кортеж и элемент $N \times N$. Функция `f` может быть применена к этому кортежу:

```
f((3, 5)) = 3 + 5 = 8
```

В таких случаях, следуя соглашениям, можно упростить запись и убрать одну пару скобок:

```
f(3, 5) = 3 + 5 = 8
```

И это все еще функция одного аргумента-кортежа, а не двух аргументов.

3.1.6 Каррирование функций

Функции кортежей можно интерпретировать по-разному. Например, функцию `f(3, 5)` можно рассматривать как функцию на *N* от множества функций на *N*. То есть предыдущий пример можно переписать так:

```
f(x)(y) = x + y
```

В таких случаях можно записать:

```
f(x) = g
```

что означает: результатом применения функции `f` к аргументу `x` является новая функция `g`. Применение этой функции `g` к аргументу `y` дает результат:

```
g(y) = x + y
```

Когда применяется функция `g`, аргумент `x` перестает быть переменной и превращается в *константу*. Он не зависит ни от аргумента `y`, ни от чего-либо еще. Если применить `f` к кортежу `(3, 5)`, то получится:

```
f(3)(5) = g(5) = 3 + 5 = 8
```

Единственное новое здесь – это область значений `f`, которая является множеством функций, а не чисел. Результатом применения `f` к целому числу является функция. Результатом применения этой результирующей функции к целому числу является целое число.

`f(x)(y)` – это каррированная форма функции `f (x, y)`. Применение этого преобразования к функции от кортежа (которую вы можете называть функцией нескольких аргументов, если хотите) называется *каррированием* в честь математика Хаскелла Карри (Haskell Curry), хотя он и не был изобретателем этого преобразования.

3.1.7 Частично-примененные функции

Каррированная форма функции сложения может показаться неестественной и вызвать вопрос: есть ли нечто подобное в реальном мире. При использовании каррированной версии аргументы рассматриваются по отдельности. Сначала берется один из аргументов, и применение функции к нему дает вам новую функцию. Эта новая функция имеет какую-либо ценность сама по себе или это просто шаг в цепочке вычислений?

В случае со сложением каррирование не выглядит особенно полезным. И кстати, сложение позволяет начать с любого из двух аргументов, так как очередность операндов в сложении не имеет значения. Промежу-

точная функция получится другой, но конечный результат от этого не изменится.

Чтобы лучше понять полезность каррирования, представьте, что вы путешествуете за границей, используя карманный калькулятор (или смартфон) для пересчета из одной валюты в другую. Что бы вы предпочли – вводить коэффициент конверсии при каждом пересчете цены или поместить коэффициент в память и вызывать его оттуда одним нажатием? Какое решение будет менее подвержено ошибкам? Рассмотрим новую функцию пары значений:

```
f(rate, price) = price / 100 * (100 + rate)
```

Ей эквивалентна следующая функция:

```
g(price, rate) = price / 100 * (100 + rate)
```

Теперь рассмотрим каррированные версии этих двух функций:

```
f(rate)(price) g(price)(rate)
```

Мы знаем, что `f` и `g` являются функциями. Но что такое `f(rate)` и `g(price)`? Да, конечно, это результаты применения `f` к `rate` и `g` к `price`. Но какова суть этих результатов?

`f(rate)` – это функция от курса, возвращающая цену. Если `rate = 9`, эта функция применит 9%-ный налог к исходной цене и вернет новую цену. Эту новую функцию можно назвать `apply9percentTax(price)`, и вполне возможно, что она сама по себе является полезным инструментом, потому что налоговые ставки меняются нечасто.

С другой стороны, `g(price)` – это функция от цены, возвращающая курс. Если цена равна 100 долл., мы получаем новую функцию, применяющую цену 100 долл. к изменяющейся ставке налога. Такая функция может показаться менее полезной, хотя многое зависит от решаемой задачи. Если задача состоит в том, чтобы вычислить, как вырастет фиксированная сумма при переменной процентной ставке, эта версия будет более полезной.

Такие функции, как `f(rate)` и `g(price)`, иногда называют частично-примененными функциями со ссылкой на формы `f(rate, price)` и `g(price, rate)`. Частично-примененные функции могут иметь огромные последствия для оценки аргументов. Я вернусь к этой теме в разделе 3.2.4.

3.1.8 Функции не имеют эффектов

Напомню, что чистые функции просто возвращают вычисленное значение и больше ничего не делают. Они не меняют никаких элементов внешнего мира (понятие *внешний мир* относится к самой функции), они не изменяют своих аргументов и не взрываются (не генерируют исключение или что-либо еще) в случае ошибки. Они могут вернуть исключение или что-то еще, например сообщение об ошибке. Но при этом исключение должно именно возвращаться, а не возбуждаться, не записывать в файл и не выводить в консоль. Более подробно о чистых функциях я расскажу в разделе 3.2.4.

3.2 Функции в Kotlin

В главе 1 мы использовали то, что в Kotlin называется функциями, но на самом деле является методами. Во многих языках программирования методы – это одно из представлений функций. Как обсуждалось в главе 2, в Kotlin методы называются функциями и для их определения используется ключевое слово `fun`. Но есть две проблемы с этими *функциями*. Данные и функции – это суть одно и то же. Любая часть данных в действительности является функцией, а любая функция – частью данных.

3.2.1 Функции как данные

Как и любые данные, каждая функция имеет тип, такой как `String` или `Int`, и может быть присвоена ссылке. Типы данных также могут передаваться как аргументы другим функциям и возвращаться функциями, в чем вы вскоре убедитесь. Функции могут храниться в структурах данных, таких как списки или ассоциативные массивы, или даже в базе данных. Но функции, объявленные с ключевым словом `fun` (например, методы Java), не могут использоваться подобными способами. В Kotlin есть все необходимые механизмы, чтобы превратить любой такой метод в истинную функцию.

3.2.2 Данные как функции

Давайте вспомним определение функции. *Функция* – это отношение между исходным и конечным множеством данных, удовлетворяющее определенным требованиям. Теперь представьте функцию как набор всех возможных элементов в качестве исходного множества и целое число 5 в качестве конечного множества. Хотите верьте, хотите нет, но эта функция удовлетворяет всем требованиям определения функции. Это особый тип функции, результат которой не зависит от аргумента. Такая функция называется *константной функцией*. Ей не нужно передавать какой-либо аргумент, и нет необходимости давать ей специальное имя. Назовем ее 5. Это чистая теория, но ее стоит запомнить. Она пригодится позже.

3.2.3 Конструкторы объектов как функции

Конструкторы объектов фактически являются функциями. В отличие от Java, где для создания объектов используется специальный синтаксис, в Kotlin используется обычный синтаксис вызова функции. (Но не синтаксис делает объекты функциями; Java-объекты тоже являются функциями.) Получить экземпляр объекта в Kotlin можно, указав имя класса и список аргументов конструктора в круглых скобках, например:

```
val person = Person("Elvis")
```

Возникает важный вопрос. Я говорил, что чистая функция всегда должна возвращать одно и то же значение для одного и того же аргумента. Если принять это во внимание, являются ли конструкторы чистыми функциями? Рассмотрим следующий пример:

```
val elvis = Person("Elvis")
val theKing = Person("Elvis")
```

Два объекта создаются с использованием одного и того же аргумента, поэтому конструктор должен вернуть одно и то же значение.


```
val elvis = Person("Elvis")
val theKing = Person("Elvis")
println(elvis == theKing) // Должна вывести "true".
```

Оператор проверки равенства возвращает `true`, только если функция `equals` была определена соответствующим образом. Такая функция автоматически определяется, если объявить `Person` как класс данных (как было показано в главе 2):

```
data class Person(val name: String)
```

Иначе вы должны сами позаботиться об определении понятия «равенство».

3.2.4 Использование функций fun в Kotlin

Раньше я уже упоминал чистые функции. Независимо от того, как они объявляются, функции, которые в Kotlin определяются с ключевым словом `fun`, не всегда представляют собой настоящие функции. То, что программисты называют функциями, настолько редко ими является, что программисты придумали отдельный термин для обозначения настоящих функций. Они называют их *чистыми функциями*. (Соответственно, все другие функции называются *нечистыми*.) В этом разделе я расскажу, что делает функцию чистой, и приведу несколько примеров таковых.

Вот какие условия необходимо соблюсти, чтобы функция/метод могла считаться чистой:

- функция не должна изменять что-либо за ее пределами, и никакие внутренние изменения не должны наблюдаться снаружи;
- не должна изменять свой аргумент;
- не должна возбуждать исключений;
- всегда должна возвращать значение;
- при вызове с тем же аргументом функция должна возвращать один и тот же результат.

Рассмотрим примеры чистых и нечистых функций (листинг 3.1).

Листинг 3.1. Чистые и нечистые функции

```
class FunFunctions {

    var percent1 = 5
    private var percent2 = 9
    val percent3 = 13

    fun add(a: Int, b: Int): Int = a + b

    fun mult(a: Int, b: Int?): Int = 5
```

```
    fun div(a: Int, b: Int): Int = a / b

    fun div(a: Double, b: Double): Double = a / b

    fun applyTax1(a: Int): Int = a / 100 * (100 + percent1)

    fun applyTax2(a: Int): Int = a / 100 * (100 + percent2)

    fun applyTax3(a: Int): Int = a / 100 * (100 + percent3)

    fun append(i: Int, list: MutableList<Int>): List<Int> {
        list.add(i)
        return list
    }

    fun append2(i: Int, list: List<Int>) = list + i
}
```

Сможете сами определить, какие из этих функций/методов являются чистыми функциями? Подумайте несколько минут, прежде чем прочитать ответ ниже. Вспомните все условия и оцените все операции, выполняемые внутри функций. Не забудьте принять во внимание то, что наблюдается снаружи, и учесть исключительные условия:

```
fun add(a: Int, b: Int): Int = a + b
```

Первая функция `add` – это чистая функция, потому что она всегда возвращает значение, зависящее только от аргументов. Она не изменяет аргументы и никак не взаимодействует с внешним миром. Эта функция может вызвать ошибку, если при сложении `a + b` произойдет переполнение значения типа `Int`. Но она не сгенерирует исключение. Результат получится ошибочный (отрицательное значение), но это совсем другая проблема. Результат всегда будет одним и тем же для одних и тех же аргументов. Но это не означает, что результат должен быть точным!

Точность

Термин «точность» сам по себе ничего не значит. Как правило, он означает соответствие результата ожиданиям. Чтобы рассуждать о точности результата, не требуется знать намерение разработчика. Обычно намерение приходится определять только по имени функции, что может приводить к недопониманию.

Вторая функция

```
fun mult(a: Int, b: Int?): Int = 5
```

тоже является чистой функцией. Тот факт, что она всегда возвращает одно и то же значение для любых аргументов, не имеет значения, так же как тот факт, что имя функции не имеет ничего общего с тем, что она возвращает. Эта функция является константной.

Функция `div`, реализующая целочисленное деление, не является чистой функцией, потому что возбуждает исключение, если делитель равен `0`:

```
fun div(a: Int, b: Int): Int = a / b
```

Чтобы сделать div чистой функцией, можно проверить второй параметр и вернуть специальное значение, если оно равно нулю. Это должно быть целое число, поэтому будет трудно выбрать осмысленное значение, но это уже другая проблема. Функция div, работающая с вещественными значениями, является чистой, потому что деление на 0.0 не вызовет исключения, но вернет Infinity – экземпляр Double, обозначающий «бесконечность»:

```
fun div(a: Double, b: Double): Double = a / b
```


Infinity поглощает сложение, т. е. сложение Infinity с любым числом вернет Infinity. Оно не полностью поглощает умножение, потому что умножение Infinity на 0.0 дает в результате NaN (Not a Number – не число). Несмотря на такое название, NaN является экземпляром Double. Рассмотрим следующий код:

```
var percent1 = 5
fun applyTax1(a: Int): Int = a / 100 * (100 + percent1)
```

Метод applyTax1 не кажется чистой функцией, потому что его результат зависит от значения percent1, которое общедоступно и может измениться между двумя вызовами функций. Как следствие, два вызова функций с одним и тем же аргументом могут вернуть разные значения: percent1 может считаться неявным параметром, но этот параметр получает значение не одновременно с явным аргументом. Это не проблема, если значение percent1 используется в функции только один раз, но, если оно читается дважды, есть вероятность его изменения между двумя операциями чтения. Если вам нужно использовать значение дважды, прочитайте его один раз и сохраните в локальной переменной. Это означает, что метод applyTax1 является чистой функцией от пары (a, percent1), но не является чистой функцией от a.

Сам класс в этом примере можно считать дополнительным неявным аргументом, потому что все его свойства доступны внутри функций. Это важное понятие. Все методы/функции экземпляра можно заменить методами/функциями, не являющимися методами/функциями экземпляра, добавив аргумент с типом включающего класса. Функцию applyTax1 можно определить за пределами FunFunctions как:

```
fun applyTax1(ff: FunFunctions, a: Int): Int = a / 100 * (100 + ff.percent1)
```

Эту функцию можно вызывать из класса, передавая ссылку this в первом аргументе, например applyTax1(this, a), или извне, если доступна ссылка на экземпляр FunFunctions. Здесь applyTax1 является чистой функцией от пары (this, a):

```
private var percent2 = 9
fun applyTax2(a: Int): Int = a / 100 * (100 + percent2)
```

Результат функции applyTax2 зависит от значения percent2, которое является изменяемым (объявлено как var). Результат applyTax2 изменит-

ся, если изменится `percent2`. Однако никакой код не изменяет эту переменную, поэтому `applyTax2` – это чистая функция. Но она небезопасная, потому что может стать нечистой без какой-либо модификации самой функции, если добавится еще одна функция, изменяющая `percent2`. Это хорошая причина стараться всегда использовать неизменяемые значения, за исключением случаев, когда без изменяемых переменных не обойтись. По умолчанию всегда используйте ключевое слово `val`:

```
val percent3 = 13
fun applyTax3(a: Int): Int = a / 100 * (100 + percent3)
```

Функция `applyTax3` – чистая функция, потому что, в отличие от `applyTax1`, использует `percent3` – неизменяемое значение. Следующая функция

```
fun append1(i: Int, list: MutableList<Int>): List<Int> {
    list.add(i)
    return list
}
```

изменяет свой аргумент, и это изменение видно снаружи, поэтому она не является чистой.

Глядя на функцию `append2`:

```
fun append2(i: Int, list: List<Int>) = list + i
```

можно подумать, что она добавляет в список новый элемент с аргументом `i`. Но это не так. Выражение `list + i` возвращает новый (неизменяемый) список, содержащий все элементы оригинального списка плюс дополнительный элемент. Функция `append2` ничего не изменяет, поэтому является чистой функцией.

3.2.5 Объектная и функциональная нотация

Как было показано выше, функции экземпляра, обращающиеся к свойствам класса, можно рассматривать как функции, получающие неявный параметр с экземпляром класса. Функции, не обращающиеся к экземпляру включающего класса, можно безопасно вынести за пределы класса. Такие функции можно поместить в объект-компаньон (и сделать их примерно эквивалентными статическим методам Java) или на уровень пакета вне какого-либо класса. Также в объект-компаньон или на уровень пакета можно поместить функции, обращающиеся к включающему экземпляру, если их неявный параметр (с включающим экземпляром) сделать явным. Рассмотрим класс `Payment` из главы 1:

```
class Payment(val creditCard: CreditCard, val amount: Int) {
    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")

    companion object {
        fun groupByCard(payments: List<Payment>): List<Payment> =
```

```
            payments.groupBy { it.creditCard }
                    .values
                    .map { it.reduce(Payment::combine) }
    }
}
```

Функция `combine` обращается к полям `creditCard` и `amount` вмещающего класса. Поэтому в таком ее виде эту функцию нельзя вынести за пределы класса или переместить в объект-компаньон. Эта функция принимает неявный параметр с экземпляром включающего ее класса. Однако этот параметр можно сделать явным, что позволит определить функцию на уровне пакета или в объекте-компаньоне:

```
fun combine(payment1: Payment, payment2: Payment): Payment =
    if (payment1.creditCard == payment2.creditCard)
        Payment(payment1.creditCard, payment1.amount + payment2.amount)
    else
        throw IllegalStateException("Cards don't match.")
```

Объявление функции в объекте-компаньоне или на уровне пакета позволяет устранить нежелательный доступ к внутренней области извне. Но меняет способ использования функции. При вызове функции изнутри класса ей можно передать ссылку `this`:

```
val newPayment = combine(this, otherPayment)
```

Это мало что меняет. Но ситуация в корне меняется, когда возникает необходимость составить цепочку из вызовов. Представьте, что нужно объединить несколько платежей. В этом случае функцию экземпляра, которая определена как

```
fun combine(payment: Payment): Payment =
    if (creditCard == payment.creditCard)
        Payment(creditCard, amount + payment.amount)
    else
        throw IllegalStateException("Cards don't match.")
```

можно использовать с применением объектной нотации:

```
val newPayment = payment1.combine(payment2).combine(payment3)
```

Такое выражение читается намного проще, чем:

```
import ...Payment.Companion.combine

val newPayment = combine(combine(payment1, payment2), payment3)
```

Этот фрагмент возбудит исключение, если кредитные карты не совпадут, но это только потому, что вы пока не научились обрабатывать ошибки функциональным способом. Эта тема будет подробно обсуждаться в главе 7.

3.2.6 Использование функций как значений

Как я уже говорил, функции можно использовать в качестве данных, но это не относится к функциям, объявленным с ключевым словом `fun`.

Kotlin позволяет рассматривать функции как данные. Kotlin имеет типы функций, и ссылки на функции могут присваиваться переменным соответствующих типов как ссылки на данные других типов. Рассмотрим следующую функцию:

```
fun double(x: Int): Int = x * 2
```

Ту же функцию можно объявить иначе:

```
val double: (Int) -> Int = { x -> x * 2 }
```

Функция `double` имеет тип `(Int) -> Int`. Слева от стрелки указан тип параметра, заключенный в круглые скобки. Справа от стрелки указан тип возвращаемого значения. За знаком равенства следует определение функции. Оно заключено в фигурные скобки и имеет форму лямбда-выражения.

В предыдущем примере лямбда-выражение состоит из имени, данного параметру, стрелки и выражения, возвращающего результат функции. Здесь использовано простое выражение, которое можно записать в одну строку. Если бы выражение было более сложным, его можно было бы записать в нескольких строках. В таком случае результатом функции будет результат выполнения последней инструкции, как в следующем примере:

```
val doubleThenIncrement: (Int) -> Int = { x ->
    val double = x * 2
    double + 1
}
```

Определение функции не обязательно должно завершаться инструкцией `return`.

Функции кортежей ничем не отличаются. Вот функция, представляющая сложение целых чисел:

```
val add: (Int, Int) -> Int = { x, y -> x + y }
```

Как видите, в типе функции (единственный) аргумент заключен в круглые скобки. В лямбда-выражении, напротив, параметры можно не заключать в круглые скобки.

Когда параметр не является кортежем (точнее, является кортежем с единственным элементом), можно использовать специальное имя `it`:

```
val double: (Int) -> Int = { it * 2 }
```

Это упрощает синтаксис, хотя делает код менее удобочитаемым, особенно когда определяется несколько вложенных функций.

> **ПРИМЕЧАНИЕ** В этом примере `double` не является именем функции. Сама функция не имеет имени. Здесь `double` – это имя ссылки соответствующего типа, поэтому ею можно манипулировать, как и любыми другими данными. Когда мы пишем
>
> ```
> val number: Int = 5
> ```
>
> мы не говорим, что `number` – это имя числа `5`. То же относится и к функциям.

Возможно, вам интересно, почему в Kotlin поддерживаются два вида функций. Если функции являются значениями, зачем тогда использовать ключевое слово `fun` для определения функций?

Как я уже говорил в начале раздела 3.2.4, функции, объявленные с помощью ключевого слова `fun`, на самом деле не являются таковыми. Вы можете называть их методами, подпрограммами, процедурами или как угодно. Они могут представлять чистые функции (всегда возвращающие одно и то же значение для данного аргумента и не имеющие никаких других эффектов, видимых извне), но их нельзя интерпретировать как данные.

Но разве без них нельзя обойтись? Можно, однако функции, объявленные с помощью ключевого слова `fun`, действую намного эффективнее. Они являются оптимизацией. Каждый раз, когда функция нужна, только чтобы передать ей аргумент и получить результат, предпочтительнее использовать версию, объявленную с помощью `fun`.

С другой стороны, каждый раз, когда функция должна использоваться в роли данных (например, для передачи другой функции, как вы вскоре увидите) или вы хотите получить ее в виде возвращаемого значения из другой функции, или сохранить в переменной или в любой другой структуре данных, необходимо использовать выражение с типом функции.

Возможно, вам интересно узнать, можно ли преобразовать функцию из одной формы в другую. Да, и это довольно просто. Нужно просто преобразовать функцию `fun` в лямбда-выражение, потому что функции `fun` нельзя создавать во время выполнения.

3.2.7 Ссылки на функции

Kotlin предлагает точно такую же возможность, что и Java, где она известна под названием *ссылка на метод*. Но, как вы помните, методы в языке Kotlin называются функциями, поэтому ссылки на методы в Kotlin называют ссылками на функции. Вот пример использования функции `fun` в лямбда-выражении:

```
fun double(n: Int): Int = n * 2

val mutliplyBy2: (Int) -> Int = { n -> double(n) }
```

То же самое можно записать так:

```
val mutliplyBy2: (Int) -> Int = { double(it) }
```

Использование ссылок на функции упрощает синтаксис:

```
val mutliplyBy2: (Int) -> Int = ::double
```

Здесь функция `double` вызывается для того же объекта, класса или пакета, что и функция `mutliplyBy2`. Если бы это была функция экземпляра другого класса, вы могли бы использовать следующий синтаксис передачи ссылки на экземпляр этого класса:

```
class MyClass {
    fun double(n: Int): Int = n * 2
}

val foo = MyClass()
val mutliplyBy2: (Int) -> Int = foo::double
```

Кроме того, если double определена в другом пакете, ее необходимо импортировать:


```
import other.package.double

val mutliplyBy2: (Int) -> Int = ::double
```

В случае определения функции в объекте-компаньоне (в некоторой степени эквивалент статического метода в Java) вы можете либо импортировать ее, либо использовать следующий синтаксис:

```
val mutliplyBy2: (Int) -> Int = (MyClass)::double
```

Это более краткая форма вызова:

```
val mutliplyBy2: (Int) -> Int = MyClass.Companion::double
```

Не забудьте использовать .Companion или круглые скобки. Иначе вы получите совершенно иной результат:


```
class MyClass {
    companion object {
        fun double(n: Int): Int = n * 2
    }
}

val mutliplyBy2: (MyClass, Int) -> Int = MyClass::double
```

В таком случае тип функции multiplyBy2 не (Int) -> Int, а (MyClass, Int) -> Int.

3.2.8 Композиция функций fun

Композиция функций fun выглядит просто:

```
fun square(n: Int) = n * n

fun triple(n: Int) = n * 3

println(square(triple(2)))

36
```

Но это не композиция функций. Этот пример создает функциональное приложение. Композиция функций – это двухместная операция над функциями, так же как сложение – это двухместная операция над числами, поэтому вы можете компоновать функции программно, используя другую функцию.

Упражнение 3.1

Напишите функцию compose (объявив ее с ключевым словом fun), которая объединяет функции целочисленного аргумента Int, возвращающие Int.

ПРИМЕЧАНИЕ Решения следуют за каждым упражнением, но желательно, чтобы вы сначала попробовали выполнить упражнение самостоятельно, не заглядывая в ответ. Код решения также приводится на веб-сайте книги. Это упражнение достаточно простое, но некоторые будут довольно сложными, поэтому кому-то из вас будет сложно удержаться и не подсмотреть ответ. Помните, что чем больше времени вы потратите на поиск решения, тем больше вы узнаете.

Подсказка

Функция `compose` принимает два параметра с типами функций `(Int) -> Int` и возвращает функцию того же типа. Функцию `compose` можно объявить с помощью ключевого слова `fun`, но функции, передаваемые в параметрах, должны быть значениями. Функцию, объявленную с помощью ключевого слова `fun`, можно преобразовать в значение (`val`), добавив перед именем `::`.

Решение

Вот решение с использованием лямбда-выражений:

```
fun compose(f: (Int) -> Int, g: (Int) -> Int): (Int) -> Int = { x -> f(g(x)) }
```

Его можно упростить до:

```
fun compose(f: (Int) -> Int, g: (Int) -> Int): (Int) -> Int = { f(g(it)) }
```


Вот как эту функцию можно использовать для объединения функций `square` и `triple`:

```
val squareOfTriple = compose(::square, ::triple)
println(squareOfTriple(2))
36
```

Теперь достаточно очевидно, насколько мощной может быть эта идея! Но остаются две большие проблемы. Во-первых, функции-параметры могут принимать только целочисленные (`Int`) аргументы и возвращать целые числа. Давайте решим их.

3.2.9 Повторное использование функций

Чтобы обеспечить более широкие возможности использования функции, ее можно преобразовать в полиморфную функцию, добавив параметры типов.

Упражнение 3.2

Преобразуйте функцию `compose` в полиморфную функцию, использовав параметры типов.

Подсказка

Добавьте параметры типов между ключевым словом `fun` и именем функции. Затем замените типы `Int` соответствующими параметрами, следя за

порядком выполнения. Помните, что вы определяете `f ° g`, т. е. сначала вы должны вызвать `g`, а затем применить `f` к результату. Укажите тип возвращаемого значения. Если типы не совпадут, код не будет компилироваться.

Решение

В этом упражнении не требуется написать новую реализацию функции. Реализация останется прежней, как в неполиморфной версии. Речь идет об определении правильной сигнатуры:

```
fun <T, U, V> compose(f: (U) -> V, g: (T) -> U): (T) -> V = { f(g(it)) }
```

Здесь отчетливо видны преимущества строгой системы типов с поддержкой параметризации. С помощью параметров типов можно не только определить функцию `compose`, способную работать с любыми типами (при условии их совпадения), но, в отличие от версии `Int`, у вас нет шанса ошибиться. Если переставить местами вызовы `f` и `g`, функция перестанет компилироваться.

3.3 Дополнительные особенности функций

До сих пор мы рассматривали приемы создания, применения и композиции функций. Но пока так и не ответили на фундаментальный вопрос: зачем нужны функции, представленные в виде данных? Разве нельзя просто использовать `fun`-версии? Прежде чем ответить на этот вопрос, рассмотрим проблему функций с несколькими аргументами.

3.3.1 Функции с несколькими аргументами?

В разделе 3.1.5 я говорил, что нет такого понятия, как функция нескольких аргументов, но есть понятие функции кортежа. Кортеж может иметь любую размерность, и некоторые кортежи с небольшим числом элементов имеют даже свои имена: пара, триплет, квартет и т. д. Существуют и другие имена, но некоторые предпочитают называть их tuple2, tuple3, tuple4 и т. д. Kotlin имеет предопределенные типы `Pair` и `Triple`. Я также говорил, что элементы аргумента можно применять один за другим и в результате применения каждого следующего элемента возвращать новую функцию, за исключением случая применения последнего элемента.

Давайте попробуем определить функцию сложения двух целых чисел. Применим функцию к первому целому числу и вернем другую функцию. Тип выглядит следующим образом:

```
(Int) -> (Int) -> Int
```

В этом синтаксисе `(Int)` – это тип аргумента, а `(Int) -> Int` – тип возвращаемого значения. Чтобы вспомнить ассоциативность символа `->`, добавьте круглые скобки вокруг возвращаемого значения, т. е.:

```
(Int) -> ((Int) -> Int)
```

Аргумент имеет тип `Int`, а тип возвращаемого значения – это тип функции, принимающей аргумент `Int` и возвращающей `Int`.

Упражнение 3.3

Напишите функцию, складывающую два значения `Int`.

Решение

Эта функция принимает аргумент `Int` и возвращает функцию, преобразующую `Int` в `Int`, т. е. требуется написать функцию типа `(Int) -> (Int) -> Int`. Назовем ее `add` и реализуем с использованием лямбда-выражений. Конечный результат показан ниже:

```
val add: (Int) -> (Int) -> Int = { a -> { b -> a + b} }
```

Если вы предпочитаете использовать более краткие имена типов, создайте псевдоним:

```
typealias IntBinOp = (Int) -> (Int) -> Int

val add: IntBinOp = { a -> { b -> a + b} }
val mult: IntBinOp = { a -> { b -> a * b} }
```

Здесь `IntBinOp` – это псевдоним типа, сформированный из слов *Integer Binary Operator* (двухместный целочисленный оператор). Число аргументов не ограничено. При желании можно определить функцию с любым числом аргументов. Как я отмечал в первой части этой главы, такие функции как `add` и `mult` называют каррированными версиями эквивалентных функций, принимающих кортежи.

3.3.2 Применение каррированных функций

Итак, мы видели, как писать каррированные версии функций. Но как их применять на практике? Точно так же, как любые другие функции. Сначала функция применяется к первому аргументу, затем результат применяется к следующему аргументу и т. д. до конца списка аргументов. Например, вот как можно применить функцию `add` к числам `3` и `5`:

```
println(add(3)(5))
8
```

3.3.3 Реализация функций высшего порядка

В разделе 3.2.8 вы написали `fun`-функцию для объединения двух функций. Она принимала аргумент-кортеж с двумя функциями и возвращала функцию. Но вместо `fun`-функции (фактически метода), можно было бы использовать функцию-значение. Такие функции, принимающие функции в аргументах или возвращающие их в виде возвращаемых значений, называют *функциями высшего порядка* (Higher-Order Function, HOF).

Упражнение 3.4

Напишите функцию-значение, объединяющую две функции, например функции `square` и `triple`.

Решение

Вы легко справитесь с этим упражнением, если последуете правильной процедуре. В первую очередь, нужно написать тип. Эта функция будет работать с двумя аргументами, а значит, это каррированная функция. Два аргумента и возвращаемый тип будут функциями типа `(Int) -> Int`.

Дадим этому типу имя `T`. Вы должны написать функцию, которая принимает аргумент типа `T` (первый аргумент) и возвращает функцию, которая преобразует `T` (второй аргумент) в `T` (возвращаемое значение). То есть функция будет иметь тип:

```
(T) -> (T) -> T
```

Если заменить `T` его значением, получится фактический тип:

```
((Int) -> Int) -> ((Int) -> Int) -> (Int) -> Int
```

Главная проблема здесь в том, что строка получилась слишком длинной! Теперь добавим реализацию, которая выглядит намного проще, чем описание типа:

```
x -> { y -> { z -> x(y(z)) } } }
```

Вот как выглядит законченный код:

```
val compose: ((Int) -> Int) -> ((Int) -> Int) -> (Int) -> Int =
    { x -> { y -> { z -> x(y(z)) } }
}
```

Если хотите, можете воспользоваться преимуществами автоматического определения типов и опустить тип возвращаемого значения. Но тогда вам придется явно указать тип каждого аргумента:

```
val compose = { x: (Int) -> Int -> { y: (Int) -> Int -> { z: Int -> x(y(z)) } } }
```

Также можно определить псевдоним типа:

```
typealias IntUnaryOp = (Int) -> Int

val compose: (IntUnaryOp) -> (IntUnaryOp) -> IntUnaryOp =
    { x -> { y -> { z -> x(y(z)) } } }
```

Попробуйте проверить этот код с использованием функций `square` и `triple`:

```
typealias IntUnaryOp = (Int) -> Int

val compose: (IntUnaryOp) -> (IntUnaryOp) -> IntUnaryOp =
    { x -> { y -> { z -> x(y(z)) } } }

val square: IntUnaryOp = { it * it }

val triple: IntUnaryOp = { it * 3 }

val squareOfTriple = compose(square)(triple)
```

Здесь сначала `compose` применяется к первому аргументу, что дает в результате новую функцию для применения ко второму аргументу. В итоге получается функция, являющаяся композицией двух аргументов-функ-

ций. Применение этой новой функции, например, к числу 2, дает результат применения `triple` к 2, к которому затем применяется `square` (что соответствует определению композиции функций):

```
println(squareOfTriple(2))
36
```

Обратите внимание на порядок параметров: `triple` применяется первой, а `square` применяется после нее к результату, возвращаемому функцией `triple`.

3.3.4 Полиморфные функции высшего порядка

У вас получилась замечательная функция `compose`, но она способна объединять только функции `(Int) -> Int`. Полиморфная функция `compose` могла бы позволить создавать функции разных типов при условии, что тип возвращаемого значения одной совпадает с типом аргумента другой.

Упражнение 3.5 (сложное)

Напишите полиморфную версию функции `compose`.

Подсказка

В этом упражнении вы столкнетесь с проблемой отсутствия полиморфных свойств в Kotlin. В языке Kotlin можно создавать полиморфные классы, интерфейсы и функции, но нельзя определять полиморфные свойства. Решение проблемы заключается в сохранении функции в функции, классе или интерфейсе, а не в свойстве.

Решение

На первый взгляд, первый шаг заключается в «параметризации типов», как это было сделано в упражнении 3.3:

```
val <T, U, V> higherCompose: ((U) -> V) -> ((T) -> U) -> (T) -> V =
    { f ->
        { g ->
            { x -> f(g(x)) }
        }
    }
```

Но это невозможно, потому что Kotlin не поддерживает параметризованные свойства. Чтобы реализовать параметризацию, необходимо создать свойство в области видимости параметра типа. Определять параметры типов могут только классы, интерфейсы и функции, объявленные с помощью ключевого слова `fun`, поэтому свойство следует определить внутри одного из этих элементов. Наиболее простой является функция `fun`:

```
fun <T, U, V> higherCompose(): ((U) -> V) -> ((T) -> U) -> (T) -> V =
    { f ->
        { g ->
            { x -> f(g(x)) }
        }
    }
```

Функция fun с именем higherCompose() не принимает параметров и всегда возвращает одно и то же значение. Это константа. Тот факт, что она определена как функция fun, не имеет значения. Она не предназначена для объединения функций. Это просто функция fun, возвращающая функцию-значение, которая объединяет функции-значения. Вы можете не указывать возвращаемый тип, но тогда вы должны будете указать типы параметров:

```
fun <T, U, V> higherCompose() =
    { f: (U) -> V ->
        { g: (T) -> U ->
            { x: T -> f(g(x)) }
        }
    }
```

Теперь можно испытать эту функцию с triple и square:

```
val squareOfTriple = higherCompose()(square)(triple)
```

Но к сожалению, этот код не компилируется:

```
Error:(79, 24) Kotlin: Type inference failed:
        Not enough information to infer parameter T in fun <T, U, V>
        higherCompose(): ((U) -> V) -> ((T) -> U) -> (T) -> V
Please specify it explicitly.
```

Компилятор сообщил, что не смог определить фактические типы параметров T, U и V. Если вы думаете, что типа параметров ((Int) -> Int) должно быть достаточно, чтобы вывести типы T, U и V, значит, вы умнее компилятора Kotlin!

Чтобы решить проблему, нужно подсказать компилятору фактические типы T, U и V. Для этого добавьте информацию о типах после имени функции, как показано ниже:

```
val squareOfTriple = higherCompose<Int, Int, Int>()(square)(triple)
```

Упражнение 3.6 (теперь простое!)

Напишите функцию higherAndThen, объединяющую функции в аргументах в обратном порядке, т. е. вызов higherCompose(f, g) эквивалентен вызову higherAndThen(g, f).

Решение

```
fun <T, U, V> higherAndThen(): ((T) -> U) -> ((U) -> V) -> (T) -> V =
    { f: (T) -> U ->
        { g: (U) -> V ->
            { x: T -> g(f(x)) }
        }
    }
```

Проверка параметров функций

При наличии сомнений относительно порядка параметров такую функцию высшего порядка следует проверить с функциями разных типов. Тестирование с функциями (Int) -> Int не даст однозначного результата,

потому что позволяет передавать аргументы с функциями в любом порядке, из-за чего будет трудно обнаружить ошибку. Вот пример проверки с использованием функций разных типов:

```
fun testHigherCompose() {

    val f: (Double) -> Int = { a -> (a * 3).toInt() }
    val g: (Long) -> Double = { a -> a + 2.0 }

    assertEquals(Integer.valueOf(9), f(g(1L)))
    assertEquals(Integer.valueOf(9),
            higherCompose<Long, Double, Int>()(f)(g)(1L))
}
```

3.3.5 Анонимные функции

До сих пор мы использовали именованные функции. Но часто можно определять функции без имен и использовать их как безымянные, или анонимные, функции. Рассмотрим пример. Следующий код:


```
val f: (Double) -> Double = { Math.PI / 2 - it }
val sin: (Double) -> Double = Math::sin
val cos: Double = compose(f, sin)(2.0)
```

можно переписать иначе, использовав анонимную функцию:

```
val cosValue: Double =
        compose({ x: Double -> Math.PI / 2 - x }, Math::sin)(2.0)
```

Здесь используется функция `compose`, которая была определена на уровне пакета с использованием ключевого слова `fun`. Но тот же прием применим к функциям высшего порядка:

```
val cos = higherCompose<Double, Double, Double>()
                ({ x: Double -> Math.PI / 2 - x })( Math::sin)

val cosValue = cos(2.0)
```

Два параметра в определении функции `cos` заключены в отдельные пары круглых скобок. В отличие от `compose` функция `higherCompose` определяется в каррированной форме, применяя параметры по одному. Также обратите внимание, что лямбда-выражение можно вынести за круглые скобки:

```
val cos = higherCompose<Double, Double, Double>()()
                { x: Double -> Math.PI / 2 - x }( Math::sin)
```

Хочу заметить, что строки с кодом в этой книге разбиты, чтобы уместить их по ширине страницы, но даже с учетом этого последняя форма может показаться несколько неудачной, однако это рекомендуемое форматирование кода на Kotlin.

Когда использовать именованные функции, а когда – анонимные

Помимо особых случаев, когда нельзя использовать анонимные функции, выбор между анонимными и именованными функциями-значе-

ниями будет зависеть только от вас. (Функции, объявленные с помощью `fun`, всегда имеют имя.) Как правило, функции, которые используются только один раз, определяются как анонимные. Под словами «используются только один раз» подразумевается, что функция пишется один раз. Это не значит, что она создается только один раз.

В следующем примере определяется функция `fun` для вычисления косинуса значения типа `Double`. Реализация функции использует две анонимных функции в форме лямбда-выражения и ссылки на функцию:

```
fun cos(arg: Double) = compose({ x -> Math.PI / 2 - x }, Math::sin)(arg)
```

Не беспокойтесь о затратах ресурсов на создание анонимных функций. Kotlin не всегда создает новые объекты, когда вызывается подобная функция. Кроме того, создание таких объектов обходится довольно дешево. Поэтому, выбирая между анонимными и именованными функциями, следует учитывать только ясность и удобство сопровождения кода. Если вас волнует производительность и возможность повторного применения, используйте ссылки на функции как можно чаще.

Преодоление проблем с автоматическим определением типов

Механизм автоматического определения типов – еще одно узкое место анонимных функций. В предыдущем примере компилятор смог определить типы двух анонимных функций, потому что знал, что функция `compose` принимает в аргументах две функции:

```
fun <T, U, V> compose(f: (U) -> V, g: (T) -> U): (T) -> V = { f(g(it)) }
```

Но так бывает не всегда. Если во втором аргументе вместо ссылки на функцию передать лямбда-выражение:

```
fun cos(arg: Double) =
    compose({ x -> Math.PI / 2 - x }, { y -> Math.sin(y)})(arg)}
```

компилятор растеряется и выведет сообщение об ошибке:

```
Error:(48, 28) Kotlin: Type inference failed: Not enough information to
infer parameter T in fun <T, U, V> compose(f: (U) -> V, g: (T) -> U):
(T) -> V
Please specify it explicitly.
Error:(48, 38) Kotlin: Cannot infer a type for this parameter.
                       Please specify it explicitly.
Error:(48, 64) Kotlin: Cannot infer a type for this parameter.
                       Please specify it explicitly.
```

Kotlin не смог определить типы аргументов. Чтобы исправить эту ошибку, нужно добавить аннотации типов:

```
fun cos(arg: Double) =
    compose({ x: Double -> Math.PI / 2 - x },
            { x: Double -> Math.sin(x)})(arg)
```

Это хорошая причина отдать предпочтение ссылкам на функции.

3.3.6 Локальные функции

Вы уже видели, как определять локальные функции-значения внутри других функций, но Kotlin позволяет также определять локальные функции `fun` внутри других функций, например:

```
fun cos(arg: Double): Double {
    fun f(x: Double): Double = Math.PI / 2 - x
    fun sin(x: Double): Double = Math.sin(x)
    return compose(::f, ::sin)(arg)
}
```

3.3.7 Замыкания

Вы видели, что при вычислении своего результата чистые функции не должны зависеть ни от чего, кроме своих аргументов. Функции в Kotlin часто обращаются к элементам вне самой функции, находящимся на уровне пакета, класса или объекта. Функции могут даже обращаться к членам объектов-компаньонов других классов или других пакетов.

Я уже говорил, что чистые функции – это функции со ссылочной прозрачностью, т. е. не имеющие наблюдаемых эффектов, кроме возврата значения. А что можно сказать о функциях, возвращаемое значение которых зависит не только от аргументов, но и от элементов в объемлющей области видимости? Вы уже видели один такой пример, когда элементы в объемлющей области видимости можно было считать неявными параметрами функций.

Это также относится к лямбда-выражениям, которые в Kotlin не имеют ограничений, свойственных лямбда-выражениям в Java: они могут обращаться к изменяемым переменным в объемлющей области видимости. Рассмотрим пример:

```
val taxRate = 0.09
fun addTax(price: Double) = price + price * taxRate
```

Здесь функция `addTax` *замкнута* на переменной `taxRate`. Важно отметить, что `addTax` не является функцией `price`, потому что не всегда возвращает один и тот же результат для одного и того же аргумента. Но ее можно представить как функцию кортежа `(price, taxRate)`.

Замыкания совместимы с чистыми функциями, если рассматривать их как дополнительные неявные аргументы. Они могут вызвать проблемы при реорганизации кода, а также при передаче функций в параметрах другим функциям. При неумеренном их использовании программы могут получаться трудно читаемыми и сложными для поддержки.

Один из способов улучшить читаемость программ – сделать их более модульными, т. е. использовать части программы как независимые модули. Этого можно достичь с помощью функций кортежей аргументов:

```
val taxRate = 0.09

fun addTax(taxRate: Double, price: Double) = price + price * taxRate

println(addTax(taxRate, 12.0))
```

Это относится и к функциям-значениям:

```
val taxRate = 0.09

val addTax = { taxRate: Double, price: Double -> price + price * taxRate }

println(addTax(taxRate, 12.0))
```

Функция `addTax` принимает один аргумент, являющийся парой значений `Double`. В отличие от Java в Kotlin допускается использовать аргументы с числом элементов больше 2. (В Java можно использовать интерфейс `Function` для одного аргумента и интерфейс `BiFunction` для пары аргументов. Для аргументов с тремя и более элементами необходимо определить свои интерфейсы.)

Но вы уже видели, что тот же результат можно получить, если использовать каррированную версию. Каррированная функция принимает единственный аргумент и возвращает функцию, принимающую один аргумент и возвращающую…, и т. д., пока не будет возвращено конечное значение. Вот каррированная версия функции-значения `addTax`:

```
val taxRate = 0.09

val addTax = { taxRate: Double ->
                { price: Double ->
                    price + price * taxRate
                }
             }

println(addTax(taxRate)(12.0))
```

Создавать каррированные версии функций `fun` не имеет большого смысла. Первую функцию можно определить с помощью `fun`, но вы должны возвращать функцию-значение, которая не может быть функцией `fun`.

3.3.8 Частичное применение функций и автоматическое каррирование

Версии с замыканием и каррированием в предыдущем примере дают одинаковые результаты и могут рассматриваться как эквивалентные. На самом деле они имеют *разную семантику*. Как я уже говорил, эти два параметра играют совершенно разные роли. Налоговая ставка `taxRate` не должна часто меняться, тогда как цена `price` может изменяться от вызова к вызову. Это ясно видно в версии с замыканием. Функция замкнута на параметре, который не изменяется (потому что это `val`). В каррированной версии оба аргумента могут изменяться при каждом вызове, даже притом что ставка налога `taxRate` будет меняться не чаще, чем в версии с замыканием.

Изменение налоговой ставки обычно требуется, например, если есть несколько налоговых ставок для разных категорий товаров или для разных пунктов доставки. В традиционном объектно-ориентированном программировании с этим поможет справиться превращение класса в параметризованный *налоговый компьютер*. Например:

```
class TaxComputer(private val rate: Double) {
    fun compute(price: Double): Double = price * rate + price
}
```

Используя этот класс, можно создать несколько экземпляров `TaxComputer` для разных налоговых ставок и использовать эти экземпляры по мере необходимости:

```
val tc9 = TaxComputer(0.09)
val price = tc9.compute(12.0)
```

Того же результата можно добиться путем частичного применения каррированной функции:


```
val tc9 = addTax(0.09)
val price = tc9(12.0)
```

Здесь использована функция `addTax`, которая была определена в конце раздела 3.3.7. Значение `tc9` теперь имеет тип `(Double) -> Double`; это функция, принимающая аргумент `Double` и возвращающая значение `Double` цены с добавленным налогом.

Как видите, каррирование и частичное применение тесно связаны. Каррирование состоит в замене функции кортежа новой функцией, которую можно применять по частям к одному аргументу за другим. Это основное отличие каррированных функций от функций кортежей: функция кортежа вычисляет все аргументы до применения функции.

При использовании каррированной версии все аргументы должны быть известны до полного применения функции, но вычислять аргументы можно по одному и применять функцию к каждому в отдельности. Вы не обязаны полностью применить функцию. Функцию трех аргументов можно преобразовать в функцию кортежа, который создает функцию одного аргумента.

Абстракцию каррирования и частичного применения функций можно сделать автоматической. В предыдущих разделах вы использовали в основном каррированные функции, а не функции кортежей. Использование каррированных функций дает большое преимущество: простоту частичного применения.

Упражнение 3.7 (простое)

Напишите функцию `fun` для частичного применения каррированной функции двух аргументов к ее первому аргументу.

Решение

Для этого ничего не нужно делать! Вот сигнатура этой функции:

```
fun <A, B, C> partialA(a: A, f: (A) -> (B) -> C): (B) -> C
```

Как видите, для частичного применения к первому аргументу достаточно применить второй аргумент (функцию) к первому:

```
fun <A, B, C> partialA(a: A, f: (A) -> (B) -> C): (B) -> C = f(a)
```

(Если вам интересно увидеть, как используется функция `partialA`, загляните в модульный тест к этому упражнению, который можно найти в примерах кода к этой книге.)

Возможно, вы заметили, что оригинальная функция имеет тип `(A) -> (B) -> C`. А что, если понадобится частично применить функцию ко второму аргументу?

Упражнение 3.8

Напишите функцию `fun` для частичного применения каррированной функции двух аргументов к ее второму аргументу.

Решение

Опираясь на предыдущий опыт, нетрудно догадаться, что решением является функция со следующей сигнатурой:

```
fun <A, B, C> partialB(b: B, f: (A) -> (B) -> C): (A) -> C
```

Это упражнение немного сложнее, но все еще достаточно простое, если внимательно рассмотреть типы. Запомните: вы постоянно должны доверять типам! Они не всегда подскажут вам немедленное решение, но обязательно приведут к нему. Эта функция имеет только одну возможную реализацию, поэтому если вы найдете реализацию, которая компилируется, то можете быть уверены, что она верная!

Итак, нам известно, что мы должны вернуть функцию типа `(A) -> C`. Начнем реализацию, написав следующий код:

```
fun <A, B, C> partialB(b: B, f: (A) -> (B) -> C): (A) -> C =
    { a: A ->
    }
```

Здесь `a` – это переменная типа `A`. После стрелки вправо нужно добавить выражение, включающее функцию `f` и переменные `a` и `b`, которое возвращает функцию типа `(A) -> C`. Функция `f` имеет тип `(A) -> (B) -> C`, поэтому мы можем начать с ее применения к `A` и получим:

```
fun <A, B, C> partialB(b: B, f: (A) -> (B) -> C): (A) -> C =
    { a: A ->
        f(a)
    }
```

Это дает нам функцию `(B) -> C`. Нам нужен тип `C`, и у нас уже есть `B`, поэтому ответ снова находится просто:

```
fun <A, B, C> partialB(b: B, f: (A) -> (B) -> C): (A) -> C =
    { a: A ->
        f(a)(b)
    }
```

Вот и все! Фактически от нас потребовалось лишь последовать за типами.

Как я уже сказал, самое главное заключается в том, что у нас уже была каррированная версия функции. Вы быстро научитесь писать такие кар-

рированные функции. Одна из задач, которая часто возникает при попытке довести абстракцию до предела и написать код, пригодный для многократного использования, – преобразование функций с кортежем аргументов в каррированные функции. Как вы только что видели, это очень просто.

Упражнение 3.9 (простое)

Преобразуйте следующую функцию в каррированную версию:

```
fun <A, B, C, D> func(a: A, b: B, c: C, d: D): String = "$a, $b, $c, $d"
```

(Я понимаю, что в целом функция бесполезная, но это всего лишь упражнение.)

Решение

И снова нам почти ничего не нужно делать, только заменить запятые стрелками вправо и добавить круглые скобки. Но давайте вспомним, что мы должны определить эту функцию в области видимости, которая принимает параметры типов, т. е. она не может быть свойством, а значит, мы должны определить ее в классе, интерфейсе или функции `fun` со всеми необходимыми параметрами типов.

Ниже приводится решение с использованием функции. Сначала напишем объемлющую функцию `fun` с параметрами типов:

```
fun <A,B,C,D> curried()
```

Затем подумаем, как должна выглядеть сигнатура требуемой функции. Первый параметр будет иметь тип `A`, поэтому запишем:

```
fun <A,B,C,D> curried(): (A) ->
```

Затем проделаем то же самое со вторым параметром типа:

```
fun <A,B,C,D> curried(): (A) -> (B) ->
```

И продолжим, пока параметры не кончатся:

```
fun <A,B,C,D> curried(): (A) -> (B) -> (C) -> (D) ->
```

Добавим возвращаемый тип:

```
fun <A,B,C,D> curried(): (A) -> (B) -> (C) -> (D) -> String
```

В реализации перечислим все параметры, разделив их стрелкой вправо и открывающими фигурными скобками (сначала вставим скобку, затем параметр, а за ним стрелку вправо):

```
fun <A,B,C,D> curried() =
    { a: A ->
        { b: B ->
            { c: C ->
                { d: D ->
                }
            }
        }
    }
```

В заключение добавим реализацию из оригинальной функции и закроем все фигурные скобки:

```
fun <A,B,C,D> curried() =
    { a: A ->
        { b: B ->
            { c: C ->
                { d: D ->
                    "$a, $b, $c, $d"
                }
            }
        }
    }
```


Этот способ создания каррированных версий применим к функциям с любым числом аргументов.

Упражнение 3.10

Напишите функцию, возвращающую каррированную версию функции `(A, B) -> C`.

Решение

И снова последуем за типами. Мы знаем, что функция принимает параметр типа `(A, B) -> C` и должна вернуть результат типа `(A) -> (B) -> C`, т. е. она имеет следующую сигнатуру:

```
fun <A, B, C> curry(f: (A, B) -> C): (A) -> (B) -> C
```

Наша реализация должна вернуть каррированную функцию двух аргументов; соответственно, начнем с того, что напишем следующий код:

```
fun <A, B, C> curry(f: (A, B) -> C): (A) -> (B) -> C =
    { a ->
        { b ->
        }
    }
```


Наконец, нам нужно получить возвращаемый тип. Для этого используем функцию `f` и применим ее к параметрам `a` и `b`:

```
fun <A, B, C> curry(f: (A, B) -> C): (A) -> (B) -> C =
    { a ->
        { b ->
            f(a, b)
        }
    }
```

И снова, если функция компилируется, она не может быть неправильной. Это одно из многочисленных преимуществ, которые дает строгая система типов! (Это не всегда верно, но в следующих главах я расскажу, как сделать так, чтобы это было верно как можно чаще.)

3.3.9 Перестановка аргументов частично примененных функций

Имея функцию двух аргументов, мы можем пожелать применить только первый аргумент и получить частично примененную функцию. Допустим, у нас есть следующая функция:

```
val addTax: (Double) -> (Double) -> Double =
    { x ->
        { y ->
            y + y / 100 * x
        }
    }
```

У нас может появиться желание применить ее к налоговой ставке и получить новую функцию одного аргумента, которую затем можно будет применить к любой цене:

```
val add9percentTax: (Double) -> Double = addTax(9.0)
```

Затем мы сможем прибавить налог к цене, как показано ниже:

```
val priceIncludingTax = add9percentTax(price);
```

Вроде бы все хорошо, но теперь представьте, что исходная функция была объявлена чуть иначе:

```
val addTax: (Double) -> (Double) -> Double =
    { x ->
        { y ->
            x + x / 100 * y
        }
    }
```

В этом случае первый аргумент представляет цену. Частичное применение к цене не имеет большого смысла. А можно ли получить версию этой функции, частично примененную к налоговой ставке? (При этом предположим, что у нас нет доступа к ее реализации.)

Упражнение 3.11

Напишите функцию `fun`, которая меняет местами аргументы каррированной функции.

Решение

Следующая функция возвращает каррированную функцию с аргументами, переставленными в обратном порядке. Ее можно обобщить на любое количество аргументов с любым порядком следования:

```
fun <T, U, V> swapArgs(f: (T) -> (U) -> V): (U) -> (T) -> (V) =
    { u -> { t -> f(t)(u) } }
```

С помощью этой функции можно получить версию функции, частично примененную к любому из двух аргументов. Например, имея функцию

вычисления суммы ежемесячного платежа по кредиту по величине процентной ставки и сумме кредита

```
val payment = { amount -> { rate -> ... } }
```

легко создать функцию одного аргумента, вычисляющую сумму ежемесячного платежа для фиксированной суммы кредита и разных процентных ставок или для фиксированной процентной ставки и разных сумм кредита.

Объявление единичной функции

Вы видели, что функции можно рассматривать как данные. Их можно передавать в аргументах другим функциям, возвращать из функций и использовать в операциях точно так же, как целые числа или строки. В будущих упражнениях вы будете применять операции к функциям, и вам потребуется нейтральный элемент для этих операций. *Нейтральный элемент* действует как `0` в операции сложения, как `1` в операции умножения и как пустая строка в операции конкатенации строк.

Нейтральный элемент является нейтральным только для данной операции. Для сложения целых чисел единица не является нейтральным элементом, а для умножения `0` – еще менее нейтральный элемент. Здесь я говорю о нейтральном элементе для композиции функций. Такая функция просто возвращает свой аргумент. По этой причине она называется *единичной* функцией. Более того, в контексте таких операций, как сложение, умножение и конкатенация строк, вместо термина «нейтральный элемент» часто используется «единичный элемент». Единичная функция в Kotlin выражается просто:


```
val identity = { it }
```

Использование правильных типов

В предыдущих примерах для представления бизнес-объектов, например цен и налоговых ставок, мы использовали стандартные типы, такие как `Int`, `Double` и `String`. В программировании это обычная практика, но может вызывать проблемы, которых следует избегать. Как я уже сказал, вы должны доверять типам больше, чем именам. Имя `"price"` не делает `Double` ценой. Оно лишь отражает наше намерение. Аналогично имя `"taxRate"`, присвоенное другому значению `Double`, показывает другое наше намерение, проверить соответствие которому не сможет ни один компилятор.

Для большей безопасности программ нужно использовать более мощные типы, которые компилятор сможет проверить. Это предотвратит неправильное использование таких типов, как, например, сложение `taxRate` и `price`. Если вы по неосторожности допустите это, компилятор увидит только сложение `Double` и `Double` – вполне законную, но в корне ошибочную операцию.

Исключение проблем с помощью стандартных типов

Рассмотрим простую задачу и как ее решение с использованием стандартных типов приводит к появлению проблем. Представьте, что у нас

есть товары, характеризующиеся названием, ценой и весом, и нужно создать накладные, отражающие продажи продукта. В этих накладных должны быть указаны: название товара, количество, общая стоимость и общий вес. Мы можем представить товар следующим классом:

```
data class Product(val name: String, val price: Double, val weight: Double)
```

Далее каждую строку в заказе можно представить классом `OrderLine`:

```
data class OrderLine(val product: Product, val count: Int) {

    fun weight() = product.weight * count

    fun amount() = product.price * count
}
```

Выглядит как старый добрый объект, инициализированный экземпляром `Product` и значением типа `Int` и представляющий одну строку в заказе. Он также имеет функции, возвращающие общую стоимость и общий вес для этой строки.

Продолжим решать задачу с использованием стандартных типов и задействуем список `List<OrderLine>` для представления заказа. Листинг 3.2 демонстрирует, как осуществляется обработка таких заказов.

Листинг 3.2. Обработка заказов

```
package com.fpinkotlin.functions.listing03_02

data class Product(val name: String, val price: Double, val weight: Double)

data class OrderLine(val product: Product, val count: Int) {

    fun weight() = product.weight * count

    fun amount() = product.price * count
}

object Store { ①

    @JvmStatic ②
    fun main(args: Array<String>) {
        val toothPaste = Product("Tooth paste", 1.5, 0.5)
        val toothBrush = Product("Tooth brush", 3.5, 0.3)
        val orderLines = listOf(
                OrderLine(toothPaste, 2),
                OrderLine(toothBrush, 3))
        val weight = orderLines.sumByDouble { it.amount() }
        val price = orderLines.sumByDouble { it.weight() }
        println("Total price: $price")
        println("Total weight: $weight")
    }
}
```

① **`Store` – это объект-синглтон**

② **Аннотация `@JvmStatic` обеспечивает возможность вызова функции `main`, как если бы она была статическим методом Java**

Если запустить эту программу, она выведет:

```
Total price: 1.9
Total weight: 13.5
```

Это замечательно, но неправильно! Ошибка очевидна, но проблема в том, что компилятор ее не заметил. (Вы можете увидеть ошибку, заглянув в код `Store`.) Но та же ошибка могла быть допущена при создании `Product`.

Единственный способ выловить эту ошибку – проверить программу, но тесты не смогут доказать правильность программы. Они могут только доказать, что нельзя доказать неправильность программы, написав другую программу (которая, кстати, тоже может оказаться неправильной). Для тех, кто не заметил ошибку (что маловероятно), подскажу, что проблема заключается в строках:

```
val weight = orderLines.sumByDouble { it.amount() }
val price = orderLines.sumByDouble { it.weight() }
```

Мы по ошибке перепутали цены и вес, и компилятор не заметил этого, потому что и то, и другое – это значения `Double`.

> **ПРИМЕЧАНИЕ** Знакомые с принципами моделирования могут вспомнить старое правило: классы не должны иметь несколько свойств одного типа. Вместо этого они должны иметь одно свойство-массив требуемого размера. В данном случае это означает, что `Product` должен иметь одно свойство-массив типа `Double` с размером 2. Это явно неправильный способ решения задачи, но это хорошее правило, чтобы запомнить его. Если вы обнаружите, что моделируете объекты с несколькими свойствами одного типа, скорее всего, вы делаете это неправильно.

Что можно предпринять, чтобы избежать таких проблем? Во-первых, вы должны понимать, что цена и вес – это не числа, а количества. Количества могут быть числами, но цена – это количество денежных единиц, а вес – количество единиц веса. Вы никогда не должны складывать килограммы с рублями.

Определение типов-значений

Для решения проблемы используем типы-значения. *Типы-значения* – это типы, представляющие значения. Вот как можно определить тип-значение для представления цены:

```
data class Price(val value: Double)
```

А так определяется тип-значение для представления веса:

```
data class Weight(val value: Double)
```

Но это не решает проблему, потому что следующее выражение считается допустимым:

```
val total = price.value + weight.value
```

В действительности мы должны определить операцию сложения в классах `Price` и `Weight`:

```
data class Price(val value: Double) {
    operator fun plus(price: Price) = Price(this.value + price.value)
}

data class Weight(val value: Double) {
    operator fun plus(weight: Weight) = Weight(this.value + weight.value)
}
```

Ключевое слово `operator` указывает, что имя функции можно использовать в инфиксной позиции в выражении. Кроме того, так как функции дано имя `plus`, в выражениях вместо этого имени можно использовать символ `+`, как в следующем примере:

```
val totalPrice = Price(1.0) + Price(2.0)
```

Нам также понадобится операция умножения, но умножение немного отличается. Сложение складывает значения одного типа, тогда как умножение может умножать значение некоторого типа на число. Умножение – некоммутативная операция, если применяется не только к числам. Вот пример реализации умножения для `Price`:

```
data class Price(val value: Double) {

    fun plus(price: Price) = Price(this.value + price.value)

    fun times(num: Int) = Price(this.value * num)
}
```

Теперь нам больше не нужно использовать `sumByDouble` для вычисления общей стоимости товаров в списке. Мы можем определить эквивалентную функцию `sumByPrice`. Если вам интересно, загляните в реализацию `sumByDouble` и попробуйте адаптировать ее для цен типа `Price`. Но есть способ лучше.

Преобразование коллекции в единственный элемент называется сверткой и выполняется функциями `fold` и `reduce`. Различия между этими функциями не всегда очевидны, но многие сводят их к двум условиям:

- наличие (`fold`) или отсутствие (`reduce`) начального значения;
- результат должен быть того же типа (`reduce`), что и элементы, или нет (`fold`).

Еще одно важное отличие: если коллекция пустая, `reduce` не имеет результата, а `fold` вернет начальное значение, указанное вами. В главе 6 вы ближе познакомитесь с операциями свертки. А пока просто будем использовать функцию `fold`, которая поддерживается всеми коллекциями в Kotlin. Эта функция принимает два параметра: начальное значение и функцию, которая объединит текущий результат со следующим элементом.

Функция `reduce` действует подобно `fold`, но не имеет начального значения. Роль начального значения в ней выполняет первый элемент коллекции, а значит, тип ее результата всегда совпадает с типом элементов

коллекции. Если применить `reduce` к пустой коллекции, она вернет `null`, ошибку или какое-то иное представление отсутствующего результата.

В следующем примере мы используем `fold` и передаем в качестве начальных значений нулевую стоимость (`Price(0.0)`) и нулевой вес (`Weight(0.0)`). Во втором параметре передаются функции сложения, которые мы только что определили. Чтобы получить функцию-значение, мы использовали лямбда-выражение:

```
val zeroPrice = Price(0.0)
val zeroWeigth = Weight(0.0)
val priceAddition = { x, y -> x + y }
```

Класс `Product` нужно немного изменить, как показано ниже:

```
data class Product(val name: String, val price: Price, val weight: Weight)
```

Класс `OrderLine` не требует изменений:

```
data class OrderLine(val product: Product, val count: Int) {

    fun weight() = product.weight * count

    fun amount() = product.price * count
}
```

Оператор `*` теперь автоматически будет замещен вызовом функций `times`, которые мы определили. Теперь можно переписать объект `Store`, задействовав новые типы и операции:

```
object Store {

    @JvmStatic
    fun main(args: Array<String>) {
        val toothPaste = Product("Tooth paste", Price(1.5), Weight(0.5))
        val toothBrush = Product("Tooth brush", Price(3.5), Weight(0.3))
        val orderLines = listOf(
                OrderLine(toothPaste, 2),
                OrderLine(toothBrush, 3))
        val weight: Weight =
                orderLines.fold(Weight(0.0)) { a, b -> a + b.weight() }
        val price: Price =
                orderLines.fold(Price(0.0)) { a, b -> a + b.amount() }
        println("Total price: $price")
        println("Total weight: $weight")
    }
}
```

Теперь любая попытка смешать типы не останется без внимания компилятора. Это означает, что сейчас вы должны указывать типы для `val weight: Weight` и `val price: Price`. Kotlin может автоматически определять типы, но, указывая их явно, вы даете компилятору возможность сообщить вам, отличаются ли типы, определенные им, от ожидаемых вами.

Но можно поступить еще лучше. Во-первых, можно добавить дополнительную проверку в типы `Price` и `Weight`. Ни один из них не должен давать возможности создавать нулевые значения кроме как внутри самого класса, чтобы получить нейтральный элемент. Чтобы обеспечить такое

ограничение, можно определить приватный конструктор и фабричную функцию. Например, для `Price`:

```
data class Price private constructor (private val value: Double) {

    override fun toString() = value.toString()

    operator fun plus(price: Price) = Price(this.value + price.value)

    operator fun times(num: Int) = Price(this.value * num)

    companion object {
        val identity = Price(0.0)

        operator fun invoke(value: Double) =
            if (value > 0)
                Price(value)
            else
                throw IllegalArgumentException(
                        "Price must be positive or null")
    }
}
```

Теперь конструктор объявлен приватным, и добавился объект-компаньон с функцией `invoke`, объявленной как `operator`, которая содержит код проверки. Имя `invoke` в комбинации с ключевым словом `operator` интерпретируется компилятором особо, так же как имена `plus` и `times`.

В данном случае выполняется перегрузка оператора `()`, соответствующего вызову функции. Благодаря этому фабричную функцию можно использовать в роли конструктора, который теперь стал приватным. Приватный конструктор вызывается из объекта-компаньона для создания результата, возвращаемого функцией. Он также используется для создания значения `identity`, используемого в операциях свертки. Теперь изменим реализацию объекта `Store` с учетом этого нововведения:

```
object Store {

    @JvmStatic
    fun main(args: Array<String>) {
        val toothPaste = Product("Tooth paste", Price(1.5), Weight(0.5))
        val toothBrush = Product("Tooth brush", Price(3.5), Weight(0.3))
        val orderLines = listOf(
                OrderLine(toothPaste, 2),
                OrderLine(toothBrush, 3))
        val weight: Weight =
                orderLines.fold(Weight.identity) { a, b ->
                    a + b.weight()
                }
        val price: Price =
                orderLines.fold(Price.identity) { a, b ->
                    a + b.amount()
                }
        println("Total price: $price")
        println("Total weight: $weight")
    }
}
```

Код создания цены и веса не изменился. Синтаксис вызова функции `invoke` похож на синтаксис вызова конструктора, использовавшийся прежде. «Нулевое» значение (`identity`), используемое в вызове `fold`, извлекается из объекта-компаньона. Его нельзя получить вызовом функции `invoke` из-за кода проверки, который возбуждает исключение при попытке создать нулевое значение.

ПРИМЕЧАНИЕ Приватный конструктор в классе данных не является приватным в полном смысле, потому что экспортируется сгенерированной функцией `copy`. Но это не проблема, потому что при этом копируется уже проверенный объект.

Итоги

- Функция описывает связь между исходным и целевым множествами. Она устанавливает соответствие между элементами исходного множества (областью определения) и элементами целевого множества (областью значений).
- Чистые функции не имеют видимых эффектов, кроме возвращаемого значения.
- Функции имеют только один аргумент, который может быть кортежем с несколькими элементами.
- Функции кортежей можно каррировать, чтобы получить возможность ее применения к элементам кортежа по одному.
- Каррированную функцию, которая применяется только к некоторым своим аргументам, называют частично примененной.
- Функции в Kotlin могут определяться с помощью ключевого слова `fun` (в действительности они являются методами) или как функции-значения, которые интерпретируются как данные.
- Функции-значения можно реализовать с использованием лямбда-выражений или ссылок на функции `fun`.
- Функции можно объединять и создавать новые.
- Лямбда-выражения и ссылки на функции можно использовать везде, где ожидаются функции-значения.
- Типы-значения повышают безопасность программ, позволяя компилятору определять проблемы.

4 Рекурсия, сорекурсия и мемоизация

Эта глава охватывает следующие темы:

- рекурсия и сорекурсия;
- рекурсивные функции;
- взаимно рекурсивные функции;
- мемоизация.

Возможность создавать рекурсивные функции присуща многим языкам программирования, но в Java она используется довольно редко. Причина в не самой удобной реализации этого механизма. К счастью, Kotlin предлагает гораздо лучшую поддержку, поэтому вы можете без опаски использовать рекурсию.

Многие алгоритмы имеют рекурсивное определение. Реализация таких алгоритмов на нерекурсивных языках заключается в преобразовании их в нерекурсивную форму. Использование языка, поддерживающего рекурсию, не только упрощает код, но также позволяет писать программы, более явно отражающие намерение (с использованием известных оригинальных алгоритмов). Такие программы, как правило, гораздо легче читать и понимать. А поскольку программирование – это в большей степени чтение программ, чем их написание, важно создавать программы, которые четко показывают, что делается, а не как это делается.

ВНИМАНИЕ Имейте в виду, что рекурсию, как и циклы, часто лучше абстрагировать в функции, а не использовать ее непосредственно.

4.1 Рекурсия и сорекурсия

Сорекурсия – это способ вычислений по шагам, когда результаты выполнения одного шага используются как входные данные следующего шага. *Рекурсия* – это та же операция, но начинающаяся с последнего шага. Возьмем в качестве примера список символов, которые требуется объединить в строку.

4.1.1 Реализация сорекурсии

Предположим, что у нас есть функция `append`, принимающая пару аргументов (`String`, `Char`) и возвращающая строку из первого аргумента с добавленным в конец символом из второго аргумента:

```
fun append(s: String, c: Char): String = "$s$c"
```

С этой функцией ничего нельзя сделать, не имея начальной строки. То есть нам нужен дополнительный элемент: пустая строка `""`. Имея два элемента, можно сконструировать требуемый результат. На рис. 4.1 показана обработка списка.

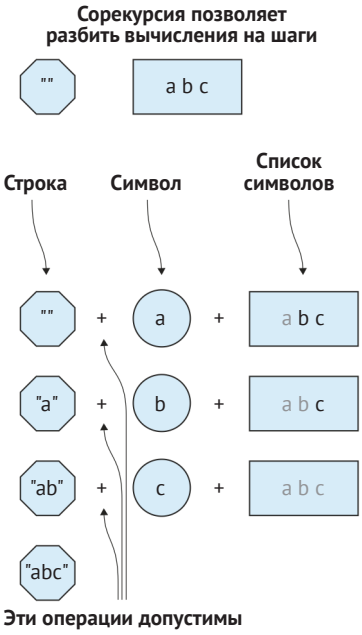


Рис. 4.1 Сорекурсивное преобразование списка символов в строку

В коде этот процесс можно описать так:

```
fun toString(list: List<Char>, s: String): String =
    if (list.isEmpty())
        s
    else
        toString(list.subList(1, list.size), append(s, list[0]))
```

Обратите внимание, что выражение `list[0]` возвращает первый элемент списка и соответствует функции, обычно известной под именем `head`. С другой стороны, выражение `list.subList(1, list.size)` соответствует функции с именем `tail`, которая возвращает остальную часть (хвост) списка. Эти операции можно абстрагировать в отдельные функции, но в этом случае важно не забыть обработать случай с пустым списком. В примере выше этого не требуется, потому что пустой список фильтруется выражением `if..else`.

Более идиоматичное решение использует функции `drop` и `first`:


```
fun toString(list: List<Char>, s: String): String =
    if (list.isEmpty())
        s
    else
        toString(list.drop(1), append(s, list.first()))
```

Использование доступа по индексу позволяет упростить сравнение сорекурсии и рекурсии. Единственная проблема этой реализации заключается в необходимости передавать в функцию `toString` пустую строку, играющую роль дополнительного аргумента (точнее, роль дополнительной части аргумента). Это решение можно упростить, написав еще одну функцию, воспользовавшись поддержкой вложенных функций в Kotlin:


```
fun toString(list: List<Char>): String {
    fun toString(list: List<Char>, s: String): String =
        if (list.isEmpty())
            s
        else
            toString(list.subList(1, list.size), append(s, list[0]))

    return toString(list, "")
}
```

Здесь мы вынуждены использовать блочную функцию – с телом в фигурных скобках, чтобы заключить в него функцию `toString`. Также потребовалось явно указать типы возвращаемых значений обеих функций по следующим причинам:

- тип возвращаемого значения включающей блочной функции должен объявляться явно, потому что иначе подразумевается тип `Unit` (который в языке Kotlin является эквивалентом типа `void` в Java);
- тип возвращаемого значения включаемой функции должен объявляться явно, потому что эта функция вызывает саму себя, из-за чего компилятор Kotlin оказывается не в состоянии определить тип возвращаемого значения.

Другое решение – добавить в функцию второй параметр со значением по умолчанию:

```
fun toString(list: List<Char>, s: String = ""): String =
    if (list.isEmpty())
        s
```

```
    else
        toString(list.subList(1, list.size), append(s, list[0]))
```

4.1.2 Реализация рекурсии

Предыдущая реализация возможна только потому, что в нашем распоряжении была функция append, которая добавляет символ в конец строки. Теперь поразмышляйте, как можно решить ту же задачу, если бы у нас имелась только следующая функция:

```
fun prepend(c: Char, s: String): String = "$c$s"
```

Можно было бы начать с конца списка. С этой целью можно перевернуть список перед вычислениями или изменить реализацию так, чтобы она возвращала последний элемент списка и список без последнего элемента. Вот одно из возможных решений:

```
fun toString(list: List<Char>): String {
    fun toString(list: List<Char>, s: String): String =
        if (list.isEmpty())
            s
        else
            toString(list.subList(0, list.size - 1),
                     prepend(list[list.size - 1], s))

    return toString(list, "")
}
```

Однако оно будет работать только со списками, поддерживающими этот тип доступа, такими как индексированный список или двусвязный список, также известный как *двусторонняя очередь*. В случае с односвязными списками нет иного выбора, кроме как перевернуть список, что далеко не эффективно.

Худший случай представляют бесконечные списки. Можно подумать, что с бесконечными списками ничего нельзя сделать, но это не так. В Kotlin к бесконечным спискам можно применять функции, которые создают другие бесконечные списки. Вы можете конструировать такие функции, чтобы получить желаемый результат, а затем обрезать результат до его вычисления.

Рассмотрим такой пример: требуется найти 100 первых простых чисел (в этом не особенно много проку, но это всего лишь пример), и для этого нужно перебрать элементы бесконечного списка целых чисел, выбрать из него простые числа и остановиться после 100-го простого числа. Конечно, в этом примере мы не можем начать с переворачивания списка. Однако решение есть – использовать рекурсию вместо сорекурсии, как показано на рис. 4.2.

На рис. 4.2 мы видим, что никаких вычислений не выполняется, пока не будет достигнуто условие завершения (в данном случае последний элемент списка). Как результат, промежуточные шаги приходится хранить где-то, пока их не станет возможно выполнить. Этот процесс можно выразить, как показано ниже:

```
fun toString(list: List<Char>): String =
    if (list.isEmpty())
        ""
    else
        prepend(list[0], toString(list.subList(1, list.size)))
```

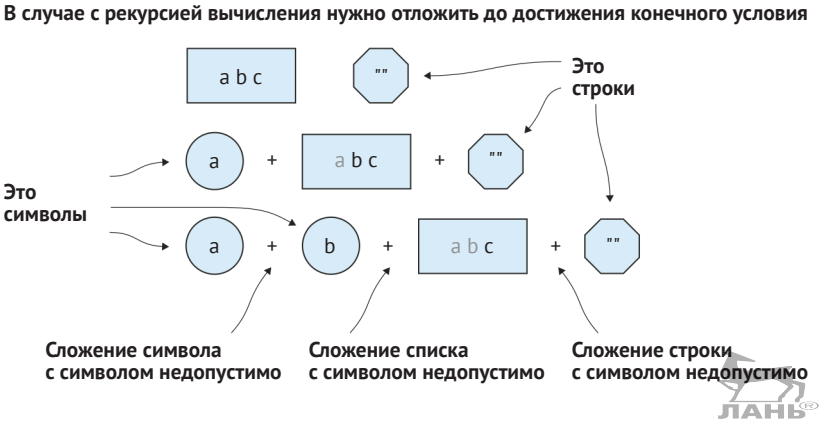


Рис. 4.2 Рекурсивное преобразование списка символов в строку

Как видите, главная проблема этого решения – необходимость хранения промежуточных шагов. Проблема объясняется ограниченностью объема памяти, которую JVM выделяет для этих целей.

4.1.3 Различия между рекурсивными и сорекурсивными функциями

Основываясь на знаниях, полученных в школе, можно подумать, что примеры рекурсии и сорекурсии – это, по сути, *рекурсивные функции*, т. е. функции, вызывающие сами себя. Если придерживаться такого определения, тогда вы правы. Но вообще это не так.

Функция является рекурсивной, если вызывает саму себя как часть вычислений. Иначе это не настоящая рекурсия. По крайней мере, это не

рекурсивный процесс, хотя он может быть сорекурсивным. Представьте метод, выводящий строку «Hello, World!» на экран и после этого вызывающий сам себя:

```
fun hello() {
    println("Hello, World!")
    hello()
}
```

Разве это не похоже больше на бесконечный цикл, чем на рекурсивный метод? Да, он реализован как рекурсивный метод, но в действительности является сорекурсивным и его легко превратить в бесконечный цикл! Самое интересное, что Kotlin способен осуществлять такое превращение автоматически.

4.1.4 Выбор между рекурсией и сорекурсией

Проблема с рекурсией заключается в том, что все языки накладывают ограничение на количество рекурсивных шагов. Теоретически основное различие между рекурсией и сорекурсией заключается в следующем:

- в сорекурсии все вычисления выполняются немедленно;
- в рекурсии все шаги должны где-то сохраняться в некоторой форме. Это позволяет отложить вычисления до наступления условия завершения, после которого в обратном порядке выполняются все отложенные прежде шаги.

Память для хранения отложенных действий часто ограничена и может переполняться. Чтобы избежать этой проблемы, лучше не использовать рекурсию и отдавать предпочтение сорекурсии.

Рисунки 4.3 и 4.4 иллюстрируют различия между рекурсией и сорекурсией. На них показано вычисление суммы для списка целых чисел. Сложение – это особый случай, потому что:

- эта операция обладает свойством коммутативности (в том смысле что a + b = b + a), несвойственным многим другим операциям, таким как конкатенация строк и символов;
- оба операнда и результат имеют один и тот же тип, что также свойственно далеко не всем операциям.

Можно подумать, что операции на рис. 4.3 и 4.4 можно преобразовать, например удалив круглые скобки. Допустим, что это невозможно. (Это всего лишь пример, чтобы показать, что получится.)

На рис. 4.3 показан сорекурсивный процесс. Каждая операция выполняется сразу, как только встречается. Как результат, необходимый объем памяти для всего процесса остается постоянным, и его изображает прямоугольник в нижней части рисунка. На рис. 4.4 показано, как ту же задачу можно решить с использованием рекурсивного подхода.

Промежуточный результат не вычисляется, пока не будет выполнен обход всех шагов. Как следствие, для рекурсивных вычислений требуется намного больше памяти; промежуточные шаги должны где-то храниться перед их обработкой в обратном порядке.

Сорекурсивное вычисление суммы первых 10 целых чисел

```
sum(1 to 10)
= 1 + sum(2 to 10)
= (1 + 2) + sum(3 to 10)
= 3 + sum(3 to 10)
= (3 + 3) + sum(4 to 10)
= 6 + sum(4 to 10)
= (6 + 4) + sum(5 to 10)
= 10 + sum(5 to 10)
= (10 + 5) + sum(6 to 10)
= 15 + sum(6 to 10)
= (15 + 6) + sum(7 to 10)
= 21 + sum(7 to 10)
= (21 + 7) + sum(8 to 10)
= 28 + sum(8 to 10)
= (28 + 8) + sum(9 to 10)
= 36 + sum(9 to 10)
= (36 + 9) + sum(10 to 10)
= 45 + sum(10 to 10)
= 45 + 10
= 55
```

Необходимый объем памяти

Рис. 4.3 Сорекурсивные вычисления

Рекурсивное вычисление суммы первых 10 целых чисел

```
sum(1 to 10)
= 10 + (9 + sum(1 to 8))
= 10 + (9 + (8 + sum(1 to 7)))
= 10 + (9 + (8 + (7 + sum(1 to 6))))
= 10 + (9 + (8 + (7 + (6 + sum(1 to 5)))))
= 10 + (9 + (8 + (7 + (6 + (5 + sum(1 to 4))))))
= 10 + (9 + (8 + (7 + (6 + (5 + (4 + sum(1 to 3)))))))
= 10 + (9 + (8 + (7 + (6 + (5 + (4 + (3 + sum(1 to 2))))))))
= 10 + (9 + (8 + (7 + (6 + (5 + (4 + (3 + (2 + sum(1 to 1)))))))))
= 10 + (9 + (8 + (7 + (6 + (5 + (4 + (3 + (2 + 1))))))))
= 10 + (9 + (8 + (7 + (6 + (5 + (4 + (3 + 3)))))))
= 10 + (9 + (8 + (7 + (6 + (5 + (4 + 6))))))
= 10 + (9 + (8 + (7 + (6 + (5 + 10)))))
= 10 + (9 + (8 + (7 + (6 + 15))))
= 10 + (9 + (8 + (7 + 21)))
= 10 + (9 + (8 + 28))
= 10 + (9 + 36)
= 10 + 45
= 55
```

Необходимый объем памяти

Рис. 4.4 Рекурсивные вычисления

Увеличение необходимого объема памяти – не самая худшая проблема рекурсии. Проблема усугубляется тем, что языки программирования

используют стек для сохранения этапов вычислений. Это разумно, потому что вычисления должны выполняться в обратном порядке. Но к сожалению, размер стека ограничен, поэтому, если шагов слишком много, стек переполнится, что приведет к сбою вычислений.

Количество шагов, которое можно без опаски поместить в стек, зависит от языка и иногда может настраиваться. В Kotlin это число примерно равно 20 000; в Java можно сохранить примерно 3000 шагов. Увеличение размера стека до более высоких значений может оказаться не лучшей идеей, потому все потоки выполнения используют один и тот же размер стека (но не одну и ту же область памяти). И часто большой размер стека приводит к напрасному расходованию памяти, учитывая, что нерекурсивные процессы занимают мало пространства на стеке.

Как видно на этих двух рисунках, сорекурсивный процесс использует постоянный объем памяти. Он не растет с увеличением количества шагов. С другой стороны, рекурсивный процесс требует все больше памяти с увеличением числа шагов (и рост может быть более крутым, чем линейный рост, как вы скоро увидите). Вот почему следует избегать рекурсивных процессов, за исключением случаев, когда число шагов будет оставаться достаточно небольшим во всех случаях. Как следствие, вы должны научиться заменять рекурсию сорекурсией всегда, когда это возможно.

4.2 Удаление хвостового вызова

На данный момент у кого-то из вас могут возникнуть сомнения. Я сказал, что сорекурсивный процесс использует постоянный объем стека. Но, как известно, когда функция вызывает саму себя, она все равно расходует пространство стека, даже если ей не нужно ничего помещать на стек. По логике вещей сорекурсия тоже должна исчерпывать стек, хотя и медленнее. Однако эту проблему можно полностью устранить. Хитрость заключается в том, чтобы превратить сорекурсивную функцию в старый добрый цикл. Это проще простого – достаточно заменить базовую реализацию

```
fun toString(list: List<Char>): String {
    fun toString(list: List<Char>, s: String): String =
        if (list.isEmpty())
            s
        else
            toString(list.subList(1, list.size), append(s, list[0]))

    return toString(list, "")
}
```

обычным императивным циклом с изменяемыми ссылками:

```
fun toStringCorec2(list: List<Char>): String {
    var s = ""
    for (c in list) s = append(s, c)
    return s
}
```

Но вам и этого не требуется делать, потому что Kotlin автоматически преобразует сорекурсивные функции в циклы!

4.2.1 Использование механизма удаления хвостового вызова

В отличие от Java Kotlin реализует механизм удаления хвостовых вызовов (Tail Call Elimination, TCE). Это означает, что, когда вызов функции самой себя является последней инструкцией в ее теле (т. е. результат вызова не используется в дальнейших вычислениях), Kotlin удаляет этот хвостовой вызов. Но при этом вы должны явно дать свое разрешение, добавив ключевое слово `tailrec` перед объявлением функции:


```
fun toString(list: List<Char>): String {
    tailrec fun toString(list: List<Char>, s: String): String =
        if (list.isEmpty())
            s
        else
            toString(list.subList(1, list.size), append(s, list[0]))

    return toString(list, "")
}
```


Kotlin обнаружит хвостовую рекурсию в этой функции и применит механизм TCE. Но напомню, что вы должны явно заявить об этом своем намерении. Поначалу это может показаться утомительным и, возможно, вы бы предпочли, чтобы Kotlin делал это автоматически. Но, как вы скоро увидите, ошибка считать рекурсию хвостовой там, где ее нет, встречается не так уж редко. Если вы укажете, что функция имеет хвостовую рекурсию, Kotlin сможет проверить ее и сообщить, если обнаружится ошибка. Иначе Kotlin будет использовать обычную, нехвостовую рекурсию, которая может привести к ошибке `StackOverflowException` во время выполнения. Что еще хуже, такая ошибка может возникать не каждый раз, потому что зависит от входных данных.

4.2.2 Преобразование циклов в хвостовую рекурсию

Использование сорекурсии вместо циклов – это смена парадигмы. Сначала вы будете продолжать мыслить в терминах первоначальной парадигмы и преобразовывать ее в новую. И только со временем вы научитесь мыслить в терминах новой парадигмы. Это верно для обучения в любой области, и обучение использованию сорекурсии – не исключение.

В предыдущих разделах я представил оригинальные понятия рекурсии и сорекурсии. А теперь я расскажу, в чем польза от преобразования императивных циклов в рекурсивные функции. (Но имейте в виду, что это лишь промежуточный шаг. Вскоре вы узнаете, как абстрагировать рекурсию и сорекурсию, чтобы вообще не касаться их.)

Мы уже видели, что сорекурсия намного эффективнее рекурсии из-за ограничений памяти, однако рекурсия имеет одно большое преимущество: она проще в реализации. Рекурсивную версию операции суммиро-

вания целых чисел от 1 до 10 можно записать в одну строку, как показано ниже, где `sum(n)` обозначает сумму целых чисел от 1 до *n*:

```
fun sum(n: Int): Int = if (n < 1) 0 else n + sum(n - 1)
```

Трудно придумать что-то более простое. Но, как видите, сложение можно выполнить только после получения результата `sum(n - 1)`. Состояние вычислений на этом этапе должно храниться в стеке, пока не будут выполнены все шаги. Чтобы получить выгоду от использования механизма TCE, эту реализацию нужно преобразовать в сорекурсивную версию. Многие программисты, которые учатся использовать сорекурсию, испытывают затруднения с этим преобразованием, хотя в нем нет ничего сложного.

Давайте сделаем перерыв и посмотрим, как традиционные программисты решают эту задачу. Они рисуют блок-схему алгоритма, используя изменяемое состояние и проверку условий. Задача настолько простая, что многие просто представляют блок-схему только в своих головах, но мы для большей ясности нарисуем ее (рис. 4.5).

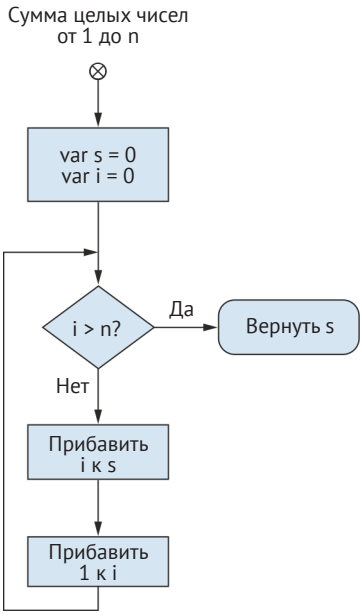


Рис. 4.5 Блок-схема алгоритма императивного вычисления суммы n первых целых чисел

Соответствующая императивная реализация использует то, что императивные программисты называют *управляющими структурами*, например циклы `for` и `while`. Поскольку в Kotlin нет цикла `for` (по крайней мере, традиционного цикла `for`, соответствующего этой блок-схеме), я использую цикл `while`:

```
fun sum(n: Int): Int {
    var sum = 0
    var idx = 0
```

```
    while(idx <= n) {
        sum += idx
        idx += 1
    }
    return sum
}
```

Ничего особенно сложного, но здесь много мест, где можно ошибиться, транслируя блок-схему в реализацию цикла `while`. Например, легко перепутать условие: что должно быть, `<=` или `<`? Когда должна увеличиваться переменная `i`, до или после увеличения переменной `s`? Очевидно, что этот стиль программирования ориентирован на умных программистов, которые не допускают подобных ошибок. Но для остальных, кому требуется писать много тестов для проверки потенциальных ошибок, возможно ли написать сорекурсивную реализацию, делающую то же самое? Конечно, возможно.

Для этого можно заменить переменные параметрами функции. Вместо функции `sum(n)` нужно написать вспомогательную функцию:

```
fun sum(n: Int, sum: Int, idx: Int): Int = ...
```

которая будет вызываться основной функцией `sum(n: Int)` с начальными значениями:

```
fun sum(n: Int, sum: Int, idx: Int): Int =
    if (idx < 1) sum else sum(n, sum + idx, idx - 1)

fun sum(n: Int) = sum(n, 0, n)
```

В действительности `n` никогда не будет изменяться, поэтому можно воспользоваться преимуществами локальных функций в Kotlin и избавиться от одного параметра, заключив вспомогательную функцию в основную и замкнув ее на параметре основной функции. Это проще показать, чем описать:

```
fun sum(n: Int): Int {
    fun sum(sum: Int, idx: Int): Int =
        if (idx < 1) sum else -sum(sum + idx, idx - 1)
    return sum(0, n)
}
```

Как я уже говорил выше, тело функции, которая содержит инструкцию `return` в конце и явно определяет тип возвращаемого значения, требуется заключить в фигурные скобки. Затем нужно добавить реализацию вспомогательной функции.

Поразмышляем о версии с циклом. В каждой итерации мы получаем модифицированные версии переменных: `s + i` и `i + 1`. Соответственно, мы должны организовать вызов вспомогательной функции самой себя с этими модифицированными параметрами:

```
fun sum(n: Int): Int {
    fun sum(s: Int, i: Int): Int = sum(s + i, i + 1)
    return sum(0, 0)
}
```

Но если запустить этот код, он никогда не завершится, потому что здесь отсутствует проверка условия завершения:

```
fun sum(n: Int): Int {
    fun sum(s: Int, i: Int): Int = if (i > n) s else sum(s + i, i + 1)
    return sum(0, 0)
}
```

Теперь осталось только сообщить Kotlin, что к этой вспомогательной функции следует применить механизм TCE, добавив ключевое слово `tailrec`:

```
fun sum(n: Int): Int {
    tailrec fun sum(s: Int, i: Int): Int =
        if (i > n) s else sum(s + i, i + 1)
    return sum(0, 0)
}
```

Когда применение механизма TCE невозможно, Kotlin выведет предупреждение:

```
Warning:(16, 5) Kotlin: A function is marked as tail-recursive
but no tail calls are found

(Перевод:
Warning:(16, 5) Kotlin: Функция отмечена как функция с хвостовой рекурсией,
но хвостовой вызов не найден.)
```

Сама программа скомпилируется, но будет создан такой код, который может переполнить стек при определенных условиях! К таким предупреждениям следует относиться с большим вниманием.

Упражнение 4.1

Реализуйте сорекурсивную функцию `add`, складывающую положительные целые числа. В реализации функции `add` не должны использоваться операторы плюс (+) и минус (-), допускаются только две функции:

```
fun inc(n: Int) = n + 1
fun dec(n: Int) = n - 1
```

Вот сигнатура функции:

```
fun add(a: Int, b: Int): Int
```

Подсказка

Теперь вы знаете достаточно, чтобы сразу написать сорекурсивную реализацию. Если у вас это не получится, напишите реализацию с циклом, а затем преобразуйте ее, как это было сделано с функцией `sum`.

Решение

Сложение двух чисел, *x* и *y*, можно выполнить с использованием следующего алгоритма:

- если *y* = 0, вернуть *x*;
- иначе – прибавить 1 к *x*, вычесть 1 из *y* и начать сначала.

Вот как можно реализовать этот алгоритм, используя цикл:

```
fun add(a: Int, b: Int): Int {
  var x = a
  var y = b
  while(y != 0) {
    x = inc(x)
    y = dec(y)
  }
  return x
}
```

Условие здесь немного изменено, чтобы упростить цикл `while`. Можно было бы использовать условие, как в описании алгоритма, но код получится не таким красивым:

```
fun add(a: Int, b: Int): Int {
  var x = a
  var y = b
  while(true) {
    if (y == 0) return x
    x = inc(x)
    y = dec(y)
  }
}
```

Также обратите внимание, что, в отличие от Java, Kotlin не позволяет использовать параметры `x` и `y` непосредственно, потому что параметры могут быть только ссылками `val`. Поэтому мы были вынуждены создать их копии. Теперь осталось только заменить переменные параметрами в вызове функции `add`:

```
tailrec fun add(x: Int, y: Int): Int = if (y == 0) x else add(inc(x), dec(y))
```

В этом упражнении вам не нужно ничего изменять. Вместо хранения текущих значений в изменяемых ссылках `var` достаточно рекурсивно вызывать функцию с новыми значениями параметров. Теперь вы можете вызвать эту функцию с любыми значениями аргументов, не опасаясь вызвать исключение `StackOverflowException`. Как вы скоро увидите, написание безопасных программ часто требует больших усилий для преобразования простой рекурсии в хвостовую. И иногда это невозможно!

4.2.3 Рекурсивные функции-значения

Как вы только что видели, определить рекурсивную функцию просто. Иногда хвостовая рекурсия дает более простое решение, как было показано в предыдущем примере. Но следует признать, что он оторван от реальности, потому что никому и в голову не придет писать функцию для выполнения сложения. Давайте попробуем придумать что-нибудь более близкое к реальности. Например, напишем функцию вычисления факториала `factorial(int n)`, которая возвращает `1`, если ее аргумент равен `0`, и `n * factorial(n - 1)` – в противном случае:

```
fun factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
```

Очевидно, что рекурсия в этой функции не является хвостовой, т. е., если передать ей аргумент со значением в несколько тысяч, она может вызвать переполнение стека. Такой код нельзя использовать в коммерческом приложении, только если вы не уверены, что глубина рекурсии никогда не превысит некоторого небольшого значения.

Итак, написать рекурсивную функцию `fun` просто. Но можно ли то же самое сказать о функциях-значениях?

Упражнение 4.2 (трудное)

Напишите рекурсивную функцию-значение `factorial`. Напомню, что функция-значение – это функция, объявленная с помощью ключевого слова `val`:

```
val factorial: (Int) -> Int =
```

Так как это учебное упражнение, использование ссылки на функцию будет считаться обманом!

Подсказка

Для решения этого упражнения можете обратиться к разделу «Отложенная инициализация» в главе 2.

Решение

Так как функция должна вызывать саму себя, значит, она должна быть определена до ее вызова, т. е. она должна быть определена до того, как вы вызовете ее! Но давайте отложим эту проблему курицы и яйца. Мы легко можем преобразовать функцию `fun` в функцию-значение с одним аргументом, использовав лямбда-выражение с той же реализацией, что и в функции `fun`:

```
val factorial: (Int) -> Int =
    { n -> if (n <= 1) n else n * factorial(n - 1) }
```

ПРИМЕЧАНИЕ Вы должны явно указать тип функции или тип лямбда-аргумента.

Самое интересное, что этот код не будет компилироваться. Компилятор будет жаловаться, что переменная `factorial` еще не инициализирована. Что это значит? Читая этот код, компилятор находится в процессе определения функции `factorial` и сталкивается с вызовом функции, которая еще не определена. Это та же проблема, что и в

```
val x: Int = x + 1
```

Решить эту проблему можно, сначала определив переменную, а затем изменив ее значение, например в блоке инициализации:

```
lateinit var x: Int
init {
    x = x + 1;
}
```

Этот код скомпилируется, потому что все члены определяются до выполнения блока инициализации. Ключевое слово `lateinit` указывает, что переменная будет инициализирована позже. Обратившись к ней до инициализации, вы получите исключение. Этот подход позволяет использовать тип, не поддерживающий значение `null`. Без `lateinit` пришлось бы использовать тип с поддержкой `null` или инициализировать ссылку фиктивным значением. Этот прием абсолютно бесполезен в предыдущем случае, но вы можете использовать его для определения функции `factorial`:


```
object Factorial {
    lateinit var factorial: (Int) -> Int
    init {
        factorial = { n -> if (n <= 1) n else n * factorial(n - 1) }
    }
}
```

Другое, более красивое решение заключается в использовании отложенной инициализации:


```
object Factorial {
    val factorial: (Int)-> Int by lazy { { n: Int ->
        if (n <= 1) n else n * factorial(n - 1)
    } }
}
```

Двойные фигурные скобки обязательны! Отложенная инициализация реализуется с использованием следующего шаблона:

```
object Factorial {
    val factorial: (Int)-> Int by lazy { ... }
}
```

где многоточие ... нужно заменить лямбда-выражением с реализацией из предыдущего примера, включая фигурные скобки:

```
{ n: Int -> if (n <= 1) n else n * factorial(n - 1) }
```

Только одно отличие – теперь Kotlin не может автоматически определить тип `n`, поэтому его нужно указать явно. Единственная проблема этого решения в том, что поле нельзя объявить как `val`, что несколько раздражает, потому что неизменность – одно из фундаментальных свойств безопасных программ. Ссылка `var` лишает нас гарантий, что значение переменной `factorial` никогда не изменится. Впрочем, эту проблему можно решить, объявив `var`-поле приватным и скопировав его значение в поле `val`:

```
object Factorial {
    private lateinit var fact: (Int) -> Int
    init {
        fact = { n -> if (n <= 1) n else n * fact(n - 1) }
    }

    val factorial = fact
}
```

В этом случае можно быть уверенным, что значение `factorial` не изменится после инициализации. Но имейте в виду, что рекурсивные функции-значения нельзя оптимизировать с помощью механизма TCE, поэтому они могут вызвать переполнение стека. Если вам понадобится функция-значение с хвостовой рекурсией, используйте ссылку на функцию.

Также обратите внимание, что эта функция не будет работать со значениями больше 16, потому что они будут вызывать арифметическое переполнение и давать отрицательный результат. Хуже того – для значений выше 33 будет возвращаться `0`, потому что умножение –2 147 483 648 (результат вызова функции с параметром 33) на 34 даст в результате `0`. Из-за этого все последующие результаты будут равны `0`. (Это объясняется тем, что тип `Int` в Kotlin представляет 32-битные значения.)

4.3 Рекурсивные функции и списки

Рекурсия и хвостовая рекурсия часто используются для обработки списков. В таких случаях список часто разбивается на две части: первый элемент, называемый *головой*, и остальную часть списка, называемую *хвостом*. Вы уже видели пример такого подхода, когда мы определяли функцию преобразования списка символов в строку. Взгляните на следующую функцию, которая вычисляет сумму элементов списка целых чисел:

```
fun sum(list: List<Int>): Int =
    if (list.isEmpty()) 0 else list[0] + sum(list.drop(1))
```

Если список пуст, функция возвращает `0`. Иначе возвращается значение первого элемента (головы списка) плюс результат применения функции `sum` к остальной части списка (хвосту). Возможно, будет понятнее, если определить вспомогательные функции `head` и `tail`, возвращающие голову и хвост списка соответственно. Не будем ограничивать эти функции поддержкой только списков целых чисел, чтобы дать возможность использовать их с любыми списками:

```
fun <T> head(list: List<T>): T =
    if (list.isEmpty())
        throw IllegalArgumentException("head called on empty list")
    else
        list[0]

fun <T> tail(list: List<T>): List<T> =
    if (list.isEmpty())
        throw IllegalArgumentException("tail called on empty list")
    else
        list.drop(1)

fun sum(list: List<Int>): Int =
    if (list.isEmpty())
        0
```

```
    else
        head(list) + sum(tail(list))
```

Или еще лучше – можно определить `head` и `tail` как функции-расширения для класса `List`:

```
fun <T> List<T>.head(): T =
    if (this.isEmpty())
        throw IllegalArgumentException("head called on empty list")
    else
        this[0]

fun <T> List<T>.tail(): List<T> =
    if (this.isEmpty())
        throw IllegalArgumentException("tail called on empty list")
    else
        this.drop(1)

fun sum(list: List<Int>): Int =
    if (list.isEmpty())
        0
    else
        list.head() + sum(list.tail())
```

В этом примере рекурсивный вызов – не последнее, что происходит внутри функции `sum`. Вот четыре последних операции, которые выполняет функция:

- вызывает функцию `head`;
- вызывает функцию `tail`;
- вызывает функцию `sum` и передает ей результат вызова функции `tail`;
- складывает результаты вызовов `head` и `sum`.

Рекурсия в этой функции не является хвостовой, поэтому к ней нельзя применить ключевое слово `tailrec`, и у вас не получится использовать ее для подсчета суммы со списками, насчитывающими больше нескольких тысяч элементов. Но эту функцию можно переписать так, чтобы рекурсивный вызов `sum` выполнялся последним:

```
fun sum(list: List<Int>): Int {
    tailrec fun sumTail(list: List<Int>, acc: Int): Int =
        if (list.isEmpty())
            acc
        else
            sumTail(list.tail(), acc + list.head())

    return sumTail(list, 0)
}
```

Вспомогательная функция `sumTail` реализует хвостовую рекурсию и потому может быть подвергнута оптимизации с использованием механизма TCE. А так как она нигде больше не используется, ее лучше поместить внутрь функции `sum`.

Конечно, вспомогательную функцию можно определить на одном уровне с основной. Но тогда желательно объявить вспомогательную функцию приватной (`private`) или внутренней (`internal`), а основную функцию – общедоступной (`public`). В данном же случае вызов вспомогательной функции из основной образует замыкание. Главное преимущество локальной вспомогательной функции перед приватной – отсутствие вероятности конфликта имен и возможность замыкания на некоторых параметрах объемлющей функции.

В языках программирования, где поддерживаются локальные функции, широко распространена практика присваивания всем вспомогательным функциям одного и того же имени, такого как `go` или `process`. Это невозможно в случае использования нелокальных функций; возникнет конфликт имен, если разные функции будут иметь одинаковые сигнатуры. В предыдущем примере вспомогательная функция для `sum` была названа `sumTail`.

Другая распространенная практика – давать вспомогательной функции такое же имя, как основной, но с символом подчеркивания в конце, например `sum_`. Также вспомогательной функции можно присвоить то же имя, что и основной, потому что они будут иметь разные сигнатуры. Неважно, какую систему вы выберете, важно, чтобы вы неукоснительно придерживались ее. В оставшейся части этой книги я буду использовать символ подчеркивания для обозначения вспомогательных функций, реализующих хвостовую рекурсию.

4.3.1 Дважды рекурсивные функции

Ни одна книга, рассказывающая о рекурсивных функциях, не может обойти стороной пример вычисления членов последовательности Фибоначчи. Для большинства из нас это абсолютно бесполезная функция, но она является одним из простейших примеров *дважды рекурсивной функции*, т. е. функции, которая на каждом шаге вызывает себя дважды. Для начала рассмотрим требования на тот случай, если вы никогда не сталкивались с этой функцией.

Последовательность Фибоначчи – это набор чисел, в котором каждое следующее число является суммой двух предыдущих. Это рекурсивное определение. Нам нужно условие завершения рекурсии, поэтому полные требования выглядят так:

- f (0) = 1;
- f (1) = 1;
- f (n) = f (n – 1) + f (n – 2).

Это оригинальная последовательность Фибоначчи, в которой первые два числа равны 1. Предполагается, что каждое число является функцией его положения в последовательности. Большинство программистов обычно предпочитают начинать с `0`, а не с `1`. Вы можете найти определение, где f (0) = 0, что не является частью оригинальной последовательности Фибоначчи. Но сути это не меняет.

Что такого интересного в этой функции? Отложим ответ на этот вопрос и попробуем написать простейшую реализацию «в лоб»:

```
fun fibonacci(number: Int): Int =
    if (number == 0 || number == 1)
        1
    else
        fibonacci(number - 1) + fibonacci(number - 2)
```

Теперь напишем простенькую программу для проверки этой функции:

```
fun main(args: Array<String>) {
    (0 until 10).forEach { print("${fibonacci(it)} ") }
}
```

Если запустить эту программу, она выведет первые десять чисел из последовательности Фибоначчи:

```
1 1 2 3 5 8 13 21 34 55
```

Опираясь на уже имеющиеся знания о рекурсии в Kotlin, можно подумать, что эта функция благополучно вычислит `f(n)` со значением `n` вплоть до нескольких тысяч, прежде чем стек переполнится. Давайте проверим. Замените `10` на `1000` и посмотрите, что получится. Запустите программу и сделайте перерыв на кофе. Когда вы вернетесь, то поймете, что программа все еще работает. Она достигнет значения где-то около 1 836 311 903 – 47-го члена последовательности (ваше достижение может отличаться, более того, вы можете даже получить отрицательное число!), – но никогда не завершится. Переполнения стека не происходит, не возникает никаких исключений – программа просто крутится и крутится, перелопачивая числа. Что же в действительности происходит?

Проблема в том, что каждый вызов функции выполняет два рекурсивных вызова. Чтобы вычислить f (n), нужно выполнить $2n$ рекурсивных вызовов. Допустим, для выполнения функции требуется 10 наносекунд. (Это число взято наугад, но вы скоро увидите, что это ничего не меняет.) Вычисление f (5000) займет $2^{5000} \times 10$ наносекунд. Представляете ли вы, как долго продлятся вычисления? Программа никогда не завершится, потому что ей потребуется больше времени, чем ожидаемая продолжительность жизни Солнечной системы (или даже Вселенной).

Чтобы создать функцию вычисления чисел Фибоначчи, имеющую практическую ценность, мы должны изменить ее так, чтобы она использовала один хвостовой рекурсивный вызов. Есть и еще одна проблема: результаты получаются настолько большими, что вскоре вы столкнетесь с арифметическим переполнением, которое сначала приведет к отрицательным значениям, а затем – к `0`.

Упражнение 4.3

Создайте версию функции `fibonacci`, реализующую хвостовую рекурсию.

Подсказка

Поразмышляв над реализацией на основе цикла, как при создании функции `sum`, вы поймете, что вам нужны две переменные для хранения двух предыдущих значений. Эти переменные затем нужно преобразовать в параметры вспомогательной функции. Параметры должны иметь тип `BigInteger`, чтобы можно было вычислять большие значения.

Решение

Для начала определим сигнатуру вспомогательной функции. Она принимает два значения типа `BigInteger` и один первоначальный аргумент и возвращает результат типа `BigInteger`:

```
tailrec fun fib(val1: BigInteger, val2: BigInteger, x: BigInteger): BigInteger
```

Теперь определим условия завершения. Если аргумент равен `0`, нужно вернуть `1`:

```
tailrec
fun fib(val1: BigInteger, val2: BigInteger, x: BigInteger): BigInteger
=
    when {
        (x == BigInteger.ZERO) -> BigInteger.ONE
        ...
    }
```

Если аргумент равен `1`, нужно вернуть сумму параметров `val1` и `val2`:

```
tailrec
fun fib(val1: BigInteger, val2: BigInteger, x: BigInteger): BigInteger
=
    when {
        (x == BigInteger.ZERO) -> BigInteger.ONE
        (x == BigInteger.ONE) -> val1 + val2
        ...
    }
```

В заключение добавим рекурсию, а для этого:

- превратим `val2` в `val1`;
- создадим новый параметр `val2`, сложив два предыдущих значения;
- вычтем 1 из аргумента;
- рекурсивно вызовем функцию и передадим ей три вычисленных значения.

Вот окончательный код:

```
tailrec
fun fib(val1: BigInteger, val2: BigInteger, x: BigInteger): BigInteger
=
    when {
        (x == BigInteger.ZERO) -> BigInteger.ONE
        (x == BigInteger.ONE) -> val1 + val2
        else -> fib(val2, val1 + val2, x - BigInteger.ONE)
    }
```

Два параметра, `val1` и `val2`, накапливают результаты `fib(n - 1)` и `fib(n - 2)`. По этой причине их часто называют *аккумуляторами*. Если хотите, можете переименовать их в `acc1` и `acc2`.

Последнее, что осталось сделать, – написать основную функцию, которая вызывает эту вспомогательную функцию с начальными параметрами, и поместить определение вспомогательной функции в тело основной:

```
fun fib(x: Int): BigInteger {
    tailrec
    fun fib(val1: BigInteger, val2: BigInteger, x: BigInteger): BigInteger
    =
        when {
            (x == BigInteger.ZERO) -> BigInteger.ONE
            (x == BigInteger.ONE) -> val1 + val2
            else -> fib(val2, val1 + val2, x - BigInteger.ONE)
        }

    return fib(BigInteger.ZERO, BigInteger.ONE,
               BigInteger.valueOf(x.toLong()))
}
```

Это лишь одна из возможных реализаций. Параметры, начальные значения и условия можно организовать немного иначе. Теперь можете попробовать вызвать `fib(10_000)` и получить результат через пару наносекунд. Ладно, через пару десятков миллисекунд, но только потому, что вывод в консоль – это довольно медленная операция. Я еще вернусь к этой проблеме ниже. В любом случае результат впечатляет – и сам результат вычислений (2090 цифр), и скорость, с какой он получен, достигнутая благодаря преобразованию двойного рекурсивного вызова в один сорекурсивный.

4.3.2 Абстракция рекурсии в списках

Одно из основных применений рекурсии – объединение первого элемента (головы) списка с результатом применения того же процесса к остальной части списка (хвосту). Вы уже видели это, когда вычисляли сумму целочисленных элементов списка:

```
fun sum(list: List<Int>): Int =
    if (list.isEmpty())
        0
    else
        list.head() + sum(list.tail())
```

Тот же принцип можно распространить на любую операцию и любой тип, не только на сложение целых чисел. Раньше мы рассмотрели пример преобразования списка символов в строку. Тот же прием можно использовать для копирования элементов списка любого типа в строку через разделитель:

```
fun <T> makeString(list: List<T>, delim: String): String =
    when {
```

```
        list.isEmpty() -> ""
        list.tail().isEmpty() ->
            "${list.head()}${makeString(list.tail(), delim)}"
        else -> "${list.head()}$delim${makeString(list.tail(), delim)}"
    }
```

Упражнение 4.4

Напишите версию функции `makeString`, использующую хвостовую рекурсию. (Попробуйте не заглядывать в реализацию функции `sum`.)

Решение

Применим тот же метод, что использовался в предыдущем примере: создадим вспомогательную функцию с дополнительным параметром, накапливающим результат. Если поместите эту вспомогательную функцию в основную, можно организовать замыкание на параметре `delim`, который не меняется в ходе рекурсии:

```
fun <T> makeString(list: List<T>, delim: String): String {
    tailrec fun makeString_(list: List<T>, acc: String): String = when
    {
        list.isEmpty() -> acc
        acc.isEmpty() -> makeString_(list.tail(), "${list.head()}")
        else -> makeString_(list.tail(), "$acc$delim${list.head()}")
    }
    return makeString_(list, "")
}
```

Ладно, это было просто, но снова и снова писать реализацию этого решения для каждой рекурсивной функции будет довольно утомительно. Можно ли абстрагировать ее? Как оказывается, можно. Сначала нужно отступить на шаг назад и окинуть взглядом всю картину целиком. Итак, что у нас есть?

1. Функция для обработки списка элементов некоторого типа, которая возвращает единственное значение другого типа. Эти типы можно обозначить параметрами типов `T` и `U`.
2. Операция с элементом типа `T` и элементом типа `U`, возвращающая результат типа `U`. Обратите внимание, что операция является функцией `(U, T) -> U`.

Кажется, что это описание отличается от того, что мы имели в примере с функцией `sum`, но в действительности здесь все то же самое, хотя `T` и `U` – это один и тот же тип (`Int`). В примере со строками тип `T` – это `Char`, а `U` – `String`. В определении функции `makeString` тип `T` уже был обобщенным, а `U` – это `String`.

Упражнение 4.5

Напишите обобщенную версию функции с хвостовой рекурсией, которую можно использовать в `sum`, `string` и `makeString`. Дайте этой функции имя `foldLeft`, затем перепишите `sum`, `string` и `makeString`, задействовав эту новую функцию.

Подсказка

Добавьте дополнительные параметры: начальное значение типа `U` (для аккумулятора) и функцию `(U, T) -> U`.

Решение

Вот сама функция `foldLeft` и три предыдущих функции, использующих эту абстракцию:

```
fun <T, U> foldLeft(list: List<T>, z: U, f: (U, T) -> U): U {
    tailrec fun foldLeft(list: List<T>, acc: U): U =
        if (list.isEmpty())
            acc
        else
            foldLeft(list.tail(), f(acc, list.head()))
    return foldLeft(list, z)
}

fun sum(list: List<Int>) = foldLeft(list, 0, Int::plus)

fun string(list: List<Char>) = foldLeft(list, "", String::plus)

fun <T> makeString(list: List<T>, delim: String) =
    foldLeft(list, "") { s, t -> if (s.isEmpty()) "$t"
                                 else "$s$delim$t" }
```

Только что созданная функция играет очень важную роль в программировании без циклов. Она позволяет абстрагировать сорекурсию безопасным способом и избавляет от необходимости тратить время и силы на реализацию своих функций, использующих хвостовую рекурсию.

Но иногда приходится идти обратным путем и использовать рекурсию вместо сорекурсии.

Представьте, что у нас есть список символов `[a, b, c]` и из него требуется получить строку `"abc"`, используя только функции `head`, `tail` и `prepend`, написанные нами в предыдущем разделе. Мы не можем обращаться к элементам списка по их индексам. Но можем написать следующую рекурсивную реализацию:

```
fun string(list: List<Char>): String =
    if (list.isEmpty())
        ""
    else
        prepend(list.head(), string(list.tail()))
```

Мы многое можем абстрагировать в этом коде, как уже абстрагировали функцию `foldLeft`. Например, тип `Char` можно абстрагировать в тип `T`, чтобы функция могла работать со списками элементов любого типа. Возвращаемый тип `String` можно абстрагировать в тип `U`, чтобы получить результат любого типа. При этом придется абстрагировать функцию `prepend`, превратив ее в обобщенную функцию типа `(T, U) -> U`. Также потребуется заменить начальное значение `""` на значение `identity` типа `U` (единичное значение), соответствующее этой функции.

Упражнение 4.6

Напишите эту абстрактную функцию и назовите ее `foldRight`. Затем перепишите функцию `string`, используя `foldRight`.

Решение

Определим сигнатуру функции, принимающей дополнительные параметры: единичное значение и функцию, используемую для свертки:

```
fun <T, U> foldRight(list: List<T>, identity: U, f: (T, U) -> U): U =
```

Если список пуст, вернем единичное значение `identity`:

```
fun <T, U> foldRight(list: List<T>, identity: U, f: (T, U) -> U): U =
    if (list.isEmpty())
        identity
```

Если список не пуст, используем код из функции `string`, заменив `prepend` функций из параметра:

```
fun <T, U> foldRight(list: List<T>, identity: U, f: (T, U) -> U): U =
    if (list.isEmpty())
        identity
    else
        f(list.head(), foldRight(list.tail(), identity, f))
```

Вот и все! Теперь можно переписать функцию `string`, используя `foldRight`:

```
fun string(list: List<Char>): String =
    foldRight(list, "", { c, s -> prepend(c, s) })
```

Обычно в коде на Kotlin последний аргумент записывается за круглыми скобками, если это функция. В этом случае запятая перед функцией не ставится:

```
fun string(list: List<Char>): String =
    foldRight(list, "") { c, s -> prepend(c, s) }
```

ПРИМЕЧАНИЕ Функция `foldRight` реализует простую, не хвостовую рекурсию, поэтому ее нельзя оптимизировать с помощью механизма TCE. Дело в том, что функцию `foldRight` невозможно реализовать с использованием хвостовой рекурсии. Единственное, что можно сделать, – определить функцию, возвращающую тот же результат, что и `foldRight`, но использующую `foldLeft` и переворачивающую список перед ее вызовом.

В программах, использующих класс `List`, нет необходимости писать свои функции `foldRight` и `foldLeft`, потому что они уже имеются в языке Kotlin (правда, функция `foldLeft` называется просто `fold`).

4.3.3 Обращение списка

Иногда операция обращения (переворачивания) списка действительно бывает полезна, даже притом что в целом она не является оптималь-

ной в смысле производительности. Предпочтительнее, конечно, найти другое решение, не требующее обращения списка, но иногда это просто невозможно. Одно из таких решений – использовать другую структуру данных, обеспечивающую доступ с обоих концов.

Реализовать функцию `reverse` с использованием цикла не составит труда: достаточно выполнить обход элементов списка в обратном порядке. Но будьте внимательны, чтобы не перепутать индексы:

```
fun <T> reverse(list: List<T>): List<T> {
    val result: MutableList<T> = mutableListOf()
    (list.size downTo 1).forEach {
        result.add(list[it - 1])
    }
    return result
}
```

Но в языке Kotlin эта реализация не потребуется, потому что в нем уже имеется функция `reversed`.

Упражнение 4.7

Определите функцию `reverse`, используя `fold`.

Подсказка

Не забывайте, что `foldRight` может вызвать переполнение стека при обработке длинных списков, поэтому, если это возможно, старайтесь использовать `foldLeft`. Также создайте функцию `prepend`, добавляющую элемент в начало списка. Не волнуйтесь о производительности. Этой проблемой мы займемся в главе 5. Реализуйте функцию так, чтобы она работала с неизменяемыми списками и использовала оператор `+`.

Решение

Функция `prepend` реализуется просто, хотя здесь есть небольшая хитрость. Оператор `+` в Kotlin объединяет списки или добавляет элемент в конец списка, но он не может добавить элемент в начало списка. Чтобы решить эту проблему, можно предварительно создать список с единственным элементом, который требуется добавить в начало:

```
fun <T> prepend(list: List<T>, elem: T): List<T> = listOf(elem) + list
```

Теперь остается только перевернуть список с помощью следующей функции:

```
fun <T> reverse(list: List<T>): List<T> =
    foldLeft(list, listOf(), ::prepend)
```

чтобы можно было выполнить свертку слева. Этот код работает, но он далеко не самый оптимальный. Можно ли определить функцию `reverse` без создания списка с одним элементом?

Упражнение 4.8

Определите функцию `reverse`, использующую только версию оператора `+`, добавляющую элемент в конец списка.

Подсказка

Чтобы решить это упражнение, нужно определить функцию `prepend`, не использующую операцию конкатенации списков. Попробуйте для начала написать функцию, копирующую список через свертку слева.

Решение

Копирование списка через свертку слева реализуется просто:

```
fun <T> copy(list: List<T>): List<T> =
    foldLeft(list, listOf()) { lst, elem -> lst + elem }
```

Функцию `prepend`, которая добавляет элемент в начало списка, тоже можно реализовать через свертку слева, использовав аккумулятор, содержащий добавляемый элемент вместо пустого списка:

```
fun <T> prepend(list: List<T>, elem: T): List<T> =
    foldLeft(list, listOf(elem)) { lst, elm -> lst + elm }
```

Теперь можно использовать прежнюю реализацию `reverse` с новой версией `prepend`:

```
fun <T> reverse(list: List<T>): List<T> =
    foldLeft(list, listOf(), ::prepend)
```

Не используйте эти версии `reverse` и `prepend` в промышленном коде. Обе предполагают полный обход списка несколько раз, поэтому они действуют довольно медленно. Для работы со списками в Kotlin используйте стандартную функцию `reversed` из класса `List`. В главе 5 вы узнаете, как создавать функциональные неизменяемые списки, обладающие прекрасной производительностью во всех случаях.

4.3.4 Сорекурсивные списки

Программистам часто приходится снова и снова делать одно и то же. И одним из примеров могут служить сорекурсивные списки, обычно списки целых чисел. Рассмотрим следующий пример на Java:

```
for (int i = 0; i < limit; i++) {
    // некоторые операции...
}
```

Этот код включает две абстракции: сорекурсивный список и некоторые операции. Сорекурсивный список – это список целых чисел от `0` (включительно) до `limit` (не включая его). Как я уже говорил, один из способов сделать программу безопаснее – довести абстракцию до предела, чтобы код можно было максимально использовать повторно. Давайте абстрагируем конструирование сорекурсивного списка.

Сорекурсивные списки создаются легко и просто: к каждому его элементу, начиная с первого (`int i = 0`), применяется выбранная функция (`i -> i++`).

Можно сначала создать список, а потом отобразить его в функцию, композицию функций или иной эффект, соответствующие коммента-

рию `некоторые операции`.... Сделаем это, использовав конкретное граничное значение. Вот пример на Java:

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

Этой реализации соответствует следующий код на Kotlin:

```
listOf(0, 1, 2, 3, 4).forEach(::println)
```

Здесь и список, и эффект абстрагированы. Но абстракцию можно расширить еще дальше.

Упражнение 4.9

Используя цикл, напишите функцию, которая конструирует список на основе начального значения, границы и функции `x -> x + 1` и имеет следующую сигнатуру:

```
fun range(start: Int, end: Int): List<Int>
```

Назовите эту функцию `range`.

Решение

В реализации функции `range` можно использовать цикл `while`:

```
fun range(start: Int, end: Int): List<Int> {
    val result: MutableList<Int> = mutableListOf()
    var index = start
    while (index < end) {
        result.add(index)
        index++
    }
    return result
}
```

Упражнение 4.10

Напишите обобщенную версию функции `range`, которая могла бы работать с разными типами и условиями. Поскольку понятие диапазона (range) подходит только для чисел, назовите эту версию `unfold`. Вот как выглядит сигнатура этой функции:

```
fun <T> unfold(seed: T, f: (T) -> T, p: (T) -> Boolean): List<T>
```

Решение

Взяв за основу реализацию функции `range`, вы лишь заменяете конкретные части абстрактными:

```
fun <T> unfold(seed: T, f: (T) -> T, p: (T) -> Boolean): List<T> {
    val result: MutableList<T> = mutableListOf()
    var elem = seed
    while (p(elem)) {
        result.add(elem)
```

```
        elem = f(elem)
    }
    return result
}
```

Упражнение 4.11

Реализуйте функцию `range`, используя `unfold`.

Решение

Здесь нет ничего сложного. Нужно лишь передать:

- параметр `start` функции `range` в вызов функции `unfold` в аргументе `seed`;
- функцию `f`: `{ x -> x + 1 }` или эквивалентную ей `{ it + 1 }`;
- предикат `p`: `{ x -> x < end }` или эквивалентный ему `{ it < end }`.

```
fun range(start: Int, end: Int): List<Int> =
    unfold(start, { it + 1 }, { it < end })
```

Сорекурсия и рекурсия состоят в дуальных отношениях. Они являются аналогами друг друга, поэтому рекурсивный процесс всегда можно превратить в сорекурсивный, и наоборот.

А теперь давайте реализуем обратный процесс.

Упражнение 4.12

Напишите рекурсивную версию `range`, использовав функции из предыдущих разделов.

Подсказка

Вам понадобится только функция `prepend`, но можете использовать другие функции, если пожелаете.

Решение

Рекурсивная версия имеет очень простую реализацию. Достаточно добавить параметр `start` в начало результата, вычисляемого этой же функцией с использованием того же параметра `end` и с применением функции `f` вместо самого параметра `start`. Это проще показать, чем объяснить:

```
fun range(start: Int, end: Int): List<Int> =
    if (end <= start)
        listOf()
    else
        prepend(range(start + 1, end), start)
```

Упражнение 4.13

Напишите рекурсивную версию `unfold`.

Подсказка

И снова начните с рекурсивной версии функции `range` и попробуйте обобщить ее.

Решение

Решение выглядит просто:

```
fun <T> unfold(seed: T, f: (T) -> T, p: (T) -> Boolean): List<T> =
    if (p(seed))
        prepend(unfold(f(seed), f, p), seed)
    else
        listOf()
```

Теперь можно переписать `range`, используя эту функцию. Но имейте в виду, что рекурсивная версия `unfold` вызовет переполнение стека через несколько тысяч рекурсивных итераций.

Упражнение 4.14

Можно ли переписать эту функцию и реализовать в ней хвостовую рекурсию? Попробуйте ответить на этот вопрос, основываясь на теоретических выкладках, а потом практически.

Подсказка

Подумайте над следующим: какой характер имеет функция `unfold` – рекурсивный или сорекурсивный?

Решение

Фактически `unfold` является сорекурсивной, подобно функции `foldLeft`, написанной нами выше. Соответственно, ее можно реализовать с использованием хвостовой рекурсии, если использовать вспомогательную функцию, принимающую дополнительный параметр аккумулятора:

```
fun <T> unfold(seed: T, f: (T) -> T, p: (T) -> Boolean): List<T> {
    tailrec fun unfold_(acc: List<T>, seed: T,
                        f: (T) -> T, p: (T) -> Boolean): List<T> =
        if (p(seed))
            unfold_(acc + seed, f(seed), f, p)
        else
            acc

    return unfold_(listOf(), seed, f, p)
}
```

Использование локальной функции позволяет упростить код и убрать неизменяющиеся параметры из вспомогательной функции (`f` и `p`), организовав замыкание с объемлющей функцией на ее параметрах:

```
fun <T> unfold(seed: T, f: (T) -> T, p: (T) -> Boolean): List<T> {
    tailrec fun unfold_(acc: List<T>, seed: T): List<T> =
        if (p(seed))
            unfold_(acc + seed, f(seed))
        else
            acc

    return unfold_(listOf(), seed)
}
```

4.3.5 Опасность строгости

Ни одна из этих версий (рекурсивная и сорекурсивная) не эквивалентна циклу `for`. Причина в том, что даже в строгих языках, таких как Java и Kotlin (строгих в отношении аргументов методов или функций), цикл `for`, подобно большинству управляющих структур, действует в отложенной манере. То есть в цикле `for` вычисления производятся в порядке: индекс, операции в теле цикла, индекс, операции в теле цикла...; тогда как при использовании функции `range` сначала вычисляется полный список индексов, а затем к нему применяется функция.

Эта проблема обусловлена использованием списков. Список – строгая структура данных. Но мы должны были начать с чего-то, поэтом выбрали списки. В главе 9 вы узнаете, как решить эту проблему с помощью ленивых коллекций.

4.4 Мемоизация

В разделе 4.3.1 мы реализовали функцию для отображения последовательности чисел Фибоначчи. Одна из проблем этой реализации состоит в следующем: чтобы напечатать последовательность чисел вплоть до `f(n)`, нужно вычислить `f(1)`, `f(2)` ... вплоть до `f(n)`, а чтобы найти каждое следующее число `f(i)`, нужно рекурсивно вычислить все предыдущие значения. В общем случае, чтобы получить последовательность до `n`, потребуется n раз вычислить `f(1)`, $n - 1$ раз вычислить `f(2)` и т. д. Общее количество вычислений будет равно сумме целых чисел от `1` до n.

В этом разделе вы узнаете, как проблема избыточных вычислений решается с помощью мемоизации. *Мемоизация*[1] – это прием, основанный на сохранении результатов вычислений в памяти, благодаря чему в будущем его можно вернуть немедленно, если придется повторить те же вычисления. Может быть есть решение лучше? Как вариант, можно попробовать написать специальную функцию `scan`. Мы сделаем это в главе 8. А пока рассмотрим другое решение: сохранение вычисленных значений в памяти, чтобы не приходилось вычислять их снова, если они понадобятся еще.

4.4.1 Мемоизация в программировании на основе циклов

В программировании на основе циклов эта проблема не возникает. Вот один из очевидных способов вывода последовательности чисел Фибоначчи:

```
fun main(args: Array<String>) {
    println(fibo(10))
}

fun fibo(limit: Int): String =
    when {
```

[1] От англ. *memoize* – запомнить. – *Прим. перев.*

```
        limit < 1 -> throw IllegalArgumentException()
        limit == 1 -> "1"
        else -> {
            var fibo1 = BigInteger.ONE
            var fibo2 = BigInteger.ONE
            var fibonacci: BigInteger
            val builder = StringBuilder("1, 1")
            for (i in 2 until limit) {
                fibonacci = fibo1.add(fibo2)
                builder.append(", ").append(fibonacci) ①
                fibo1 = fibo2 ②
                fibo2 = fibonacci ③
            }
            builder.toString()
        }
    }
```

① **Накопление текущего результата в аккумуляторе (`StringBuffer`)**
② **Сохранение `f(n – 1)` для следующей итерации**
③ **Сохранение `f(n)` для следующей итерации**

Несмотря на то что эта программа полна проблем, избавлять от которых призвано функциональное программирование, она работает и делает это намного эффективнее, чем функциональная версия. Причина – в мемоизации (запоминании) промежуточных результатов.

Как я уже сказал, мемоизация – это прием, основанный на запоминании результатов вычислений в памяти, которые потом можно вернуть немедленно, при попытке повторно выполнить те же вычисления. Применительно к функциям эта методика заключается в запоминании результатов предыдущих вызовов, чтобы они могли вернуть результат намного быстрее при повторном вызове с теми же аргументами.

Это может показаться несовместимым с принципами, которые я перечислил выше, потому что для запоминания результатов необходимо изменяемое состояние, что является побочным эффектом. Но в действительности здесь нет никаких противоречий, потому что результат функции остается неизменным для одних и тех же аргументов. (Можно даже сказать, что он не просто неизменный, а тот же самый, потому что не вычисляется снова!) Побочный эффект сохранения результатов не наблюдается снаружи функции. В традиционном программировании поддержание состояния является универсальным способом вычисления результатов, поэтому мемоизация даже не замечается.

4.4.2 Мемоизация рекурсивных функций

Рекурсивные функции часто неявно используют прием мемоизации. В примере с рекурсивной функцией, вычисляющей числа Фибоначчи, мы хотели вернуть последовательность, поэтому вычисляли каждое число, из-за чего производились избыточные вычисления. Самое простое решение – переписать функцию так, чтобы она сразу возвращала строку с последовательностью.

Упражнение 4.15

Напишите функцию, использующую хвостовую рекурсию, которая принимает целое число n и возвращает строку с последовательностью чисел Фибоначчи от 0 до n, перечисленных через запятую и пробел.

Подсказка

Одно из решений – использовать в роли аккумулятора экземпляр StringBuilder. StringBuilder – это изменяемая структура, но ее изменяемость не видна вовне. Другое решение – вернуть список чисел и затем преобразовать его в строку String. Это решение проще, потому что позволяет отделить задачу от визуального представления: сначала вернуть список, а затем написать функцию для преобразования в строку нужного формата.

Решение

В листинге 4.1 представлено решение с использованием списка List в роли аккумулятора.

Листинг 4.1 Рекурсивное вычисление чисел Фибоначчи с неявной мемоизацией

```
fun fibo(number: Int): String {
    tailrec fun fibo(acc: List<BigInteger>,
                     acc1: BigInteger,
                     acc2: BigInteger, x: BigInteger): List<BigInteger>
    =
        when (x) {
            BigInteger.ZERO -> acc
            BigInteger.ONE -> acc + (acc1 + acc2)
            else -> fibo(acc + (acc1 + acc2), acc2, acc1 + acc2, ①
                         x - BigInteger.ONE)
        }

    val list = fibo(listOf(), ②
                    BigInteger.ONE, BigInteger.ZERO,
                    BigInteger.valueOf(number.toLong()))
    return makeString(list, ", ") ③
}

fun <T> makeString(list: List<T>, separator: String): String =
    when {
        list.isEmpty() -> ""
        list.tail().isEmpty() -> list.head().toString()
        else -> list.head().toString() + ④
                foldLeft(list.tail(), "") { x, y -> x + separator + y
    }
}
```

① Первый знак + в этой строке – оператор конкатенации списков; остальные осуществляют сложение больших целых BigInteger

② Вызов вспомогательной функции fibo для получения списка с числами Фибоначчи

③ Преобразование списка в строку вызовом `makeString`. Эта функция написана только как упражнение. Вместо нее можно использовать стандартную функцию `list.joinToString(", ")`
④ Знак + должен находиться в конце этой строки, а не в начале следующей, иначе код не скомпилируется

4.4.3 Неявная мемоизация

Этот пример демонстрирует использование неявной мемоизации. Не нужно считать этот способ лучшим решением задачи. Многие задачи можно решить проще, если взглянуть на них с другой стороны. Давайте попробуем изменить точку зрения на эту задачу.

Вместо набора чисел последовательность Фибоначчи можно рассматривать как набор пар (кортежей из двух элементов). То есть вместо последовательности

```
1, 1, 2, 3, 5, 8, 13, 21, ...
```

можно попробовать сгенерировать такую последовательность:

```
(1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...
```

Здесь каждый следующий кортеж конструируется на основе предыдущего. Второй элемент n-го становится первым элементом кортежа $n + 1$. Второй элемент кортежа с порядковым номером $n + 1$ равен сумме двух элементов n-го кортежа. Вот как можно написать соответствующую функцию в Kotlin:

```
val f = { x: Pair<BigInteger, BigInteger> ->
    Pair(x.second, x.first + x.second) }
}
```

Или, если использовать деструктурирующее объявление:

```
val f = { (a, b): Pair<BigInteger, BigInteger> -> Pair(b, a + b)}
```

Для замены рекурсивной функции ее сорекурсивной версией, понадобятся две вспомогательных функции: `map` и `iterate`.

Упражнение 4.16

Определите функцию `iterate`, которая действовала бы как `unfold`, но вместо рекурсивного вызова самой себя до выполнения некоторого условия она должна вызывать себя определенное число раз.

Подсказка

Для начала скопируйте функцию `unfold` и замените последний параметр и условие.

Решение

Вместо предиката в третьем аргументе функция должна принимать целое число:

```
fun <T> iterate(seed: T, f: (T) -> T, n: Int): List<T> {
```

Функция использует другую вспомогательную функцию, реализующую хвостовую рекурсию, идентичную той, что используется функцией `unfold`, кроме условия:

```
fun <T> iterate(seed: T, f: (T) -> T, n: Int): List<T> {
    tailrec fun iterate_(acc: List<T>, seed: T): List<T> =
        if (acc.size < n)
            iterate_(acc + seed, f(seed))
        else
            acc

    return iterate_(listOf(), seed)
}
```

Упражнение 4.17

Определите функцию `map`, которая применяет функцию `(T) -> U` к каждому элементу в `List<T>` и производит `List<U>`.

Подсказка

Вы можете реализовать хвостовую рекурсию в самой функции или использовать `foldLeft` или `foldRight`. Попробуйте начать с функции `copy`, которую мы написали, когда определяли функцию `reverse`.

Решение

Вот вариант рекурсивного решения:

```
fun <T, U> map(list: List<T>, f: (T) -> U): List<U> {
    tailrec fun map_(acc: List<U>, list: List<T>): List<U> =
        if (list.isEmpty())
            acc
        else
            map_(acc + f(list.head()), list.tail())

    return map_(listOf(), list)
}
```

Проще и безопасней было бы использовать `foldLeft`, уже отлаженную абстракцию рекурсии. Напомню, как выглядит функция `copy`:

```
fun <T> copy(list: List<T>): List<T> =
    foldLeft(list, listOf()) { lst, elem -> lst + elem}
```

Нам нужно только применить функцию-аргумент к каждому элементу в процессе копирования:

```
fun <T, U> map(list: List<T>, f: (T) -> U): List<U> =
    foldLeft(list, listOf()) { acc, elem -> acc + f(elem)}
```

Упражнение 4.18

Определите сорекурсивную версию функции вычисления чисел Фибоначчи, которая производит строковое представление последовательности с первыми *n* числами Фибоначчи.

Решение

Мы должны организовать итерации с использованием пары двух первых чисел и функции, вычисляющей следующую пару на основе предыдущей. В результате получится список пар. Затем этот список можно отобразить с помощью функции, возвращающей первый элемент из каждой пары, и преобразовать получившийся список в строку:

```
fun fiboCorecursive(number: Int): String {
    val seed = Pair(BigInteger.ZERO, BigInteger.ONE)
    val f = { x: Pair<BigInteger, BigInteger> -> Pair(x.second, x.first
                    + x.second) }
    val listOfPairs = iterate(seed, f, number + 1)
    val list = map(listOfPairs) { p -> p.first }
    return makeString(list, ", ")
}
```

4.4.4 Автоматическая мемоизация

Мемоизацию можно использовать для ускорения вычислений не только в рекурсивных, но и в любых других функциях. Давайте рассмотрим, как выполняется умножение. Чтобы перемножить, например, два числа – 234 и 686, – мы берем ручку с листком бумаги или калькулятор. Но, если нас попросят умножить 9 на 7, мы можем дать ответ немедленно, не прибегая к вычислениям, потому что мы помним таблицу умножения.

Мемоизованная функция действует аналогично, но ей нужно выполнить вычисления один раз, чтобы запомнить результат. Представьте функцию, умножающую заданное число на 2:

```
fun double(x: Int) = x * 2
```

Результаты вычисления этой функции можно сохранять в ассоциативном массиве. Вот как это можно сделать, используя традиционные приемы программирования: проверку условия и управление потоком выполнения:

```
val cache = mutableMapOf<Int, Int>() ①

fun double(x: Int) =
    if (cache.containsKey(x)) { ②
        cache[x] ③
    } else {
        val result = x * 2 ④
        cache.put(x, result) ⑤
        result ⑥
    }
```

① **Для хранения результатов используется изменяемый ассоциативный массив**
② **Поиск результата в массиве**
③ **Если найден, вернуть результат**
④ **Если не найден, вычислить результат**
⑤ **Добавить результат в массив**
⑥ **Вернуть результат**

Так получилось, что проверка условия и управление потоком выполнения уже абстрагированы в функции `computeIfAbsent`:

```
val cache: MutableMap<Int, Int> = mutableMapOf()

fun double(x: Int) = cache.computeIfAbsent(x) { it * 2 }
```

Или, если вы предпочитаете функцию-значение:

```
val double: (Int) -> Int = { cache.computeIfAbsent(it) { it * 2 } }
```

Но здесь есть две проблемы:

- эти модификации необходимо применить ко всем функциям, которые требуется мемоизовать;
- ассоциативный массив, используемый для хранения результатов, доступен извне.

Вторую проблему легко решить, поместив ассоциативный массив и функцию в отдельный объект и объявив массив приватным членом. Вот пример такой функции `fun`:

```
object Doubler {

    private val cache: MutableMap<Int, Int> = mutableMapOf()

    fun double(x: Int) = cache.computeIfAbsent(x) { it * 2 }
}
```

А вот как можно использовать этот объект:

```
val y = Doubler.double(x);
```

В этом решении ассоциативный массив недоступен извне, и вторую проблему можно считать решенной. Но как можно решить первую проблему?

Начнем с определения требований. Нам нужен некоторый способ, позволяющий следующее:

```
val f: (Int) -> Int = { it * 2 }
val g: (Int) -> Int = Memoizer.memoize(f)
```

В этом случае мы могли бы использовать мемоизованную функцию взамен оригинальной. Все значения, возвращаемые функцией `g`, в первый раз вычисляются оригинальной функцией `f`, а при последующих вычислениях возвращаются из кеша. Кроме того, если создать третью функцию

```
val f: (Int) -> Int = { it * 2 }
val g: (Int) -> Int = Memoizer.memoize(f)
val h: (Int) -> Int = Memoizer.memoize(f)
```

значения, кешированные функцией `g`, не должны быть доступны функции `h`; `g` и `h` должны использовать разные кеши (если только не мемоизовать функцию `memoize`!).

В листинге 4.2 показан класс `Memoizer`, имеющий удивительно простую реализацию.

Листинг 4.2 Класс Memoizer

```
class Memoizer<T, U> private constructor() {

    private val cache = ConcurrentHashMap<T, U>()

    private fun doMemoize(function: (T) -> U): (T) -> U =
        { input ->
            cache.computeIfAbsent(input) { ①
                function(it)
            }
        }

    companion object {

        fun <T, U> memoize(function: (T) -> U): (T) -> U = ②
            Memoizer<T, U>().doMemoize(function)
    }
}
```

① Выполняет вычисления, если необходимо, вызывая оригинальную функцию
② Возвращает мемоизованную версию функции в аргументе

В листинге 4.3 показано, как использовать этот класс. Программа имитирует продолжительные вычисления, чтобы нагляднее продемонстрировать выгоды мемоизации функций.

Листинг 4.3 Демонстрация класса Memoizer

```
fun longComputation(number: Int): Int { ①
    Thread.sleep(1000) ②
    return number
}

fun main(args: Array<String>) {
    val startTime1 = System.currentTimeMillis()
    val result1 = longComputation(43)
    val time1 = System.currentTimeMillis() - startTime1
    val memoizedLongComputation =
        Memoizer.memoize(::longComputation) ③
    val startTime2 = System.currentTimeMillis()
    val result2 = memoizedLongComputation(43)
    val time2 = System.currentTimeMillis() - startTime2
    val startTime3 = System.currentTimeMillis()
    val result3 = memoizedLongComputation(43)
    val time3 = System.currentTimeMillis() - startTime3
    println("Call to nonmemoized function: result = " +
            "$result1, time = $time1")
    println("First call to memoized function: result = " +
            "$result2, time = $time2")
    println("Second call to nonmemoized function: result = " +
            "$result3, time = $time3")
}
```

① Функция для мемоизации
② Имитация продолжительных вычислений
③ Мемоизация функции

Эта программа выводит следующие результаты:

```
Call to non memoized function: result = 43, time = 1000
First call memoized function: result = 43, time = 1001
Second call to nonmemoized function: result = 43, time = 0
```

Теперь вы можете превращать обычные функции в мемоизованные единственным вызовом функции из класса `Memoizer`, но, чтобы использовать этот прием в промышленном коде, вам необходимо позаботиться о потенциальной проблеме нехватки памяти. Этот прием можно использовать, когда число возможных вариантов входных данных невелико и все результаты можно хранить в памяти, не опасаясь ее переполнения. В противном случае для хранения результатов лучше использовать слабые ссылки.

4.4.5 Мемоизация функций нескольких аргументов

Как я уже отмечал, в нашем мире не существует такого понятия, как функция нескольких аргументов. Функции отражают связь между двумя наборами данных (исходным и целевым). Они не могут иметь несколько аргументов. Функции, которые выглядят как функции нескольких аргументов, в действительности являются:

- функциями кортежей;
- функциями, возвращающими функции, возвращающими функции ... возвращающими результат.

В любом случае мы имеем функцию одного аргумента, поэтому с легкостью можем применить к ней объект `Memoizer`.

Функции кортежей могут показаться самым простым вариантом. Мы можем использовать класс `Pair` или `Triple`, реализованные в Kotlin, или определить свой класс, если потребуется объединить больше трех элементов. Но в действительности второй вариант намного проще, правда, при этом необходимо использовать каррированные версии функций, как описывалось в разделе «Каррирование функций» в главе 3.

Мемоизация каррированных функций осуществляется просто, хотя и немного не так, как было показано выше. Вы должны мемоизовать каждую функцию:

```
val mhc = Memoizer.memoize { x: Int ->
                Memoizer.memoize { y: Int ->
                    x + y
                }
            }
```

Тот же прием используется для мемоизации функции трех аргументов:

```
val f3 = { x: Int -> { y: Int -> { z: Int -> x + y - z } } }

val f3m = Memoizer.memoize { x: Int ->
                Memoizer.memoize { y: Int ->
                    Memoizer.memoize { z: Int -> x + y - z }
                }
            }
```

В листинге 4.4 показан пример тестирования мемоизованной функции трех аргументов.

Листинг 4.4 Пример тестирования мемоизованной функции трех аргументов

```
val f3m = Memoizer.memoize { x: Int ->
    Memoizer.memoize { y: Int ->
        Memoizer.memoize { z: Int ->
            longComputation(z) - (longComputation(y) +
                                  longComputation(x))
        }
    }
}

fun main(args: Array<String>) {
    val startTime1 = System.currentTimeMillis()
    val result1 = f3m(41)(42)(43)
    val time1 = System.currentTimeMillis() - startTime1
    val startTime2 = System.currentTimeMillis()
    val result2 = f3m(41)(42)(43)
    val time2 = System.currentTimeMillis() - startTime2
    println("First call to memoized function: result = " +
            "$result1, time = $time1")
    println("Second call to memoized function: result = " +
            "$result2, time = $time2")
}
```

Эта программа выводит следующее:

```
First call to memoized function: result = -40, time = 3003
Second call to memoized function: result = -40, time = 0
```

Вывод наглядно показывает, что первый вызов мемоизованной функции длился 3003 миллисекунды, а второй вернул результат немедленно.

С другой стороны, мемоизация функции кортежей может показаться проще, если определить свой класс `Tuple`, потому что в этом случае есть возможность использовать `data`-класс, для которого Kotlin автоматически создаст функции `equals` и `hashCode`. Ниже показан пример реализации класса `Tuple4` (если вам достаточно кортежей с двумя или тремя элементами, можете использовать стандартные классы `Pair` и `Triple` соответственно):

```
data class Tuple4<T, U, V, W>(val first: T,
                              val second: U,
                              val third: V,
                              val fourth: W)
```

В листинге 4.4 показан пример тестирования мемоизованной функции, принимающей аргумент типа `Tuple4`.

Листинг 4.5 Мемоизованная функция, принимающая аргумент типа Tuple4

```
val ft = { (a, b, c, d): Tuple4<Int, Int, Int, Int> ->
    longComputation(a) + longComputation(b)
        - longComputation(c) * longComputation(d) }

val ftm = Memoizer.memoize(ft)

fun main(args: Array<String>) {
    val startTime1 = System.currentTimeMillis()
    val result1 = ftm(Tuple4(40, 41, 42, 43))
    val time1 = System.currentTimeMillis() - startTime1
    val startTime2 = System.currentTimeMillis()
    val result2 = ftm(Tuple4(40, 41, 42, 43))
    val time2 = System.currentTimeMillis() - startTime2
    println("First call to memoized function: result = " +
            "$result1, time = $time1")
    println("Second call to memoized function: result = " +
            "$result2, time = $time2")
}
```

4.5 Являются ли мемоизованные функции чистыми?

Мемоизация подразумевает сохранение состояния между вызовами функции. Поведение мемоизованной функции зависит от текущего ее состояния, но при этом функция всегда возвращает одно и то же значение для одних и тех же входных данных. Отличается только время, необходимое на возврат результата. Мемоизованные функции остаются чистыми, если оригинальные функции – чистые.

Разница во времени вычислений может превратиться в источник проблем. Например, оригинальной версии функции вычисления чисел Фибоначчи может потребоваться много лет для вычисления результата, что может стать проблемой. Часто ускорить вычисления не составляет большого труда. Но если это не так, значит, где-то есть куда большая проблема!

Итоги

- Рекурсивная функция вызывает саму себя и использует результат рекурсивного вызова в последующих вычислениях.
- Рекурсивная функция помещает текущее состояние вычислений на стек перед вызовом самой себя.
- Размер стека по умолчанию в Kotlin весьма ограничен. Если количество рекурсивных вызовов окажется слишком большим, вы получите исключение `StackOverflowException`.

- Хвостовая рекурсия – это рекурсия, когда рекурсивный вызов является последним (*хвостовым*) действием в теле функции.
- Хвостовую рекурсию в Kotlin можно оптимизировать с помощью механизма удаления хвостовых вызовов (Tail Call Elimination, TCE).
- Лямбда-выражения тоже могут быть рекурсивными.
- Мемоизация позволяет функциям запоминать вычисленные результаты, чтобы потом быстрее возвращать их.
- Мемоизацию можно сделать автоматической.

5 Обработка данных с использованием списков

Эта глава охватывает следующие темы:

- классификация структур данных;
- вездесущие односвязные списки;
- важность неизменяемости;
- обработка с помощью рекурсивных функций.

Структуры данных – одно из важнейших понятий в программировании и в повседневной жизни. Мир, как мы его видим, – это гигантская структура данных, состоящая из более простых структур, которые, в свою очередь, состоят из еще более простых структур. Всякий раз, моделируя какие-либо объекты или факты, мы получаем структуры данных.

Есть множество типов структур данных. В информатике структуры данных, представляющие множество однотипных данных, называют общим термином «коллекции». Коллекция – это группа элементов данных, связанных друг с другом некоторыми отношениями. В простейшем случае отношением является принадлежность к некоторой группе.

В этой главе рассматриваются структуры данных и способы создания собственной реализации односвязного списка. В Kotlin есть свои списки – как изменяемые, так и неизменяемые. Но неизменяемые списки в Kotlin на самом деле не являются таковыми и не поддерживают возможности совместного использования данных, что ухудшает эффективность таких операций, как добавление и удаление элементов. Неизменяемый список, который мы создадим в этой главе, реализует эффективные операции со стеком и является неизменным.

5.1 Классификация коллекций данных

Структура данных – это структурированный фрагмент данных. Коллекции данных – это одна из категорий структур данных. Классифицировать коллекции данных можно по-разному. Класс коллекций можно разделить на линейные, ассоциативные и графовые коллекции:

- *линейные коллекции – это коллекции, в которых элементы связаны одним измерением*. В них каждый элемент связан со следующим. Типичным примером линейной коллекции может служить список;
- *ассоциативные коллекции – это коллекции, которые можно рассматривать как функции*. Для данного объекта `o` функция `f(o)` возвращает `true` или `false` в зависимости от принадлежности этого объекта коллекции. В отличие от линейных коллекций между элементами коллекции нет никакой связи. Эти коллекции не упорядочены, хотя вполне можно определить некоторый искусственный порядок следования элементов. Типичными примерами ассоциативных коллекций являются множество и ассоциативный массив (который также иногда называют отображением (map) или словарем). Функциональная реализация ассоциативных массивов будет представлена в главе 11;
- *графы – это коллекции, в которых каждый элемент связан с несколькими другими элементами*. Наиболее ярким примером может служить дерево, в частности, бинарное, каждый элемент которого связан с двумя другими элементами. О деревьях мы поговорим в главе 10.

5.2 Разные типы списков

Все наше внимание в этой главе мы сосредоточим на наиболее типичном представителе линейных коллекций: списке. Список – это, пожалуй, самая распространенная структура данных из используемых в программировании, поэтому они часто применяются для изучения разнообразных идей, связанных со структурами данных.

> **ВАЖНО** Все, о чем рассказывается в этой главе, не является характерным только для списков и используется при работе со многими другими структурами данных (которые могут вообще не быть коллекциями).

Списки можно разбить на подкатегории по множеству разных аспектов, включая следующие:

- *способ доступа* – одни списки обеспечивают доступ только с одного конца, другие – с обоих концов. Списки могут поддерживать запись с одного конца, а чтение – с другого. Наконец, некоторые списки позволяют обратиться к произвольному элементу по его позиции в списке, которую также называют *индексом*;

- *способ упорядочения* – в некоторых списках чтение элементов производится в порядке их записи. Структуры этого типа называют очередями FIFO (first in, first out – первым пришел, первым ушел). В других элементы извлекаются в порядке, обратном порядку записи (очереди LIFO, last in, first out – последним пришел, первым ушел). А некоторые списки позволяют извлекать элементы в совершенно другом порядке;
- *реализация* – способы доступа и упорядочения тесно связаны с реализацией, которую вы выбираете для своего списка. Если вы решите представить список как цепочку, связав каждый следующий элемент с предыдущим, вы получите совершенно другой результат с точки зрения способа доступа, отличающийся от способа доступа к индексированному массиву. Или если вы решите связать каждый элемент с двумя соседними элементами – следующим и предыдущим, – вы получите список, поддерживающий доступ с обоих концов.

На рис. 5.1 представлены разные типы списков с разными способами доступа. Он иллюстрирует принципы, лежащие в основе каждого типа списка, но не способ, которым они реализованы.

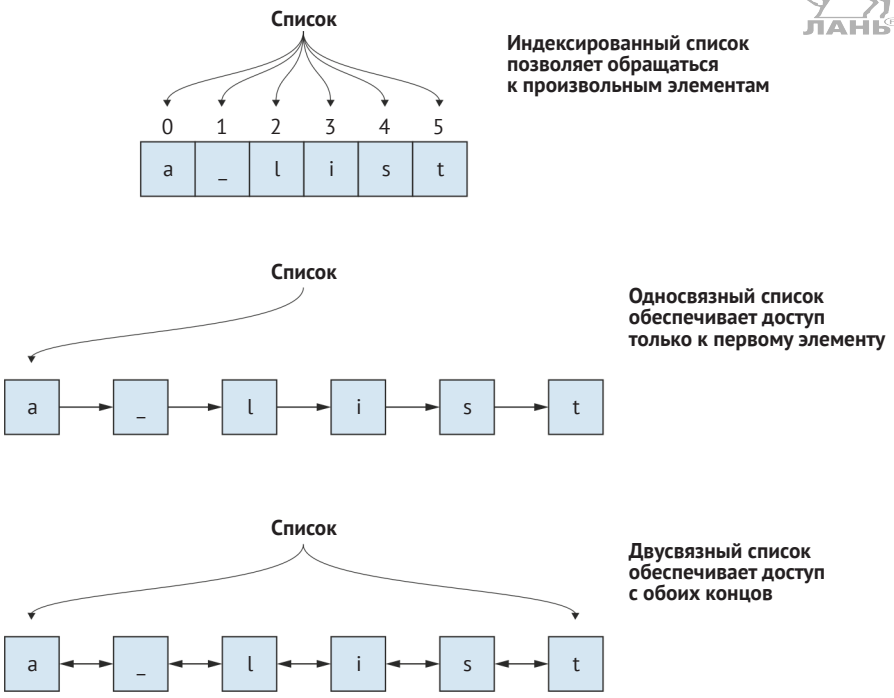


Рис. 5.1 Разные типы списков с разными способами доступа к их элементам

5.3 Производительность списков

Одним из важнейших критериев при выборе типа списка является производительность различных видов операций. Производительность часто выражается в форме нотации *Большого О*. Эта нотация в основном используется в математике, а в информатике она указывает, как изменяется сложность алгоритма при изменении объема входных данных. Применительно к спискам эта нотация показывает, как меняется производительность в зависимости от длины списка. Например:

- O(1) – указывает, что скорость выполнения операции постоянна (ее можно интерпретировать так: время, необходимое для обработки одного элемента, нужно умножить на 1, чтобы получить время обработки n элементов);
- O(log(n)) – время, необходимое для обработки одного элемента, нужно умножить на log(n), чтобы получить время обработки n элементов;
- O(n) – время, необходимое для обработки одного элемента, нужно умножить на n, чтобы получить время обработки n элементов;
- O(n^2) – время, необходимое для обработки одного элемента, нужно умножить на n^2, чтобы получить время обработки n элементов.

Структуру данных с производительностью O(1) для всех типов операций можно считать идеальной. К сожалению, это невозможно. Каждый тип списка предлагает разную производительность для разных операций. Индексированные списки обеспечивают производительность O(1) для операции извлечения данных и близкую к O(1) для вставки. Односвязные списки обеспечивают производительность O(1) для вставки и извлечения на одном конце и O(n) – на другом.

Выбор оптимальной структуры – это всегда компромисс. Чаще всего выбирается структура с производительностью O(1) для наиболее частых операций, и как неизбежное зло принимается производительность O(log(n)) или даже O(n) для других операций, которые выполняются значительно реже.

Имейте в виду, что этот способ оценки производительности имеет большое значение для структур, способных масштабироваться до бесконечности, но не относится к структурам, которыми вы будете манипулировать, потому что они всегда будут ограничены объемом доступной памяти. Структура с временем доступа O(n) может оказаться быстрее, чем структура с производительностью O(1), из-за ограничения размера. Если время обработки одного элемента в первой структуре намного меньше, ограниченность памяти может помешать второй структуре проявить свои преимущества. Часто лучше иметь производительность O(n) с временем доступа к элементу в одну наносекунду, чем O(1) с временем доступа в одну миллисекунду. (Скорость обработки последней превысит скорость обработки первой только на коллекциях, насчитывающих более 1 млн элементов.)

5.3.1 Время в обмен на объем памяти и сложность

Как было показано выше, выбор реализации структуры данных часто заключается в поиске компромисса между временем выполнения разных операций. Обычно выбор падает на реализацию, которая быстрее выполняет одни операции и медленнее – другие в зависимости от того, какие операции используются наиболее часто. Но есть и другие компромиссы в обмен на время.

Представьте, что вам нужна структура, позволяющая извлекать элементы в отсортированном порядке, начиная с наименьшего. Вы можете выбрать реализацию, которая сортирует элементы при вставке или, напротив, хранит элементы в порядке поступления, но производит поиск наименьшего элемента при извлечении. Принимая решение о выборе той или иной реализации, важно учитывать, будет ли извлеченный элемент удаляться из структуры. Если нет, тогда к элементу можно будет обратиться несколько раз, и в этом случае, вероятно, лучше сортировать элементы во время вставки, чтобы избежать многократной сортировки при извлечении одного и того же элемента. Типичным представителем структур этого типа может служить так называемая *приоритетная очередь*, которая обычно используется для организации ожидания поступления некоторого элемента. Вы можете проверять очередь снова и снова, пока не будет получен ожидаемый элемент. В подобных случаях предпочтительнее осуществлять сортировку элементов во время вставки.

Но что, если требуется иметь возможность доступа к элементам с использованием нескольких разных порядков сортировки? Например, в порядке их вставки или в обратном порядке. Этому случаю наиболее полно соответствует двусвязный список, изображенный на рис. 5.1. Вероятно, в такой ситуации элементы должны сортироваться во время поиска.

Возможно, вы предпочтете реализацию с временем доступа O(1) с одного конца и O(n) с другого или придумаете другую структуру, с временем доступа O(log(n)) с обоих концов. Еще одно решение – хранить два списка, один с элементами в порядке вставки, а другой – в обратном порядке. В этом случае увеличится время вставки, но время извлечения с любого конца будет равно O(1). Один из недостатков такого подхода состоит в увеличении расходуемого объема памяти, поэтому, как видите, выбор оптимальной структуры также может быть выбором компромисса между временем и объемом памяти.

Также вы можете придумать структуру, минимизирующую время вставки и извлечения с обоих концов. Такие структуры уже изобретены, и вам остается только реализовать их, но они намного сложнее, поэтому придется искать компромисс между временем и сложностью реализации.

5.3.2 Отказ от изменения на месте

Большинство структур данных изменяется в процессе выполнения программы, потому что в них добавляются новые элементы и удаляются

старые. Для реализации таких операций можно использовать два подхода. Первый – это *изменение на месте*.

Под изменением на месте подразумевается изменение элементов данных путем изменения самой структуры. Когда все программы были однопоточными, этот подход считался хорошей идеей (хотя на самом деле это не так). Но сейчас с развитием многопоточного программирования ситуация изменилась. Это касается не только изменения элементов, но также вставки, удаления, сортировки и всех других операций, которые изменяют структуру. Если программе позволить изменять структуры данных, эти структуры нельзя совместно использовать, не прибегая к сложным механизмам защиты, которые при неправильной реализации могут приводить к взаимоблокировкам, неоправданным блокировкам, простою потоков выполнения, устареванию данных и множеству других проблем.

Изменение на месте

В 1981 году, в своей статье «The transaction concept: virtues and limitations» Джим Грей (Jim Gray) писал[1]:

Обновление на месте: отравленное яблоко?

Когда бухгалтерия велась с использованием глиняных табличек или бумаги и чернил, бухгалтеры разработали четкие правила надежного бухгалтерского учета. Одно из таких правил – ведение бухгалтерии с двумя записями, чтобы обеспечить самоконтроль вычислений, благодаря чему ошибки обнаруживаются очень быстро, практически сразу после их появления, а не намного позже, при проверке результата (когда они вообще могут не обнаружиться). Второе правило – никто и никогда не должен менять бухгалтерские книги; если обнаруживается ошибка, она отмечается, и в книгу добавляется новая запись, исправляющая ее. Бухгалтерские книги представляют полную историю сделок в бизнесе…

Есть ли решение этих проблем? Да, есть: использовать неизменяемые структуры данных. Многие программисты оказываются в шоке, когда впервые узнают об этом решении. Как можно делать что-то полезное со структурами данных, если их нельзя изменять? В конце концов, мы часто начинаем с пустых структур и постепенно добавляем в них данные. Как это делать, если их нельзя изменять?

Ответ прост. Как и в случае учета с двойной записью, для представления нового состояния вместо изменения существующих данных нужно создать новые. Вместо добавления элемента в существующий список нужно создать новый список с дополнительным элементом. Основное преимущество такого подхода: если во время вставки со списком рабо-

[1] *Jim Gray*. The transaction concept: virtues and limitations. Technical Report 81.3 (Tandem Computers, June 1981), http://www.hpl.hp.com/techreports/tandem/TR-81.3.pdf.

тает какой-то другой поток, это изменение не отразится на нем, потому что тот поток не увидит изменения. Обычно эта идея сразу вызывает два возражения:

- если другой поток не увидит изменения, он будет обрабатывать устаревшие данные;
- если создание копии списка с новым элементом требует времени и дополнительной памяти, использование неизменяемых структур может ухудшить производительность программы.

Оба аргумента ошибочны. Поток, обрабатывающий устаревшие данные, фактически обрабатывает данные, имевшиеся на тот момент, когда он начал их читать. Если элемент будет вставлен после завершения операции, проблем с параллельной обработкой не возникнет. Но если он будет вставлен во время операции, что произойдет с изменяемой структурой данных? В отсутствие должной защиты от одновременного доступа данные могут быть повреждены или будет получен ложный результат (или и то, и другое). При наличии механизма защиты вставка будет заблокирована до завершения операции первым потоком. Во втором случае конечный результат получится точно таким же, как при использовании неизменяемой структуры.

Аргумент относительно производительности верный, если используются структуры данных, подразумевающие полное копирование при каждом изменении, как это имеет место для неизменяемых списков в Kotlin. Однако эта проблема легко решается с применением специальных структур, реализующих совместное использование данных, как будет показано далее в этой главе.

5.4 Какие виды списков доступны в Kotlin?

Kotlin предлагает два вида списков: изменяемые и неизменяемые. Оба вида базируются на списках Java, но снабжены огромным количеством дополнительных функций благодаря системе функций-расширений в Kotlin.

Изменяемые списки действуют подобно спискам в Java. Их можно изменять, добавляя, вставляя или удаляя элементы; в этом случае предыдущая версия списка теряется. Неизменяемые списки, напротив, нельзя изменить, по крайней мере, напрямую. Добавление элемента в неизменяемый список создает копию исходного списка, включающую новый элемент. Эти списки работают надежно, но производительность некоторых операций ниже оптимальной, потому что по своей природе эти списки не являются неизменяемыми. Их неизменность реализована с использованием известного приема *защитного копирования*. Хотя термин «защитное копирование» подразумевает создание копии для защиты себя от параллельных изменений, производимых другими потоками, этот прием также может использоваться для защиты других потоков от собственных изменений.

Нужны ли более эффективные неизменяемые списки, чем предлагаемые Kotlin, – спорный вопрос. Ответ на него во многом зависит от конкретной ситуации. Если вам нужна высокопроизводительная неизменяемая структура LIFO, такая как стек, несомненно, лучше иметь что-то более эффективное, чем неизменяемый список в Kotlin. Но в любом случае, даже если вам не нужен высокопроизводительный список LIFO, знакомство с приемами создания подобных структур очень важно для разработки безопасных программ. Каждая функция, которую вы напишете для работы с неизменяемыми постоянными списками, настолько будет обогащать ваши знания, что вы не сможете этого избежать.

5.4.1 Использование постоянных структур данных

Как я уже сказал, создание копии структуры данных перед вставкой элемента требует времени, что влечет снижение производительности. Но этого не происходит, если используется общий доступ к данным, на котором основаны неизменяемые постоянные структуры данных. На рис. 5.2 показано, как можно удалять и добавлять элементы для создания нового неизменяемого односвязного списка с оптимальной производительностью.

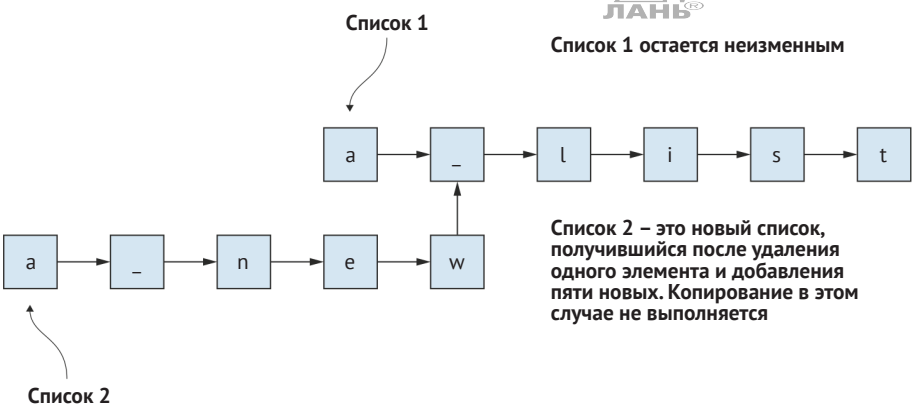


Рис. 5.2 Удаление и добавление элементов без изменения и копирования списка

Как показано на рис. 5.2, копирование списка здесь не выполняется. Такой список выполняет операции удаления и вставки намного эффективнее, чем изменяемый список. Функциональные структуры данных (неизменяемые и постоянные) не всегда медленнее своих изменяемых аналогов. Часто они работают даже быстрее (хотя могут отставать в некоторых операциях). Но в любом случае они намного безопаснее.

5.4.2 Реализация неизменяемого, постоянного, односвязного списка

На рис. 5.1 и 5.2 изображена теоретическая структура односвязного списка. На самом деле такой список нельзя реализовать, потому что элементы не могут быть связаны друг с другом непосредственно. Они должны

быть представлены специальными элементами, поддерживающими такие ссылки, чтобы в них можно было хранить любые элементы. Решение состоит в том, чтобы определить рекурсивную структуру списка, включающую:

- элемент, являющийся первым элементом списка, который также называют *головой*;
- остальную часть списка, которая сама является списком и называется *хвостом*.

Вы уже видели обобщенный элемент, состоящий из двух элементов разных типов: `Pair`. Односвязный список элементов типа `A` на самом деле является структурой `Pair<A, List<A>>`. Класс `Pair` закрыт для расширения, но вы можете определить свой класс:

```
open class Pair<A, B>(val first: A, val second: B)

class List<A>(val head: A, val tail: List<A>): Pair<A, List<A>>(head, tail)
```

Но, как рассказывалось в главе 4, каждая рекурсия должна определять граничный случай. По соглашению этот граничный случай называется `Nil` и соответствует пустому списку. А поскольку у `Nil` нет ни головы, ни хвоста, это не экземпляр `Pair`. Тогда в соответствии с новым определением списком может быть:

- пустой список (`Nil`);
- пара из элемента и списка.

Вместо класса `Pair` со свойствами `first` и `second` создадим свой класс `List` с двумя свойствами: `head` и `tail`. Это упростит обработку граничного случая `Nil`. `Nil` можно объявить как объект-одиночку, т. е. синглтон, чтобы всегда имелся только один экземпляр пустого списка. В данном случае мы создадим его как список `List<Nothing>`, который можно преобразовать в список любого типа. Определить эти элементы можно так:

```
open class List<A>

object Nil : List<Nothing>()

class Cons<A>(private val head: A, private val tail: List<A>): List<A>
()
```

Но здесь есть большой недостаток: любой будет способен расширить класс `List`, вследствие чего могут получиться несогласованные реализации списков, и любой сможет определить подклассы `Nil` и `Cons`, которые являются специфическими деталями реализации и не должны быть доступными для наследования. Решить эту проблему можно, объявив класс `List` запечатанным и определив подклассы `Nil` и `Cons` внутри класса `List`:

```
sealed class List<A> {

    internal object Nil: List<Nothing>()

    internal class Cons<A>(private val head: A,
                           private val tail: List<A>): List<A>()
}
```

На рис. 5.3 представлена первая версия реализации нашего списка. Для опробования нашего класса `List` в деле нам понадобится несколько функций. Эти функции показаны в листинге 5.1.

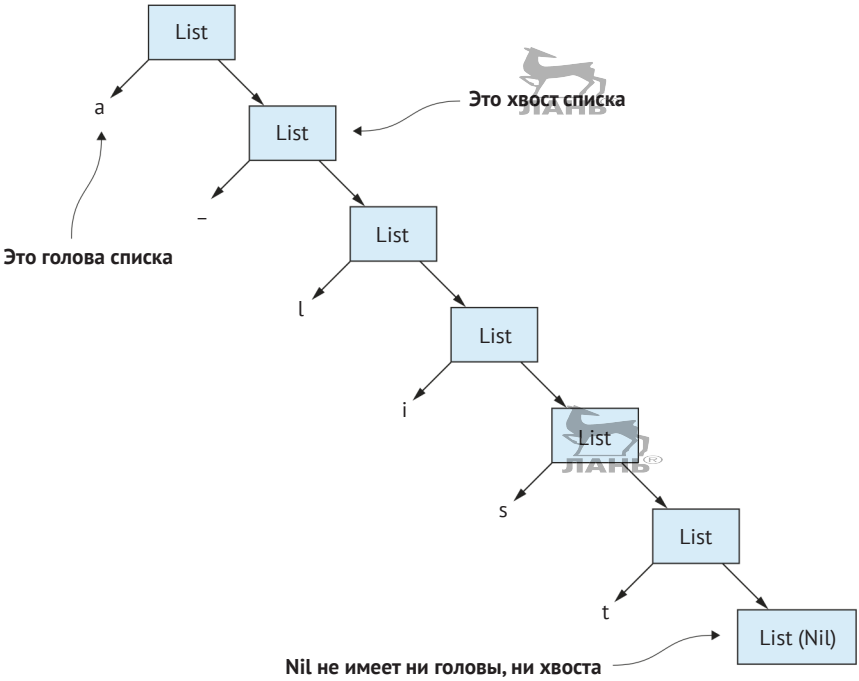


Рис. 5.3 Первая реализация односвязного списка

Листинг 5.1. Односвязный список

```
sealed class List<A> { ①

    abstract fun isEmpty(): Boolean ②

    private object Nil : List<Nothing>() { ③ ④

        override fun isEmpty() = true

        override fun toString(): String = "[NIL]"
    }

    private class Cons<A>(
        internal val head: A,
        internal val tail: List<A>) : List<A>() { ⑤

        override fun isEmpty() = false

        override fun toString(): String = "[${toString("", this)}NIL]"

        tailrec private fun toString(acc: String, list: List<A>):String =
            when (list) { ⑥
                is Nil -> acc
                is Cons -> toString("$acc${list.head}, ", list.tail)
```

```
            }
    }

    companion object {

        operator
        fun <A> invoke(vararg az: A): List<A> = ⑦
                az.foldRight(Nil as List<A>) { a: A, list: List<A> ->
                    Cons(a, list) ⑧
                }
    }
}
```

① **Запечатанные классы являются неявно абстрактными и имеют неявные приватные конструкторы**
② **Абстрактная функция `isEmpty` имеет уникальную реализацию в каждом расширяющем классе**
③ **Расширяющие классы объявляются приватными и определяются внутри класса `List`**
④ **Подкласс `Nil` представляет пустой список**
⑤ **Подкласс `Cons` представляет непустые списки**
⑥ **Функция `toString` реализована как сорекурсивная функция (подробности см. в главе 4)**
⑦ **Функция `invoke` объявлена с ключевым словом `operator` и может вызываться по имени класса, как `ClassName()`**
⑧ **Первый аргумент в вызове `foldRight` – это объект `Nil`, который явно приводится к типу `List<A>`**

Класс `List` (см. листинг 5.1) реализован как *запечатанный (sealed) класс*. Запечатанные классы позволяют определять *алгебраические типы данных* (Algebraic Data Types, ADT), т. е. типы, имеющие ограниченный набор подтипов. Запечатанные классы являются неявно абстрактными и имеют неявный приватный конструктор. Здесь класс `List` параметризуется параметром типа `A`, который представляет тип элементов списка.

Класс `List` содержит два приватных подкласса для представления двух возможных форм, которые может принимать `List`: `Nil` – пустой список и `Cons` – непустой список. (Имя *Cons* означает *construction*, или *конструкция*.) Класс `Cons` принимает параметры типа `A` (голова) и `List<A>` (хвост). По соглашению функция `toString` всегда добавляет `NIL` как последний элемент, поэтому реализация возвращает `[NIL]`. Эти параметры объявлены внутренними (`internal`), поэтому они не будут видны за пределами файла или модуля, в котором объявлен класс `List`. Подклассы объявлены приватными, поэтому списки можно создавать только с помощью вызова функции `invoke` объекта-компаньона.

Кроме того, класс `List` определяет абстрактную функцию `isEmpty()`, которая возвращает `true`, если список пуст, и `false` – в противном случае. Объявление функции `invoke` с модификатором `operator` позволяет вызывать ее с использованием упрощенного синтаксиса:

```
val List<Int> list = List(1, 2, 3)
```

Это не вызов конструктора (конструктор `List` является приватным), а вызов функции `invoke` объекта-компаньона. Функция `foldRight` – это стандартная функция Kotlin для массивов и коллекций. Мы уже определили подобную функцию в главе 4, но я еще вернусь к ней далее в этой главе.

5.5 Совместное использование данных в операциях со списками

Одним из огромных преимуществ неизменяемых постоянных структур данных, таких как односвязный список, является более высокая производительность, достигаемая за счет повторного использования данных. Вы уже видели, что доступ к первому элементу списка, на которое ссылается свойство `head`, происходит немедленно. Удаление первого элемента осуществляется так же быстро – в этом случае нужно лишь вернуть значение свойства `tail`. Теперь посмотрим, как получить новый список с дополнительным элементом.

Упражнение 5.1

Реализуйте функцию `cons`, добавляющую элемент в начало списка. (Напомню, что имя *cons* означает *construction*, или *сконструировать*.)

Подсказка

Эта функция должна иметь одинаковую реализацию для обоих подклассов, поэтому ее можно определить как конкретную функцию в классе `List`.

Решение

Эта функция создает новый список, хвостом которого является текущий список, а головой – новый элемент:

```
fun cons(a: A): List<A> = Cons(a, this)
```

Упражнение 5.2

Реализуйте функцию `setHead`, которая замещает первый элемент списка новым значением.

Подсказка

Пустой список не имеет головы, а значит, ее нельзя изменить, и в этой ситуации следует возбудить исключение. (В следующей главе вы узнаете, как обработать эту ситуацию более безопасным способом.)

Решение

Эту функцию можно реализовать в классе `List`, воспользовавшись преимуществами конструкции `when` и механизма интеллектуального приведения типов:

```
fun setHead(a: A): List<A> = when (this) {
    Nil -> throw IllegalStateException("setHead called on an empty list")
    is Cons -> tail.cons(a)
}
```

Как видите, нет необходимости явно приводить тип `List` к типам `Nil` и `Cons`. Это приведение Kotlin выполнит автоматически. Также обратите внимание, что нет необходимости добавлять ветку `else`, потому что класс

объявлен запечатанным. В этом случае Kotlin видит, что внутри `when` обрабатываются все подклассы.

Тем из вас, кому прежде доводилось программировать на языке Java, может не понравиться этот стиль программирования. Здесь `is` является эквивалентом оператора `instanceof` в Java, использование которого обычно считается плохой практикой. Однако в самом операторе `instanceof` нет ничего неправильного. Его применение считается плохой практикой, потому что это нарушает правила объектно-ориентированного программирования.

ПРИМЕЧАНИЕ Kotlin – не только язык объектно-ориентированного программирования (ООП). В действительности это мультипарадигмальный язык, в котором предпочтение отдается инструментам, лучше всего справляющимся с поставленной задачей, и вы вольны в выборе приемов программирования.

С другой стороны, проверка типов только потому, что это не запрещено, – тоже не лучшее решение. Выбирая способ решения задачи, также следует рассмотреть инструменты ООП, которые могут оказаться лучше. Может, здесь как раз тот случай? На первый взгляд кажется, что нет. Давайте разберемся с этим вопросом, создав абстрактную функцию `setHead` в родительском классе `List` и конкретные реализации в `Nil` и `Cons`:

```
sealed class List<A> {

    abstract fun setHead(a: A): List<A>

    private object Nil: List<Nothing>() {

        override fun setHead(a: Nothing): List<Nothing> =
            throw IllegalStateException("setHead called on an empty list")
            ...
    }

    private class Cons<A>(internal val head: A,
                          internal val tail: List<A>) : List<A>() {
        override fun setHead(a: A): List<A> = tail.cons(a)
    ...
```

Как будто бы все хорошо, но в действительности это не так. Если попытаться вызвать `setHead` для пустого списка, вы получите исключение `ClassCastException` вместо ожидаемого `IllegalStateException`. Причина в том, что, когда функция `setHead` класса `Nil` получает аргумент типа `A`, она преобразует его в тип `Nothing`, потому что он указан в параметре типа реализации `setHead`. Это вызовет исключение `ClassCastException`, так как `Nothing` не является супертипом `A`. (Это тип `A` является супертипом `Nothing`.) Тип `Nothing` можно использовать, чтобы скомпилировать код, но его экземпляр нельзя создать во время выполнения. В результате не получится вызвать ни одну функцию с параметром типа `Nothing`.

Означает ли это, что реализовать `setHead` как абстрактную функцию в классе `List` невозможно? Нет. Проблему легко решить, объявив абстрактный класс `Empty<A>` и объект `Nil<Nothing>`, реализующий этот класс.

Это лучший вариант, если бы нам требовалось определить много функций, каждая из которых принимает аргумент типа `A`. Но нам нужно определить только функции `setHead` и `cons`. Кроме того, функция `cons` может иметь одинаковую реализацию в обоих подклассах, поэтому лучше оставить ее как есть. В главе 11 мы используем абстрактный класс с реализацией объекта-синглтона для представления пустых деревьев.

5.6 Дополнительные операции со списками

Использование общих данных позволяет также определить другие операции, зачастую действующие гораздо эффективнее, чем аналогичные операции с изменяемыми списками. В оставшейся части этого раздела мы расширим возможности списка, повторно используя общие данные.

Упражнение 5.3

Возврат свойства `tail` списка действует так же, как операция удаления первого элемента, но при этом сам список не изменяется. Напишите обобщенную функцию `drop`, которая удаляет первые `n` элементов из списка. Эта функция должна не удалять элементы из имеющегося списка, а возвращать новый список, соответствующий предполагаемому результату. В этом списке не будет ничего нового, потому что он повторно использует уже имеющиеся данные. На рис. 5.4 показано, как должна действовать функция.

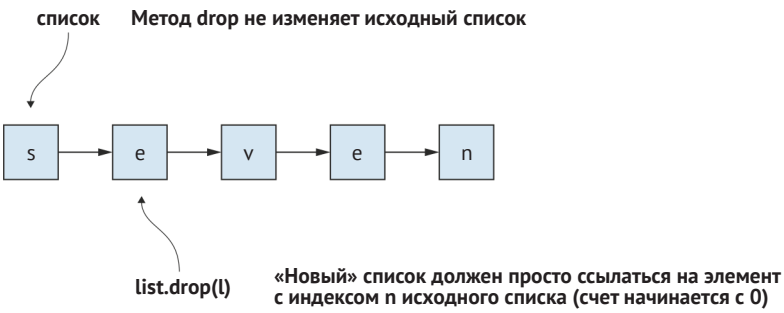


Рис. 5.4 Удаление первых n элементов из списка без изменения или создания чего-либо

Функция имеет следующую сигнатуру:

```
fun drop(n: Int): List<A>
```

Если аргумент `n` больше длины списка, функция `drop` должна вернуть пустой список. (Если хотите, можете дать этой функции имя `dropAtMost`.)

Подсказка

Используйте сорекурсию для реализации функции `drop`. Можете определить абстрактную функцию в классе `List` и две разных реализации в `Nil` и `Cons`. Или, может быть, вам удастся найти более удачное решение?

Решение

Определение абстрактной функции в классе List выглядит просто – только добавьте ключевое слово abstract перед ее сигнатурой. Реализация в классе Nil должна вернуть this. А ниже показана рекурсивная реализация в классе Cons:

```
override fun drop(n: Int): List<A> = if (n == 0) this else tail.drop(n - 1)
```

Но эта реализация может вызвать переполнение стека при больших значениях n (где-то между 10 000 и 20 000), если, конечно, список достаточно длинный. Чтобы избежать этой проблемы, рекурсию следует заменить сорекурсией, как было показано в главе 4, добавив дополнительный параметр. Казалось бы, сделать это просто:

```
override fun drop(n: Int): List<A> {
  tailrec fun drop(n: Int, list: List<A>): List<A> =
      if (n <= 0) list else drop(n - 1, list.tail())
  return drop(n, this)
}
```

Но эта реализация не обрабатывает случай, когда n больше длины списка. Использование сорекурсии не позволяет передать граничный случай в реализацию Nil. Поэтому следует явно проверить, является ли значение экземпляром Nil, не полагаясь на полиморфизм. Но, не имея возможности положиться на полиморфизм, мы можем не объявлять абстрактную функцию в классе List с двумя конкретными реализациями в подклассах, а ограничиться одной конкретной реализацией в List:

```
fun drop(n: Int): List<A> {
    tailrec fun drop(n: Int, list: List<A>): List<A> =
        if (n <= 0) list else when (list) {
            is Cons -> drop(n - 1, list.tail)
            is Nil -> list
        }
    return drop(n, this)
}
```

5.6.1 Преимущества объектной нотации

Определение функций в классах – это лишь одно из возможных решений. Функции можно также определять за пределами классов, на уровне пакета. Как я уже говорил, объявление функции внутри класса просто добавляет в нее неявный параметр this. То есть функцию drop можно определить так:

```
class List<A> {
    ...
}

fun <A> drop(aList: List<A>, n: Int): List<A> {
    tailrec fun drop_(list: List<A>, n: Int): List<A> = when (list) {
        List.Nil -> list
        is List.Cons -> if (n <= 0) list else drop_(list.tail, n - 1)
```

```
    }
    return drop_(aList, n)
}
```


В этой версии вспомогательная функция становится ненужной, потому что ее сигнатура в точности совпадает с сигнатурой основной функции:

```
class List<A> {
    ...
}

tailrec fun drop(list: List<A>, n: Int): List<A> = when (list) {
    List.Nil -> list
    is List.Cons -> if (n <= 0) list else drop(list.tail, n - 1)
}
```

Единственный недостаток такого решения в том, что теперь классы `Nil` и `Cons` должны быть объявлены внутренними (`internal`), а не приватными (`private`), потому что, в отличие от Java, объемлющий класс не имеет доступа к приватным внутренним или вложенным классам. Но эту проблему легко решить, определив функцию в *объекте-компаньоне класса*:


```
companion object {
    tailrec fun <A> drop(list: List<A>, n: Int): List<A> = when (list)
        {
            Nil -> list
            is Cons -> if (n <= 0) list else drop(list.tail, n - 1)
        }

    ...
}
```

В этом случае функцию можно вызвать, добавив перед ее именем имя класса, как это делается в языке Java, когда вызываются статические функции:

```
fun main(args: Array<String>) {
    val list = List(1, 2, 3)
    println(List.drop(list, 2))
}
```

Также функцию можно импортировать. Однако функции экземпляров часто проще в использовании, чем функции, объявленные на уровне пакета или в объектах-компаньонах, потому что позволяют вызывать функции с использованием объектной нотации, которая легче читается. Например, вот как реализуется удаление двух элементов из списка целых чисел с последующей заменой первого элемента результата на 0 при использовании функций уровня пакета:

```
val newList = setHead(drop(list, 2), 0);
```

Чтобы добавить новую функцию в процесс обработки, нужно добавить ее имя слева, а дополнительные аргументы, кроме самого списка, – справа, как показано на рис. 5.5.

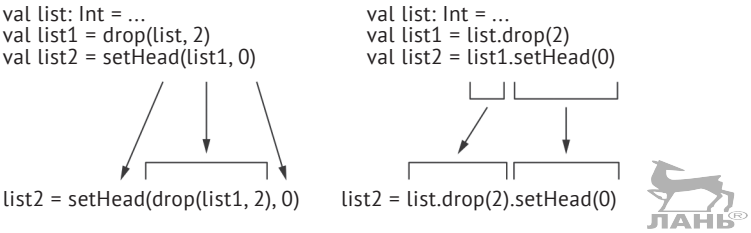


Рис. 5.5 Без использования объектной нотации (слева) цепочки вызовов функций труднее читать. Объектная нотация (справа) помогает писать более понятный код

Использование объектной нотации делает код более понятным:

```
val newList = list.drop(2).setHead(0);
```

Разработчики библиотек в таких случаях должны бы предлагать оба варианта. При наличии функции в объекте-компаньоне добавить версию функции экземпляра не составит никакого труда:

```
fun drop(n: Int): List<A> = drop(this, n)
```

Теперь у нас есть лучшее из обоих миров. Нам не нужна вспомогательная функция (хотя функцию в объекте-компаньоне вполне можно считать вспомогательной) и не нужна конкретная реализация в каждом подклассе. Чтобы функция в объекте-компаньоне была недоступна извне, ее можно объявить приватной. Но в этом случае ее также нужно поместить в класс списка, и в результате мы оказываемся в исходной точке.

На вопрос о выборе того или иного решения нет однозначного ответа. Выбирайте то решение, которое лучше соответствует вашим потребностям или вашему стилю. Решение с меньшим количеством кода обычно предпочтительнее, потому что чем больше кода, тем сложнее его поддерживать. Кроме того, старайтесь минимизировать код, видимый извне.

Упражнение 5.4

Реализуйте функцию `dropWhile` для удаления элементов из головы списка, пока выполняется условие. Вот сигнатура этой функции:

```
fun dropWhile(p: (A) -> Boolean): List<A>
```

Решение

Допустим, мы выбрали подход с использованием объекта-компаньона, вот как можно реализовать вспомогательную функцию:

```
private
tailrec fun <A> dropWhile(list: List<A>,
                          p: (A) -> Boolean): List<A> = when (list) {
    Nil -> list
    is Cons -> if (p(list.head)) dropWhile(list.tail, p) else list
}
```

Ниже приводится реализация функции экземпляра в классе `List`, которая вызывает вспомогательную функцию:

```
fun dropWhile(p: (A) -> Boolean): List<A> = dropWhile(this, p)
```

5.6.2 Объединение списков

Объединение списков, когда один список добавляется в конец другого и в результате получается новый, – это самая обычная операция над списками. Было бы хорошо иметь возможность связать два списка, но это невозможно. Решить задачу можно, добавляя элементы одного списка в другой. Но мы можем добавлять элементы только в начало (в голову) списка, поэтому, чтобы объединить списки `list1` и `list2`, следует добавлять элементы из `list1` в начало `list2`, начиная с конца, как показано на рис. 5.6.

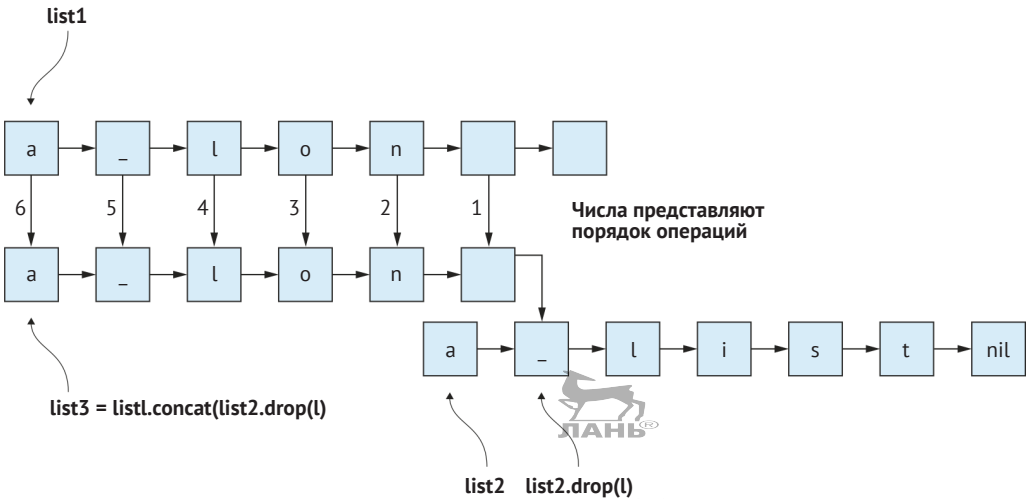


Рис. 5.6 Повторное использование существующих данных при объединении списков. Как видите, оба исходных списка сохраняются, и список list2 повторно используется в новом списке. Но мы не можем действовать, как показано на рисунке, потому что сначала нужно получить доступ к последнему элементу списка list1. Это невозможно из-за особенностей организации списков

Сделать это можно так: сначала перевернуть список `list1`, создав тем самым новый, и затем добавлять элементы по одному в `list2`, начав с головы перевернутого списка. Но у нас нет функции `reverse`. Можно ли обойтись без нее? Да, можно. Давайте посмотрим, как это сделать:

- если список `list1` пуст, вернуть `list2`;
- в противном случае вернуть результат добавления первого элемента списка `list1` (т. е. `list1.head`) к конкатенации остальной части списка `list1` (т. е. `list1.tail`) и `list2`.

Это рекурсивное определение транслируется в следующий код:

```
fun <A> concat(list1: List<A>, list2: List<A>): List<A> = when (list1)
{
    Nil -> list2
    is Cons -> concat(list1.tail, list2).cons(list1.head)
}
```

Чтобы иметь возможность вызывать функцию с использованием объектной нотации, можно добавить функцию экземпляра в класс `List`, которая вызывает функцию из объекта-компаньона и передает ссылку `this` в первом аргументе:

```
fun concat(list: List<A>): List<A> = concat(this, list)
```

Красота этого решения (для одних читателей) в том, что для иллюстрации его работы не требуется рисунок. Это типичное математическое определение, преобразованное в код.

Основной недостаток этого определения (для других читателей) заключается в том, что по той же причине его трудно изобразить на рисунке. Кому-то это может показаться странным, но в действительности здесь нет ничего странного. Это решение не представляет процесс объединения списков (который можно было бы изобразить в виде блок-схемы). Оно выражает результат. И этот код не вычисляет результат. Он и есть результат!

ПРИМЕЧАНИЕ Программирование с использованием функций, заменяющих управляющие структуры, часто требует мыслить в терминах конечного результата, а не способов его получения. Функциональный код – это прямой перевод определения в код.

Очевидно, что этот код может вызвать ошибку переполнения стека, если список `list1` окажется слишком длинным, а вот длина списка `list2` не является предметом для беспокойства. То есть это решение позволяет без опаски добавлять короткие списки в начало списков сколь угодно большой длины.

Важно отметить, что элементы первого списка добавляются в начало второго списка в обратном порядке. Это явно отличается от распространенного представления конкатенации, когда второй список добавляется в хвост первого, что, кстати, плохо подходит для случая односвязных списков.

Если вам потребуется объединить списки произвольной длины, вспомните, о чем рассказывалось в главе 4: чтобы обезопасить функцию `concat` от переполнения стека, рекурсию нужно заменить сорекурсией. К большому сожалению, это невозможно. Как следствие, это решение ограничено размером стека. Далее вы увидите, как решить эту проблему, пожертвовав временем ради экономии памяти (в стеке). А сейчас если вы думаете, что задание выполнено, то это не совсем так, потому что функцию `concat` еще нужно обобщить. Представьте, например, применение этой функции в ходе более обобщенной операции. Можно ли абстрагировать эту функцию, сделать ее безопасной в смысле переполнения стека, а затем повторно использовать для реализации многих других операций? Подождите и увидите!

Возможно, вы заметили, что сложность этой операции (и, следовательно, время для ее выполнения) пропорциональна длине первого списка. При объединении `list1` и `list2` с длинами `n1` и `n2` сложность со-

ставит O(n1), т. е. она не зависит от `n2`. В зависимости от величин `n1` и `n2` эта операция может оказаться намного эффективнее объединения двух изменяемых списков в императивном программировании.

5.6.3 Удаление элементов с конца списка

Иногда требуется удалить элементы, находящиеся в конце списка. Хотя односвязный список не является идеальной структурой данных для такой операции, мы все равно можем ее реализовать.

Упражнение 5.5

Напишите функцию удаления последнего элемента из списка. Функция должна возвращать новый список. Реализуйте ее как функцию экземпляра со следующей сигнатурой:

```
fun init(): List<A>
```

Возможно, вам покажется странным, что я выбрал имя `init` вместо, например, `dropLast`. Я заимствовал это имя из Haskell (http://zvon.org/other/haskell/Outputprelude/init_f.html).

Подсказка

Подумайте, может быть, эту функцию можно выразить в терминах другой функции, о которой я уже говорил. Возможно, сейчас самое время создать эту вспомогательную функцию.

Решение

Чтобы удалить последний элемент, нужно пройти от начала списка до конца и создать новый (в обратном порядке), потому что последним элементом в списке должен быть `Nil`. Это следствие особенностей создания списков с помощью объектов `Cons`. В результате получается обратный список, который нужно перевернуть. То есть нам остается только реализовать функцию `reverse`:

```
tailrec fun <A> reverse(acc: List<A>, list: List<A>): List<A> =
    when (list) {
        Nil -> acc
        is Cons -> reverse(acc.cons(list.head), list.tail)
    }
```

Это реализация функции в объекте-компаньоне. А функция экземпляра в классе `List` приводится ниже:

```
fun reverse(): List<A> = reverse(List.invoke(), this)
```

Обратите внимание, что для вызова функции `invoke` без аргумента нельзя использовать синтаксис `List()`. Ее нужно вызвать явно, иначе Kotlin решит, что вызывается конструктор, и выдаст исключение, потому что класс `List` является абстрактным. Мы также можем выбрать другую организацию, например поместить вспомогательную функцию в вызывающую:

```
fun reverse(): List<A> {
    tailrec fun <A> reverse(acc: List<A>,
                            list: List<A>): List<A> = when (list) {
        Nil -> acc
        is Cons -> reverse(acc.cons(list.head), list.tail)
    }
    return reverse(List.invoke(), this)
}
```

Теперь легко можно реализовать функцию `init`, использующую `reverse`:

```
fun init(): List<A> = reverse().drop(1).reverse()
```

Это реализация для класса `Cons`. В классе `Nil` функция `init` должна возбуждать исключение.

5.6.4 Использование рекурсии для свертки списков с помощью функций высшего порядка

В главе 4 мы познакомились с операцией свертки списков. Эту операцию также можно применять к неизменяемым постоянным спискам. В случае с изменяемыми списками мы могли выбирать между рекурсивной и итеративной реализациями. В случае с неизменяемыми списками нет причин использовать итеративный подход. Давайте рассмотрим, как выполняется свертка, на примере списка целых чисел.

Упражнение 5.6

Напишите функцию, вычисляющую сумму элементов постоянного списка целых чисел, использовав рекурсивный подход. Реализацию можно поместить в объект-компаньон класса `List` или на уровень пакета в том же файле, если вместо приватных используются внутренние подклассы. Объявление на уровне пакета может выглядеть предпочтительнее, потому что оно характерно для `List<Int>`.

Решение

Вот как выглядит определение рекурсивного решения:

- `0`, если список пуст;
- `head + sum`, если хвост `tail` – непустой список.

Вот как это решение выражается в коде на Kotlin:

```
fun sum(ints: List<Int>): Int = when (ints) {
    Nil -> 0
    is Cons -> ints.head + sum(ints.tail)
}
```

Но этот код не компилируется, потому что `Nil` не является подтипом `List<Int>`.

5.6.5 Вариантность

Проблема в том, что, хотя тип Nothing является дочерним для всех типов (включая Int), значение Nothing всегда можно преобразовать в любой другой тип, но нельзя преобразовать List<Nothing> в List <Int>. Дело в том, что, как рассказывалось в главе 2, тип List инвариантен к A. Чтобы добиться желаемого, нужно сделать List ковариантным к A, т. е. объявить его как List <out A>:

```
sealed class List<out A> {
...
```

Но, встретив такое объявление класса List, компилятор сообщит об ошибке (среди множества других ошибок):

```
Error:(7, 17) Kotlin: Type parameter A is declared as 'out'
                            but occurs in 'in' position in type A
(Перевод:
Error:(7, 17) Kotlin: Параметр типа A объявлен как 'out',
                            но находится в позиции 'in' в типе A
)
```

Ошибка в следующей строке:

```
fun cons(a: A): List<A> = Cons(a, this)
```

Она означает, что класс List не может иметь функции с параметром типа A. Параметр подается на вход функции, т. е. находится в позиции in. Функции могут только возвращать тип A (в позиции out).

Подробнее о вариантности

Представьте себе корзину яблок. С этой корзиной можно выполнить только две операции:

- положить яблоко в корзину;
- извлечь яблоко из корзины.

Яблоко (Apple) – это Фрукт (Fruit – родительский класс для класса Apple), а Gala – это подкласс класса Apple. Обратное неверно. Fruit – это не Apple. Даже притом что иногда утверждение «фрукт – это яблоко» бывает верным, в общем случае оно не соответствует действительности. Точно так же Apple – это не Gala.

Вы можете положить Gala *в* корзину с яблоками, но вы не можете положить фрукт, потому что фрукт может не быть яблоком. С другой стороны, если вам нужен фрукт, вы можете взять его из корзины с яблоками. Но если вам нужно яблоко Gala, вы не сможете этого сделать, потому что в корзине могут быть другие сорта яблок.

А если корзина пустая? Не лучше ли было бы сказать, что в ней нет того, что вам нужно? Без такой возможности вам понадобится пустая корзина для яблок, еще одна пустая корзина для апельсинов и еще одна для каждого возможного типа объектов.

Можно использовать пустую корзину типа Any (эквивалент типа Object в Java). И хотя в пустую корзину этого типа можно положить что угодно, из нее ничего нельзя извлечь, потому не известно, какому типу принадлежит извлеченный объект. В Kotlin эта проблема решается с помощью типа Nothing. В отличие от Any, который является родительским типом любого другого типа, Nothing является дочерним типом всех типов.

Объявив параметр A как out, мы говорим, что List<Gala> – это List<Apple>, потому что Gala – это Apple. И наоборот, List<Apple> не является List<Gala>, потому что Apple не является Gala. И это обстоятельство позволяет иметь один пустой список; все, что нужно сделать, это объявить пустой список как List<Nothing>. List<Nothing> – это List<Apple>, потому что Nothing – это Apple. Но List<Nothing> также является List<Tiger>, потому что Nothing также является Tiger. Если вам трудно понять это, представьте, что Nothing – это Absence of (отсутствие). Nothing означает отсутствие Apple и отсутствие Tiger, а также отсутствие чего-либо еще.

Устранение проблемы изменением вариантности

Одна из задач компилятора – предотвращать ошибки программиста. Если объявить параметр типа List как out A, компилятор не позволит использовать тип A в позиции in, а значит, вы не сможете поместить A в List<A>, используя функцию экземпляра List, принимающую параметр типа A. То есть нельзя написать такой код:

```
sealed class List<out A> {
    fun cons(a: A): List<A> = Cons(a, this) { // Ошибка компиляции
        ...
    }

    internal class Cons<out A>(internal val head: A,
                               internal val tail: List<A>): List<A>()

    internal object Nil: List<Nothing>()
}
```

Встретив такой код, компилятор выведет сообщение об ошибке:

```
Type parameter A is declared as 'out' but occurs in 'in' position in type `A`
(Перевод:
Параметр типа A объявлен как 'out', но находится в позиции 'in' в типе `A`)
```

Вы можете подумать, что это несправедливо, потому что точно знаете, что добавить A в List<A> можно. Но здесь вы не правы. Чтобы понять, почему, представьте, как будет выглядеть реализация абстрактной функции cons в каждом подклассе:

```
sealed class List<out A> {

    abstract fun cons(a: A): List<A>

    internal class Cons<out A>(internal val head: A,
                               internal val tail: List<A>): List<A>() {
        ...
    }
```

```
    internal object Nil: List<Nothing>() {

        override fun cons(a: Nothing): List<Nothing> = Cons(a, this) // Ошибка
    }
}
```

При попытке скомпилировать такой код вы получите дополнительную ошибку (кроме упомянутой выше) в реализации `Nil`: компилятор отметит реализацию функции `cons` как *недостижимый код*. Почему? Потому что `this` в классе `Nil` ссылается на `List<Nothing>`, и вызов `Nil.cons(1)` приведет к тому, что число `1` будет преобразовано в `Nothing`. Это невозможно, потому что `Nothing` является подтипом `Int`, а не наоборот. Теперь у нас две проблемы, которые нужно решить:

- компилятор не позволяет использовать `A` в позиции `in`, хотя известно, что в некоторых случаях это вполне допустимо;
- проблема приведения типа в классе `Nil`, которую нужно предотвратить, чтобы исключить ситуации, когда возникает первая проблема.

Чтобы понять, что происходит в классе `Nil`, вспомните, что Kotlin – *строгий язык*, т. е. аргументы функций вычисляются независимо от того, используются они или нет. Проблема не в реализации функции:

```
Cons(a, this)
```

Перед добавлением нового экземпляра `A` в список, ссылку `this`, которая представляет тип `List<Nothing>`, можно безопасно привести к типу `List<A>`. Проблема с аргументом функции:

```
override fun cons(a: Nothing)
```

Когда функция получает аргумент `A`, он сразу же приводится к типу аргумента-получателя `Nothing`, что и вызывает ошибку. Это следствие строгости языка. И это прискорбно, потому что сразу после этого элемент можно было бы добавить в `List<A>` (преобразовав `Nil` в `List<A>`). Приведение аргумента к типу `Nothing` фактически не требуется. Чтобы решить эту проблему, нужно:

- запретить компилятору генерировать ошибку, взяв на себя ответственность за использование `A` в позиции `in`, что можно сделать с помощью аннотации `@UnsafeVariance`:

  ```
  fun cons(a: @UnsafeVariance A): List<A>
  ```

- устранить приведение типа `A` к подтипу, поместив реализацию в родительский класс:

  ```
  sealed class List<out A> {
      fun cons(a: @UnsafeVariance A):List<A> = Cons(a, this)
      ...
  ```

Здесь мы сообщаем компилятору, что он не должен беспокоиться о проблеме вариантности в функции `cons`: всю полноту ответственности мы берем на себя. Теперь тот же прием можно применить к функциям `setHead` и `concat`:

```
sealed class List<out A> {

    fun setHead(a: @UnsafeVariance A): List<A> = when (this) {
        is Cons -> Cons(a, this.tail)
        Nil -> throw IllegalStateException("setHead called on an empty list")
    }

    fun cons(a: @UnsafeVariance A): List<A> = Cons(a, this)

    fun concat(list: List<@UnsafeVariance A>): List<A> = concat(this, list)
    ...
```

Это необходимо сделать только для функций, принимающих параметр типа `A` или `List<A>`. Применять этот прием к функциям без параметров, а также к функции `dropWhile`, принимающей параметр типа `(A) -> Boolean`, не нужно. Также вы должны убедиться, что при использовании этого трюка ни одно небезопасное приведение типа никогда не потерпит неудачу. Как обычно, большая свобода предполагает большую ответственность.

Обратите внимание, что есть еще одно решение: создать абстрактный класс `Empty<A>` для представления пустых списков и затем определить объект-синглтон `Nil<Nothing>`. Это позволит определить абстрактные функции в родительском классе `List` и конкретные реализации в `Cons` и `Empty`. Вот как можно определить функцию `concat`:

```
sealed class List<A> {

    abstract fun concat(list: List<A>): List<A>

    abstract class Empty<A> : List<A>() {
        override fun concat(list: List<A>): List<A> = list
    }

    private object Nil : Empty<Nothing>()

    private class Cons<A>(private val head: A,
                          private val tail: List<A>) : List<A>() {
        override fun concat(list: List<A>): List<A> =
                Cons(this.head, list.concat(this.tail))
    }
}
```

Это решение может показаться более привлекательным для программистов на Java, потому что позволяет избежать проверки типов, как в этом примере:

```
fun <A> concat(list1: List<A>, list2: List<A>): List<A> = when (list1)
{
    Nil -> list2
    is Cons -> Cons(list1.head, concat(list1.tail, list2))
}
```

В любом случае благодаря использованию вариантности `out` наша функция `sum` теперь прекрасно компилируется и работает!

Упражнение 5.7

Напишите функцию, вычисляющую произведение элементов списка вещественных чисел, использовав рекурсивный подход.

Решение

Вот определение рекурсивного вычисления произведения элементов непустого списка:

```
head * product of tail
```

Но что нужно вернуть, если список пуст? Вы легко ответите на этот вопрос, вспомнив школьный курс математики. Также ответ можно вывести из предыдущего определения для непустого списка.

Подумайте, что случится после применения рекурсивной формулы ко всем элементам. Вы должны будете умножить полученный результат на произведение всех элементов пустого списка. То есть у вас нет другого выбора, кроме как сказать, что произведение всех элементов пустого списка равно 1.

Это та же ситуация, как в примере вычисления суммы элементов, где мы использовали `0` в роли суммы всех элементов пустого списка. `0` – это единичное значение, или нейтральный элемент, для операции суммирования, а `1` – это единичное значение, или нейтральный элемент, для произведения. То есть функцию `product` можно реализовать так:

```
fun product(ints: List<Int>): Int = when (ints) {
  List.Nil -> 1
  is List.Cons -> ints.head * product(ints.tail)
}
```

Операция вычисления произведения имеет одно важное отличие от операции суммирования: она имеет *поглощающий элемент*, удовлетворяющий следующему условию: *a * поглощающий элемент = поглощающий элемент * a = поглощающий элемент*.

Поглощающим элементом для произведения является число `0`. Поглощающий элемент для любой операции (если имеется) также называется *нулевым элементом*. Получив нулевой элемент, вычисления можно прервать, т. е., как говорят, выполнить вычисления *по короткой схеме*, как здесь:

```
fun product(ints: List<Double>): Double = when (ints) {
    List.Nil -> 1.0
    is List.Cons -> if (ints.head == 0.0)
                        0.0
                    else
                        ints.head * product(ints.tail)
}
```

Но давайте забудем об этой оптимизации и посмотрим на определения `sum` и `product`. Нет ли здесь единого шаблона, который можно было бы абстрагировать? Поставим функции рядом друг с другом (изменив имя параметра для единообразия):

```
fun sum(ints: List<Int>): Int = when (ints) {
    List.Nil -> 0
    is List.Cons -> ints.head + sum(ints.tail)
}
```

```
fun product(ints: List<Double>): Double = when (ints) {
    List.Nil -> 1.0
    is List.Cons -> ints.head * product(ints.tail)
}
```

Теперь устраним различия, заменив их обобщенной нотацией:

```
fun sum(list: List<Type>): Type = when (list) {
    List.Nil -> identity
    is List.Cons -> list.head operator operation(list.tail)
}

fun product(list: List<Type>): Type = when (list) {
    List.Nil -> identity
    is List.Cons -> ints.head operator operation(list.tail)
}
```

Как видите, получилось две абсолютно идентичные функции, отличающиеся только значениями для `Type`, `operation`, `identity` и `operator`. Если найти способ абстрагировать эти общие элементы, вам останется только передать информацию о переменных элементах, и вы получите единую реализацию. Такая обобщенная функция называется *сверткой*, и мы познакомились с ней в главе 4. В этой главе вы узнали, что есть два вида свертки: `foldRight` и `foldLeft`.

В листинге 5.2 показаны общие части операций `sum` и `product`, абстрагированные в функцию `foldRight`, которая принимает список для свертки, единичное значение и функцию высшего порядка, представляющую операцию. Аргумент `identity`, как нетрудно догадаться, представляет единичное значение для данной операции, а функция имеет каррированную форму. (Подробнее о каррировании рассказывается в главе 3.) Также в листинге 5.2 определены функции, представляющие операции.

Листинг 5.2 Реализация foldRight и использующие ее функции sum и product

```
fun <A, B> foldRight(list: List<A>, ①
                     identity: B, ②
                     f: (A) -> (B) -> B): B = ③
    when (list) {
        List.Nil -> identity
        is List.Cons -> f(list.head)(foldRight(list.tail, identity, f))
    }

fun sum(list: List<Int>): Int = ④
    foldRight(list, 0) { x -> { y -> x + y } }

fun product(list: List<Double>): Double = ④
    foldRight(list, 1.0) { x -> { y -> x * y } }
```

① **A и B представляют типы**
② **Единичное значение для операции свертки**
③ **Функция f имеет каррированную форму и представляет оператор**
④ **Функции (`sum` и `product`), реализующие операции**

Переменная часть `Type` была заменена двумя типами, `A` и `B`, потому что тип результата свертки не всегда совпадает с типом элементов списка. Здесь мы пошли чуть дальше по пути абстрагирования, чем это необходимо для операций вычисления суммы и произведения, но это скоро нам пригодится. Переменная часть `operation` – это имена двух функций.

Операция свертки не зависит от конкретного арифметического действия. Ее можно использовать, например, для преобразования списка символов в строку. В этом случае `A` и `B` будут двумя разными типами: `Char` и `String`. Также операцию свертки можно применить для преобразования списка строк в одну строку. Сможете теперь заметить, как реализовать `concat` с использованием свертки?

Кстати, `foldRight` сама напоминает односвязный список. Если представить список 1, 2, 3 как

```
Cons(1, Cons(2, Cons(3, Nil)))
```

сразу становится заметным его сходство с правой сверткой:

```
f(1, f(2, f(3, identity)))
```

Возможно, вы уже поняли, что `Nil` – это единичный элемент для операции конкатенации списков, хотя можно обойтись и без него, если список списков для конкатенации не пустой. В этом случае операция будет называться уже не *сверткой* (fold), а *сокращением* (reduce). Такое возможно, только если результат имеет тот же тип, что и элементы списков. Выразить это в коде можно, передав `Nil` и `cons` в `foldRight` в качестве единичного значения и функции, используемой для свертывания:

```
foldRight(List(1, 2, 3), List()) { x: Int ->
        { y: List<Int> ->
            y.cons(x)
        }
    }
```

Этот вызов вернет новый список с теми же элементами и в том же порядке, в чем легко убедиться, выполнив следующий код:

```
println(foldRight(List(1, 2, 3), List()) { x: Int ->
        { y: List<Int> ->
            y.cons(x)
        }
    })
```

Этот код выведет:

```
[1, 2, 3, NIL]
```

Далее приводится расшифровка происходящего на каждом шаге:

```
foldRight(List(1, 2, 3), List(), x -> y -> y.cons(x));
foldRight(List(1, 2), List(3), x -> y -> y.cons(x));
foldRight(List(1), List(2, 3), x -> y -> y.cons(x));
foldRight(List(), List(1, 2, 3), x -> y -> y.cons(x));
```

Функцию `foldRight` следует поместить в объект-компаньон класса `List` и добавить функцию экземпляра, которая будет передавать ссылку `this` в вызов `foldRight`:

```
fun <B> foldRight(identity: B, f: (A) -> (B) -> B): B =
    foldRight(this, identity, f)
```

Упражнение 5.8

Напишите функцию для вычисления длины списка. Она должна использовать функцию `foldRight`.

Решение

Эту функцию можно определить непосредственно в классе `List`:

```
fun length(): Int = foldRight(0) { _ -> { it + 1} }
```

Так как первый параметр, представляющий элемент списка, не используется, по соглашениям для него выбрано имя _ (символ подчеркивания). Для второго параметра используется имя `it`. На каждом шаге рекурсии к значению счетчика прибавляется 1. Если параметр не используется, его можно опустить. В этом случае объявление функции упрощается до:

```
fun length(): Int = foldRight(0) { { it + 1} }
```

Эта реализация, кроме того, что она рекурсивная (а значит, есть опасность переполнения стека), имеет низкую производительность. Но даже если ее преобразовать в сорекурсивную версию, ее производительность все равно останется равной $0(n)$, т. е. время, необходимое для вычислений, прямо пропорционально длине списка. В следующих главах вы увидите, как получить длину односвязного списка за постоянное время.

Упражнение 5.9

Функция `foldRight` – рекурсивная, поэтому она несет риск переполнения стека. Насколько быстро это произойдет, зависит от нескольких факторов, наиболее важным из которых является размер стека. Определите сорекурсивную функцию `foldLeft`, которая не вызывает переполнения стека. Вот ее сигнатура:

```
public <B> foldLeft(identity: B, f: (B) -> (A) -> B): B
```

Подсказка

Если вы забыли, чем `foldLeft` отличается от `foldRight`, вернитесь к главе 4.

Решение

Реализация вспомогательной функции в объекте-компаньоне напоминает функцию для `foldRight`, отличаясь лишь параметром, в котором принимает функцию: `(B) -> (A) -> B` вместо `(A) -> (B) -> B`. Если список во втором параметре представлен экземпляром `Nil` (пустой список), функция `foldLeft` должна вернуть аккумулятор `acc`, как это делает `foldRight`. Если список не пустой (`Cons`), `foldLeft` должна применить функцию из параметра к аккумулятору и голове списка и затем вызвать саму себя:

```
tailrec fun <A, B> foldLeft(acc: B, list: List<A>, f: (B) -> (A) -> B):
B =
    when (list) {
        List.Nil -> acc
        is List.Cons -> foldLeft(f(acc)(list.head), list.tail, f)
    }
```

Эта вспомогательная функция вызывается главной функцией из класса `List`:

```
fun <B> foldLeft(identity: B, f: (B) -> (A) -> B): B =
    foldLeft(identity, this, f)
```

Упражнение 5.10

Используйте новую функцию `foldLeft`, чтобы создать новые версии `sum`, `product` и `length`, не страдающие проблемой переполнения стека.

Решение

Вот версия `sum`, использующая функцию `foldLeft`:

```
fun sum(list: List<Int>): Int = list.foldLeft(0, { x -> { y -> x + y } })
```

Версия функции `product`:

```
fun product(list: List<Double>): Double =
    list.foldLeft(1.0, { x -> { y -> x * y } })
```

И версия функции `length`:

```
fun length(): Int = foldLeft(0) { { _ -> it + 1} }
```

И снова второй параметр функции `length` (представляющий каждый элемент списка в каждом вызове функции) игнорируется. Первый параметр представлен ключевым словом `it`. Поэтому, в соответствии с соглашениями, мы можем опустить параметр, обозначаемый символом подчеркивания, и не использовать явное имя для первого параметра, как в примере ниже:

```
fun length(): Int = foldLeft(0) { i -> { i + 1} }
```

Эта функция почти так же неэффективна, как версия с `foldRight`, и в промышленном коде ее лучше не использовать. Она не страдает проблемой переполнения стека, но действует медленно, потому что должна обойти все элементы списка, чтобы подсчитать их.

Упражнение 5.11

Напишите функцию переворачивания списка, используя `foldLeft`.

Подсказка

Обратите внимание на тип единичного значения. Его нужно указать явно.

Решение

Перевернуть список с использованием левой свертки просто. Нужно начать с пустого списка в качестве аккумулятора и последовательно до-

бавлять элементы из первого списка в аккумулятор с помощью функции cons:

```
fun reverse(): List<A> =
    foldLeft(Nil as List<A>) { acc -> { acc.cons(it) } }
```

Как видите, в данном случае нужно явно передать параметр Nil и привести его к типу List. Если этого не сделать, Kotlin использует Nil как тип возвращаемого значения. В предыдущей версии reverse мы использовали вспомогательную функцию, принимающую аргумент типа List, поэтому приведение типа производилось неявно. Мы не можем использовать вызов List() непосредственно, потому что это будет расценено компилятором как вызов конструктора класса List, что сделать невозможно, потому что это абстрактный класс. Чтобы обойти эту проблему, функцию invoke следует вызвать явно:

```
fun reverse(): List<A> =
    foldLeft(List.invoke()) { acc -> { acc.cons(it) } }
```

Упражнение 5.12

Перепишите foldRight, использовав в новой версии foldLeft.

Подсказка

Используйте только что реализованные функции.

Решение

В результате получится медленная версия foldRight, но не страдающая проблемой переполнения стека:

```
fun <B> foldRightViaFoldLeft(identity: B, f: (A) -> (B) -> B): B =
    this.reverse().foldLeft(identity) { x -> { y -> f(y)(x) } }
```

Обратите внимание, что аналогично можно определить функцию foldLeft в терминах foldRight, хотя такое решение не имеет практической ценности. Фактически реализация foldRight через foldLeft тоже не особенно полезна. Как будет показано в главе 9, функция foldRight в основном используется для свертки длинных или бесконечных ленивых коллекций без полного их вычисления. Вызов reverse заставляет вычислить весь список, что уничтожает волшебство ленивых коллекций.

5.6.6 Безопасная рекурсивная версия foldRight

Как я уже говорил, рекурсивная реализация foldRight показана в этой книге только для демонстрации основных идей, но она страдает проблемой переполнения стека и не должна использоваться в промышленном коде. Также обратите внимание, что это – статическая реализация. Реализацию в виде функции экземпляра можно сделать намного проще в использовании и разрешить включать ее в цепочки вызовов с использованием объектной нотации.

Упражнение 5.13

Используя знания, полученные в главе 4, напишите сорекурсивную версию `foldRight` без явного использования `foldLeft`. Назовите эту функцию `coFoldRight`.

Подсказка

Напишите вспомогательную функцию в объекте-компаньоне и основную функцию в классе `List`. Имейте в виду, что здесь есть одна хитрость, из-за которой вы должны объявить вспомогательную функцию приватной.

Решение

Хитрость заключается в том, чтобы передать вспомогательной функции уже перевернутый список. Сама вспомогательная функция выполняет только часть работы. Вот она:

```
private tailrec fun <A, B> cofoldRight(acc: B,
                                       list: List<A>,
                                       identity: B,
                                       f: (A) -> (B) -> B): B =
    when (list) {
        List.Nil -> acc
        is List.Cons ->
            cofoldRight(f(list.head)(acc), list.tail, identity, f)
    }
```


Теперь напишем основную функцию:

```
fun <B> cofoldRight(identity: B, f: (A) -> (B) -> B): B =
            cofoldRight(identity, this.reverse(), identity, f)
```

К сожалению, эта реализация имеет ту же проблему, что и версия, использующая `foldLeft`: она вынуждает вычислить весь список, что исключает главное достоинство свертки справа.

Упражнение 5.14

Реализуйте функцию `concat` в терминах `foldLeft` или `foldRight`. Поместите эту реализацию в объект-компаньон, заменив предыдущую рекурсивную реализацию, а затем вызовите эту функцию из функции-расширения, объявленной за пределами класса `List`.

Решение

Функцию `concat` легко реализовать с использованием правой свертки:

```
fun <A> concatViaFoldRight(list1: List<A>, list2: List<A>): List<A> =
        foldRight(list1, list2) { x -> { y -> Cons(x, y) } }
```

Реализация с использованием левой свертки напоминает функцию `reverse`, которая применяет `foldLeft` к предварительно перевернутому списку и использует второй список как аккумулятор. Обратите внимание, что в ней используется ссылка на функцию (`x::cons`):

```
fun <A> concatViaFoldLeft(list1: List<A>, list2: List<A>): List<A> =
        list1.reverse().foldLeft(list2) { x -> x::cons }
```

Если использование ссылки на функцию кажется вам неудобным, ее можно заменить лямбда-выражением:

```
fun <A> concat(list1: List<A>, list2: List<A>): List<A> =
        list1.reverse().foldLeft(list2) { x -> { y -> x.cons(y) } }
```

Здесь также можно использовать конструктор списка, как в реализации на основе `foldRight`:

```
fun <A> concat(list1: List<A>, list2: List<A>): List<A> =
        list1.reverse().foldLeft(list2) { x -> { y -> Cons(y, x) } }
```

Реализация на основе `foldLeft` менее эффективна, потому что требует предварительно перевернуть список. С другой стороны, она не страдает проблемой переполнения стека.

Упражнение 5.15

Напишите функцию, которая преобразует список списков в плоский список, содержащий все элементы из всех вложенных списков.

Подсказка

Это преобразование выполняется как последовательность операций конкатенации. Его можно сравнить с суммированием элементов списка целых чисел, только в этом случае вместо чисел используются списки, а взамен сложения выполняется конкатенация. В остальном это та же самая функция `sum`.

Решение

И снова используем ссылку на функцию вместо лямбда-выражения, чтобы представить вторую часть функции: выражение `{ x -> x::concat }` эквивалентно выражению `{ x -> { y -> x.concat(y) } }`:

```
fun <A> flatten(list: List<List<A>>): List<A> =
        list.foldRight(Nil) { x -> x::concat }
```

Чтобы избавиться от проблемы переполнения стека, вместо `foldRight` можно использовать `coFoldRight`:

```
fun <A> flatten(list: List<List<A>>): List<A> =
        list.coFoldRight(Nil) { x -> x::concat }
```

5.6.7 Отображение и фильтрация списков

Мы можем определить множество полезных абстракций для работы со списками. Одна из таких абстракций – изменение всех элементов списка применением некоторой общей функции.

Упражнение 5.16

Напишите функцию, которая принимает список целых чисел и умножает каждое число на 3.

Подсказка

Попробуйте использовать функции, которые мы уже написали. Не используйте (со)рекурсию явно. Ваша цель – абстрагировать рекурсию один раз и затем использовать эту абстракцию везде, где она может понадобиться.

Решение

Мы должны применить `foldRight` с пустым списком в роли единичного значения и функцию, добавляющую в список каждый элемент, умноженный на 3:

```
fun triple(list: List<Int>): List<Int> =
        List.foldRight(list, List()) { h ->
            { t: List<Int> ->
                t.cons(h * 3)
            }
        }
```

Мы должны явно указать тип для единичного значения или для параметра `t`. Это обусловлено ограниченными возможностями механизма автоматического определения типов в Kotlin. Также можно использовать левую свертку, чтобы устранить проблему переполнения стека, но при этом придется сначала перевернуть исходный список, а затем перевернуть результат.

Упражнение 5.17

Напишите функцию, которая преобразует каждый элемент списка `List<Double>` в строку `String`.

Решение

Эту операцию можно представить как конкатенацию пустого списка требуемого типа (`List<String>`) с исходным списком, каждый элемент которого преобразуется в строку перед добавлением в аккумулятор. Получившаяся реализация напоминает функцию `concat`:

```
fun doubleToString(list: List<Double>): List<String> =
    List.foldRight(list, List()) { h ->
        { t: List<String> ->
            t.cons(h.toString())
        }
    }
```

Упражнение 5.18

Добавьте в класс `List` функцию `map`, которая не страдает проблемой переполнения стека и позволяет изменить каждый элемент в списке применением заданной функции. На этот раз объявите ее как функцию экземпляра `List`. Вот ее сигнатура:

```
fun <B> map(f: (A) -> B): List<B>
```

Решение

Чтобы избавиться от проблемы переполнения стека, используем левую свертку, а затем перевернем результат:

```
fun <B> map(f: (A) -> B): List<B> =
    foldLeft(Nil) { acc: List<B> -> { h: A -> Cons(f(h), acc) }
}.reverse()
```

Также можно использовать безопасную функцию `coFoldRight` и получить тот же результат (но при этом придется перевернуть исходный список вместо результата):

```
fun <B> map(f: (A) -> B): List<B> =
    coFoldRight(Nil) { h -> { t: List<B> -> Cons(f(h), t) } }
```

Упражнение 5.19

Напишите функцию `filter`, удаляющую из списка элементы, которые не удовлетворяют заданному условию. Реализуйте ее как функцию экземпляра. Вот сигнатура этой функции:

```
fun filter(p: (A) -> Boolean): List<A>
```

Решение

Ниже показана реализация в родительском классе `List`, использующая `coFoldRight`:

```
fun filter(p: (A) -> Boolean): List<A> =
    coFoldRight(Nil) { h -> { t: List<A> -> if (p(h)) Cons(h, t) else t
} }
```

Упражнение 5.20

Напишите функцию `flatMap`, которая принимает список `List<A>`, к каждому его элементу применяет функцию `A -> List<B>` и возвращает `List<B>`. Вот ее сигнатура:

```
fun <B> flatMap(f: (A) -> List<B>): List<B> =
```

Для списка `List(1,-1,2,-2,3,-3)`, например, вызов

```
List(1,2,3).flatMap { i -> List(i, -i) }
```

должен вернуть

```
List(1,-1,2,-2,3,-3).
```

Решение

Функцию `flatMap` можно представить как композицию функции `map`, которая использует функцию, возвращающую список, и функции `flatten`, преобразующей список списков в плоский список. Вот как выглядит простейшая реализация (в классе `List`):

```
fun <B> flatMap(f: (A) -> List<B>): List<B> = flatten(map(f))
```

Упражнение 5.21

Напишите новую версию `filter` на основе `flatMap`.

Решение

Вот одна из возможных реализаций:


```
fun filter(p: (A) -> Boolean): List<A> =
    flatMap { a -> if (p(a)) List(a) else Nil }
```

Обратите внимание на тесную связь между функциями `map`, `flatten` и `flatMap`. Если передать в вызов `map` функцию, возвращающую список, вы получите список списков. К результату можно применить функцию `flatten` и получить единый список, содержащий все элементы из списков, вложенных в исходный список. Тот же результат можно получить, непосредственно применив `flatMap`.

Итоги

- Структуры данных – одно из важнейших понятий в программировании, потому что позволяют обрабатывать наборы данных как одно целое.
- Односвязный список – это эффективная структура данных для программирования с использованием функций. Он обладает преимуществом неизменяемых списков и позволяет вносить некоторые изменения, такие как вставка и удаление элемента в первой позиции за постоянное (и короткое) время. Это объясняется тем, что, в отличие от списков в Kotlin (неизменяемых), эти операции не связаны с копированием элементов.
- Использование общих данных позволяет увеличить производительность некоторых операций, хотя и не всех.
- Для конкретных случаев использования можно создавать специализированные структуры данных, чтобы добиться более высокой производительности.
- Свертку списков можно реализовать методом рекурсивного применения функций.
- Применение сорекурсии для свертки списков избавляет от проблемы переполнения стека.
- После определения функций `foldRight` и `foldLeft` исчезает необходимость повторно использовать (со)рекурсию для обработки списков. `foldRight` и `foldLeft` абстрагируют (со)рекурсию.

6 Необязательные данные

Эта глава охватывает следующие темы:

- пустая ссылка `null`, или «ошибка на миллиард долларов»;
- альтернативы ссылке `null`;
- тип `Option` для представления необязательных данных;
- применение функций к необязательным значениям;
- конструирование необязательных значений.

Представление необязательных данных всегда вызывало сложности в компьютерных программах. Идея необязательных данных проста. В повседневной жизни легко представить отсутствие чего-либо, когда это что-то хранится в контейнере, – чем бы оно ни было, его можно представить в виде пустого контейнера. Например, отсутствие яблок можно представить как пустую корзину для яблок. Отсутствие бензина в автомобиле можно представить как пустой бензобак. Но представить отсутствие данных в компьютерных программах немного сложнее.

В большинстве случаев данные представлены ссылками, указывающими на них, поэтому наиболее очевидным способом представления отсутствующих данных является использование никуда не указывающего указателя. Такие указатели мы называем *пустыми (или нулевыми) указателями*. *Ссылка* в языке Kotlin – это указатель на значение.

В большинстве языков программирования ссылки можно изменять и заставлять их указывать на новое значение. По этой причине вместо термина «ссылка» часто используется термин «переменная». Слово «переменная» не вызывает путаницы, если все ссылки допускают возмож-

ность записи в них указателей на новые значения. В таких случаях принято использовать термин «переменная» вместо «ссылка». Но ситуация меняется, когда используются ссылки, которые нельзя переназначить. В таком случае вы получаете два типа переменных: которые могут изменяться и которые не могут, и тогда безопаснее использовать термин «ссылка».

Некоторые ссылки можно инициализировать значением `null`, а затем изменить, чтобы они указывали на конкретные значения. Более того, их можно изменить снова, записав указатель `null` после удаления данных. В таких случаях их вполне можно называть *переменными*, т. е. *ссылками на переменные*.

С другой стороны, некоторые ссылки нельзя создать, не инициализировав их конкретными значениями, и после этого их нельзя изменить, чтобы они указывали на другие значения. Такие ссылки иногда называют *константами*. Например, ссылки в Java обычно являются переменными, если не объявлены как `final`, что делает их константами. В Kotlin ссылки на переменные объявляются с ключевым словом `var`, а непеременные ссылки, которые также называют *неизменяемыми*, объявляются как `val`.

В первый момент может показаться, что на необязательные данные можно ссылаться только с помощью `var`. Какой смысл создавать ссылку на отсутствующие данные, если потом в нее нельзя записать ссылку на некоторые существующие данные? Но давайте подумаем, как еще можно представить отсутствие данных? Самый распространенный способ – использовать так называемую *пустую ссылку*. Пустая ссылка – это ссылка, не указывающая ни на что, пока в нее не будет записан указатель на некоторые данные.

В этой главе вы узнаете, как обращаться с необязательными данными – отсутствующими значениями, отсутствие которых не является результатом ошибки. Я начну с обсуждения проблем с пустым указателем. После этого я расскажу, как Kotlin обрабатывает пустые ссылки и альтернативы пустым ссылкам, имеющиеся в Kotlin. А затем покажу, как создать и использовать тип `Option` для работы с необязательными данными. Этот тип вы сможете использовать для композиции функций, даже если фактические данные отсутствуют.

6.1 Проблемы с пустым указателем

Одной из наиболее распространенных ошибок в программах, допускающих возможность появления пустых ссылок, является исключение `NullPointerException`. Это исключение возникает при попытке разыменовать идентификатор, когда обнаруживается, что он ни на что не указывает: программа ожидает найти некоторые данные и обнаруживает, что они отсутствуют. Говорят, что такой идентификатор указывает *в никуда*. В этом разделе я расскажу о проблемах использования пустых ссылок.

В 1965 году, занимаясь разработкой языка ALGOL (это объектно-ориентированный язык), Тони Хоар (Tony Hoare) изобрел пустую ссылку. Позднее он неоднократно сожалел об этом. Вот что он сказал[1] в 2009 году:

> Я называю это своей ошибкой на миллиард долларов... Я хотел гарантировать абсолютную безопасность любых операций со ссылками, включив автоматическую проверку в компилятор. Но я не смог устоять перед соблазном добавить пустую ссылку (null reference), потому что реализовать ее было так просто. Это решение привело к бесчисленному количеству ошибок и уязвимостей, ставших причиной миллиардных убытков за последние 40 лет.

В настоящее время общепринятым считается правило избегать пустых ссылок, но на практике оно применяется далеко не всегда. Стандартная библиотека Java, которая также используется в программах на Kotlin, содержит методы и конструкторы, принимающие необязательные параметры, которые по умолчанию получают значение `null`. Возьмем, к примеру, класс `java.net.Socket`. Он определяет следующий конструктор:

```
public Socket(String address,
              int port,
              InetAddress localAddr,
              int localPort) throws IOException
```

Согласно документации[2],

> Если в параметре `localAddr` передается `null`, это эквивалентно передаче локального адреса.

Здесь пустая ссылка является допустимой. Этот способ представления отсутствия данных не является характерным только для объектов. Порт (`localPort`) тоже может отсутствовать; но в Java он не может иметь значения `null`, потому что это примитив:

> Нулевое значение в параметре `localPort` позволяет системе выбрать свободный порт в операции `bind`.

Такого рода значения иногда называют *сигнальными* или *контрольными*. Они используется не как фактические значения (в данном случае не имеется в виду порт с номером 0), а сигнализируют об отсутствии значения. В стандартной библиотеке Java можно найти массу других примеров представления отсутствующих данных.

Использование значения `null` для представления отсутствия данных опасно, потому что отсутствие локального адреса может оказаться непреднамеренным, а обусловлено некоторой предыдущей ошибкой. Но

---

[1] *Tony Hoare*. Null References: The Billion Dollar Mistake // QCon. August 25, 2009. http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/Socket.html.

это не вызовет исключения. Программа продолжит работать, хотя и не так, как задумано.

Есть и другие случаи законного использования пустых ссылок. Попытавшись извлечь значение из `HashMap`, используя ключ, отсутствующий в хеше, вы получите `null`. Это ошибка? Трудно сказать. Возможно, ключ правильный, но не был добавлен в хеш, или, может быть, ключ действительно правильный и должен присутствовать в хеше, но не был включен из-за другой ошибки. Сам ключ тоже может быть `null` и иметь непустое значение, и это не вызовет исключения, потому что ключ `null` поддерживается в `HashMap`. Полный бардак. Как профессиональный программист, вы знаете, что с этим делать:

- никогда не использовать ссылку, не проверив ее (вы должны выполнять такую проверку для каждого параметра функции);
- никогда не извлекать значение из хеша без предварительной проверки наличия соответствующего ключа;
- никогда не пытаться получить элемент из списка, не убедившись, что список не пуст и содержит достаточно элементов, если вы обращаетесь к элементу по его индексу.

Следуя этим правилам, вы никогда не столкнетесь с исключением `NullPointerException` или `IndexOutOfBoundsException` и сможете мирно сосуществовать с пустыми ссылками. Но, забыв использовать какое-то правило, вы почти наверняка столкнетесь с этими неприятными исключениями, и, чтобы этого не произошло, желательно иметь какой-то более простой и безопасный способ обработки отсутствия значений. В этой главе вы узнаете, как обращаться с отсутствием значений, которое не является результатом ошибки. Данные этого вида называют *необязательными данными*.

Для работы с необязательными данными есть масса приемов. Один из самых известных и наиболее часто используемых – это список. Когда кроме обычных значений функция может вернуть `null`, некоторые программисты возвращают список. Список может содержать 1 или 0 элементов. Это вполне надежный способ, но он имеет несколько существенных недостатков:

1. Невозможно гарантировать, что список не будет содержать более одного элемента. Что делать, получив список с несколькими элементами?
2. Как отличить список, который должен содержать не более одного элемента, от обычного списка?
3. Класс `List` определяет множество функций, помогающих обрабатывать несколько элементов. Эти функции останутся для вас бесполезными.
4. Списки – сложные структуры, а вам не нужны лишние сложности. Было бы достаточно более простой реализации.

6.2 Как пустые ссылки обрабатываются в Kotlin

В главе 2 вы узнали, как обрабатываются пустые ссылки в Kotlin. Обычные типы в Kotlin не поддерживают значение `null`. Если ссылка принадлежит к одному из таких типов, она никогда не получит значения `null`, будь то `val` (константа) или `var` (переменная). Константа `val` с пустой ссылкой – довольно бессмысленная конструкция. Но переменная `var` со значением `null` вполне может пригодиться. Иногда бывает желательно объявить ссылку некоторого типа, не присваивая сразу же конкретного значения. И при этом ссылку нужно объявить так, чтобы она имела правильную область видимости. Рассмотрим следующий пример:

```
while(enumeration.hasNext()) {
    result = process(result, enumeration.next())
} use(result)
```

В этом примере элементы перечисления `enumeration` используются внутри цикла, который перебирает содержимое перечисления до его исчерпания. Где в этом примере следует объявить переменную `result`, и какое начальное значение следует присвоить ей? Одно из решений, которые приходят на ум: обработать первый элемент перечисления вне цикла:

```
var result = process(enumeration.next())
while(enumeration.hasNext()) {
    result = process(result, enumeration.next())
} use(result)
```

Но этот код вызовет ошибку, если в перечислении `enumeration` нет ни одного элемента. Другое типичное решение – объявить `result` как пустую ссылку:

```
var result = null
while(enumeration.hasNext()) {
    result = process(result, enumeration.next())
} use(result)
```

Теперь функция `process` должна предусматривать обработку `null` в `result`. (В предыдущем примере использовалась перегруженная версия, принимающая единственный аргумент.) Теперь пустая ссылка `result` означает, что выполняется первая итерация. Но Kotlin не скомпилирует этот код.

Kotlin поддерживает пустые ссылки, но требует, чтобы вы сообщили компилятору, что знаете, что ссылка может быть пустой. Это не избавит от исключения `NullPointException`, но заставит вас принять на себя ответственность за это. Это как отказ от ответственности: используя пустую ссылку, вы действуете на свой страх и риск. Компилятор Kotlin оказывает обширную помощь программисту, использующему изменяемые ссылки. Программисты, предпочитающие использовать неизменяемые значения, напротив, не используют пустые ссылки. Однако особенности работы с типами, поддерживающими значение `null`, в Kotlin все еще могут

здорово помочь в разработке безопасных программ. Программисты на Kotlin могут гарантировать, что в их коде никогда не будет пустых ссылок. (Краткое описание типов с поддержкой и без поддержки значения `null` вы найдете в главе 2.)

Как рассказывалось в главе 2, ссылкам обычных типов нельзя присвоить значение `null`. Чтобы получить такую возможность, нужно использовать типы с поддержкой `null`. Такие типы записываются так же, как обычные, но с вопросительным знаком (`?`) в конце. Ссылке `Int` нельзя присвоить `null`, но ссылке типа `Int?` можно. С этого момента мы будем использовать только типы, не поддерживающие пустые ссылки, чтобы никогда не встретиться с исключением `NullPointException` в своем коде.

6.3 Альтернативы пустым ссылкам

Теперь мы знаем, что в нашем коде никогда не возникнет исключение `NullPointerException`, верно? Не совсем так, потому что многие функции в библиотеках Kotlin принимают параметры с поддержкой `null` и могут возвращать пустые ссылки. Функции, способные принимать `null` в аргументах, не являются проблемой, потому что типы без поддержки `null` являются потомками типов с такой поддержкой. `Int` является подтипом `Int?`, `String` – подтипом `String?` и т. д. Поэтому мы всегда сможем вызвать функцию, принимающую тип с поддержкой `null`, передав аргумент соответствующего типа без такой поддержки.

Однако возможность возврата пустого значения намного более сложная проблема. Рассмотрим пример, использующий ассоциативный массив `Map`. Следующий код не компилируется в Kotlin:

```
val map: Map<String, Person> = ...
val person: Person = map["Joe"]
```

Причина в том, что `map["Joe"]` – это синтаксический сахар, под которым скрывается вызов `map.get("Joe")`, а функция `get` класса `Map` может вернуть экземпляр `Person`, если в массиве имеется ключ `"Joe"` и он содержит непустое значение, или «ничто». Как функция может вернуть «ничто»? Она может вернуть `null`. Даже запретив пустые ссылки в своем коде, вы почти наверняка будете получать их от каких-либо сторонних библиотек. Можно, конечно, отказаться от сторонних библиотек и использовать только свои. Но это решение требует много работы. Другое решение – обернуть вызовы функций, возвращающие `null`, некоторыми функциями-обертками, возвращающие что-то еще. Но что?

Использование библиотек – не единственная возможная причина проблемы. Вот пример функции, возвращающей необязательные данные:

```
fun mean(list: List<Int>): Double = when {
    list.isEmpty() -> TODO("What should you return?")
    else -> list.sum().toDouble() / list.size
}
```

Функция `mean` – это пример частичной функции, с которыми мы познакомились в главе 3. Она определена для всех списков чисел, кроме пустого списка. Что должна вернуть эта функция, получив пустой список? Можно вернуть сигнальное (или контрольное) значение. Но какое значение выбрать? Так как функция должна вернуть значение типа `Double`, можно воспользоваться значением, которое определено в классе `Double`:

```
fun mean(list: List<Int>): Double = when {
    list.isEmpty() -> Double.NaN
    else -> list.sum().toDouble() / list.size
}
```

Это вполне работоспособное решение, потому что `Double.NaN` (Not a Number – не число) является допустимым значением типа `Double`. Но у нас остаются нерешенными три проблемы:

1. Как применить тот же принцип к функции, возвращающей `Int`? В классе `Int` нет специального значения, эквивалентного значению `NaN`.
2. Как сообщить пользователю, что функция может вернуть сигнальное значение?
3. Что должна вернуть обобщенная функция, как, например, представленная ниже?

```
fun <T, U> f(list: List<T>): U = when {
    list.isEmpty() -> ???
    else -> ... (некоторые вычисления, возвращающие значение типа U)
}
```

Еще одно решение – возбудить исключение:

```
fun mean(list: List<Int>): Double = when {
    list.isEmpty() -> throw IllegalArgumentException("Empty list")
    else -> list.sum().toDouble() / list.size
}
```

Но это очень неуклюжее решение, которое создает проблем больше, чем решает:

1. Исключения обычно используются для представления ошибочных результатов, но здесь нет ошибки. Здесь просто нет результата, потому что нет входных данных! Или вызов функции с пустым списком следует рассматривать как ошибку?
2. Какое исключение следует возбудить? Свое? Или выбрать какое-то стандартное?
3. Функция, возбуждающая исключение, перестает быть чистой функцией и теряет способность компоноваться с другими функциями, потому что придется прибегнуть к помощи управляющих структур, таких как `try...catch`, которые являются не чем иным, как современной формой `goto`.

Можно также вернуть значение `null` и переложить ответственность за его обработку на вызывающий код:

```
fun mean(list: List<Int>): Double? = when {
    list.isEmpty() -> null
    else -> list.sum().toDouble() / list.size
}
```

В обычных языках возврат значения `null` – худшее из возможных решений. Например, вот к чему оно приводит в таких языках, как Java:

- вынуждает (в идеале) вызывающий код проверить результат на равенство `null` и выполнить соответствующие действия;
- вызывает крах программы с исключением `NullPointerException`, если используется упакованный тип и при его распаковке не получается преобразовать пустую ссылку в элементарное значение;
- как и в случае с решением на основе исключения, функция теряет способность к компоновке с другими функциями;
- потенциальная проблема может распространиться далеко от точки ее происхождения; если вызывающая программа забудет проверить результат, исключение `NullPointerException` может возникнуть в другом коде, далеко от точки вызова функции.

Язык Kotlin смягчает проблему:

- он вынуждает вызывающий код обработать возможное значение `null`, явно определив тип с его поддержкой;
- крах при распаковке в Kotlin невозможен, потому что в нем не используются элементарные типы и, соответственно, не производится упаковка/распаковка (по крайней мере, на уровне пользователя);
- предлагает операторы, поддерживающие композицию функций, возвращающих типы с поддержкой `null` (подробности см. в главе 2);
- он не решает проблему распространения ошибки, но смягчает ее, распространяя обязательство предусмотреть обработку `null`.

Как я уже говорил, решение в Kotlin лучше, чем отсутствие решения во многих других языках, но оно все еще далеко от идеала. Лучшим решением было бы попросить пользователя указать специальное значение, которое необходимо вернуть, если данные недоступны. Например, следующая функция выполняет поиск максимального значения в списке. Она использует Kotlin-функцию `List.max()`, возвращающую `Int?`, но обрабатывает случай с пустым списком, чтобы дать вам возможность заставить свою функцию возвращать `Int` вместо `Int?`:

```
fun max(list: List<Int>, default: Int): Int = when {
    list.isEmpty() -> default
    else -> list.max()
}
```

Но этот код не скомпилируется, потому что `list.max()` возвращает значение типа `Int?` (сама функция никогда не вернет `null`, но Kotlin этого не видит). Вот как выглядит идиоматическое решение на Kotlin:

```
fun max(list: List<Int>, default: Int): Int = list.max() ?: default
```

А как быть в случаях, когда нет значения по умолчанию? Иногда значение по умолчанию может быть определено позже. Или нужно применить эффект к результату, если значение имеется, и ничего не делать в ином случае. Взгляните на следующий пример, использующий предыдущую функцию max:

```
val max = max(listOf(1, 2, 3), 0)

println("The maximum is $max")
```

Если список пуст, этот код выведет `The maximum is 0` (Максимальное значение 0) что может быть неверно. В конце концов, максимум не равен 0; в данном случае нет максимума. Можно попробовать написать:

```
val max = max(listOf(1, 2, 3), 0)

print(if (max != 0) "The maximum is $max\n" else "")
```

Но и это решение окажется неправильным, если, к примеру, список будет содержать только нули. Можно прибегнуть к традиционному решению:

```
val max = listOf(1, 2, 3).max()

if (max != null) println("The maximum is $max")
```

Но теперь вы вынуждены явно обрабатывать пустые ссылки. Однако не все так плохо, и Kotlin предлагает лучшее решение!

6.4 Тип Option

В оставшейся части этой главы мы напишем свой тип `Option` для работы с необязательными (optional) данными. Наш тип `Option` будет напоминать тип `List`. Мы реализуем его как абстрактный запечатанный класс `Option`, включающий два внутренних подкласса, представляющих наличие или отсутствие данных.

Подкласс, представляющий отсутствие данных, назовем `None`, а подкласс, представляющий наличие данных, – `Some`. Экземпляр `Some` будет содержать соответствующее значение. На рис. 6.1 показано, как тип `Option` может изменить способ композиции функций, а в листинге 6.1 показан код с реализацией типа `Option`.

Листинг 6.1. Тип Option

```
sealed class Option<out A> { ①

    abstract fun isEmpty(): Boolean

    internal object None: Option<Nothing>() { ②

        override fun isEmpty() = true

        override fun toString(): String = "None"

        override fun equals(other: Any?): Boolean =
                                        other === None ③
```

```
        override fun hashCode(): Int = 0
    }

    internal data class Some<out A>(internal val value: A) : Option<A> () { ④
        override fun isEmpty() = false
    }

    companion object {

        operator fun <A> invoke(a: A? = null): Option<A> ⑤
            = when (a) {
                null -> None
                else -> Some(a)
            }
    }
}
```

① **Тип `Option` объявляется ковариантным к `A`**
② **`None` – это объект-синглтон, используемый для представления всех типов**
③ **Любые отсутствующие данные считаются равными**
④ **Экземпляры `Some` считаются равными, если имеют равные значения, именно поэтому мы используем объявление `data class`**
⑤ **Функция `invoke` объявлена с использованием ключевого слова `operator`**

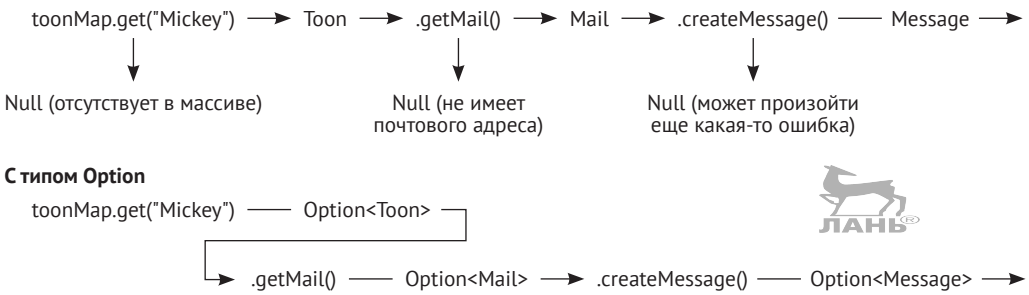


Рис. 6.1 Использование типа Option для представления необязательных данных позволяет компоновать функции, даже если данные могут отсутствовать. Без типа Option композиция функций не будет давать в результате функцию, потому что получившаяся программа потенциально может возбудить исключение NullPointerException

Как видите, класс `Option` близко напоминает класс `List`. Они оба являются абстрактными классами с двумя внутренними реализациями. Подкласс `None` соответствует `Nil`, а подкласс `Some` – `Cons`. Также обратите внимание, что использование `null` в качестве значения по умолчанию для параметра функции `invoke` может выглядеть как грязный трюк. При желании можно было бы объявить перегруженную версию функции без аргументов. Теперь мы можем использовать `Option` для определения функции `max`:

```
fun max(list: List<Int>): Option<Int> = Option(list.max())
```

Как таковой класс `Option` довольно бесполезен. Он позволяет лишь проверить наличие результата и распечатать его. В него можно добавить

функцию, возвращающую значение, но эта функция должна генерировать исключение или возвращать null, получив при вызове None. Возврат null делает бессмысленным создание Option. Возбуждение исключения – тоже плохо, потому что теряется возможность композиции, которую дают другие результаты. Знакомые с Java знают, что в классе Optional в Java есть метод get, возвращающий значение, если оно имеется, и генерирующий исключение в противном случае.

Вот что по этому поводу сказал архитектор языка Java Брайан Гетц (Brian Goetz)[1]:

> Обращение: НИКОГДА не вызывайте Optional.get, если нет полной уверенности, что он не вернет null; используйте один из безопасных методов, таких как orElse или ifPresent. Оглядываясь назад, можно сказать, что методу get мы должны были бы дать другое имя, такое как getOrElseThrowNoSuchElementException, что сделало бы его использование более ясным. Но, как говорят, все мы сильны задним умом.

Действительность намного проще: в Java-классе Optional необязательно было определять метод get, поэтому мы не стали добавлять его в свой класс Option, потому что Option – это вычислительный контекст для безопасной обработки необязательных данных. Необязательные данные потенциально небезопасны, и поэтому нам необходим безопасный контекст для работы с ними. Как следствие, вы *никогда не должны* извлекать значение из этого контекста, предварительно не сделав его безопасным.

6.4.1 Извлечение значения из Option

Многие функции, написанные нами для класса List, также могут пригодиться в классе Option. Но перед тем как заняться этими функциями, рассмотрим несколько примеров использования типа Option. Помните значения по умолчанию?

Упражнение 6.1

Реализуйте функцию getOrElse, возвращающую значение, если оно имеется, или указанное значение по умолчанию. Вот как могла бы выглядеть сигнатура функции:

```
fun getOrElse(default: A): A
```

Решение

Реализуем эту функцию в классе Option и определим ее параметр с типом A, несовместимым с ковариантностью. Отменить проверку вариантности можно с помощью аннотации @UnsafeVariance, как описывалось в главе 5:

[1] Ответ Брайана Гетца на вопрос «Should Java 8 getters return optional type?» (Должен ли метод чтения в Java 8 возвращать тип Optional?) на Stackoverflow (12 октября 2014 года), https://stackoverflow.com/questions/26328555.

```
fun getOrElse(default: @UnsafeVariance A): A = when (this) {
    is None -> default
    is Some -> value
}
```

Теперь мы можем прозрачно использовать функции, возвращающие экземпляры `Option`, как показано ниже:

```
val max1 = max(listOf(3, 5, 7, 2, 1)).getOrElse(0)
val max2 = max(listOf()).getOrElse(0)
```

Здесь `max1` получит значение 7 (максимальное значение в списке), а `max2` – значение 0 (значение по умолчанию). Но у нас все еще могут возникнуть проблемы. Взгляните на следующий пример:

```
fun getDefault(): Int = throw RuntimeException()

fun main(args: Array<String>) {
    val max1 = max(listOf(3, 5, 7, 2, 1)).getOrElse(getDefault())
    println(max1)
    val max2 = max(listOf()).getOrElse(getDefault())
    println(max2)
}
```

Это довольно искусственный пример. Функция `getDefault` – это не совсем функция, она просто демонстрирует, что может произойти. Что выведет этот пример? Если вы думаете, что она выведет 7 и затем возбудит исключение, тогда внимательно еще раз изучите этот код.

Данный пример ничего не выведет и сразу возбудит исключение, потому что Kotlin – строгий язык. Параметры функции вычисляются до ее выполнения независимо от того, нужны они или нет. То есть параметр функции `getOrElse` будет вычислен в любом случае, независимо от того, является ли результат вызова функции `max` экземпляром класса `Some` или `None`. Тот факт, что в некоторых случаях вычислять параметр функции не нужно, не имеет никакого значения. Это обстоятельство не важно, когда параметр является литералом, но очень важно, когда это вызов функции. Функция `getDefault` будет вызываться в любом случае, поэтому первая строка возбудит исключение и ничего не успеет вывести на экран. Обычно такое поведение весьма далеко от желаемого.

Упражнение 6.2

Исправьте предыдущую проблему: добавьте новую версию функции `getOrElse` с отложенным вычислением параметра.

Подсказка

Вместо литерального значения используйте функцию без аргументов, возвращающую значение.

Решение

Вот реализация этой функции в классе `Option`:

```
fun getOrElse(default: () -> @UnsafeVariance A): A = when (this) {
    is None -> default()
```

```
    is Some -> value
}
```

Теперь параметр будет вычисляться вызовом указанной функции только в отсутствие значения. Пример с функцией max можно переписать так:

```
val max1 = max(listOf(3, 5, 7, 2, 1)).getOrElse(::getDefault)
println(max1)
val max2 = max(listOf()).getOrElse(::getDefault)
println(max2)
```

На этот раз программа выведет число 7 и только потом возбудит исключение.

6.4.2 Применение функций к необязательным значениям

Функция map является одной из важнейших в классе List. Она позволяет применить функцию (A) -> B к каждому элементу списка List<A> и получить список List<B>. Учитывая, что класс Option напоминает список, который может содержать не более одного элемента, к нему можно применить ту же идею.

Упражнение 6.3

Напишите функцию map для преобразования Option<A> в Option<B> применением функции (A) -> B.

Подсказка

Решить задачу можно, определив абстрактную функцию в классе Option и конкретные реализации в каждом из подклассов. Также можно определить конкретную реализацию непосредственно в классе Option. В первом случае важно не забыть позаботиться о типе в классе None. Вот как выглядит сигнатура абстрактной функции в классе Option:

```
abstract fun <B> map(f: (A) -> B): Option<B>
```

Решение

Реализация в None должна просто вернуть синглтон None. Обратите внимание, что функция map должна иметь тип (Nothing) -> B:

```
override fun <B> map(f: (Nothing) -> B): Option<B> = None
```

Реализация в Some лишь немногим сложнее. В ней нужно извлечь фактическое значение, применить к нему указанную функцию и заключить результат в новый экземпляр Some:

```
override fun <B> map(f: (A) -> B): Option<B> = Some(f(value))
```

А вот вариант с конкретной реализацией непосредственно в классе Option:

```
fun <B> map(f: (A) -> B): Option<B> = when (this) {
    is None -> None
    is Some -> Some(f(value))
}
```

6.4.3 Композиция функций с типом Option

Как вы скоро увидите, функции типа `(A) -> B` не имеют большого распространения в безопасных программах. Прежде всего, потому что функции, возвращающие необязательные значения, порождают определенные проблемы. В конце концов вам придется проделывать лишнюю работу, обертывая значения экземплярами `Some`. Но со временем вы увидите, что эти операции требуются очень редко.

При объединении функций в цепочку вычислений мы часто начинаем со значения, полученного в ходе некоторых предыдущих вычислений, и затем последовательно передаем промежуточные результаты из функции в функцию, нигде не сохраняя их. Вам чаще придется использовать функции `(A) -> Option<B>`, чем `(A) -> B`. Вспомните класс `List`. Вам это ничего не напоминает? Да, это уже известная нам функция `flatMap`.

Упражнение 6.4

Напишите функцию экземпляра `flatMap`, которая принимает функцию `(A) -> Option<B>` и возвращает `Option<B>`.

Подсказка

Можно было бы определить разные реализации в обоих подклассах, но вы должны попытаться написать уникальную реализацию, работающую с обоими подклассами, и поместить ее в класс `Option`. Вот ее сигнатура:

```
fun <B> flatMap(f: (A) -> Option<B>): Option<B>
```

Можете использовать функции, которые мы уже написали (`map` и `getOrElse`).

Решение

Проще всего было бы определить абстрактную функцию в классе `Option` и конкретные реализации в классах `None` и `Some`, возвращающие `None` и `Some(f(value))` соответственно. Это, пожалуй, самое эффективное решение. Но более красиво выглядит следующее решение: сначала применить функцию `f`, что даст в результате `Option<Option<B>>`, а затем вызвать `getOrElse` с экземпляром `None` в качестве значения по умолчанию, чтобы извлечь значение (`Option<B>`):

```
fun <B> flatMap(f: (A) -> Option<B>): Option<B> =
    map(f).getOrElse(None)
```

Упражнение 6.5

Кроме `flatMap`, отображающей функцию, которая возвращает `Option`, нам также нужна версия `getOrElse`, принимающая значение по умолчанию типа `Option`. Напишите функцию `orElse` со следующей сигнатурой:

```
fun orElse(default: () -> Option<A>): Option<A>
```

Подсказка

Как нетрудно догадаться, реализация этой функции не должна извлекать значение. Именно так обычно используется тип `Option` – через компози-

цию Option, а не обертывание и извлечение значений. В результате одна и та же реализация будет работать для обоих подклассов. Не забывайте, однако, о вариантности.

Решение

Решение заключается в том, чтобы применить функцию _ -> this, получить результат Option <Option<A>>, а затем для этого результата вызвать getOrElse с указанным значением по умолчанию. В соответствии с соглашениями символ подчеркивания (_) в определениях функций используется для обозначения неиспользуемых аргументов. Здесь он обозначает значение, содержащееся в Option. Однако это значение используется косвенно, потому что содержится в Option (если имеется). Решить проблему с вариантностью можно с помощью аннотации @UnsafeVariance:

```
fun orElse(default: () -> Option<@UnsafeVariance A>): Option<A> =
    map { _ -> this }.getOrElse(default)
```

Синтаксис функции можно немного упростить:

```
fun orElse(default: () -> Option<@UnsafeVariance A>): Option<A> =
    map { this }.getOrElse(default)
```

Синтаксис { x } – это простейший способ записать константную функцию, возвращающую x, независимо от используемого аргумента.

Упражнение 6.6

В главе 5 мы написали функцию filter для удаления из списка всех элементов, не соответствующих заданному условию, выраженному в форме предиката (функции, возвращающей Boolean). Напишите аналогичную функцию для Option. Вот ее сигнатура:

```
fun filter(p: (A) -> Boolean): Option<A>
```

Подсказка

Так как Option похож на класс List, который может содержать не более одного элемента, реализация выглядит тривиально просто. В подклассе None нужно вернуть None (или this), а в подклассе Some нужно вернуть исходный экземпляр this, если он соответствует условию, и None в противном случае. Но попробуйте написать универсальную реализацию в классе Option.

Решение

Задача легко решается с помощью flatMap, которая вызовет функцию-предикат только для экземпляра Some:

```
fun filter(p: (A) -> Boolean): Option<A> =
    flatMap { x -> if (p(x)) this else None }
```

Обратите внимание, что использование имени p (от слова *predicate*) является распространенной практикой для обозначения функций, возвращающих Boolean.

6.4.4 Примеры использования Option

Знакомые с Java-классом `Optional` знают, что `Optional` содержит метод `isPresent()`, который позволяет проверить, содержит ли `Optional` конкретное значение. (`Optional` имеет другую реализацию, не основанную на двух разных подклассах.) Мы могли бы реализовать такую функцию в своем Kotlin-классе `Option`, но с именем `isSome()`, потому что она будет проверять, является ли экземпляр представителем `Some` или `None`. Мы также могли бы назвать ее `isNone()`, что выглядело бы более логичным, поскольку эта функция была бы эквивалентом функции `List.isEmpty()`.

Функция `isSome()` может пригодиться в некоторых ситуациях, но это не лучший способ использовать класс `Option`. Если в программе потребуется проверить `Option` с помощью `isSome()` перед вызовом какого-либо метода `getOrThrow()`, чтобы получить значение, такой прием мало будет отличаться от проверки ссылки на `null` перед ее разыменованием. Единственное отличие: забыв выполнить проверку, вы рискуете получить исключение `IllegalStateException`, `NoSuchElementException` или любое другое, выбранное для реализации `getOrThrow()` в `None`, вместо `NullPointerException`.

Лучший способ – воспользоваться приемом композиции. Для этого нужно определить все необходимые функции для всех случаев использования, соответствующих всем действиям со значением, которые вы предприняли бы, убедившись, что оно не равно `null`. Например, можно было бы:

- передать значение в другую функцию;
- применить некоторый эффект к значению;
- применить функцию или эффект к самому значению, если оно не равно `null`, или к значению по умолчанию.

Первый и третий варианты использования уже возможны благодаря функциям, написанным выше, а применить эффект можно несколькими разными способами, о которых вы узнаете в главе 12.

В качестве примера рассмотрим, как с помощью класса `Option` изменить способ использования ассоциативных массивов. В листинге 6.2 используется функция-расширение для класса `Map`, возвращающая `Option` при попытке получить значение указанного ключа. Стандартная реализация `Map.get(key)` в языке Kotlin, которую можно использовать как `Map[key]`, возвращает `null`, если ключ не найден. Мы используем функцию-расширение `getOption`, чтобы вернуть `Option`.

Листинг 6.2 Использование Option для получения значения из Map

```
import com.fpinkotlin.optionaldata.exercise06.Option
import com.fpinkotlin.optionaldata.exercise06.getOrElse

data class Toon (val firstName: String,
                 val lastName: String,
                 val email: Option<String> = Option()) {

    companion object {
        operator fun invoke(firstName: String,
```

```
                            lastName: String,
                            email: String? = null) =
            Toon(firstName, lastName, Option(email))
    }
}

fun <K, V> Map<K, V>.getOption(key: K) =
    Option(this[key]) ①

fun main(args: Array<String>) {

    val toons: Map<String, Toon> = mapOf(
            "Mickey" to Toon("Mickey", "Mouse", "mickey@disney.com"),
            "Minnie" to Toon("Minnie", "Mouse"),
            "Donald" to Toon("Donald", "Duck", "donald@disney.com"))

    val mickey =
        toons.getOption("Mickey").flatMap { it.email } ②
    val minnie = toons.getOption("Minnie").flatMap { it.email }
    val goofy = toons.getOption("Goofy").flatMap { it.email }

    println(mickey.getOrElse { "No data" })
    println(minnie.getOrElse { "No data" })
    println(goofy.getOrElse { "No data" })
}
```

① **Функция-расширение, реализующая прием проверки перед использованием, чтобы исключить возможность возврата пустой ссылки**

② **Композиция `Option` через `flatMap`**

Обратите внимание, что `Option` позволяет также внедрить в реализацию `Map` шаблон проверки наличия ключа вызовом `containsKey` перед вызовом `get`.

Эта простая программа помогает увидеть, как можно объединять разные функции, возвращающие `Option`. Вам не нужно ничего проверять, и вы не рискуете получить `NullPointerException` при попытке получить адрес электронной почты из экземпляра `Toon`, у которого ее нет, или даже экземпляр `Toon`, которого нет в массиве. В листинге 6.3 показано более идиоматическое решение для Kotlin, использующее операторы для работы с типами, поддерживающими значение `null`.

Листинг 6.3 Использование типов с поддержкой null более идиоматическим способом

```
fun main(args: Array<String>) {

    val toons: Map<String, Toon> = mapOf(
            "Mickey" to Toon("Mickey", "Mouse", "mickey@disney.com"),
            "Minnie" to Toon("Minnie", "Mouse"),
            "Donald" to Toon("Donald", "Duck", "donald@disney.com"))

    val mickey = toons["Mickey"]?.email ?: "No data"
    val minnie = toons["Minnie"]?.email ?: "No data"
    val goofy = toons["Goofy"]?.email ?: "No data"

    println(mickey)
```

```
    println(minnie)
    println(goofy)
}
```

Как видите, идиоматическое решение выглядит проще. Но здесь маленькая проблемка. Обе программы производят один и тот же вывод:

```
Some(value=mickey@disney.com)
None
No data
```

Первая строка – это электронная почта Микки (Mickey). Вторая строка содержит слово «None» (Нет), потому что у Минни (Minnie) нет электронной почты. Третья строка содержит текст «No data» (Нет данных), потому что в ассоциативном массиве нет персонажа Гуфи (Goofy). Очевидно, что нужен какой-то способ, который поможет отличить эти две ситуации. К сожалению, ни типы с поддержкой `null`, ни класс `Option` не способны помочь в этом. В главе 7 я покажу, как решить эту проблему.

Упражнение 6.7

Реализуйте функцию-значение `variance` в терминах `flatMap`, вычисляющую дисперсию. Дисперсия массива значений помогает представить, как эти значения распределены вокруг среднего значения. Если все значения близки к среднему, дисперсия низкая. Нулевая дисперсия получается, когда все значения равны среднему. Дисперсия вычисляется как среднее значений `Math.pow(x - m, 2)`, вычисляемых для каждого элемента `x` в массиве, где `m` – среднее значение в массиве. Функция должна быть реализована на уровне пакета (вне класса `Option`). Вот ее сигнатура:

```
val variance: (List<Double>) -> Option<Double>
```

Подсказка

Прежде чем взяться за эту функцию, напишите сначала функцию `mean`. Если у вас возникнут затруднения, обращайтесь к главе 5 или используйте следующую версию:

```
val mean: (List<Double>) -> Option<Double> = { list ->
    when {
        list.isEmpty() -> Option()
        else -> Option(list.sum() / list.size)
    }
}
```

Решение

Написав функцию `mean`, вы с легкостью напишете `variance`:

```
val variance: (List<Double>) -> Option<Double> = { list ->
    mean(list).flatMap { m ->
        mean(list.map { x ->
            Math.pow((x - m), 2.0)
        })
    }
}
```

Условие, требующее реализовать функции как функции-значения, не является обязательным, но тогда вы не сможете использовать их для передачи в аргументах функциям высшего порядка. Однако в случаях непосредственного использования функции fun намного удобнее. Если вы предпочитаете пользоваться функциями fun, когда это возможно, используйте следующее решение:

```
fun mean(list: List<Double>): Option<Double> =
    when {
        list.isEmpty() -> Option()
        else -> Option(list.sum() / list.size)
    }

fun variance(list: List<Double>): Option<Double> =
    mean(list).flatMap { m ->
        mean(list.map { x ->
            Math.pow((x - m), 2.0)
        })
    }
```

Как видите, функции fun проще в использовании, потому что имеют более простые типы. По этой причине рекомендуется всегда использовать функции fun вместо функций-значений, если это возможно. Кроме того, между ними легко переключаться. Представьте, что есть некая функция

```
fun aToBfunction(a: A): B {
    return ...
}
```

Эквивалентную ей функцию-значение можно записать так:

```
val aToBfunction: (A) -> B = { a -> aToBfunction(a) }
```

Или, если использовать ссылку на функцию:

```
val aToBfunction: (A) -> B = ::aToBfunction
```

И наоборот, функцию fun можно определить в терминах функции-значения:

```
fun aToBfunction(a: A): B = aToBfunction(a)
```

Как показывает реализация variance, применение функции flatMap позволяет конструировать вычисления, состоящие из множества этапов, каждый из которых может завершиться неудачей. Вся цепочка вычислений будет прервана при первой же встретившейся ошибке, потому что None.flatMap(f) сразу возвращает None, не пытаясь применить f.

6.4.5 Другие способы комбинирования типа Options

Решение об использовании типа Option может иметь значительные последствия. В частности, некоторые разработчики считают, что их прежний код от этого устареет. Что можно сделать, если нужна функция типа (Option<A>) -> Option<B>, но в данный момент есть только функции типа

(A) -> B? Неужели придется переписать все библиотеки? Конечно, нет! Вы можете просто адаптировать эти функции.

Упражнение 6.8

Определите функцию lift, которая принимает функцию типа (A) -> B и возвращает функцию (Option<A>) -> Option<B>. Используйте для этого функции, которые уже определили. На рис. 6.2 показано, как работает функция lift.

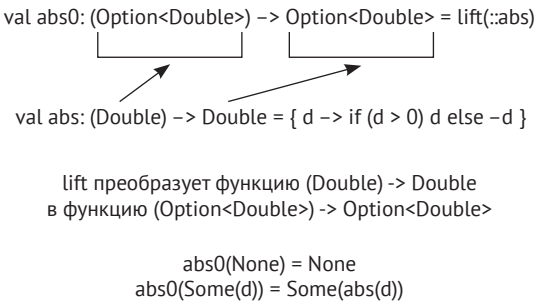


Рис. 6.2 Функция lift

Подсказка

Создайте функцию fun на уровне пакета и используйте в ней функцию map.

Решение

Решение выглядит очень просто:

```
fun <A, B> lift(f: (A) -> B): (Option<A>) -> Option<B> = { it.map(f) }
```

Большинство ваших библиотек будет содержать только функции fun. Преобразовать функцию fun, которая принимает аргумент типа A и возвращает значение типа B, в функцию (Option<A>) -> Option<B> легко. Например, вот как можно преобразовать функцию String.toUpperCase:

```
val upperOption: (Option<String>) -> Option<String> =
                                  lift { it.toUpperCase() }
```

Также можно использовать ссылку на функцию:

```
val upperOption: (Option<String>) -> Option<String> =
                           lift(String::toUpperCase)
```

Упражнение 6.9

Предыдущую функцию lift бессмысленно применять к функциям, генерирующим исключения. Напишите версию lift, которая может работать с функциями, генерирующими исключения.

Решение

Чтобы решить это упражнение, нужно завернуть реализацию функции, возвращаемой функцией lift, в блок try...catch, возвращающий None в случае исключения:

```
fun <A, B> lift(f: (A) -> B): (Option<A>) -> Option<B> = {
    try {
        it.map(f)
    } catch (e: Exception) {
        Option()
    }
}
```

Иногда может также понадобиться преобразовать функцию `(A) -> B` в функцию `(A) -> Option<B>`. Для этого можно использовать тот же прием:

```
fun <A, B> hLift(f: (A) -> B): (A) -> Option<B> = {
    try {
        Option(it).map(f)
    } catch (e: Exception) {
        Option()
    }
}
```

Обратите внимание, что это не самый лучший способ, потому что он теряет исключение. В главе 7 я покажу, как решить эту проблему.

А как быть с функциями, принимающими два аргумента? Допустим, вы хотите вызвать Java-метод `Integer.parseInt(String s, int radix)` и передать ему аргументы `Option<String>` и `Option<Integer>`. Как это сделать? Сначала из этого Java-метода нужно создать функцию-значение:

```
val parseWithRadix: (Int) -> (String) -> Int =
        { radix -> { string -> Integer.parseInt(string, radix) } }
```

Здесь я поставил аргументы в обратном порядке и создал каррированную функцию. В этом есть определенный смысл, потому что перестановка аргументов производит (путем применения только первого аргумента) функцию, которая позволяет выполнять *парсинг с заданным основанием любых строк*:

```
val parseHex: (String) -> Int = parseWithRadix(16)
```

Без перестановки аргументов получится функция, которая выполняет *парсинг заданной строки с любым основанием*. Этот второй вариант функции менее полезен, хотя это зависит от вашего конкретного случая использования.

Упражнение 6.10

Напишите функцию `map2`, которая принимает аргументы `Option<A>` и `Option<B>` и функцию `(A, B) -> C` в каррированной форме и возвращает `Option<C>`.

Подсказка

Используйте функцию `flatMap` и, если посчитаете нужным, функцию `map`.

Решение

Вот решение, использующее обе функции – `flatMap` и `map`. Для вас важно понять суть этого шаблона, потому что он часто будет встречаться вам в вашей практике. Мы еще вернемся к нему в главе 8:

```
fun <A, B, C> map2(oa: Option<A>,
                   ob: Option<B>,
                   f: (A) -> (B) -> C): Option<C> =
                           oa.flatMap { a -> ob.map { b -> f(a)(b) } }
```

С помощью `map2` вы теперь сможете использовать любые функции двух аргументов, как если бы они изначально предназначались для работы с типом `Option`.

А как быть с функциями, принимающими большее число аргументов? Взгляните на пример функции `map3`:

```
fun <A, B, C, D> map3(oa: Option<A>,
                      ob: Option<B>,
                      oc: Option<C>,
                      f: (A) -> (B) -> (C) -> D): Option<D> =
            oa.flatMap { a ->
                ob.flatMap { b ->
                    oc.map { c ->
                        f(a)(b)(c)
                    }
                }
            }
```

Заметили общий шаблон? (Последней здесь используется `map`, а не `flatMap`, только потому, что `f` возвращает простое значение. Если бы она возвращала `Option`, вы могли бы использовать тот же шаблон, заменив `map` на `flatMap`.)

6.4.6 Комбинирование List и Option

Как вы наверняка понимаете, простой возможности сконструировать экземпляр `Option` будет достаточно не всегда. В какой-то момент вам понадобится, чтобы любой новый тип, который вы определите, был совместим с любыми другими типами. В предыдущей главе мы определили тип `List`, и для создания полезных программ нам нужно научиться сочетать типы `List` и `Option`.

Чаще других будет встречаться операция преобразования `List<Option<A>>` в `Option<List<A>>`. Тип `List<Option<A>>` получается отображением типа `List<B>` с функцией `(B) -> Option<A>`. Обычно в результате должно получиться значение `Some<List<A>>`, если все элементы имеют значение `Some<A>`, и `None`, если хотя бы один элемент является значением `None<A>`. Это единственный возможный результат. Иногда можно игнорировать результаты `None` и получить список `List<A>`, но это совсем другой случай использования.

Упражнение 6.11

Напишите функцию `sequence` на уровне пакета, которая преобразует `List<Option<A>>` в `Option<List<A>>`. Она должна возвращать `Some<List<A>>`, если все значения в исходном списке являются экземплярами `Some`, или `None<List<A>>`, если в списке присутствует хотя бы одно значение `None`. Вот ее сигнатура:

```
fun <A> sequence(list: List<Option<A>>): Option<List<A>>
```

Обратите внимание, что здесь должен использоваться класс `List` из главы 5, а не встроенный в Kotlin класс `List`.

Подсказка

Проверьте список и выясните, не пустой ли он, и, если не пустой, выполните рекурсивный вызов `sequence`. Не забывайте также об абстракциях рекурсии `foldRight` и `foldLeft` – вы можете использовать любую из них для реализации `sequence`.

Решение

Вот версия, явно использующая рекурсию, которую можно было бы использовать при наличии общедоступных функций `list.head()` и `list.tail()` в классе `List` (в действительности этих функций нет, поэтому данное решение не компилируется:

```
fun <A> sequence(list: List<Option<A>>): Option<List<A>> {
    return if (list.isEmpty())
        Option(List())
    else
        list.head().flatMap({ hh ->
            sequence(list.tail()).map({ x -> x.cons(hh) }) })
}
```

В свое время мы отказались от реализации функций `list.head()` и `list.tail()`, потому что они могут генерировать исключения при вызове для пустого списка. Сейчас же мы можем написать эти функции так, чтобы они возвращали `Option`. К счастью, функцию `sequence` можно также реализовать с использованием `foldRight` и `map2`:

```
fun <A> sequence(list: List<Option<A>>): Option<List<A>> =
        list.foldRight(Option(List())) { x ->
            { y: Option<List<A>> -> map2(x, y) { a ->
                { b: List<A> -> b.cons(a) } }
            }
        }
```

Обратите внимание, что, к сожалению, Kotlin не может автоматически определить типы параметров `y` и `b`. (В этом Kotlin уступает Java!) Теперь рассмотрим следующий пример:

```
val parseWithRadix: (Int) -> (String) -> Int = { radix ->
    { string ->
        Integer.parseInt(string, radix)
    }
}
val parse16 = hLift(parseWithRadix(16))
val list = List("4", "5", "6", "7", "8", "9", "A", "B")
val result = sequence(list.map(parse16))
println(result)
```

Это решение дает желаемый результат, но оно далеко не оптимально, потому что обе функции, `map` и `sequence`, вызывают `foldRight`.

Упражнение 6.12

Определите функцию `traverse`, которая дает тот же результат, но вызывает `foldRight` только один раз. Вот ее сигнатура:

```
fun <A, B> traverse(list: List<A> , f: (A) -> Option<B>): Option<List<B>>
```

Затем реализуйте `sequence` в терминах `traverse`.

Подсказка

Вместо явной рекурсии используйте функцию `foldRight`.

Решение

Сначала определим функцию `traverse`:

```
fun <A, B> traverse(list: List<A> , f: (A) -> Option<B>): Option<List<B>> =
    list.foldRight(Option(List())) { x ->
        { y: Option<List<B>> ->
            map2(f(x), y) { a ->
                { b: List<B> ->
                    b.cons(a)
                }
            }
        }
    }
```

Затем переопределим `sequence` в терминах `traverse`:

```
fun <A> sequence(list: List<Option<A>>): Option<List<A>> =
                            traverse(list) { x -> x }
```

6.4.7 Когда и как использовать тип Option

Как я уже говорил в главе 2 и в начале этой главы, Kotlin поддерживает другой способ обработки необязательных данных. Возникает вопрос: какой подход лучше использовать – типы с поддержкой `null` или `Option`:

1. Типы с поддержкой `null` удобны для внутреннего использования в некоторых функциях, таких как генераторы, когда возвращаемое значение `null` указывает на достижение условия завершения. Но такие значения `null` не должны покидать пределы функций; они всегда должны использоваться в строго ограниченной области видимости и могут возвращаться только локальными функциями.
2. Тип `Option` лучше всего подходит для действительно необязательных данных. Но обычно отсутствие данных является результатом ошибок, которые традиционные программисты предпочитают обрабатывать с помощью исключений. Возврат `None` вместо исключения – это все равно что перехватить исключение и молча проглотить его. Возможно, это не ошибка на миллиард долларов, но это все еще большая ошибка. В главе 7 вы узнаете, как исправить эту ситуацию. После этого вам больше никогда не понадобится тип `Option`. Но не волнуйтесь; все, что вы узнали в этой главе, все равно будет чрезвычайно полезно.

Тип `Option` – это самая простая форма представления данных, которую вы будете использовать снова и снова. Это параметризуемый тип, и он поддерживает функцию для создания `Option<A>` из `A`. Тип `Option` также имеет функцию `flatMap`, которую можно использовать для создания экземпляров `Option`. Хотя сам по себе он не особенно полезен, но на его примере мы познакомились с фундаментальным понятием, которое называется *монадой* (monad). Тип `List` имеет те же характеристики. Особенно важно, что оба эти класса имеют много общего. Пусть вас не пугает термин «монада»; в нем нет ничего страшного. Рассматривайте его как шаблон проектирования (хотя в действительности это гораздо больше, чем шаблон).

Итоги

- Нам нужен способ представления необязательных данных, т. е. данных, которые могут отсутствовать. Подтип `Some` представляет присутствующие данные, а подтип `None` – отсутствующие.
- Для представления необязательных данных Kotlin использует типы с поддержкой `null`. Они защищают от исключения `NullPointerException` при использовании типов, не поддерживающих `null`, и вынуждают явно обрабатывать значение `null` при их использовании.
- Пустые (`null`) ссылки – самый непрактичный и опасный способ представления отсутствующих данных. Сигнальные значения и пустые списки – еще два способа представления отсутствующих данных, но они плохо сочетаются.
- Тип `Option` предлагает лучший способ представления необязательных данных.
- Функции высшего порядка `map` и `flatMap` позволяют применять другие функции к значениям типа `Option`. Функции, работающие с фактическими значениями, легко адаптировать для работы с экземплярами `Option`.
- Тип `List` можно объединить с типом `Option`. Экземпляр `List<Option>` можно преобразовать в экземпляр `Option<List>` с помощью функции `sequence`.
- Экземпляры `Option` можно сравнивать. Экземпляры подтипа `Some` равны, если равны фактические значения, завернутые в них. Так как в программе существует только один экземпляр `None`, все ссылки `None` считаются равными.
- Тип `Option` способен представить результат вычислений, в ходе которых может генерироваться исключение, но при этом информация об исключении теряется.

7 Обработка ошибок и исключений

Эта глава охватывает следующие темы:

- сохранение информации об ошибке в типе `Either`;
- обработка ошибок с помощью типа `Result`;
- доступ к данным в типе `Result`;
- адаптация функций для работы с типом `Result`.

В главе 6 вы узнали, как обращаться с необязательными данными без использования пустых ссылок с применением типа данных `Option`. Этот тип данных, как вы видели, идеально подходит для решения проблемы отсутствия данных, когда это не является результатом ошибки. Но это не самый эффективный способ обработки ошибок, потому что, хотя он и позволяет сообщить об отсутствии данных, с его помощью нельзя узнать причину отсутствия. Все отсутствующие данные обрабатываются одинаково, и вызывающий код должен сам попытаться выяснить, что именно произошло. Однако часто это невозможно.

В этой главе мы решим различные упражнения, которые научат обрабатывать ошибки и исключения в Kotlin. Один из важнейших навыков, который вы приобретете, – представление отсутствия данных из-за ошибки, чего не позволяет добиться тип `Option`. Сначала перечислим проблемы, которые предстоит решить, а затем рассмотрим типы `Either` и `Result`:

- тип `Either` удобно использовать для функций, которые могут возвращать значения двух разных типов;

- тип `Result` удобно использовать, когда требуется вернуть данные или сообщить об ошибке.

После обсуждения типов `Either` и `Result` и нескольких упражнений их использования мы рассмотрим дополнительные приемы работы с типом `Result`.

7.1 Проблемы с отсутствующими данными

В большинстве случаев отсутствие данных является результатом ошибки во входных данных или в вычислениях. Это два совершенно разных случая, но они приводят к одному и тому же результату: отсутствию данных.

В традиционном программировании, когда функция или метод принимает параметр с объектом, программисты знают, что должны проверить этот параметр на равенство `null`. Но что следует делать, если параметр действительно имеет значение `null`, часто не определено.

Вспомним пример из листинга 6.2 в главе 6:

```
val goofy = toons.getOption("Goofy").flatMap { it.email }

println(goofy.getOrElse({ "No data" }))
```

Этот пример выводил строку «No data» (Нет данных) из-за того, что ключ `Goofy` отсутствовал в ассоциативном массиве. Это можно было бы считать рядовой ситуацией. Но взгляните на следующий код:


```
    val toon = getName()
            .flatMap(toons::getOption)
            .flatMap(Toon::email)

    println(toon.getOrElse{"No data"})
}

fun getName(): Option<String> = ???
```

Что следует сделать, если пользователь введет пустую строку? Очевидное решение: проверить ввод и вернуть `Option<String>`. При отсутствии допустимой строки можно вернуть `None`. Кроме того, такая операция может вызвать исключение. Программа могла бы выглядеть так:

```
    val toon = getName()
        .flatMap(toons::getOption)
        .flatMap(Toon::email)

    println(toon.getOrElse{"No data"})
}

fun getName(): Option<String> = try {
    validate(readLine())
} catch (e: IOException) {
    Option()
}

fun validate(name: String?): Option<String> = when {
```

```
    name?.isNotEmpty() ?: false -> Option(name)
    else -> Option()
}
```

Теперь поразмышляем, что может случиться, если выполнить такой код:

- он благополучно выполнится и выведет адрес электронной почты;
- возникнет исключение `IOException`, и в консоли появится сообщение «No data»;
- пользователь ввел недопустимое имя, и в консоли появится сообщение «No data»;
- пользователь ввел допустимое имя, но оно отсутствует в массиве; в консоли появится сообщение «No data»;
- имя присутствует в массиве, но у персонажа с таким именем нет электронной почты; в консоли появится сообщение «No data».

Нам нужно вывести в консоль разные сообщения, чтобы указать, что произошло в каждом конкретном случае. Если ограничиться только известными нам типами, можно использовать `Pair<Option<T>, Option<String>>` в качестве типа возвращаемого значения каждой функции, но это сложно. `Pair` – это *тип-произведение*, т. е. число элементов, которые можно представить типом `Pair<T, U>`, равно произведению числа возможных значений типа `T` на число возможных значений типа `U`. Нам этого совсем не нужно, потому что для каждого существующего значения `T` мы будем иметь `None` для `U`.

Точно так же для каждого существующего значения `U` мы будем иметь `None` для `T`. Нам нужен *тип-сумма*, т. е. такой тип `E<T, U>`, который будет содержать либо `T`, либо `U`, но не `T` и `U` одновременно. Такой тип называется *типом-суммой*, потому что число возможных реализаций является суммой числа возможных реализаций `T` и `U`. *Тип-произведение*, такой как `Pair<T, U>`, напротив, имеет число возможных реализаций, которое является произведением числа возможных реализаций `T` и `U`.

7.2 Тип Either

Для случаев, когда функция может вернуть значение одного из двух разных типов, например данные или ошибку, мы будем использовать специальный тип: тип `Either`. Определить тип, который может содержать `A` или `B`, легко. Нужно лишь немного модифицировать тип `Option`, изменив тип `None`, чтобы он мог содержать значение. Также нужно изменить имена подтипов. Два приватных подкласса внутри типа `Either` мы назовем `Left` и `Right`, как показано в листинге 7.1.

Листинг 7.1 Тип Either

```
sealed class Either<out A, out B> {

    internal
    class Left<out A, out B>(private val value: A): Either<A, B>() {
```

```
        override fun toString(): String = "Left($value)"
    }

    internal
    class Right<out A, out B>(private val value: B) : Either<A, B>() {
        override fun toString(): String = "Right($value)"
    }

    companion object {
        fun <A, B> left(value: A): Either<A, B> = Left(value)
        fun <A, B> right(value: B): Either<A, B> = Right(value)
    }
}
```

Теперь мы легко можем заменить тип `Option` на тип `Either` для представления значений, которые могут отсутствовать из-за ошибки. Для этого достаточно параметризовать `Either` типом данных и типом ошибки.

В соответствии с соглашениями подкласс `Right` используется для представления успеха (т. е. right – верный), а подкласс `Left` – для представления ошибки. Мы не будем использовать имя `Wrong` (неверный) для подкласса, потому что тип `Either` может использоваться для хранения данных того или иного типа, оба из которых действительные.

Давайте выберем тип для представления ошибки. Можно выбрать тип `String`, чтобы иметь возможность передать сообщение об ошибке, или тип какого-нибудь исключения. Например, функцию `max`, возвращающую максимальное значение в списке, которую мы написали в главе 6, можно изменить следующим образом:

```
fun <A: Comparable<A>> max(list: List<A>): Either<String, A> =
when(list) {
    is List.Nil -> Either.left("max called on an empty list")
    is List.Cons -> Either.right(list.foldLeft(list.head) { x -> { y ->
            if (x.compareTo(y) == 0) x else y
        }
    })
}
```

Чтобы тип `Either` имел практическую пользу, нам нужен способ композиции функций, использующих его, например когда результат функции, возвращающей `Either`, передается в другую функцию, возвращающую другой экземпляр `Either`. Для поддержки композиции функций, возвращающих `Either`, нужно определить те же функции, которые мы определили в классе `Option`.

Упражнение 7.1

Определите функцию `map`, преобразующую (или отображающую) `Either<E, A>` в `Either<E, B>` с использованием заданной функции `(A) -> B`. Вот как выглядит сигнатура такой функции `map`:

```
abstract fun <B> map(f: (A) -> B): Either<E, B>
```

Подсказка

Имена `E` и `A` для параметров выбраны так, чтобы было проще понять, какая из сторон отображается. Имя `E` – это сокращение от слова *error* (ошибка). Точно также можно было бы определить две функции `map` (например, с именами `mapLeft` и `mapRight`) для преобразования одной или другой стороны экземпляра `Either`. Мы создадим смещенную версию `Either`, поддерживающую отображение только одной стороны.

Решение

Реализация `Left` немного сложнее реализации `None` в `Option`, потому что требуется сконструировать новый экземпляр `Either`, хранящий то же значение ошибки, что и оригинал:

```
override fun <B> map(f: (A) -> B): Either<E, B> = Left(value)
```

Реализация в `Right` в точности повторяет реализацию в `Some`:

```
override fun <B> map(f: (A) -> B): Either<E, B> = Right(f(value))
```

Упражнение 7.2

Определите функцию `flatMap` для преобразования `Either<E, A>` в `Either<E, B>`, которая принимает функцию `(A) -> Either<E, B>`. Вот как выглядит сигнатура функции `flatMap`:

```
abstract fun <B> flatMap(f: (A) -> Either<E, B>): Either<E, B>
```

Решение

Реализация в `Left` в точности повторяет реализацию функции `map`:

```
override fun <B> flatMap(f: (A) -> Either<E, B>): Either<E, B> =
    Left(value)
```

Реализация в `Right` повторяет реализацию `Option.flatMap`:

```
override fun <B> flatMap(f: (A) -> Either<E, B>): Either<E, B> =
    f(value)
```

Обратите внимание, что параметр `E` оставлен инвариантным. Нам не нужно заботиться о вариантности, потому что мы вскоре избавимся от этого параметра.

Упражнение 7.3

Определите функции `getOrElse` и `orElse` со следующими сигнатурами:

```
fun getOrElse(defaultValue: () -> @UnsafeVariance A): A

fun orElse(defaultValue: () -> Either<E, @UnsafeVariance A>): Either<E, A>
```

Подсказка

Помните о вариантности!

Решение

Чтобы отключить проверку вариантности, типы аргументов в обеих функциях следует снабдить аннотацией `@UnsafeVariance`. Функция

getOrElse возвращает содержащееся значение, если this – это экземпляр Right, или результат вызова функции defaultValue в противном случае. В подклассе Right мы должны заменить спецификатор private доступа к свойству value спецификатором internal, чтобы разрешить доступ из реализации функции getOrElse в суперклассе:

```
fun getOrElse(defaultValue: () -> @UnsafeVariance A): A = when (this) {
    is Right -> this.value
    is Left -> defaultValue()
}
```

Функция orElse будет вызывать константную функцию map, возвращающую this, и затем применять getOrElse к результату:

```
fun orElse(defaultValue: () -> Either<E, @UnsafeVariance A>): Either<E, A> =
    map { this }.getOrElse(defaultValue)
```

Класс Either справляется со своей задачей, но он далек от идеала. Проблема в том, что мы не получаем никакой дополнительной информации в случае отсутствия значения. Нам возвращается значение по умолчанию, и мы не можем определить, является ли оно результатом вычисления или результатом ошибки. Для правильной обработки ошибок нам нужна смещенная версия Either, где левый тип известен. Вместо класса Either (который, кстати, имеет много других интересных вариантов использования) можно создать специализированную версию с известным фиксированным типом для класса Left.

Первый вопрос, который может у вас возникнуть: «Какой тип выбрать?» На ум приходят два разных типа: String и RuntimeException. Строка может содержать сообщение об ошибке, так же как исключение, но многие ошибочные ситуации порождают исключение. Использование String в качестве типа, передаваемого значением Left, заставит вас игнорировать информацию в исключении и использовать только текст сообщения. Поэтому в качестве значения Left лучше использовать RuntimeException. А если у вас есть только текстовое сообщение, вы легко сможете превратить его в исключение.

7.3 Тип Result

В действительности нам нужен тип, представляющий данные или ошибку. Так как этот тип обычно представляет результат вычислений, которые могут потерпеть неудачу, назовем его Result. Он аналогичен типу Option, кроме того, что подклассы называются Success и Failure, как показано в листинге 7.2. Этот класс очень похож на класс Option и дополнительно сохраняет исключение.

Листинг 7.2 Класс Result

```
import java.io.Serializable

sealed class Result<out A>: Serializable { ①

    internal ②
```

```
    class Failure<out A>(
        internal val exception: RuntimeException): Result<A>() { ③

        override fun toString(): String =
            "Failure(${exception.message})"
    }

    internal ②
    class Success<out A>(internal val value: A) : Result<A>() {

        override fun toString(): String = "Success($value)"
    }

    companion object {

        operator fun <A> invoke(a: A? = null): Result<A> = when (a) {
            null -> Failure(NullPointerException()) ⑥
            else -> Success(a)
        }

        fun <A> failure(message: String): Result<A> =
            Failure(IllegalStateException(message)) ⑦

        fun <A> failure(exception: RuntimeException): Result<A> =
            Failure(exception) ⑧

        fun <A> failure(exception: Exception): Result<A> =
            Failure(IllegalStateException(exception)) ⑨
    }
}
```

① Класс `Result` имеет только один параметр типа, соответствующий значению успеха
② Конструкторы объявлены внутренними
③ Подкласс `Failure` содержит поле типа `RuntimeException`
⑥ Если `Result` конструируется со значением `null`, немедленно определяется ошибка
⑦ Если экземпляр `Failure` создается с сообщением, это сообщение обертывается исключением `RuntimeException` (точнее, его подклассом `IllegalStateException`)
⑧ Если экземпляр `Failure` создается с исключением `RuntimeException`, исключение сохраняется как есть
⑨ Если экземпляр `Failure` создается с контролируемым исключением, он обертывается исключением `RuntimeException`

Для полноценного использования `Result` необходимы те же функции, что уже имеются в классах `Option` и `Either`, но немного отличающиеся логикой работы.

Упражнение 7.4

Определите функции `map`, `flatMap`, `getOrElse` и `orElse` в классе `Result`. Для `getOrElse` можно определить две функции: одну – принимающую значение, и другую – принимающую функцию, которая возвращает значение по умолчанию. Вот их сигнатуры:

```
fun <B> map(f: (A) -> B): Result<B>
fun <B> flatMap(f: (A) -> Result<B>): Result<B>
fun <A> getOrElse(defaultValue: A): A
fun <A> orElse(defaultValue: () -> Result<A>): Result<A>
```

Подсказка

Не забудьте предусмотреть обработку любых исключений, которые могут генерировать ваши реализации, и позаботьтесь о вариантности.

Решение

Все перечисленные функции напоминают одноименные функции в классе `Either`. Вот реализации `map` и `flatMap` в классе `Success`:

```
override fun <B> map(f: (A) -> B): Result<B> = try {
    Success(f(value))
} catch (e: RuntimeException) {
    Failure(e)
} catch (e: Exception) {
    Failure(RuntimeException(e))
}

override fun <B> flatMap(f: (A) -> Result<B>): Result<B> = try {
    f(value)
} catch (e: RuntimeException) {
    Failure(e)
} catch (e: Exception) {
    Failure(RuntimeException(e))
}
```

А вот реализации в классе `Failure`:

```
override fun <B> map(f: (A) -> B): Result<B> =
    Failure(exception)

override fun <B> flatMap(f: (A) -> Result<B>): Result<B> =
    Failure(exception)
```

Функция `getOrElse` может пригодиться, когда значение по умолчанию определяется как литерал, который не требуется вычислять. В этом случае нет необходимости использовать отложенные вычисления. Чтобы избежать проблем с вариантностью, следует добавить аннотацию: `@UnsafeVariance`:

```
fun getOrElse(defaultValue: @UnsafeVariance A): A = when (this) {
    is Success -> this.value
    is Failure -> defaultValue
}
```

Функция `orElse` используется, когда значение по умолчанию – вычисляемое. Так как в ходе вычислений может возникнуть исключение, необходимо предусмотреть его обработку, как показано ниже:

```
fun orElse(defaultValue: () -> Result<@UnsafeVariance A>): Result<A> =
    when (this) {
        is Success -> this
        is Failure -> try {
            defaultValue()
    } catch (e: RuntimeException) {
        Result.failure<A>(e)
```

```
    } catch (e: Exception) {
        Result.failure<A>(RuntimeException(e))
    }
}
```

Необходимость обработки исключений зависит от реализаций используемых функций. Если вы будете использовать только свои реализации, вам может не потребоваться перехватывать исключения при условии, что вы никогда не будете возбуждать их. Но если вы будете использовать функции из стандартной библиотеки (что вполне вероятно), следует предусмотреть обработку исключений, применяя главный принцип безопасности: всегда перехватывать и никогда не возбуждать. Также обратите внимание, что у вас может возникнуть желание определить следующую функцию по аналогии с `Option`:

```
fun getOrElse(defaultValue: () -> A): A
```

Однако эта реализация не дает абсолютной уверенности, что она никогда не возбудит исключение. Подумайте, что нужно вернуть, если константная функция `() -> A` возбудит исключение?

7.4 Приемы использования типа Result

После добавления функций появляется возможность использовать класс `Result` для безопасной композиции функций, представляющих вычисления, которые либо выполняются, либо завершаются неудачей. Это важно, потому что `Result` и подобные типы часто рассматриваются как контейнеры, которые могут содержать или не содержать значение, что в принципе неверно.

Тип `Result` – это вычислительный контекст для значения, которое может присутствовать или отсутствовать. Он используется не путем извлечения значения, а путем использования экземпляров `Result` с применением его функций. Например, давайте изменим пример `ToonMail`, чтобы продемонстрировать использование этого класса. Сначала определим специальную функцию-расширение `get` для `Map`, которая возвращает `Result.Failure`, если ключ отсутствует в ассоциативном массиве, как показано в листинге 7.3. Назовем эту новую функцию `getResult`.

Листинг 7.3 Новая функция Map.getResult, возвращающая Result

```
fun <K, V> Map<K, V>.getResult(key: K) = when {
    this.containsKey(key) -> Result(this[key]) ①
    else -> Result.failure("Key $key not found in map") ②
}
```

① Если ключ присутствует в ассоциативном массиве, вернуть `Success` с извлеченным объектом
② Иначе вернуть `Failure` с сообщением об ошибке

Затем изменим класс `Toon`, как показано в листинге 7.4.

Листинг 7.4 Класс Toon с измененным свойством email

```
data
class Toon private constructor (val firstName: String, ①
        val lastName: String,
        val email: Result<String>) { ②

    companion object {
        operator fun invoke(firstName: String, ③
                            lastName: String) =
            Toon(firstName, lastName, ④
                 Result.failure("$firstName $lastName has no mail"))

        operator fun invoke(firstName: String, ③
                            lastName: String,
                            email: String) =
            Toon(firstName, lastName, Result(email)) ⑥
    }
}
```

① Конструктор объявлен приватным
② Свойство email теперь имеет тип Result (т. е. может быть экземпляром Success или Failure)
③ Перегруженная функция invoke
④ Если адрес электронной почты отсутствует, в качестве значения по умолчанию использовать Result.Failure
⑥ Если объект содержит адрес электронной почты, он обертывается классом Result

Теперь изменим программу ToonMail, как показано в листинге 7.5.

Листинг 7.5 Измененная программа, использующая тип Result

```
fun main(args: Array<String>) {

    val toons: Map<String, Toon> = mapOf(
        "Mickey" to Toon("Mickey", "Mouse", "mickey@disney.com"),
        "Minnie" to Toon("Minnie", "Mouse"),
        "Donald" to Toon("Donald", "Duck", "donald@disney.com"))

    val toon = getName()
        .flatMap(toons::getResult) ①
        .flatMap(Toon::email)

    println(toon)
}

fun getName(): Result<String> = try { ②
    validate(readLine())
} catch (e: IOException) {
    Result.failure(e)
}

fun validate(name: String?): Result<String> = when {
    name?.isNotEmpty() ?: false -> Result(name)
    else -> Result.failure("Invalid name $name")
}
```

① Методы, возвращающие Result, объединяются через flatMap
② Функция getName позволяет ввести имя с клавиатуры и возвращает Failure, если имя отсутствует в массиве или было возбуждено исключение

При желании можно изменить функцию getName, чтобы она возвращала возникшее исключение, обернутое экземпляром Failure.

Обратите внимание, как компонуются разные операции, возвращающие Result. Нам нет необходимости извлекать значение из Result, которое может оказаться исключением. Для составления таких цепочек используется функция flatMap. Попробуем выполнить программу с разными входными значениями, например:

```
"Mickey"
"Minnie"
"Goofy"
```

пустое значение (просто нажмите клавишу **Enter**).

Вот что выведет программа в каждом из перечисленных случаев:

```
Success(mickey@disney.com)
Failure(Minnie Mouse has no mail)
Failure(Key Goofy not found in map)
Failure(Invalid name )
```

Как будто бы все хорошо, но это не так. Дело в том, что для Minnie (не имеет электронной почты) и Goofy (отсутствует в массиве) программа сообщила об ошибке. В действительности эти случаи могут быть вполне законными. В конце концов, если отсутствие адреса электронной почты рассматривать как ошибку, тогда мы должны запретить создавать экземпляры Toon без него.

Очевидно, что это не ошибка, а всего лишь отсутствие необязательных данных. То же относится и к самому ассоциативному массиву. Отсутствие ключа может быть ошибкой (при условии, что он должен быть там), но с точки зрения массива ключи – это необязательные данные. Может показаться, что это не проблема, потому что у нас уже есть тип Option (который вы разработали в главе 6). Но посмотрите, как мы компонуем свои функции:

```
toon = getName()
        .flatMap(toons::getResult)
        .flatMap(Toon::email)
```

Такой способ возможен только потому, что getName, Map.getResult и Toon.email возвращают Result. Если Map.getResult и Toon.email будут возвращать Option, их нельзя будет скомпоновать с getName. Однако у нас есть возможность преобразования между типами Result и Option. Например, в класс Result можно добавить функцию toOption:

```
abstract fun toOption(): Option<A>
```

Тогда в подклассе Success она будет иметь реализацию:

```
override fun toOption(): Option<A> = Option(value)
```

а в подклассе Failure:

```
override fun toOption(): Option<A> = Option()
```

Вот как можно было бы использовать такую функцию:

```
Option<String> result =
    getName().toOption().flatMap(toons::getResult).flatMap(Toon::emmail)
```

Но тогда мы потеряем все преимущества использования `Result`! Если теперь внутри функции `getName` возникнет исключение, оно будет потеряно после преобразования `Failure` в функции `toOption` и программа просто ничего не выведет.

Кто-то из вас мог бы подумать, что следует пойти другим путем и преобразовать `Option` в `Result`. Этот прием будет работать (хотя в нашем примере придется вызывать новую функцию `toResult` в обоих экземплярах `Option`, возвращаемых функциями `Map.get` и `Toon.getMail`), но это утомительно. Поскольку при таком подходе часто придется преобразовывать `Option` в `Result`, лучше это преобразование встроить в класс `Result`. Для этого достаточно добавить новый подкласс, соответствующий случаю `None`. Случай `Some` не нуждается в преобразовании, потому что это тот же самый `Success`. В листинге 7.6 показан новый класс `Result` с новым подклассом `Empty`.

Листинг 7.6 Новый класс Result, обрабатывающий ошибки и необязательные данные

```
sealed class Result<out A>: Serializable {

    abstract fun <B> map(f: (A) -> B): Result<B>

    abstract fun <B> flatMap(f: (A) -> Result<B>): Result<B>

    fun getOrElse(defaultValue: @UnsafeVariance A): A = when (this) {
        is Result.Success -> this.value
        else -> defaultValue ①
    }

    fun getOrElse(defaultValue: () -> @UnsafeVariance A): A = when (this) {
        is Result.Success -> this.value
        else -> defaultValue() ①
    }

    fun orElse(defaultValue: () -> Result<@UnsafeVariance A>): Result<A> =
        when (this) {
            is Success -> this
            else -> try { ①
                defaultValue()
            } catch (e: RuntimeException) {
                Result.failure<A>(e)
            } catch (e: Exception) {
                Result.failure<A>(RuntimeException(e))
            }
        }

    internal object Empty: Result<Nothing>() { ④

        override fun <B> map(f: (Nothing) -> B): Result<B> = Empty

        override fun <B> flatMap(f: (Nothing) -> Result<B>): Result<B> = Empty
```

```
        override fun toString(): String = "Empty"
    }

    internal
    class Failure<out A>(private val exception: RuntimeException): Result<A>()
    {
        override fun <B> map(f: (A) -> B): Result<B> = Failure(exception)

        override fun <B> flatMap(f: (A) -> Result<B>): Result<B> =
            Failure(exception)

        override fun toString(): String =
            "Failure(${exception.message})"
    }

    internal class Success<out A>(internal val value: A) : Result<A>()
    {
        override fun <B> map(f: (A) -> B): Result<B> = try {
            Success(f(value))
        } catch (e: RuntimeException) {
            Failure(e)
        } catch (e: Exception) {
            Failure(RuntimeException(e))
        }

        override fun <B> flatMap(f: (A) -> Result<B>): Result<B> = try
        {
            f(value)
        } catch (e: RuntimeException) {
            Failure(e)
        } catch (e: Exception) {
            Failure(RuntimeException(e))
        }

        override fun toString(): String = "Success($value)"
    }

    companion object {

        operator fun <A> invoke(a: A? = null): Result<A> = when (a) {
            null -> Failure(NullPointerException())
            else -> Success(a)
        }

        operator fun <A> invoke(): Result<A> = Empty ⑤

        fun <A> failure(message: String): Result<A> =
                Failure(IllegalStateException(message))

        fun <A> failure(exception: RuntimeException): Result<A> =
                Failure(exception)

        fun <A> failure(exception: Exception): Result<A> =
                Failure(IllegalStateException(exception))
    }
}
```

① **Функции `getOrElse` и `orElse` теперь обрабатывают случай `Empty`**

④ По аналогии с экземпляром None в Option класс Result содержит экземпляр-синглтон Empty, параметризованный типом Nothing
⑤ По аналогии с Option версия функции invoke без входных параметров возвращает синглтон Empty

Теперь снова изменим приложение ToonMail, как показано в листингах 7.7, 7.8 и 7.9.

Листинг 7.7 Функция getResult

```
fun <K, V> Map<K, V>.getResult(key: K) = when {
    this.containsKey(key) -> Result(this[key])
    else -> Result.Empty ①
}
```

① Теперь функция возвращает Result.Empty, если указанный ключ отсутствует в ассоциативном массиве

Листинг 7.8 Класс Toon, использующий Result.Empty для представления необязательных данных

```
data class Toon private constructor (val firstName: String,
                                     val lastName: String,
                                     val email: Result<String>) {

    companion object {
        operator fun invoke(firstName: String,
                            lastName: String) =
            Toon(firstName, lastName, Result.Empty) ①

        operator fun invoke(firstName: String,
                            lastName: String,
                            email: String) =
            Toon(firstName, lastName, Result(email))
    }
}
```

① Если экземпляр конструируется без адреса электронной почты, свойству присваивается значение Result.Empty

Листинг 7.9 Приложение ToonMail, правильно обрабатывающее отсутствие необязательных данных

```
fun main(args: Array<String>) {

    val toons: Map<String, Toon> = mapOf(
        "Mickey" to Toon("Mickey", "Mouse", "mickey@disney.com"),
        "Minnie" to Toon("Minnie", "Mouse"),
        "Donald" to Toon("Donald", "Duck", "donald@disney.com"))

    val toon = getName()
        .flatMap(toons::getResult)
        .flatMap(Toon::email)

    println(toon)
}
```

```
fun getName(): Result<String> = try {
    validate(readLine())
} catch (e: IOException) {
    Result.failure(e)
}

fun validate(name: String?): Result<String> = when {
    name?.isNotEmpty() ?: false -> Result(name)
    else -> Result.failure(IOException()) ①
}
```

① **Функция `validate` теперь имитирует исключение `IOException`, если имя не было введено**

Для входных данных – `"Mickey"`, `"Minnie"`, `"Goofy"` и `""` – эта программа выведет:

```
Success(mickey@disney.com)
Empty
Empty
Failure(java.io.IOException)
```

Может показаться, что чего-то не хватает, потому что два разных случая `Empty` оказались неразличимы, но это не так. Сообщения об ошибках не нужны для необязательных данных. Если вы считаете, что сообщение необходимо, значит, ваши данные не являются необязательными.

7.5 Дополнительные способы использования Result

До сих пор мы рассматривали ограниченные варианты использования `Result`. Мы никогда не должны использовать `Result` для прямого доступа к значению в нем (если оно существует). Предыдущий пример использования `Result` соответствует простейшему конкретному варианту композиции функций: получить результат одного вычисления и использовать его в качестве входных данных в следующем вычислении.

Существуют и другие, более специфические варианты использования. Например, мы можем использовать значение в `Result`, только если оно соответствует некоторому предикату (т. е. некоторому условию). Иногда может потребоваться преобразовать ошибку `Failure` во что-то другое или превратить ее в успешное исключение. Также может потребоваться передать несколько результатов на вход одной функции. Возможно, могут понадобиться некоторые вспомогательные функции, создающие экземпляры `Result` из результатов вычислений, для работы с устаревшим кодом. Наконец, иногда бывает необходимо применить какие-то эффекты к результатам.

7.5.1 Применение предикатов

На практике часто приходится применять предикаты к результатам вычислений. (*Предикат* – это функция, возвращающая значение типа

Boolean.) Этот случай легко абстрагировать, поэтому сделаем это сейчас и только один раз.

Упражнение 7.5

Напишите функцию filter, принимающую условие в форме функции (A) -> Boolean и возвращающую результат Result<A>, который может быть экземпляром Success или Failure в зависимости от соответствия обернутого значения заданному условию. Вот сигнатура этой функции:

```
fun filter(p: (A) -> Boolean): Result<A>
```

Напишите вторую функцию, принимающую условие в первом аргументе и String во втором и использующую строковый аргумент для создания возможного экземпляра Failure.

Подсказка

Можно определить абстрактную функцию в классе Result и реализовать ее в подклассах, но попробуйте включить конкретную реализацию непосредственно в Result. Используйте при этом функции, которые были определены ранее.

Решение

Определим функцию fun, которая принимает обернутое значение, применяет к нему функцию-значение из аргумента и возвращает тот же экземпляр Result, если условие выполняется, или Empty (или Failure) – в противном случае. Все это можно реализовать с помощью функции flatMap:

```
fun filter(p: (A) -> Boolean): Result<A> = flatMap {
    if (p(it))
        this
    else
        failure("Condition not matched")
}

fun filter(message: String, p: (A) -> Boolean): Result<A> = flatMap {
    if (p(it))
        this
    else
        failure(message)
}
```

Упражнение 7.6

Напишите fun-функцию exists, которая принимает функцию-значение (A) -> Boolean и возвращает true, если обернутое значение соответствует условию, и false – в противном случае. Вот сигнатура этой функции:

```
fun exists(p: (A) -> Boolean): Boolean
```

Подсказка

И снова постарайтесь определить единую реализацию в классе Result. Можете использовать любые функции, написанные выше.

Решение

Решение состоит в том, чтобы применить функцию `map` к `Result<T>`, получить `Result<Boolean>` и затем использовать `getOrElse` со значением по умолчанию `false`. Использовать константную функцию для организации отложенных вычислений не требуется, потому что `false` – это литерал:

```
fun exists(p: (A) -> Boolean): Boolean = map(p).getOrElse(false)
```

Выбор имени `exists` для этой функции может показаться сомнительным. Но это та же самая функция, которую можно применить к списку и получить `true`, если хотя бы один элемент удовлетворяет условию. Поэтому имеет смысл использовать то же имя.

Кто-то из вас может заметить, что эта реализация также подойдет для функции `forAll`, которая возвращает `true`, если все элементы в списке удовлетворяют условию. Вы можете выбрать другое имя или определить функцию `forAll` в классе `Result` с той же реализацией. Но вы должны понимать, в чем `List` и `Result` похожи, а чем отличаются.

7.6 Преобразование ошибок

Иногда полезно заменить ошибку чем-то другим. Это все равно что перехватить одно исключение и возбудить другое. Это может понадобиться, чтобы заменить текст сообщения более подходящим или добавить дополнительную информацию, которая поможет пользователю определить причину проблемы. Например, такое сообщение, как «Файл конфигурации не найден», было бы гораздо более полезным, если бы оно содержало путь к каталогу, где программа пыталась найти файл.

Упражнение 7.7

Напишите функцию `mapFailure`, которая принимает строку и преобразует один экземпляр `Failure` в другой, используя строку как сообщение об ошибке. Если `Result` является экземпляром `Empty` или `Success`, эта функция ничего не должна делать.

Подсказка

В родительском классе определите абстрактную функцию.

Решение

Вот как выглядит объявление абстрактной функции в родительском классе:

```
abstract fun mapFailure(message: String): Result<A>
```

Реализации в `Empty` и `Success` должны просто возвращать `this`. Вот реализация в `Empty`:

```
override fun mapFailure(message: String): Result<Nothing> = this
```

А вот реализация в `Success`:

```
override fun mapFailure(message: String): Result<A> = this
```

Реализация в `Failure` обертывает имеющееся исключение новым экземпляром `Failure` с заданным сообщением и возвращает его:

```
override fun mapFailure(message: String): Result<A> =
    Failure(RuntimeException(message, exception))
```

Вы можете выбрать тип исключений `RuntimeException` или какой-то более конкретный его подтип. Также могла бы пригодиться функция преобразования `Empty` в `Failure` с заданным сообщением.

7.7 Дополнительные фабричные функции

Вы уже видели, как создавать экземпляры `Success` и `Failure` из значения. Однако есть ряд ситуаций, встречающихся настолько часто, что для них тоже желательно было бы определить свои фабричные функции. Для адаптации старых библиотек часто требуется создать `Result` из некоторого значения, в том числе и `null`, и сопроводить его конкретным сообщением об ошибке. Для этого в объекте-компаньоне можно определить функцию со следующей сигнатурой:

```
operator fun <A> invoke(a: A? = null, message: String): Result<A>
```

Также могут пригодиться функции, создающие `Result` из функции `(A) -> Boolean` и из экземпляра `A`:

```
operator fun <A> invoke(a: A? = null, p: (A) -> Boolean): Result<A>
operator fun <A> invoke(a: A? =
                null, message: String, p: (A) -> Boolean): Result<A>
```

Упражнение 7.8

Реализуйте функции `invoke`, описанные выше.

Подсказка

Вы должны выбрать, что возвращать в каждом случае.

Решение

Это довольно простое упражнение. Вот несколько возможных реализаций, возвращающих `Empty`, когда сообщение об ошибке не указано, и `Failure` – в противном случае:

```
operator fun <A> invoke(a: A? = null, message: String): Result<A> =
    when (a) {
        null -> Failure(NullPointerException(message))
        else -> Success(a)
    }

operator fun <A> invoke(a: A? = null, p: (A) -> Boolean): Result<A> =
    when (a) {
        null -> Failure(NullPointerException())
        else -> when {
            p(a) -> Success(a)
            else -> Empty
```

```
        }
    }

operator fun <A> invoke(a: A? = null,
                        message: String,
                        p: (A) -> Boolean): Result<A> =
    when (a) {
        null -> Failure(NullPointerException())
        else -> when {
            p(a) -> Success(a)
            else -> Failure(IllegalArgumentException(
                "Argument $a does not match condition: $message"))
        }
    }
```

Обратите внимание, что здесь перед применением предиката не потребовалось аргумент `a` типа `A?` приводить к типу `A`. Компилятор Kotlin понимает, что после проверки в первой инструкции `when` он не может иметь значения `null`.

7.8 Применение эффектов

До сих пор мы не применяли никаких эффектов к значениям, заключенным в `Result`, кроме его получения (через `getOrElse`). Такое положение дел сводит на нет преимущество использования `Result`. С другой стороны, мы еще не изучили требуемые при этом методы безопасного применения эффектов. Эффектом может быть все, что изменяет что-то во внешнем мире, например вывод сообщения в консоль, в файл, в базу данных, в поле изменяемого компонента или его отправка в сеть.

> **ВНИМАНИЕ** Прием, который я покажу ниже, небезопасен, поэтому его следует использовать только после выполнения всех вычислений. Эффекты должны применяться в специальной и ограниченной области кода, и дальнейшие вычисления не должны затрагивать значений, к которым применялся эффект. Если потребуется применить эффекты безопасным способом, а затем продолжить некоторые вычисления, подождите до главы 12, где я опишу все необходимые для этого приемы.

Применение эффекта в стандартном программировании состоит из извлечения значения (если оно есть) из `Result` и выполнения некоторого действия с ним. В безопасном программировании все делается наоборот: эффект передается в экземпляр `Result`, который применит его к содержащемуся значению (если имеется).

Эффекты в языке Kotlin можно представить с помощью функций, которые ничего не возвращают, а просто производят требуемый эффект. Их вообще не следовало бы называть функциями, но иного решения в Kotlin нет. В Java эффекты реализуют интерфейс `Consumer`. В Kotlin такие функции возвращают тип `Unit`.

Упражнение 7.9

Напишите функцию `forEach`, которая принимает эффект в параметре и применяет его к значению, содержащемуся в экземпляре `Result`.

Подсказка

В классе `Result` объявите абстрактную функцию, а в подклассах – конкретные ее реализации.

Решение

Вот объявление абстрактной функции в `Result`:

```
abstract fun forEach(effect: (A) -> Unit)
```

Обратите внимание, что обе функции, `forEach` и `effect`, возвращают `Unit`. Если функция `fun` возвращает `Unit`, объявление типа возвращаемого значения можно опустить. Реализация в `Failure` ничего не делает:

```
override fun forEach(effect: (A) -> Unit) {}
```

Реализация в `Empty` тоже ничего не делает, но имеет немного иную сигнатуру:

```
override fun forEach(effect: (Nothing) -> Unit) {}
```

Реализация в `Success` просто применяет эффект к значению:

```
override fun forEach(effect: (A) -> Unit) {
    effect(value)
}
```

Такая реализация `forEach` прекрасно подошла бы для класса `Option`, созданного нами в главе 6, но не для класса `Result`. Обычно в случае ошибки требуется выполнить какие-то особые действия.

Упражнение 7.10

Напишите функцию `forEachOrElse` для этого случая. Она должна поразному обрабатывать экземпляры `Failure` и `Empty`. Вот ее сигнатура в классе `Result`:

```
abstract fun forEachOrElse(onSuccess: (A) -> Unit,
                           onFailure: (RuntimeException) -> Unit,
                           onEmpty: () -> Unit)
```

Решение

Каждая из трех реализаций вызывает свою функцию:

```
// Success
override fun forEachOrElse(onSuccess: (A) -> Unit,
                           onFailure: (RuntimeException) -> Unit,
                           onEmpty: () -> Unit) = onSuccess(value)

// Failure
override fun forEachOrElse(onSuccess: (A) -> Unit,
                           onFailure: (RuntimeException) -> Unit,
```

```
                            onEmpty: () -> Unit) = onFailure(exception)
// Empty
override fun forEachOrElse(onSuccess: (Nothing) -> Unit,
                           onFailure: (RuntimeException) -> Unit,
                           onEmpty: () -> Unit) = onEmpty()
```

Обратите внимание, что мы не возбуждаем никаких исключений. Любые манипуляции с исключениями – это прерогатива вызывающего кода. Если программист захочет возбудить исключение, он передаст выражение { `throw it` } во втором аргументе.

Упражнение 7.11

Функция `forEachOrElse` работает прекрасно, но она не оптимальна. Фактически `forEach` действует точно так же, как `forEachOrElse` с определенными значениями аргументов. Сможете исправить ситуацию?

Подсказка

Сделайте все аргументы необязательными.

Решение

Решение заключается в определении значений по умолчанию для всех трех параметров. Вот новое объявление абстрактной функции в классе `Result`:

```
abstract fun forEach(onSuccess: (A) -> Unit = {},
                     onFailure: (RuntimeException) -> Unit = {},
                     onEmpty: () -> Unit = {})
```

Новую функцию можно переименовать и заменить ею прежнюю функцию `forEach`. Реализации в подклассах не изменились. Теперь можно вызвать `forEach`, определив эффекты только для `Success` и `Empty`, и игнорировать `Failure` (например) с помощью именованных аргументов:

```
val result: Result<Int> = if (z % 2 == 0) Result(z) else Result()
result.forEach({ println("$it is even") },
               onEmpty = { println("This one is odd") })
```

Обратите внимание на отсутствие имени у первого аргумента. Имена должны указываться только для аргументов, которые передаются не в своей позиции. `onEmpty` – это третий аргумент функции, а здесь он передается во второй позиции, поэтому его имя нужно указать.

Ниже приводится пример одного из часто встречающихся случаев применения функции `forEach`. Здесь используется гипотетическая функция `log`, объявленная на уровне пакета:

```
val result = getComputation()

result.forEach(::println, ::log)
```

Напомню еще раз, что эти функции не являются настоящими. Они просто реализуют простой и удобный способ использования `Result`. Но подробнее эту тему мы рассмотрим в главе 12.

7.9 Дополнительные способы комбинирования с типом Result

Тип `Result` имеет примерно похожую область применения, что и тип `Option`. В главе 6 мы определили функцию `lift` для композиции по экземплярам `Option`, которая преобразует функцию `(A) -> B` в функцию `(Option<A>) -> Option<B>`. Аналогичную функцию можно определить для `Result`, что мы и сделаем в серии упражнений, следующей далее.

Упражнение 7.12

Напишите функцию `lift` для `Result` с сигнатурой, как показано ниже. Поместите ее на уровень пакета.

```
fun <A, B> lift(f: (A) -> B): (Result<A>) -> Result<B>
```

Решение

Вот простейший вариант реализации:

```
fun <A, B> lift(f: (A) -> B): (Result<A>) -> Result<B> = { it.map(f) }
```

В отличие от `Option` здесь не требуется перехватывать исключение, которое может возбудить функция `f`, потому что оно уже обрабатывается функцией `map`.

Упражнение 7.13

Напишите функцию `lift2` для преобразования функции `(A) -> (B) -> C` и функцию `lift3` для преобразования функции `(A) -> (B) -> (C) -> D` со следующими сигнатурами:

```
fun <A, B, C> lift2(f: (A) -> (B) -> C):
    (Result<A>) -> (Result<B>) -> Result<C>
fun <A, B, C, D> lift3(f: (A) -> (B) -> (C) -> D):
    (Result<A>) -> (Result<B>) -> (Result<C>) -> Result<D>
```

Решение

Вот как выглядит решение:

```
fun <A, B, C> lift2(f: (A) -> (B) -> C):
    (Result<A>) -> (Result<B>) -> Result<C> =
        { a ->
            { b ->
                a.map(f).flatMap { b.map(it)}
            }
        }

fun <A, B, C, D> lift3(f: (A) -> (B) -> (C) -> D):
    (Result<A>) -> (Result<B>) -> (Result<C>) -> Result<D> =
        { a ->
            { b ->
                { c ->
```

```
                    a.map(f).flatMap { b.map(it)}.flatMap { c.map(it) }
                }
            }
        }
```

Вы наверняка заметили знакомый шаблон. Таким способом можно определить `lift` с любым числом параметров.

Упражнение 7.14

В главе 6 мы определили функцию `map2`, которая принимает `Option<A>`, `Option<B>` и функцию `(A) -> (B) -> C` и возвращает `Option<C>`. Определите аналогичную функцию для `Result`.

Подсказка

Не используйте функцию из `Option`. Задействуйте функцию `lift2` из упражнения 7.13.

Решение

Вот как была определена функция в классе `Option`:

```
fun <A, B, C> map2(oa: Option<A>,
                   ob: Option<B>,
                   f: (A) -> (B) -> C): Option<C> =
    oa.flatMap { a -> ob.map { b -> f(a)(b) } }
```


Тот же шаблон использовался в реализации `lift2`. Поэтому версия `map2` для `Result` выглядит проще:

```
fun <A, B, C> map2(a: Result<A>,
                   b: Result<B>,
                   f: (A) -> (B) -> C): Result<C> =
    lift2(f)(a)(b)
```

Одно из типичных применений этой функции – вызов функций или конструкторов с аргументами типа `Result`, возвращаемыми другими функциями. Вернемся к предыдущему примеру `ToonMail`. Чтобы заполнить ассоциативный массив `Toon`, можно предложить пользователю ввести имя, фамилию и адрес электронной почты с помощью следующих функций:

```
fun getFirstName(): Result<String> = Result("Mickey")

fun getLastName(): Result<String> = Result("Mouse")

fun getMail(): Result<String> = Result("mickey@disney.com")
```

Конечно, истинная реализация должна быть иной. Но мы еще не научились безопасно принимать ввод из консоли, поэтому пока будем пользоваться этими фиктивными реализациями. Вот как можно создать экземпляр `Toon` с помощью этих реализаций:

```
var createPerson: (String) -> (String) -> (String) -> Toon =
                            { x -> { y -> { z -> Toon(x, y, z) } } }

val toon = lift3(createPerson)(getFirstName())(getLastName()) (getMail())
```

Но на этом мы исчерпали доступные возможности абстракции. На практике может понадобиться вызывать функции или конструкторы с более чем тремя аргументами. В таких ситуациях можно прибегнуть к следующему шаблону, который иногда называют *включением* (comprehension):

```
val toon = getFirstName()
    .flatMap { firstName ->
        getLastName()
            .flatMap { lastName ->
                getMail()
                    .map { mail -> Toon(firstName, lastName, mail) }
            }
    }
```

Шаблон включения дает два преимущества:

- позволяет использовать любое число аргументов;
- не требует определения функции.

Обратите внимание: чтобы использовать `lift3`, совсем необязательно определять отдельную функцию, правда, при этом придется указать типы аргументов, потому что возможности автоматического определения типов в Kotlin не безграничны:

```
val toon2 = lift3 { x: String ->
    { y: String ->
        { z: String ->
            Toon(x, y, z)
        }
    }
}(getFirstName())(getLastName())(getMail())
```

Некоторые языки программирования предлагают синтаксический сахар для таких конструкций, например:

```
for {
    firstName in getFirstName(),
    lastName in getLastName(),
    mail in getMail()
} return new Toon(firstName, lastName, mail)
```

В Kotlin нет ничего подобного, но это не вызывает никаких сложностей. Обратите внимание, что вызовы `flatMap` и `map` – вложенные. Все начинается с вызова первой функции (или с экземпляра `Result`), затем последовательно вызываются `flatMap`, и, наконец, вызов `map` отображается в вызов конструктора или другой функции. Например, чтобы вызвать функцию с пятью параметрами, когда имеется только пять экземпляров `Result`, можно использовать следующий прием:

```
val result1 = Result(1)
val result2 = Result(2)
val result3 = Result(3)
val result4 = Result(4)
```

```
val result5 = Result(5)

fun compute(p1: Int, p2: Int, p3: Int, p4: Int, p5: Int) =
    p1 + p2 + p3 + p4 + p5

val result = result1.flatMap { p1: Int ->
    result2.flatMap { p2 ->
        result3.flatMap { p3 ->
            result4.flatMap { p4 ->
                result5.map { p5 ->
                    compute(p1, p2, p3, p4, p5)
                }
            }
        }
    }
}
```

Это несколько надуманный пример, но он наглядно показывает, как можно расширить шаблон. Обратите внимание, что последний вызов `map` вместо `flatMap` не является жестким требованием. Функция `map` используется просто потому, что функция `compute` возвращает простое значение. Если бы она возвращала `Result`, вместо `map` мы должны были бы использовать `flatMap`:

```
val result1 = Result(1)
val result2 = Result(2)
val result3 = Result(3)
val result4 = Result(4)
val result5 = Result(5)

fun compute(p1: Int, p2: Int, p3: Int, p4: Int, p5: Int) =
    Result(p1 + p2 + p3 + p4 + p5)

val result = result1.flatMap { p1: Int ->
    result2.flatMap { p2 ->
        result3.flatMap { p3 ->
            result4.flatMap { p4 ->
                result5.flatMap { p5 ->
                    compute(p1, p2, p3, p4, p5)
                }
            }
        }
    }
}
```

Но, так как в конце чаще всего используется конструктор, а конструкторы всегда возвращают значения, отличные от `Result`, для вызова последней функции приходится использовать `map`.

Итоги

- Необходимо иметь возможность определять, когда отсутствие данных вызвано ошибкой. Тип `Option` не дает ее, поэтому нам понадобился тип `Result`. Вы также можете использовать тип `Either` для представления данных, которые могут иметь один тип из двух – `Left` или `Right`.
- Тип `Either` может иметь функции `map` и `flatMap`, подобно типу `Option`, но их можно определить отдельно для каждой стороны (`Right` и `Left`).
- Тип `Either` можно сделать смещенным, если одна сторона (например, `Left`) всегда будет представлять один и тот же тип (например, `RuntimeException`). Такой смещенный тип называется `Result`. Успех в нем представлен подтипом `Success`, а ошибка – подтипом `Failure`.
- Один из вариантов использования типа `Result` заключается в извлечении заключенного в него значения, если оно присутствует, или использование значения по умолчанию – в противном случае. Вычисление значения по умолчанию, если оно не является литералом, можно отложить.
- Комбинирование типа `Option` (представляющего необязательные данные) с типом `Result` (представляющим данные или ошибки) является утомительным занятием. Этот вариант использования упрощается добавлением в `Result` подтипа `Empty`, что делает тип `Option` бесполезным.
- При необходимости ошибки можно отображать, например, в более конкретные сообщения.
- Добавление нескольких фабричных функций упростило создание экземпляров `Result` в некоторых ситуациях, таких как использование данных с поддержкой `null` или условных данных, которые представлены данными и условием.
- Применять эффекты к `Result` можно с помощью функции `forEach`. Эта функция позволяет применить разные эффекты к `Success`, `Failure` и `Empty`.
- Функции `(A) -> B` можно преобразовывать (с помощью функции `lift`) в функции `(Result<A>) -> Result<B>`. Аналогично можно преобразовывать функции `(A) -> (B) -> C` (с помощью `lift2`) в функции `(Result<A>) -> (Result<B>) -> Result<C>`.
- Для вычислений с любым количеством данных в форме `Result` можно использовать шаблон включения.

8 Дополнительные операции со списками

Эта глава охватывает следующие темы:

- ускорение обработки списков за счет мемоизации;
- комбинирование типов `List` и `Result`;
- реализация доступа к элементам списка по индексам;
- развертывание списков;
- автоматизация параллельной обработки списков.

В главе 5 мы создали свою первую структуру данных – односвязный список. На тот момент мы не владели всеми методами, необходимыми, для того чтобы сделать эту структуру полноценным инструментом обработки данных. Одним из особенно полезных инструментов, которого нам не хватало, был способ представления операций, создающих необязательные данные, или операций, приводящих к ошибке.

В главах 6 и 7 вы узнали, как представлять необязательные данные и ошибки. В этой главе вы узнаете, как комбинировать операции, которые возвращают необязательные данные или ошибки со списками. Также мы разработали несколько функций, которые действуют не самым оптимальным образом, например `length`, и тогда я говорил, что позднее вы познакомитесь с более эффективными методами реализаций этих операций. Это время настало, и в данной главе вы узнаете, как реализовать эти операции более эффективно. Вы также увидите, как автоматически распараллеливать некоторые операции со списками, чтобы использовать преимущества многоядерной архитектуры современных компьютеров.

8.1 Проблемы функции length

Свертка списка начинается с начального значения и затем последовательно добавляет к нему каждый элемент списка. Для этого, очевидно, требуется время, пропорциональное длине списка. Можно ли ускорить эту операцию? В качестве примера применения операции свертки мы написали функцию `length` (упражнение 5.10) со следующей реализацией:

```
fun length(): Int = foldLeft(0) { { _ -> it + 1} }
```

Эта реализация основана на операции свертки списка, которая на каждом шаге прибавляет 1 к результату. Начальное значение равно 0, а значения самих элементов игнорируются. Это обстоятельство позволяет использовать одну и ту же реализацию для всех списков. Поскольку элементы списка игнорируются, их тип не имеет значения. Но сравните предыдущую операцию с операцией, которая вычисляет сумму списка целых чисел:

```
fun sum(list: List<Int>): Int = list.foldRight(0) { x -> { y -> x + y } }
```

Основное отличие заключается в том, что функция `sum` может работать только с целыми числами, тогда как функция `length` работает со списками любых типов. Обратите внимание, что `foldRight` и `foldLeft` – это только средства абстракции рекурсии. Длину списка можно определить как 0 для пустого списка и 1 плюс длина хвоста для непустого списка. Точно так же сумму списка целых чисел можно рекурсивно определить как 0 для пустого списка и значение заголовка плюс сумма хвоста для непустого.

Аналогичным образом к спискам можно применять другие операции, и среди них есть такие, для которых тип элементов списка не имеет значения:

1. *Хеш-код списка можно вычислить как сумму хеш-кодов его элементов.* Так как хеш-код – это целое число (по крайней мере, для объектов в языке Kotlin), данная операция не зависит от типов объектов.
2. *Строковое представление списка, возвращаемое функцией* `toString`*, можно вычислить как конкатенацию представлений* `toString` *элементов списка.* И снова фактический тип элементов не имеет значения.

Некоторые операции могут зависеть от некоторых характеристик типа элементов, но не от конкретного типа. Например, функции `max`, которая возвращает максимальный элемент списка, понадобится только механизм сравнения `Comparator` или тип, реализующий интерфейс `Comparable`. А более общую функцию `sum` можно определить для типов, реализующих интерфейс `Summable` с его функцией `plus`.

8.2 Проблема производительности

Все эти функции можно реализовать с использованием операции свертки, но такой подход имеет большой недостаток: время, необходимое для вычисления результата, прямо пропорционально длине списка. Пред-

ставьте, что у вас есть список с миллионом элементов и вы решили проверить его длину. Подсчет числа элементов может показаться единственно возможным решением (собственно, так и работает функция `length`, основанная на операции свертки). Но, если вы решите добавлять элементы в список, пока их число не достигнет миллиона, вы едва ли будете проверять его длину после каждой операции добавления.

В таких ситуациях принято хранить счетчик элементов где-то еще и увеличивать его на единицу после добавления каждого элемента. Возможно, вам придется вычислить длину списка один раз, если изначально список непустой, но и все. С этим приемом мемоизации мы познакомились в главе 4. Теперь остается только решить, где хранить этот счетчик. Ответ очевиден: в самом списке.

8.3 Преимущества мемоизации

Обновление счетчика элементов в списке требует некоторого времени, поэтому операция добавления элемента будет выполняться немного медленнее, чем без обновления счетчика. Может показаться, что в этом случае мы меняем шило на мыло (время на время). Составляя список из одного миллиона элементов, мы потеряем в миллион раз больше времени, чем необходимо для увеличения счетчика на единицу. Однако эта потеря будет компенсироваться в операциях получения длины списка, длительность таких операций будет близка к 0 (и, как нетрудно догадаться, останется постоянной). Возможно, общее время, потраченное на увеличение счетчика, будет равно затратам на один вызов `length`. Но если программа вызывает `length` больше одного раза, выигрыш становится абсолютно очевидным.

8.3.1 Недостатки мемоизации

Прием мемоизации имеет несколько недостатков. В этом разделе я опишу их и дам несколько советов по использованию этого приема.

Мемоизация может превратить функцию с временем выполнения O(*n*), прямо пропорциональным количеству элементов, в функцию с временем выполнения O(1) – постоянным временем. Это огромное преимущество, но оно требует дополнительных затрат времени на вставку элементов. Однако замедление вставки, как правило, не является большой проблемой. Гораздо более важной является проблема увеличения расходуемого объема памяти.

Структуры данных, реализующие изменение на месте, не подвержены этой проблеме. В изменчивом списке ничто не мешает хранить длину списка как изменяемое целое, занимающее всего 32 бита. Но в неизменяемом списке придется запомнить длину каждого элемента. Точно подсчитать увеличение размера сложно, но, если размер узла в односвязном списке составляет около 40 байт (для самого узла) плюс две 32-битные ссылки для головы и хвоста (в 32-битной JVM), мы получим около 100 байт на элемент. В этом случае добавление счетчика с длиной приведет

к увеличению требуемого объема памяти чуть более чем на 30 %. Если запоминаемые значения будут ссылками, такими как ссылки на максимальное или минимальное значение в списках объектов `Comparable`, результат будет тем же. В 64-битной JVM вычисления становятся еще сложнее из-за некоторой оптимизации размера ссылок, но суть, я надеюсь, вы поняли.

> **ПРИМЕЧАНИЕ** За дополнительной информацией о размере ссылок на объекты в JVM обращайтесь к документации на сайте Oracle с описанием сжатых указателей на объекты и улучшений производительности JVM (https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html).

Вам решать, использовать ли мемоизацию в своих структурах данных. Это может быть допустимо для функций, которые часто вызываются и не создают новых объектов для своих результатов. Например, функции `length` и `hashCode` возвращают целые числа, а функции `max` или `min` возвращают ссылки на уже существующие объекты, поэтому они могут быть хорошими кандидатами. С другой стороны, функция `toString` создает новые строки, которые придется запоминать, поэтому, вероятно, ее мемоизация приведет к расходованию значительных объемов памяти. Другой фактор, который необходимо учитывать, – как часто используется функция. Функция `length` может использоваться чаще, чем `hashCode`, потому что списки обычно не используются в роли ключей в ассоциативных массивах.

Упражнение 8.1

Напишите мемоизованную версию функции `length` из упражнения 3.8. Вот ее сигнатура в классе `List`:

```
abstract fun lengthMemoized(): Int
```

Решение

Реализация в классе `Nil` должна возвращать 0:

```
override fun lengthMemoized(): Int = 0
```

Прежде чем приступать к реализации версии в классе `Cons`, сначала нужно добавить в класс поле для мемоизации:

```
internal class Cons<out A>(internal val head: A,
                           internal val tail: List<A>): List<A>() {

    private val length: Int = tail.lengthMemoized() + 1

    ...
```

После этого мемоизованная версия `lengthMemoized` должна просто возвращать `length`:

```
override fun lengthMemoized() = length
```

Эта версия будет работать намного быстрее. Интересно отметить связь между функциями `length` и `isEmpty`. Кто-то из вас мог бы подумать,

что функция `isEmpty` действует эквивалентно выражению `length == 0`. С логической точки зрения это действительно так, но с точки зрения реализации и производительности разница может быть огромна.

Кстати, Kotlin допускает гораздо более компактное решение, если использовать абстрактное свойство. Для свойств `val` Kotlin автоматически генерирует методы чтения и позволяет определять абстрактные свойства, которые реализуются в расширяющем классе. Вот объявление такого свойства в классе `List`:


```
abstract val length: Int
```

В объекте `Nil` этому свойству присваивается 0:

```
override val length = 0
```

А в классе `Cons` оно инициализируется в точности, как в предыдущей реализации функции:

```
override val length = tail.length + 1
```

Запоминание максимального или минимального значения в списке объектов `Comparable` можно организовать точно так же, но ситуация осложняется, если максимальное или минимальное значение может быть удалено из списка. К минимальным или максимальным элементам часто обращаются, чтобы получить элементы по приоритету. В этом случае функция `compareTo` элементов сравнивает их приоритеты.

Мемоизация приоритетов позволит сразу определить, какой элемент имеет максимальный приоритет, но она теряет всякий смысл, потому что программа часто будет удалять соответствующие элементы. Для таких случаев необходима другая структура данных, о которой я расскажу в главе 11.

8.3.2 Оценка увеличения производительности

Как я уже сказал, вам решать, стоит ли мемоизовать какие-либо функции в классе `List`. А принять решение вы сможете, немного поэкспериментировав. Измерение доступного объема памяти до и после создания списка с одним миллионом целых чисел показывает, что мемоизация незначительно увеличивает потребление памяти.

Конечно, это не самый точный метод измерения, тем не менее он показывает, что в среднем объем доступной памяти уменьшается примерно на 22 Мб в обоих случаях (с мемоизацией и без), и эта величина варьируется в диапазоне от 20 до 25 Мб. Это показывает, что теоретическое увеличение на 4 Мб (1 млн * 4 байта) не так значительно, как ожидалось. Но увеличение производительности оказалось огромным. Десять попыток определить длину списка могут в сумме занять 200 мс в варианте без мемоизации, а в версии с мемоизацией это время равно 0 (слишком короткое, чтобы его можно было измерить в миллисекундах). Правда, время добавления элемента увеличилось (добавление единицы к длине хвоста и сохранение результата), удаление элемента не требует затрат, поскольку длина хвоста уже хранится в памяти.

8.4 Комбинирование List и Result

В главе 7 вы видели, насколько похожи структуры данных `Result` и `List`. Главное их отличие – объем занимаемой памяти, но они обе используют одинаковые функции, такие как `map` и `flatMap`. Вы также видели, как можно комбинировать списки со списками и результаты с результатами. Теперь вы узнаете, как комбинировать результаты со списками.

8.4.1 Списки, возвращающие Result

Вероятно, вы уже заметили, что я стараюсь избегать прямого обращения к элементам результатов и списков. Попытка обратиться к голове или хвосту списка вызывает исключение, если список пустой (представлен значением `Nil`), а возбуждение исключения – один из самых прямых способов сделать программы небезопасными. Но вы знаете, что можно безопасно обратиться к значению `Result`, если указать значение по умолчанию, которое будет использоваться в случае ошибочного или пустого результата. Можно ли сделать то же самое в отношении головы списка? Именно так – нет, но можно вернуть `Result`.

Упражнение 8.2

Напишите функцию `headSafe` в `List<A>`, возвращающую `Result<A>`.

Подсказка

Объявите следующую абстрактную функцию в `List` и добавьте конкретные реализации в подклассы:

```
abstract fun headSafe(): Result<A>
```

Решение

Реализация в классе `Nil` должна вернуть пустой результат:

```
override fun headSafe(): Result<Nothing> = Result()
```

Реализация в классе `Cons` должна вернуть `Success` со значением головы списка:

```
override fun headSafe(): Result<A> = Result(head)
```

Упражнение 8.3

Напишите функцию `lastSafe`, возвращающую `Result` с последним элементом списка.

Подсказка

Не используйте рекурсию явно, а вместо этого воспользуйтесь функциями, написанными в главе 5. Попробуйте определить единственную реализацию в классе `List`.

Решение

Решить эту задачу можно несколькими способами. Сначала мы рассмотрим простейшее решение и обсудим его недостатки, а затем я покажу

более удачное решение, лишенное этих недостатков. Итак, вот тривиальное решение, использующее явную рекурсию:

```
fun lastSafe(): Result<A> = when (this) {
    Nil -> Result()
    is Cons -> when (tail) {
        Nil -> Result(head)
        is Cons -> tail.lastSafe()
    }
}
```

Это решение имеет несколько недостатков. Во-первых, оно рекурсивное, т. е. его следует преобразовать в сорекурсивное. Сделать это несложно, но для этого придется передать список в виде аргумента:

```
tailrec fun <A> lastSafe(list: List<A>): Result<A> = when (list) {
    List.Nil -> Result()
    is List.Cons<A> -> when (list.tail) {
        List.Nil -> Result(list.head)
        is List.Cons -> lastSafe(list.tail)
    }
}
```

Более удачное решение основано на использовании свертки, которая абстрагирует рекурсию. От нас требуется только определить правильную функцию для свертки. Мы всегда должны сохранять последнее значение, если оно имеется. Вот как может выглядеть такая функция:

```
{ _: Result<A> -> { y: A -> Result(y) } }
```

Теперь осталось только применить `foldLeft` к списку и использовать `Result()` в качестве единичного значения:

```
fun lastSafe(): Result<A> =
    foldLeft(Result()) { _: Result<A> -> { y: A -> Result(y) } }
```

Упражнение 8.4

Попробуйте заменить `headSafe` единой реализацией в классе `List`, использующей свертку. Какие достоинства и недостатки имеет такая реализация?

Решение

Решить задачу можно так:

```
fun headSafe(): Result<A> =
    foldRight(Result()) { x: A -> { _: Result<A> -> Result(x) } }
```

Единственное достоинство этой функции – возможность проявить свою изобретательность и представить замысловатое решение. Разрабатывая функцию `lastSafe`, мы знали, что требуется обойти весь список, чтобы найти последний элемент. Чтобы найти первый элемент, выполнять обход списка не требуется.

Функция `foldRight` в этом решении просто переворачивает список и затем выполняет обход элементов результата, чтобы найти последний

элемент (который является первым в оригинальном списке). Очень неэффективно! И кстати, это именно то, что делает функция `lastSafe`: она переворачивает список и извлекает первый элемент из результата. Кроме чувства гордости за находчивость, эта реализация не дает ничего. Но если вы все же решите написать единую реализацию в классе `List`, тогда используйте следующее сопоставление с образцом:

```
fun headSafe(): Result<A> = when (this) {
    Nil -> Result()
    is Cons -> Result(head)
}
```

8.4.2 Преобразование List<Result> в Result<List>

Список с результатами некоторых вычислений часто имеет вид `List<Result>`. Например, применение функции `(T) -> Result<U>` к списку элементов типа `T` даст в результате `List<Result<U>>`. Такие значения часто бывает нужно комбинировать с функциями, принимающими `List<T>`. Это означает, что необходим способ преобразования `List<Result<U>>` в `List<U>`, похожий на тот, что реализует функция `flatMap`. Огромная разница заключается в том, что в данном случае задействованы два разных типа данных: `List` и `Result`. Есть несколько стратегий организовать такое преобразование:

1. Отбросить все ошибочные и пустые результаты и создать список `List<U>` из действительных значений. Если действительных значений не останется, результатом сможет служить пустой список.
2. Отбросить все ошибочные и пустые результаты и создать список `List<U>` из действительных значений. Если действительных значений не останется, результатом сможет служить экземпляр `Failure`.
3. Если действительный результат должен получаться, только если список не содержит ошибочных и пустых значений, тогда результатом может служить список `List<U>`, возвращаемый как `Success<List<U>>`, в противном случае можно вернуть `Failure<List<U>>`.

В первом случае мы получим соответствие списку результатов, каждый из которых является необязательным. Во втором случае будет действительный результат, только если в списке имеется хотя бы один действительный результат. И в третьем случае мы получим действительный результат, только если в списке нет ошибочных и пустых результатов.

Упражнение 8.5

Напишите функцию `flattenResult`, которая принимает `List<Result<A>>` и возвращает `List<A>` со всеми действительными значениями из оригинального списка, игнорируя ошибочные и пустые значения. Функция должна быть определена на уровне пакета и иметь следующую сигнатуру:

```
fun <A> flattenResult(list: List<Result<A>>): List<A>
```

Попробуйте не использовать рекурсию явно, а вместо нее используйте функции из классов `List` и `Result`.

Подсказка

Имя функции явно намекает, что вы должны сделать.

Решение

Чтобы решить эту задачу, нужно каждый элемент `Result<A>` преобразовать в `List<A>` (если он действительный) или в пустой список (если он ошибочный или пустой), используя следующую функцию:

```
{ ra -> ra.map { List(it) }.getOrElse(List()) }
```

Эта функция имеет тип `(Result<A>) -> List<A>`. Теперь осталось только применить ее к `List<Result<A>>` с помощью `flatMap`:

```
fun <A> flattenResult(list: List<Result<A>>): List<A> =
    list.flatMap { ra -> ra.map { List(it) }.getOrElse(List()) }
```

Упражнение 8.6

Напишите функцию `sequence`, преобразующую `List<Result<A>>` в `Result<List<A>>`. Она должна возвращать `Success<List<A>>`, если все элементы в оригинальном списке являются экземплярами `Success`, или `Failure<List<A>>` – в противном случае. Вот ее сигнатура:

```
fun <A> sequence(list: List<Result<A>>): Result<List<A>>
```

Подсказка

И снова используйте `foldRight` вместо явной рекурсии. Вам также понадобится функция `map2` из класса `Result`.

Решение

Вот реализация, использующая `foldRight` и `Result.map2`:

```
import com.fpinkotlin.common.map2

...

fun <A> sequence(list: List<Result<A>>): Result<List<A>> =
    list.foldRight(Result(List())) { x ->
        { y: Result<List<A>> ->
            map2(x, y) { a -> { b: List<A> -> b.cons(a) } }
        }
    }
```

И снова предпочтительнее было бы иметь реализацию на основе `foldLeft`, не страдающую проблемой переполнения стека. Если вы решите написать ее, не забудьте перевернуть результат перед возвратом.

Эта реализация обрабатывает пустые значения (`Empty`), как если бы они были ошибочными (`Failure`), и возвращает первый встретившийся недопустимый элемент, который может быть экземпляром `Failure` или `Empty`. Однако такое поведение может оказаться нежелательным.

Если следовать логике, что `Empty` представляет необязательные данные, тогда перед преобразованием из списка нужно отфильтровать все элементы `Empty`. А для этого необходимо использовать функцию `isEmpty` из класса `Result`, возвращающую `true` для экземпляра подкласса `Empty` и `false` для экземпляров `Success` и `Failure`:

```
fun <A> sequence2(list: List<Result<A>>): Result<List<A>> =
    list.filter{ !it.isEmpty() }.foldRight(Result(List())) { x ->
        { y: Result<List<A>> ->
            map2(x, y) { a -> { b: List<A> -> b.cons(a) } }
        }
    }
```

Упражнение 8.7

Напишите более обобщенную версию функции `traverse`, которая выполняет обход списка `List<A>`, применяет функцию `(A) -> Result<B>` к каждому элементу и возвращает `Result<List<B>>`. Вот ее сигнатура:


```
fun <A, B> traverse(list: List<A>, f: (A) -> Result<B>): Result<List<B>>
```

Затем напишите новую версию `sequence` в терминах `traverse`.

Подсказка

Вместо явной рекурсии используйте `foldRight` или, чтобы избавиться от опасности переполнения стека, ее безопасную версию `coFoldRight`.

Решение

Сначала определим функцию `traverse`:

```
fun <A, B> traverse(list: List<A>, f: (A) -> Result<B>): Result<List<B>> =
    list.foldRight(Result(List())) { x ->
        { y: Result<List<B>> ->
            map2(f(x), y) { a -> { b: List<B> -> b.cons(a) } }
        }
    }
```

Затем переопределим функцию `sequence`:

```
fun <A> sequence(list: List<Result<A>>): Result<List<A>> =
    traverse(list, { x: Result<A> -> x })
```

8.5 Абстракции операций со списками

Многие распространенные операции с типом данных `List` заслуживают обобщения, чтобы потом не приходилось писать один и тот же код снова и снова. Вы постоянно будете обнаруживать новые варианты использования, которые можно реализовать путем комбинирования основных функций. Никогда не стесняйтесь оформлять эти варианты использования в виде новых функций в классе `List`. Следующие упражнения демонстрируют несколько наиболее распространенных операций со списками:

- упаковка (zipping) и распаковка (unzipping) списков;
- преобразование списка пар в пару списков;
- преобразование списка любого списка в пару списков.

8.5.1 Упаковка и распаковка списков

Упаковка (zipping) – это процесс объединения двух списков в один путем объединения элементов с одинаковыми индексами. *Распаковка* (unzipping) – это обратный процесс, заключающийся в создании двух списков из одного путем деления элементов, как, например, создание двух списков координат `x` и `y` из одного списка точек.

Упражнение 8.8

Напишите функцию `zipWith`, объединяющую элементы двух списков разных типов с использованием заданной функции и создающую новый список. Вот ее сигнатура:

```
fun <A, B, C> zipWith(list1: List<A>,
                      list2: List<B>,
                      f: (A) -> (B) -> C): List<C>
```

Эта функция принимает списки `List<A>` и `List<B>` и создает список `List<C>`, объединяя элементы с помощью функции `(A) -> (B) -> C`.

Подсказка

Объединение должно завершаться по достижении конца кратчайшего списка.

Решение

В этом решении придется явно использовать рекурсию, потому что в рекурсии участвуют сразу два списка, а для этого случая у нас нет никаких готовых абстракций. Вот как выглядит решение:

```
fun <A, B, C> zipWith(list1: List<A>,
                      list2: List<B>,
                      f: (A) -> (B) -> C): List<C> {
    tailrec
    fun zipWith(acc: List<C>,
                list1: List<A>,
                list2: List<B>): List<C> = when (list1) {
        List.Nil -> acc
        is List.Cons -> when (list2) {
            List.Nil -> acc
            is List.Cons ->
                zipWith(acc.cons(f(list1.head)(list2.head)),
                        list1.tail, list2.tail)
        }
    }
    return zipWith(List(), list1, list2).reverse()
}
```

Сорекурсивная вспомогательная функция `zipWith` вызывается с пустым списком в качестве начального значения аккумулятора. Если один из двух списков-аргументов пуст, сорекурсия останавливается и возвращается текущий аккумулятор. Иначе вычисляется новое значение путем применения функции к значениям голов обоих списков, и рекурсивно вызывается вспомогательная функция, которой передаются хвосты обоих списков из аргументов.

Упражнение 8.9

Предыдущее упражнение заключалось в создании списка, объединяющего элементы с одинаковыми индексами из двух исходных списков. В этом упражнении вы должны написать функцию `product`, которая создает список со всеми возможными комбинациями элементов из двух исходных списков. Например, для списков `List("a", "b", "c")` и `List("d", "e", "f")` и при использовании функции конкатенации строк `product` должна вернуть `List("ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf")`.

Подсказка

Для решения этого упражнения явная рекурсия не нужна.

Решение

Решение напоминает шаблон включения, который использовался в главе 7 для комбинирования `Result`. Единственное отличие в том, что здесь число комбинаций равно произведению количества элементов в списках, тогда как при комбинировании `Result` число комбинаций всегда ограничивалось одним:

```
fun <A, B, C> product(list1: List<A>,
                      list2: List<B>,
                      f: (A) -> (B) -> C): List<C> =
    list1.flatMap { a -> list2.map { b -> f(a)(b) } }
```

> **ПРИМЕЧАНИЕ** Таким способом можно объединить больше двух списков. Единственная проблема в том, что число комбинаций растет в геометрической прогрессии.

Чаще всего `product` и `zipWith` применяются, когда в качестве объединяющей функции нужно использовать конструктор. Вот пример использования конструктора `Pair`:

```
product(List(1, 2), List(4, 5, 6)) { x -> { y: Int -> Pair(x, y) } }
zipWith(List(1, 2), List(4, 5, 6)) { x -> { y: Int -> Pair(x, y) } }
```

Первое выражение создаст список всех возможных пар, составленных из элементов двух списков:

```
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), NIL]
```

Второе выражение создаст список пар элементов с одинаковыми индексами:

```
[(1, 4), (2, 5), NIL]
```

Упражнение 8.10

Напишите функцию `unzip`, преобразующую список пар в пару списков. Вот ее сигнатура:

```
fun <A, B> unzip(list: List<Pair<A, B>>): Pair<List<A>, List<B>>
```

Подсказка

Вместо явной рекурсии используйте `foldRight`.

Решение

Для реализации достаточно применить `foldRight` к списку и использовать пару пустых списков в качестве единичного значения:

```
fun <A, B> unzip(list: List<Pair<A, B>>): Pair<List<A>, List<B>> =
    list.coFoldRight(Pair(List(), List())) { pair ->
        { listPair: Pair<List<A>, List<B>> ->
            Pair(listPair.first.cons(pair.first),
                 listPair.second.cons(pair.second))
        }
    }
```

Упражнение 8.11

Обобщите функцию `unzip`, чтобы она могла преобразовывать списки любого типа в пару списков с помощью функции, которая принимает объект с типом элемента списка и создает пару `Pair`. Например, из списка экземпляров платежей `Payment` она должна создать пару списков: один содержит кредитные карты, использованные для осуществления платежей, а другой – суммы платежей. Реализуйте ее как функцию экземпляра `List` со следующей сигнатурой:

```
fun <A1, A2> unzip(f: (A) -> Pair<A1, A2>): Pair<List<A1>, List<A2>>
```

Подсказка

Решение выглядит почти так же, как в упражнении 8.10.

Решение

Важно отметить, что результат функции преобразования элемента списка должен использоваться дважды. Чтобы не применять функцию дважды, можно использовать многострочное лямбда-выражение:

```
fun <A1, A2> unzip(f: (A) -> Pair<A1, A2>): Pair<List<A1>, List<A2>> =
    this.coFoldRight(Pair(Nil, Nil)) { a ->
        { listPair: Pair<List<A1>, List<A2>> ->
            val pair = f(a)
            Pair(listPair.first.cons(pair.first),
                 listPair.second.cons(pair.second))
        }
    }
```

Решение будет выглядеть интереснее, если использовать функцию `let` из стандартной библиотеки Kotlin:

```
fun <A1, A2> unzip(f: (A) -> Pair<A1, A2>): Pair<List<A1>, List<A2>> =
    this.coFoldRight(Pair(Nil, Nil)) { a ->
        { listPair: Pair<List<A1>, List<A2>> ->
            f(a).let {
                Pair(listPair.first.cons(it.first),
                     listPair.second.cons(it.second))
            }
        }
    }
```


Почему это решение интереснее? Здесь нет никаких веских причин, кроме того, что оно мне нравится! Если вы другого мнения, тогда просто используйте многострочную версию (которая, кстати, должна работать чуть быстрее). В любом случае теперь можно переписать версию из упражнения 8.10, как показано ниже:

```
fun <A, B> unzip(list: List<Pair<A, B>>): Pair<List<A>, List<B>> =
    list.unzip { it }
```

8.5.2 Доступ к элементам по индексам

В главе 5 мы создали свою первую структуру данных – односвязный список. Односвязный список – не лучшая структура для доступа к элементам по индексам, но иногда просто необходимо использовать индексы. Как обычно, мы должны абстрагировать эту процедуру в функциях класса `List`.

Упражнение 8.12

Напишите функцию `getAt`, которая принимает индекс и возвращает соответствующий элемент. Функция не должна возбуждать исключение, если индекс окажется за границами списка.

Подсказка

На этот раз сначала напишем версию с явной рекурсией. Затем попробуем ответить на следующие вопросы:

1. Можно ли сделать то же самое с использованием операции свертки? Какая это должна быть свертка – слева или справа?
2. Почему версия с явной рекурсией лучше?
3. Есть ли более эффективная реализация?

Напомню: впервые с хвостовой рекурсией мы встретились в главе 4, а операцию свертки исследовали в главах 3 и 5.

Решение

Решение с явной рекурсией выглядит просто:

```
fun getAt(index: Int): Result<A> {
    tailrec fun <A> getAt(list: List<A>, index: Int): Result<A> =
        when (list) {
            Nil -> Result.failure("Dead code. Should never execute.")
            is Cons ->
                if (index == 0)
```

```
                Result(list.head)
            else
                getAt(list.tail, index - 1)
        }

    return if (index < 0 || index >= length())
        Result.failure("Index out of bound")
    else
        getAt(this, index)
}
```

Первым делом функция проверяет значение индекса, чтобы убедиться, что он не отрицательный и меньше длины списка. Если эти два условия не выполняются, она возвращает `Failure`. В противном случае вызывается вспомогательная функция для сорекурсивной обработки. Вспомогательная функция проверяет, равен ли индекс 0. Если равен, она возвращает голову списка. Иначе она рекурсивно вызывает себя, передавая хвост списка с уменьшенным значением индекса.

Вариант `Nil` во вспомогательной функции – это «мертвый» код. Если основная функция получит `Nil`, значение `index` в любом случае будет либо меньше 0, либо больше или равно длине списка (которая сама равна 0). Следовательно, хвост аргумента никогда не будет экземпляром `Nil`. В свете этого функцию можно переписать так:

```
fun getAt(index: Int): Result<A> {
    tailrec fun <A> getAt(list: Cons<A>, index: Int): Result<A> =
        if (index == 0)
            Result(list.head)
        else
            getAt(list.tail as Cons, index - 1)

    return if (index < 0 || index >= length())
        Result.failure("Index out of bound")
    else
        getAt(this as Cons, index)
}
```

Также у кого-то из вас может появиться соблазн использовать функцию `let` из стандартной библиотеки:

```
fun getAt(index: Int): Result<A> {
    tailrec fun <A> getAt(list: List<A>, index: Int): Result<A> = // Предупреждение
        (list as Cons).let {
            if (index == 0)
                Result(list.head)
            else
                getAt(list.tail, index - 1)
        }

    return if (index < 0 || index >= length())
        Result.failure("Index out of bound")
    else
        getAt(this, index)
}
```

Но эта версия будет скомпилирована с предупреждением, потому что рекурсия во вспомогательной функции перестала быть хвостовой. Как следствие, она способна вызвать переполнение стека.

Применение сорекурсии выглядит более удачным решением. Но можно ли вместо явной рекурсии использовать свертку как абстракцию этой самой рекурсии? Да, можно, и это должна быть свертка слева, правда, решение получится довольно замысловатым:

```
fun getAtViaFoldLeft(index: Int): Result<A> =
    Pair(Result.failure<A>("Index out of bound"), index).let {
        if (index < 0 || index >= length())
            it
        else
            foldLeft(it) { ta ->
                { a ->
                    if (ta.second < 0)
                        ta
                    else
                        Pair(Result(a), ta.second - 1)
                }
            }
    }.first
```

Сначала определяется единичное значение. Поскольку это значение должно содержать и результат, и индекс, оно имеет тип `Pair` и хранит экземпляр `Failure`. Далее следует проверить допустимость индекса. Если индекс имеет недопустимое значение, в качестве временного результата выбирается `it` (единичное значение). Иначе выполняется свертка слева с функцией, возвращающей либо уже вычисленный результат (`ta`), если индекс меньше 0, либо новый экземпляр `Success`. Это решение может показаться лучше, но у него есть два недостатка:

- оно менее понятно, хотя это довольно субъективно и решать вам;
- оно менее эффективно, потому что свертка продолжается до конца списка даже после достижения элемента с искомым индексом.

Упражнение 8.13 (сложное)

Найдите решение, в котором версия на основе свертки завершается сразу после достижения искомого элемента.

Подсказка

Для этого потребуется специальная версия `foldLeft`, а также специальная версия `Pair`.

Решение

Прежде всего, нам нужна специальная версия `foldLeft`, которая прекращает выполнение, встретив поглощающий (нулевой) элемент для операции свертки. Вообразите список целых чисел, который нужно свернуть с использованием операции умножения. Поглощающим элементом для умножения является число 0, потому что умножение любого числа на 0

дает 0. Вот как выглядит определение версии foldLeft в классе List, выполняющей вычисления *по короткой схеме*:

```
abstract fun <B> foldLeft(identity: B, zero: B, f: (B) -> (A) -> B): B
```

Нулевой элемент

Поглощающий элемент операции часто называют еще *нулевым* элементом, хотя он не всегда выражается в виде числа 0. Число 0 является поглощающим элементом только для операции умножения. Для операции сложения положительных целых чисел поглощающим элементом является бесконечность.

Реализация в классе Nil возвращает параметр identity:

```
override fun <B> foldLeft(identity: B,
                          zero: B,
                          f: (B) -> (Nothing) -> B): B =
    identity
```

А вот как выглядит реализация в классе Cons:

```
override fun <B> foldLeft(identity: B, zero: B, f: (B) -> (A) -> B): B
{
    fun <B> foldLeft(acc: B,
                     zero: B,
                     list: List<A>, f: (B) -> (A) -> B): B = when
        (list) {
            Nil -> acc
            is Cons ->
                if (acc == zero)
                    acc
                else
                    foldLeft(f(acc)(list.head), zero, list.tail, f)
        }

    return foldLeft(identity, zero, this, f)
}
```

Как видите, эта версия прерывает рекурсию и возвращает значение аккумулятора, как только оно станет равно нулевому значению (zero). Теперь нужно определить нулевое значение для нашей свертки.

Роль нулевого значения в нашем случае может сыграть пара Pair<Result<A>, Int>, в которой элемент Int равен -1 (первое значение меньше 0). Можно ли для этой цели использовать стандартный тип Pair? Увы, нет, потому что он должен иметь особую функцию equals, которая возвращает true при равенстве целочисленных элементов пар независимо от значения элемента Result<A>. Учитывая все вышесказанное, требуемую функцию можно реализовать так:

```
fun getAt(index: Int): Result<A> {
    data class Pair<out A>(val first: Result<A>, val second: Int) {
        override fun equals(other: Any?): Boolean = when {
            other == null -> false
            other.javaClass == this.javaClass ->
```

```
                (other as Pair<A>).second == second
            else -> false
        }

        override fun hashCode(): Int =
            first.hashCode() + second.hashCode()
    }

    return Pair<A>(Result.failure("Index out of bound"), index)
        .let { identity ->
            Pair<A>(Result.failure("Index out of bound"), -1).let {zero ->
                if (index < 0 || index >= length())
                    identity
                else
                    foldLeft(identity, zero) { ta: Pair<A> ->
                        { a: A ->
                            if (ta.second < 0)
                                ta
                            else
                                Pair(Result(a), ta.second - 1)
                        }
                    }
            }
        }.first
}
```

Теперь свертка будет останавливается сразу после обнаружения искомого элемента. Новую функцию `foldLeft` можно использовать для прерывания любых вычислений с нулевым элементом. (Помните: *нулевое значение* – это не 0.) Кроме того, вместо нулевого значения можно использовать предикат и заставить функцию прервать рекурсию, когда предикат вернет `true`:

```
abstract fun <B> foldLeft(acc: B, p: (B) -> Boolean, f: (B) -> (A) -> B): B
```

Реализация в `Nil`, как и прежде, должна вернуть `identity`:

```
override fun <B> foldLeft(identity: B,
                          p: (B) -> Boolean,
                          f: (B) -> (Nothing) -> B): B = identity
```

Реализация в `Cons` напоминает предыдущую, только вместо проверки равенства `acc` значению `zero` она вызывает предикат с аккумулятором `acc`:

```
override fun <B> foldLeft(identity: B,
                          p: (B) -> Boolean,
                          f: (B) -> (A) -> B): B {
    fun <B> foldLeft(acc: B,
                     p: (B) -> Boolean,
                     list: List<A>): B = when (list) {
        Nil -> acc
        is Cons ->
            if (p(acc))
                acc
            else
```

```
                foldLeft(f(acc)(list.head), p, list.tail, f)
        }

        return foldLeft(identity, p, this)
    }
```

А вот реализация `getAt`, использующая эту версию `foldLeft`:

```
fun getAt(index: Int): Result<A> {
    val p: (Pair<Result<A>, Int>) -> Boolean = { it.second < 0 }
    return Pair<Result<A>, Int>(Result.failure("Index out of bound"), index)
        .let { identity ->
            if (index < 0 || index >= length())
                identity
            else
                foldLeft(identity, p) { ta: Pair<Result<A>, Int> ->
                    { a: A ->
                        if (p(ta))
                            ta
                        else
                            Pair(Result(a), ta.second - 1)
                    }
                }
    }.first
}
```

8.5.3 Разбиение списков

Иногда требуется разбить список на две части в определенной позиции. Хотя односвязный список далеко не идеален для подобных операций, такое разбиение реализуется относительно просто. Разбиение списка имеет несколько полезных применений, среди которых параллельная обработка его частей с использованием нескольких потоков.

Упражнение 8.14

Напишите функцию `splitAt`, которая принимает целое число и возвращает два списка, разбив исходный список в указанной позиции. Она не должна возбуждать исключение `IndexOutOfBoundException`. Вместо этого индекс со значением меньше 0 должен интерпретироваться как 0, а индекс со значением больше максимального индекса должен интерпретироваться как максимальный индекс.

Подсказка

Используйте явную рекурсию.

Решение

Рекурсивное решение выглядит просто:

```
fun splitAt(index: Int): Pair<List<A>, List<A>> {
    tailrec fun splitAt(acc: List<A>,
                        list: List<A>, i: Int): Pair<List<A>, List<A>> =
        when (list) {
```

```
            Nil -> Pair(list.reverse(), acc)
            is Cons -> if (i == 0)
                Pair(list.reverse(), acc)
            else
                splitAt(acc.cons(list.head), list.tail, i - 1)
        }

    return when {
        index < 0        -> splitAt(0)
        index > length() -> splitAt(length())
        else             ->
            splitAt(Nil, this.reverse(), this.length() - index)
    }
}
```


Основная функция использует рекурсию для корректировки значения индекса, правда, она выполнит рекурсивный вызов не более одного раза. Вспомогательная функция похожа на функцию `getAt`, но сначала переворачивает список. Функция накапливает элементы, пока не достигнет позиции индекса, поэтому накопленный список будет содержать элементы в правильном порядке, но оставшуюся часть списка необходимо перевернуть обратно.

Упражнение 8.15 (менее сложное, если вы решили упражнение 8.13)

Попробуйте придумать реализацию, использующую свертку вместо явной рекурсии.

Подсказка

Реализацию, выполняющую обход всего списка, написать легко. Реализацию, выполняющую обход только до указанного индекса, написать сложнее. Для этого нужна новая специальная версия `foldLeft`, действующая по короткой схеме и возвращающая как первую, так и вторую часть списка.

Решение

Вот как можно реализовать функцию, выполняющую обход всего списка:

```
fun splitAt(index: Int): Pair<List<A>, List<A>> {
    val ii = if (index < 0) 0
             else if (index >= length()) length() else index
    val identity = Triple(Nil, Nil, ii)
    val rt = foldLeft(identity) { ta: Triple<List<A>, List<A>, Int> ->
        { a: A ->
            if (ta.third == 0)
                Triple(ta.first, ta.second.cons(a), ta.third)
            else
                Triple(ta.first.cons(a), ta.second, ta.third - 1)
        }
    }

    return Pair(rt.first.reverse(), rt.second.reverse())
}
```

Результат свертки сначала накапливается в первом списке аккумулятора, а после достижения заданного индекса (с учетом его корректировки, чтобы избежать выхода за границы списка) обход продолжается, но оставшиеся элементы накапливаются уже во втором списке аккумулятора.

Одна из проблем этой реализации заключается в том, что второй список с накопленными оставшимися значениями нужно перевернуть. То есть функция не только выполняет обход оставшейся части списка, но делает это дважды: один раз, чтобы накопить элементы в обратном порядке, и второй – чтобы получить окончательный результат.

Во избежание этой проблемы можно изменить специальную версию `foldLeft` и заставить ее возвращать не только результат, полученный по короткой схеме (поглощающий, или нулевой, элемент), но и остальную часть списка в нетронутом виде. Для этого функция должна возвращать `Pair`:

```
abstract fun <B> foldLeft(identity: B, zero: B,
                          f: (B) -> (A) -> B): Pair<B, List<A>>
```

Соответственно, нужно изменить реализацию в классе `Nil`:

```
override
fun <B> foldLeft(identity: B, zero: B, f: (B) -> (Nothing) -> B):
    Pair<B, List<Nothing>> = Pair(identity, Nil)
```

и изменить реализацию в `Cons`, добавив возврат остальной части списка:

```
override fun <B> foldLeft(identity: B, zero: B, f: (B) -> (A) -> B):
                    Pair<B, List<A>> {
    fun <B> foldLeft(acc: B, zero: B, list: List<A>, f: (B) -> (A) -> B):
            Pair<B, List<A>> =
        when (list) {
            Nil -> Pair(acc, list)
            is Cons -> if (acc == zero)
                Pair(acc, list)
            else
                foldLeft(f(acc)(list.head), zero, list.tail, f)
        }

    return foldLeft(identity, zero, this, f)
}
```

Теперь можно переписать `splitAt`, задействовав в ней новую версию `foldLeft`:

```
fun splitAt(index: Int): Pair<List<A>, List<A>> {

    data class Pair<out A>(val first: List<A>, val second: Int) {
        override fun equals(other: Any?): Boolean = when {
            other == null -> false
            other.javaClass == this.javaClass ->
                                (other as Pair<A>).second == second
            else -> false
        }
```

```
        override fun hashCode(): Int =
            first.hashCode() + second.hashCode()
    }

    return when {
        index <= 0 -> Pair(Nil, this)
        index >= length -> Pair(this, Nil)
        else -> {
            val identity = Pair(Nil as List<A>, -1)
            val zero = Pair(this, index)
            val (pair, list) = this.foldLeft(identity, zero) { acc ->
                { e -> Pair(acc.first.cons(e), acc.second + 1) }
            }
            Pair(pair.first.reverse(), list)
        }
    }
}
```

И снова нам потребовалась специальная версия класса `Pair` с особой функцией `equals`, которая сравнивает только вторые элементы в парах. Обратите внимание, что второй список в результате не потребовалось переворачивать.

Когда не следует использовать свертку

Возможность использовать свертку совсем не означает, что вы должны делать это. Предыдущие упражнения – это просто упражнения. Занимаясь разработкой библиотек, вы должны выбирать наиболее оптимальную реализацию.

Хорошая библиотека должна иметь функциональный интерфейс и соответствовать требованиям безопасного программирования. То есть все функции должны быть истинными функциями без побочных эффектов и сохранять ссылочную прозрачность. Происходящее внутри библиотеки не имеет значения.

Функциональную библиотеку в императивном окружении, таком как JVM, можно сравнить с компилятором функционального языка. Скомпилированный код всегда будет основан на изменяющихся областях памяти и регистрах, потому что именно так действует компьютер.

Функциональная библиотека дает больше возможностей. Одни функции могут быть реализованы в функциональном стиле, другие – в императивном; это не имеет значения. Разбиение односвязного списка или поиск элемента по индексу работают намного быстрее, если реализованы императивно, а не функционально, потому что односвязный список неприспособлен для таких операций.

Функциональный подход не в том, чтобы реализовывать функции, использующие индексы, на основе свертки, а в том, чтобы вообще избегать таких функций. Если вам нужны структуры с такими функциями, лучше создать специализированные структуры, как будет показано в главе 10.

8.5.4 Поиск подсписков

Часто при работе со списками требуется выяснить, содержится ли какой-то список в другом (более длинном) списке, т. е. является ли список подсписком другого списка.

Упражнение 8.16

Напишите функцию `hasSubList`, которая проверяет, является ли список подсписком другого списка. Например, список `(3, 4, 5)` является подсписком списка `(1, 2, 3, 4, 5)`, но не списка `(1, 2, 4, 5, 6)`. Вот сигнатура этой функции:

```
fun hasSubList(sub: List<@UnsafeVariance A>): Boolean
```

Подсказка

Сначала реализуйте функцию `startsWith`, чтобы определить, не начинается ли список с искомого подсписка. После этого вы сможете реализовать рекурсивное применение этой функции ко всем элементам, начиная с первого.

Решение

Рекурсивную версию `startsWith` можно реализовать в классе `List`, но при этом важно не забыть запретить проверку вариантности параметра:

```
fun startsWith(sub: List<@UnsafeVariance A>): Boolean {
    tailrec fun startsWith(list: List<A>, sub: List<A>): Boolean =
        when (sub) {
            Nil -> true
            is Cons -> when (list) {
                Nil -> false
                is Cons -> if (list.head == sub.head)
                    startsWith(list.tail, sub.tail)
                else
                    false
            }
        }

    return startsWith(this, sub)
}
```

После этого реализация `hasSubList` выглядит очень просто:

```
fun hasSubList(sub: List<@UnsafeVariance A>): Boolean {
    tailrec
    fun <A> hasSubList(list: List<A>, sub: List<A>): Boolean =
        when (list) {
            Nil -> sub.isEmpty()
            is Cons ->
                if (list.startsWith(sub))
                    true
                else
                    hasSubList(list.tail, sub)
```

```
        }

    return hasSubList(this, sub)
}
```

8.5.5 Разные функции для работы со списками

Можно придумать множество других полезных функций для работы со списками. Следующие упражнения дадут вам некоторую практику в этой области. Предлагаемые решения – не единственно возможные. Не стесняйтесь изобретать свои собственные.

Упражнение 8.17

Напишите функцию `groupBy` со следующими характеристиками:

- принимает функцию `(A) -> B`;
- возвращает ассоциативный массив `Map`, ключи в котором являются результатами применения указанной функции к каждому элементу списка, а значения – списками элементов, соответствующих каждому ключу.

Например, пусть есть список экземпляров `Payment`, таких как:

```
data class Payment(val name: String, val amount: Int)
```

Следующий код должен создать `Map` с парами ключ/значение, где каждый ключ – это имя `name` человека, выполнившего платеж, а его значение – список с экземпляров `Payment`, представляющих эти платежи:

```
val map: Map<String, List<Payment>> = list.groupBy { x -> x.name }
```

Подсказка

Используйте неизменяемую версию `Map<B, List<A>>`. Прежде чем добавить новое значение, нужно проверить, присутствует ли соответствующий ключ в массиве. Если ключ присутствует, можно добавить значение в соответствующий ему список. Иначе нужно создать новый ключ, использовав имя из одноэлементного списка, содержащего добавляемое значение. Это легко реализовать с помощью функции `getOrDefault` из класса `Map`.

Решение

Ниже показана императивная версия. Мне нечего сказать о ней, потому что это самый обычный традиционный императивный код с изменяемым локальным состоянием. Если потребуется сохранить порядок следования элементов в подсписках, список сначала нужно перевернуть:

```
fun <B> groupBy(f: (A) -> B): Map<B, List<A>> =
    reverse().foldLeft(mapOf()) { mt: Map<B, List<A>> ->
        { t ->
            val key = f(t)
            mt + (key to (mt[key] ?: Nil).cons(t))
        }
    }
```

Здесь применяется оператор «Элвис» (`?:`), потому что используется неизменяемый ассоциативный массив, возвращающий тип с поддержкой `null`. Использование внутренних типов, поддерживающих `null`, вполне приемлемо для безопасного программирования, если не позволять этим типам просачиваться наружу из ваших функций. Здесь такое просачивание было допущено.

Намного лучше было бы, если бы в отсутствие искомого ключа ассоциативный массив возвращал `Result.Empty` вместо `null`. Впрочем, также можно использовать функцию `getOrDefault`:

```
fun <B> groupBy(f: (A) -> B): Map<B, List<A>> =
    reverse().foldLeft(mapOf()) { mt: Map<B, List<A>> ->
    { t ->
        val k = f(t)
        mt + (k to (mt.getOrDefault(k, Nil)).cons(t))
    }
}
```

Более идиоматично было бы использовать функцию `let` из стандартной библиотеки Kotlin:

```
fun <B> groupBy(f: (A) -> B): Map<B, List<A>> =
    reverse().foldLeft(mapOf()) { mt: Map<B, List<A>> ->
        { t ->
            f(t).let {
                mt + (it to (mt.getOrDefault(it, Nil)).cons(t))
            }
        }
    }
```

Но в этом мало проку, потому что решается проблема внутри функции, где это не требуется, а не снаружи, где вызывающий код все еще может использовать обычный способ доступа и получать `null`. В главе 11 вы узнаете, как создать свой неизменяемый ассоциативный массив, исправляющий эту проблему.

На переворачивание списка в самом начале тратится определенное время, поэтому вы можете предпочесть использовать `foldRight` вместо `foldLeft`, правда, при этом возникает опасность переполнения стека. Вот решение с `foldRight`:

```
fun <B> groupBy(f: (A) -> B): Map<B, List<A>> =
    foldRight(mapOf()) { t ->
        { mt: Map<B, List<A>> ->
            f(t).let { mt + (it to (mt.getOrDefault(it, Nil)).cons(t)) }
        }
    }
```

Упражнение 8.18

Напишите функцию `unfold`, которая принимает начальный элемент `S` и функцию `f: (S) -> Option<Pair<A, S>>` и возвращает список `List<A>`, получаемый в результате последовательного применения `f` к значению `S`,

пока в результате получается значение `Some`. Например, следующий код должен вернуть список с числами от 0 до 9:

```
unfold(0) { i ->
    if (i < 10)
        Option(Pair(i, i + 1))
    else
        Option()
}
```

Решение

Простая и небезопасная рекурсивная версия реализуется просто:

```
fun <A, S> unfold_(z: S, f: (S) -> Option<Pair<A, S>>): List<A> =
    f(z).map({ x ->
                unfold_(x.second, f).cons(x.first)
            }).getOrElse(List.Nil)
```

К сожалению, несмотря на красоту решения, эта версия вызывает переполнение стека, успевая выполнить чуть больше 1000 шагов рекурсии. Чтобы решить эту проблему, ее нужно преобразовать в сорекурсивную версию:

```
fun <A, S> unfold(z: S, getNext: (S) -> Option<Pair<A, S>>): List<A> {
    tailrec fun unfold(acc: List<A>, z: S): List<A> {
        val next = getNext(z)
        return when (next) {
            Option.None -> acc
            is Option.Some ->
                unfold(acc.cons(next.value.first), next.value.second)
        }
    }

    return unfold(List.Nil, z).reverse()
}
```

Однако сорекурсивная версия требует переворачивать результат. Для небольших списков это несущественно, но может вызвать значительные затраты, если число элементов очень велико.

Эта реализация требует, чтобы класс `Option` находился в том же модуле с классом `List`. Под модулем в данном случае подразумевается:

- модуль IntelliJ IDEA;
- проект Maven;
- набор исходных файлов Gradle;
- набор файлов, компилируемый в одном вызове задачи Ant.

Для этого упражнения класс `Option` был скопирован из модуля `common` в модуль `advancedlisthandling`. В реальной жизни проще было бы сделать обратное – скопировать класс `List` в модуль `common`.

Вот пример использования `unfold`:

```
fun main(args: Array<String>) {
    val f: (Int) -> Option<Pair<Int, Int>> =
```

```
        { it ->
            if (it < 10_000) Option(Pair(it, it + 1)) else Option()
        }
    val result = unfold(0, f)
    println(result)
}
```

Здесь `unfold` генерирует значения, пока функция `next` не вернет `None`. Если вам понадобится использовать функцию, которая может сгенерировать ошибку, используйте класс `Result` вместо `Option`:

```
fun <A, S> unfold(z: S,
                  getNext: (S) -> Result<Pair<A, S>>): Result<List<A>>
{
    tailrec fun unfold(acc: List<A>, z: S): Result<List<A>> {
        val next = getNext(z)
        return when (next) {
            Result.Empty -> Result(acc)
            is Result.Failure -> Result.failure(next.exception)
            is Result.Success ->
                unfold(acc.cons(next.value.first), next.value.second)
        }
    }

    return unfold(List.Nil, z).map(List<A>::reverse)
}
```

Упражнение 8.19

Напишите функцию `range`, которая принимает два целых числа и возвращает список всех целых чисел, которые больше или равны первому и меньше второго.

Подсказка

Используйте функции, написанные выше.

Решение

Это упражнение решается легко, если использовать функцию из упражнения 8.18:

```
fun range(start: Int, end: Int): List<Int> {
    return unfold(start) { i ->
        if (i < end)
            Option(Pair(i, i + 1))
        else
            Option()
    }
}
```

Упражнение 8.20

Напишите функцию `exists`, которая принимает функцию `(A) -> Boolean`, представляющую условие, и возвращает `true`, если хотя бы один элемент в списке удовлетворяет этому условию. Вместо явной рекурсии используйте функции, написанные выше.

Подсказка

Не нужно подвергать проверке все элементы списка. Функция должна завершаться при обнаружении первого же элемента, удовлетворяющего условию.

Решение

Рекурсивную функцию можно было бы определить так:

```
fun exists(p:(A) -> Boolean): Boolean =
    when (this) {
        Nil -> false
        is Cons -> p(head) || tail.exists(p)
    }
```

Так как оператор || выполняет вычисления по короткой схеме, рекурсивный процесс прекращается сразу, как только будет обнаружен первый элемент, удовлетворяющий условию, выраженному в виде предиката p.

Но эта не хвостовая рекурсия, и данная реализация может вызвать переполнение стека, если список окажется достаточно длинным и в первой тысяче элементов (или что-то около того) не окажется значения, удовлетворяющего условию. Кроме того, для пустого списка она возбудит исключение. Поэтому вы должны определить абстрактную функцию в классе List с конкретной реализацией в подклассе Nil. Лучшее решение дает применение функции foldLeft с нулевым параметром:

```
fun exists(p: (A) -> Boolean): Boolean =
    foldLeft(false, true) { x -> { y: A -> x || p(y) } }.first
```

Упражнение 8.21

Напишите функцию forAll, которая принимает функцию (A) -> Boolean, представляющую условие, и возвращает true, если все элементы списка удовлетворяют условию.

Подсказка

Вместо явной рекурсии используйте функции, написанные выше. И снова – она не всегда должна проверять все элементы списка. Функция forAll должна получиться похожей на функцию exists.

Решение

Решение очень похоже на реализацию функции exists, но имеет два отличия: единичное и нулевое значения меняются местами и вместо оператора || используется оператор &&:

```
fun forAll(p: (A) -> Boolean): Boolean =
    foldLeft(true, false) { x -> { y: A -> x && p(y) } }.first
```

Другое возможное решение основано на использовании функции exists:

```
fun forAll(p: (A) -> Boolean): Boolean = !exists { !p(it) }
```

Эта версия проверяет присутствие в списке элемента, не удовлетворяющего условию.

8.6 Автоматическое распараллеливание операций со списками

Большинство операций со списками можно реализовать с использованием свертки. Операция *свертки* предполагает выполнение операции столько раз, сколько элементов в списке. Для длинных списков и продолжительных операций на выполнение свертки может потребоваться значительное время. Поскольку в настоящее время большинство компьютеров оборудовано многоядерными процессорами (или даже несколькими процессорами), у вас может возникнуть желание найти способ для параллельной обработки списков.

Чтобы выполнить свертку параллельно, нужно только одно (кроме многоядерного процессора, конечно): реализовать дополнительную операцию, объединяющую результаты параллельных вычислений.

8.6.1 Не все вычисления могут выполняться параллельно

Используем в качестве примера список целых чисел. У вас не получится напрямую распараллелить вычисление среднего значения по всем целым числам в списке. Можно разбить список на четыре части (если компьютер оборудован четырьмя процессорами) и найти среднее значение для каждой. Но нельзя вычислить среднее значение для всего списка по средним значениям подсписков.

С другой стороны, поиск среднего значения подразумевает вычисление суммы всех элементов с последующим делением на их количество. А вычисление суммы легко распараллелить: вычислить суммы для подсписков, а затем найти общую сумму.

Это частный случай, в котором для свертки подсписков и объединения промежуточных результатов используется одна и та же операция (сложение). Однако так бывает не всегда. Возьмем, к примеру, список символов, в процессе свертки которого символы последовательно добавляются в строку. Чтобы объединить промежуточные результаты, нужна другая операция: конкатенация строк.

8.6.2 Деление списка на подсписки

Прежде всего, мы должны разделить список на подсписки, и эта операция должна выполняться автоматически. Возникает важный вопрос – сколько подсписков должно получиться на выходе? Идеальным может показаться, если число подсписков будет равно числу процессоров (ядер), но это не совсем так. Количество процессоров (или, точнее, количество логических ядер) – не самый важный критерий.

Намного важнее, чтобы обработка всех подсписков занимала одинаковое время. Но это во многом зависит от типа вычислений. Можно раз-

делить список на четыре подсписка, решив выделить четыре потока для параллельной обработки, но тогда некоторые потоки могут быстро завершиться, а другим придется выполнять гораздо более длительные вычисления. Это разрушит преимущество распараллеливания, потому что большая часть вычислений будет производиться одним потоком.

Намного лучше было бы разделить список на большое количество подсписков, а затем отправить эти подсписки в пул потоков. При таком подходе как только поток завершит обработку подсписка, он получит новый для обработки. Итак, наша первая задача – создать функцию, которая делит список на подсписки.

Упражнение 8.22

Напишите функцию `divide(depth: Int)`, которая делит список на множество подсписков. Она должна разделить список на два подсписка, затем рекурсивно разделить каждый подсписок еще на два подсписка и т. д. по числу в параметре `depth`. Эта функция должна быть реализована в классе `List` и иметь следующую сигнатуру:

```
fun divide(depth: Int): List<List<A>>
```

Подсказка

Сначала определите новую версию функции `splitAt`, возвращающую список списков вместо `Pair<List, List>`. Назовите ее `splitListAt`. Вот сигнатура этой новой функции:

```
fun splitListAt(index: Int): List<List<A>>
```

Решение

Функция `splitListAt` – это немного измененная версия функции `splitAt`:

```
fun splitListAt(index: Int): List<List<A>> {
    tailrec fun splitListAt(acc: List<A>,
                            list: List<A>, i: Int): List<List<A>> =
        when (list) {
            Nil -> List(list.reverse(), acc)
            is Cons -> if (i == 0)
                List(list.reverse(), acc)
            else
                splitListAt(acc.cons(list.head), list.tail, i - 1)
        }

    return when {
        index < 0        -> splitListAt(0)
        index > length() -> splitListAt(length())
        else             ->
            splitListAt(Nil, this.reverse(), this.length() - index)
    }
}
```

Эта функция всегда возвращает список с двумя списками. Теперь можно определить функцию `divide`:

```
fun divide(depth: Int): List<List<A>> {
    tailrec
    fun divide(list: List<List<A>>, depth: Int): List<List<A>> =
        when (list) {
            Nil -> list // мертвый код
            is Cons ->
                if (list.head.length() < 2 || depth < 1)
                    list
                else
                    divide(list.flatMap { x ->
                        x.splitListAt(x.length() / 2)
                    }, depth - 1)
        }

    return if (this.isEmpty())
        List(this)
    else
        divide(List(this), depth)
}
```

Случай `Nil` в выражении `when` – это «мертвый» код, потому что локальная функция никогда не будет вызываться со значением `Nil` в параметре. Далее можно использовать явное приведение к типу `Cons`. Также обратите внимание, что ключевое слово `tailrec` можно опустить, потому что количество шагов рекурсии не превысит *log (длина_списка)*. Вы не сможете уместить в куче список достаточно длинный, чтобы вызвать переполнение стека.

8.6.3 Параллельная обработка подсписков

Для параллельной обработки подсписков нужна специальная версия функции. Эта специальная версия будет принимать дополнительный параметр `ExecutorService`, представляющий пул потоков, которые будут задействованы для параллельной обработки.

Упражнение 8.23

Напишите функцию `parFoldLeft` в классе `List<A>`, которая принимает те же параметры, что и `foldLeft` плюс `ExecutorService` и функцию `(B) -> (B) -> B`, и возвращает `Result<B>`. Дополнительная функция будет использоваться для объединения результатов обработки подсписков. Вот сигнатура этой функции:

```
fun <B> parFoldLeft(es: ExecutorService,
                    identity: B,
                    f: (B) -> (A) -> B,
                    m: (B) -> (B) -> B): Result<B>
```

Решение

Сначала определим желаемое число подсписков и разделим исходный список:

```
divide(1024)
```

Затем разделим список на подсписки с помощью функции, которая сразу будет отправлять задания в `ExecutorService`. Каждое задание выполняет свертку подсписка и возвращает экземпляр `Future`. Список экземпляров `Future` передается в функцию, вызывающую функцию `get` каждого экземпляра `Future` и создает список результатов (по одному результату для каждого подсписка). Мы должны перехватить возможные исключения. Наконец, список результатов объединяется с помощью второй функции и в случае успеха возвращается `Success`. Иначе возвращается `Failure`:

```
fun <B> parFoldLeft(es: ExecutorService,
                    identity: B,
                    f: (B) -> (A) -> B,
                    m: (B) -> (B) -> B): Result<B> =
    try {
        val result: List<B> = divide(1024).map { list: List<A> ->
            es.submit<B> { list.foldLeft(identity, f) }
        }.map<B> { fb ->
            try {
                fb.get()
            } catch (e: InterruptedException) {
                throw RuntimeException(e)
            } catch (e: ExecutionException) {
                throw RuntimeException(e)
            }
        }
        Result(result.foldLeft(identity, m))
    } catch (e: Exception) {
        Result.failure(e)
    }
```

Реализацию теста производительности этой функции вы найдете в примерах к книге (https://github.com/pysaumont/fpinkotlin). Тест десять раз вычисляет значение Фибоначчи для 35 000 случайных чисел в диапазоне от 1 до 30 с использованием медленного алгоритма. На восьмиядерном компьютере мы получили следующие результаты:

```
Duration serial 1 thread: 140933
Duration parallel 2 threads: 70502
Duration parallel 4 threads: 36337
Duration parallel 8 threads: 20253
```

Упражнение 8.24

В предыдущем упражнении, чтобы организовать автоматическое распараллеливание, мы делили список на подсписки с помощью свертки, но параллельные вычисления можно организовать и без свертки. Это, пожалуй, самый простой способ автоматического распараллеливания операций со списками.

Напишите функцию `parMap`, которая автоматически применяет указанную функцию к каждому элементу списка параллельно. Вот сигнатура этой функции:

```
fun <B> parMap(es: ExecutorService, g: (A) -> B): Result<List<B>>
```

Подсказка

В этом упражнении почти ничего не нужно придумывать. Достаточно просто передать каждый элемент и функцию в `ExecutorService` и получить результаты.

Решение

Вот как выглядит решение этого упражнения:

```
fun <B> parMap(es: ExecutorService, g: (A) -> B): Result<List<B>> =
    try {
        val result = this.map { x ->
            es.submit<B> { g(x) }
        }.map<B> { fb ->
            try {
                fb.get()
            } catch (e: InterruptedException) {
                throw RuntimeException(e)
            } catch (e: ExecutionException) {
                throw RuntimeException(e)
            }
        }
        Result(result)
    } catch (e: Exception) {
        Result.failure(e)
    }
```

Реализацию теста производительности этой функции вы найдете в примерах к книге. С его помощью вы сможете определить, насколько производительнее работает эта версия. Но имейте в виду, что прирост производительности во многом зависит от технических возможностей компьютера, на котором выполняется программа.

Эта параллельная версия создает отдельное задание для каждого элемента списка. Она действует эффективнее для коротких списков и продолжительных вычислений. Для длинных списков и быстрых вычислений прирост производительности может оказаться незначительным или даже отрицательным.

Итоги

- Мемоизация позволяет ускорить обработку списков.
- Список `List` экземпляров `Result` можно преобразовать в экземпляр `Result` со списком `List`.
- Два списка можно упаковать в один. Также списки можно распаковывать и получать пары списков.

- Используя явную рекурсию, можно организовать доступ к элементам списка по индексам.
- Можно написать специальную версию `foldLeft`, которая прекращает свертку по достижении нулевого значения.
- Можно реализовать развертывание списков, используя функцию преобразования и конечное условие.
- Списки можно автоматически делить на подсписки и обрабатывать их параллельно.

9 Ленивые вычисления

Эта глава охватывает следующие темы:

- важность понимания ленивых вычислений;
- реализация ленивых вычислений в Kotlin;
- комбинирование ленивых вычислений;
- создание ленивого списка;
- обработка бесконечных потоков данных.

Представьте, что вы бармен в небольшом баре, где посетителям предлагается всего два продукта: кофе эспрессо и апельсиновый сок. Клиент входит в бар и садится за стойку. У вас есть два варианта:

1. Приготовить эспрессо, отжать порцию апельсинового сока, подать и то, и другое и позволить клиенту сделать выбор.
2. Спросить у клиента, что он желает, и подать требуемый напиток.

В программировании второй подход называют *ленивым*, а первый – *строгим*. Какая стратегия лучше – выбирать вам. Эта глава рассказывает не о морали, а об эффективности.

В примере выше возможны и другие варианты. Вы могли бы приготовить эспрессо и апельсиновый сок, до того как войдет посетитель. Как только он войдет, вы сможете спросить, чего он хочет, немедленно подать нужный напиток и затем приготовить новую порцию для следующего возможного клиента. Такое решение может показаться странным, но это не так.

Теперь представьте, что кроме напитков вы также предлагаете яблочный и клубничный пироги. Стали бы вы ждать, пока придет клиент и вы-

берет тот или иной пирог, чтобы приготовить его? Или вы приготовили бы по одному пирогу заранее, чтобы не заставлять клиента ждать? Выбрав ленивую стратегию, вы подождете, пока клиент закажет, скажем, яблочный пирог. Затем приготовите яблочный пирог, отрежете ломтик и подадите его посетителю. Это, безусловно, ленивый подход, но обратите внимание, что одновременно вы готовите несколько ломтиков яблочного пирога.

Языки программирования работают точно так же. И, как в примере выше, нормальный путь часто настолько очевиден, что никто даже не задумывается об этом, пока не возникает проблема. В первой части этой главы я сравню строгий и ленивый подходы. Kotlin – строгий язык, но он позволяет реализовать ленивый подход. Во второй части главы я покажу разные приемы использования ленивых (отложенных) вычислений на Kotlin: как реализовать ленивые вычисления, как их комбинировать, как создать ленивый список и, наконец, как обрабатывать бесконечные потоки данных.

9.1 Строгий и ленивый подходы

О некоторых языках говорят, что они ленивые; другие называют строгими. В таких случаях речь обычно идет о том, как языки вычисляют аргументы функции или метода. Но на самом деле все языки ленивы, потому что лень – это истинная сущность программирования.

Программирование заключается в комбинировании программных инструкций, которые будут производить вычисления после запуска программы. Если бы языки были по-настоящему строгими, каждая отдельная инструкция в программе была бы выполнена немедленно, сразу после нажатия клавиши **Enter**, что, собственно, и происходит при использовании оболочки REPL (Read, Eval, Print Loop – цикл чтения, выполнения и вывода результатов).

Не означает ли это, что программа, написанная на строгом языке, таком как Kotlin или Java, в виде комбинации строгих элементов, не может иметь ленивую конструкцию? Конечно, нет. Строгие языки строги в отношении вычисления аргументов метода/функции, но большинство других конструкций ленивы по своей природе. Возьмем для примера структуру `if...else`:

```
val result = if (testCondition())
        getIfTrue()
    else
        getIfFalse()
```

Очевидно, что функция `testCondition` будет выполняться всегда. Но в зависимости от результата, возвращаемого `testCondition`, вызываться будет только одна из функций – `getIfTrue` или `getIfFalse`. Структура `if...else` строго относится к условию, но она ленива в отношении своих ветвей. Если бы `getIfTrue` и `getIfFalse` были эффектами, вы не смогли бы сделать ничего полезного с абсолютно строгой структурой `if...else`, потому

что оба эффекта имели бы место в любом случае. Напротив, если бы это были чистые функции, это не изменило бы результат программы. Оба значения были бы вычислены, но в `result` попало бы только то, которое соответствует условию. Однако такой подход впустую потратил бы время на вычисление ненужного значения, хотя и оставался бы вполне пригодным для использования.

Строгость и ленивость относятся не только к управляющим структурам и аргументам функций, но и ко всему в программировании. Например, взгляните на следующее объявление:

```
val x: Int = 2 + 3
```

Здесь `x` вычисляется немедленно и получает значение `5`. Так как Kotlin – строгий язык, он сразу же выполнит сложение. А теперь взгляните на другой пример:

```
val x: Int = getValue()
```

В Kotlin функция `getValue` будет вызвана немедленно в точке объявления ссылки `x`. Но в ленивом языке, напротив, функция `getValue` будет вызываться, только когда программе потребуется использовать значение, на которое ссылается `x`. Это огромная разница. Например, взгляните на следующую программу:

```
fun main(args: Array<String>) {
    val x = getValue()
}

fun getValue(): Int {
    println("Returning 5")
    return 5
}
```

Эта программа выведет в консоль текст `Returning 5`, потому что функция `getValue` вызывается немедленно, хотя возвращаемое ею значение нигде не используется. В ленивом языке эта программа не выполнила бы никаких вычислений и ничего бы не вывела в консоль.

9.2 Строгие вычисления в Kotlin

Kotlin создавался как строгий язык. Вычисления в нем выполняются немедленно. Аргументы передаются в функции *по значению*, т. е. они сначала вычисляются, а затем в функцию передаются вычисленные значения. В ленивых языках, напротив, аргументы передаются *по имени*, т. е. не вычисляются в момент вызова. Пусть вас не смущает тот факт, что аргументы функций в Kotlin часто являются ссылками. Ссылки – это адреса, и эти адреса передаются по значению.

Одни языки программирования – строгие (как Kotlin и Java), другие – ленивые. Одни являются строгими по умолчанию и допускают возможность использовать ленивые вычисления, а другие – ленивыми по умолчанию и допускают возможность использовать строгие вычисле-

ния. Kotlin, однако, не всегда строг. Вот некоторые ленивые конструкции в Kotlin:

- логические операторы `||` и `&&`;
- `if ... else`;
- цикл `for`;
- цикл `while`.

Kotlin также предлагает ленивые структуры данных, такие как `Sequence`, способные лениво вычислять свои свойства, как будет показано ниже.

Рассмотрим логические операторы `||` и `&&`. Эти операторы не вычисляют свои операнды, если этого не требуется для вычисления результата. Если операнд слева от `||` имеет значение `true`, результат будет равен `true` независимо от значения второго операнда, поэтому нет смысла вычислять второй операнд. Аналогично, если операнд слева от `&&` имеет значение `false`, результат будет равен `false` независимо от значения второго операнда, а значит, нет смысла вычислять его.

Это ничего вам не напоминает? В главе 8 мы создали специальную версию функции `foldLeft`, которая прекращает рекурсию, обнаружив *поглощающий элемент* (также называемый *нулевым элементом*). Здесь `false` – поглощающий элемент для операции `&&`, а `true` – поглощающий элемент для операции `||`.

А теперь представьте, что мы решили сымитировать логические операторы в виде функций. Листинг 9.1 демонстрирует одну из возможных реализаций.

Листинг 9.1 Логические функции and и or

```
fun main(args: Array<String>) {
    println(or(true, true))
    println(or(true, false))
    println(or(false, true))
    println(or(false, false))
    println(and(true, true))
    println(and(true, false))
    println(and(false, true))
    println(and(false, false))
}

fun or(a: Boolean, b: Boolean): Boolean = if (a) true else b

fun and(a: Boolean, b: Boolean): Boolean = if (a) b else false
```

Логические операторы дают более простой способ сделать то же самое, но наша цель – избежать использования этих операторов. Как вы думаете, у нас это получилось? Запустив программу, получаем в консоли следующий вывод:

```
true
true
true
false
```

```
true
false
false
false
```

Вроде бы все неплохо. Но давайте теперь запустим программу в листинге 9.2.

Листинг 9.2 Проблема строгости

```
fun main(args: Array<String>) {
    println(getFirst() || getSecond())
    println(or(getFirst(), getSecond()))
}

fun getFirst(): Boolean = true

fun getSecond(): Boolean = throw IllegalStateException()

fun or(a: Boolean, b: Boolean): Boolean = if (a) true else b

fun and(a: Boolean, b: Boolean): Boolean = if (a) b else false
```

Эта программа выведет:

```
true
Exception in thread "main" java.lang.IllegalStateException
```

Очевидно, что функция `or` не эквивалентна оператору `||`. Разница в том, что `||` вычисляет свои операнды лениво, т. е. второй операнд не вычисляется, если первый имеет значение `true`, потому что этого не требуется для вычисления результата. Функция `or`, напротив, строго вычисляет свои аргументы, т. е. второй аргумент вычисляется, даже если его значение не требуется для получения результата, поэтому она всегда генерирует исключение `IllegalStateException`. В главах 6 и 7 мы столкнулись с этой проблемой в функции `getOrElse`, где ее аргумент вычислялся всегда, даже если этого не требовалось.

9.3 Ленивые вычисления в Kotlin

Ленивые вычисления необходимы во многих случаях. Фактически Kotlin использует их в реализации таких конструкций, как `if ... then`, циклы и блоки `try ... catch`. Без этого, например, блок `catch` будет вычисляться даже в отсутствие исключения.

Ленивая реализация является обязательным условием, когда речь идет об обработке ошибок, а также когда требуется манипулировать бесконечными структурами данных. Для реализации таких вычислений Kotlin предлагает использовать прием делегирования. Вот как это работает:

```
val first: Boolean by Delegate()
```

где `Delegate` – это класс, реализующий следующую функцию:

```
operator fun getValue(thisRef: Any?, property: KProperty<*>): Boolean
```

Обратите внимание на два важных обстоятельства:

1. Класс `Delegate`, который может иметь любое имя по вашему выбору, не должен реализовать какой-либо интерфейс. Он должен просто объявить и реализовать функцию `getValue` с сигнатурой, как показано выше, которая будет вызываться с использованием механизма рефлексии.
2. Если вместо `val` вам понадобится объявить `var`, тогда в классе `Delegate` придется также реализовать функцию для изменения значения:

```
operator fun setValue(thisRef: Any?, property: KProperty<*>, value: Boolean).
```

Kotlin также предлагает стандартные делегаты для реализации ленивых вычислений, в том числе и `Lazy`:

```
val first: Boolean by lazy { ... }
```

Это короткий вариант более развернутого объявления:

```
class Lazy {
    operator fun getValue(thisRef: Any?,
                          property: KProperty<*>): Boolean = ...
}

val first: Boolean by Lazy()
```

Стандартный интерфейс `Lazy` в языке Kotlin на самом деле немного сложнее. С его использованием предыдущий пример с логической функцией `or` можно реализовать так:

```
fun main(args: Array<String>) {
    val first: Boolean by lazy { true }
    val second: Boolean by lazy { throw IllegalStateException() }
    println(first || second)
    println(or(first, second))
}

fun or(a: Boolean, b: Boolean): Boolean = if (a) true else b
```

К сожалению, это не нестоящие ленивые вычисления, в чем можно убедиться, запустив программу:

```
true
Exception in thread "main" java.lang.IllegalStateException
```

Здесь функции, используемые для инициализации ссылок `first` и `second`, не вызываются в месте их объявления; эти вычисления действительно выполняются лениво (т. е. откладываются). Но они вызываются, когда функция `or` получает аргументы `first` и `second`, а не когда они фактически используются, а это уже совсем не ленивые вычисления, иначе исключение не возникло бы, потому что аргумент `second` нигде не используется!

9.4 Реализация ленивых вычислений

Реализовать ленивые вычисления в Kotlin, как это сделано в ленивых языках, невозможно. Но для достижения этой цели можно использовать правильные типы. Давайте вспомним, что мы делали в главах 5, 6 и 7. В главе 5 мы научились абстрагировать списки, объединив два взаимосвязанных понятия – тип элементов и еще нечто, представленное типом `List`. Это «нечто» является составным понятием, охватывающим мощность (список может содержать любое количество элементов, включая `0`, `1` или более), а также упорядоченность элементов. Мы могли бы развить абстракцию дальше и разделить понятия мощности и упорядоченности. Но мы решили думать о понятии `List` в целом и в результате получили параметризованный тип `List<A>`.

В главе 6 мы абстрагировали понятие необязательности, объединив два взаимосвязанных понятия: тип данных (пусть это будет `A`) и тип `Option`. И в главе 7 мы абстрагировали возможность ошибки в другой составной тип, `Result<A>`. Закономерность, как мне кажется, очевидна. С одной стороны, мы имеем простой тип `A`, а с другой – своего рода модель или приемы, применяемые к этому простому типу. Ленивые вычисления тоже можно считать моделью, а значит, их можно реализовать как тип. Давайте назовем это `Lazy<A>`.

Вы могли бы возразить, что ленивые вычисления можно представить намного проще, использовав константную функцию. Мы уже использовали эту идею в предыдущих главах, и теперь вы знаете, что константная функция – это функция, принимающая любой аргумент любого типа и всегда возвращающая одно и то же значение. Такая функция может иметь следующий тип:

```
() -> Int
```

Здесь константная функция возвращает `Int`. Создать ленивое целое значение можно так:

```
val x: () -> Int = { 5 }
```

Но в этом мало смысла, потому что 5 – это литерал и глупо использовать ленивую инициализацию ссылки на литеральное значение. Но вернемся к предыдущему примеру и используем ленивые логические значения:

```
fun main(args: Array<String>) {
    val first = { true }
    val second = { throw IllegalStateException() }
    println(first() || second())
    println(or(first, second))
}

fun or(a: () -> Boolean, b: () -> Boolean): Boolean =
    if (a()) true else b()
```

Теперь программа будет работать, как ожидалось, и выведет:

```
true
true
```

Одно из отличий нашей реализации от реализации в ленивых языках – мы изменили типы. Ленивый тип `A` – это не `A`, поэтому мы были вынуждены изменить функцию `or`, чтобы она соответствовала новым типам.

Но есть еще одно более существенное отличие от по-настоящему ленивых вычислений: если использовать значение дважды, функция будет вызвана дважды. Для вычисления значения требуется время, а такие избыточные вычисления будут впустую расходовать процессорное время. Это то, что мы называем *вызов при передаче по имени*, т. е. значение аргумента не вычисляется до того, как оно потребуется, но вычисляется каждый раз, когда оно необходимо! То же самое можно реализовать в виде функции `fun`, но при этом придется указать тип возвращаемого значения для функции, вызывающей исключение, потому что Kotlin не сможет сам определить его:

```
fun main(args: Array<String>) {
    fun first() = true
    fun second(): Boolean = throw IllegalStateException()
    println(first() || second())
    println(or(::first, ::second))
}

fun or(a: () -> Boolean, b: () -> Boolean): Boolean =
    if (a()) true else b()
```

Сможете ли вы придумать решение этой проблемы, чтобы функция вызывалась только в первый раз, когда это необходимо? Этот способ вычислений называется *вызовом по необходимости*. Если вспомнить, о чем рассказывалось в главе 4, вы легко найдете решение – оно заключается в мемоизации.

Упражнение 9.1

Реализуйте тип `Lazy<A>`, работающий подобно мемоизованной функции `() -> A`. Этот тип должен поддерживать следующий способ его использования:

```
fun main(args: Array<String>) {
    val first = Lazy {
        println("Evaluating first")
        true
    }

    val second = Lazy {
        println("Evaluating second")
        throw IllegalStateException()
    }
```

```
    println(first() || second())
    println(first() || second())
    println(or(first, second))
}

fun or(a: Lazy<Boolean>, b: Lazy<Boolean>): Boolean =
    if (a()) true else b()
```

Эта программа должна вывести следующее:

```
Evaluating first
true
true
true
```

Подсказка

Для данной задачи достаточно, чтобы тип `Lazy` просто расширял (наследовал) функцию `() -> A`. Это не единственное решение, но оно упрощает применение типа. Используйте конструкцию `by lazy` для мемоизации и избегайте явного изменения состояния.

Решение

Вот решение на основе конструкции `by lazy`:

```
class Lazy<out A>(function: () -> A): () -> A {

    private val value: A by lazy(function)

    operator override fun invoke(): A = value
}
```

Как видите, эта реализация не изменяет состояния. Изменение состояния абстрагировано в виде конструкции `by lazy`, предлагаемой языком Kotlin. В главе 14 вы узнаете, как тот же прием использовать для абстрагирования общего изменяемого состояния.

9.4.1 Комбинирование ленивых значений

В предыдущем примере мы комбинировали ленивые логические значения с помощью функции `or`:

```
fun or(a: Lazy<Boolean>, b: Lazy<Boolean>): Boolean =
    if (a()) true else b()
```

Но это было обманом, потому что мы полагались на выражение `if ... else`, чтобы избежать вычисления второго аргумента. Представьте функцию, комбинирующую две строки:

```
fun constructMessage(greetings: String,
                     name: String): String = "$greetings, $name!"
```

Теперь вообразите, что вычисление каждого параметра является ресурсоемкой операцией, и желательно получить результат `constructMessage`, применяя ленивый подход, чтобы его можно было использовать или не использовать в зависимости от внешнего условия. Если условие

не выполняется, строки `name` и `greetings` не должны вычисляться. Например, допустим, что условием является четность случайного целого числа. Вот как это должно работать:

```
val greetings = Lazy {
    println("Evaluating greetings")
    "Hello"
}

val name: Lazy<String> = Lazy {
    println("computing name")
    "Mickey"
}

val message = constructMessage(greetings, name)
val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0
println(if (condition) <скомбинировать и вывести сообщение>
        else "No greetings when time is odd")
```

Теперь нужно переписать функцию `constructMessage` так, чтобы она откладывала вычисление аргументов, например:

```
fun constructMessage(greetings: Lazy<String>,
                     name: Lazy<String>): String =
    "${greetings()}, ${name()}!"
```

Здесь нет никакой реальной пользы от применения ленивых значений, потому что они будут вычисляться до комбинирования, в момент вызова функции `constructMessage`, даже если условие не будет выполнено. Нам же желательно, чтобы функция `constructMessage` возвращала ленивый результат, т. е. `Lazy<String>`, без вычисления параметров.

Упражнение 9.2

Напишите ленивую версию функции `constructMessage`.

Подсказка

Каждая часть сообщения должна вычисляться только один раз независимо от количества вызовов функции.

Решение

Вот функция, которая выполняет ленивую конкатенацию двух аргументов, а также код, демонстрирующий, что вычисления выполняются только один раз:

```
fun constructMessage(greetings: Lazy<String>,
                     name: Lazy<String>): Lazy<String> =
    Lazy { "${greetings()}, ${name()}!" }

fun main(args: Array<String>) {
    val greetings = Lazy {
        println("Evaluating greetings")
        "Hello"
    }

    val name1: Lazy<String> = Lazy {
```

```
        println("Evaluating name")
        "Mickey"
    }

    val name2: Lazy<String> = Lazy {
        println("Evaluating name")
        "Donald"
    }

    val defaultMessage = Lazy {
        println("Evaluating default message")
        "No greetings when time is odd"
    }

    val message1 = constructMessage(greetings, name1)
    val message2 = constructMessage(greetings, name2)
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) message1() else defaultMessage())
    println(if (condition) message1() else defaultMessage())
    println(if (condition) message2() else defaultMessage())
}
```

Запустите программу несколько раз. Когда условие выполняется, она должна вывести:

```
Evaluating greetings
Evaluating name
Hello, Mickey!
Hello, Mickey!
Evaluating name
Hello, Donald!
```

Как видите, аргумент `greeting` вычисляется только один раз, а аргумент `name` – два раза (и получает два разных значения), даже притом что функция вызывается трижды.

> **ПРИМЕЧАНИЕ** Это не совсем так. С точки зрения функции аргументы вычисляются при каждом ее вызове как экземпляры `Lazy`, но только в первом вызове происходит дальнейшее вычисление экземпляров `Lazy`.

Когда условие не выполняется, программа выводит:

```
Evaluating default message
No greetings when time is odd
No greetings when time is odd
No greetings when time is odd
```

Как видите, в этом случае вычисляется только сообщение по умолчанию.

Условие должно быть *внешним*, т. е. оно не должно содержать ленивых данных. В противном случае данные будут вычислены при проверке условия. Также есть возможность определить ленивые каррированные `val`-функции. С каррированными функциями мы познакомились в главе 3.

Упражнение 9.3

Напишите ленивую каррированную val-версию функции constructMessage.

Решение

Здесь нет ничего сложного:

```
val constructMessage: (Lazy<String>) -> (Lazy<String>) -> Lazy<String>
=
    { greetings ->
        { name ->
            Lazy { "${greetings()}, ${name()}!" }
        }
    }

fun main(args: Array<String>) {
    val greetings = Lazy {
        println("Evaluating greetings")
        "Hello"
    }

    val name1: Lazy<String> = Lazy {
        println("Evaluating name")
        "Mickey"
    }

    val name2: Lazy<String> = Lazy {
        println("Evaluating name")
        "Donald"
    }

    val message1 = constructMessage(greetings)(name1)
    val message2 = constructMessage(greetings)(name2)
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) message1() else defaultMessage())
    println(if (condition) message2() else defaultMessage())
}
```


Эта программа дает тот же результат, но синтаксис вызова функции пришлось немного изменить. Она может служить хорошим примером использования каррированных функций. Возможно, вам придется приветствовать многих людей, используя один и тот же текст сообщения, но с разными именами. Это можно реализовать так:

```
val constructMessage: (Lazy<String>) -> (Lazy<String>) -> Lazy<String>
=
    { greetings ->
        { name ->
            Lazy { "${greetings()}, ${name()}!" }
        }
    }

fun main(args: Array<String>) {
```

```
    val greetings = Lazy {
        println("Evaluating greetings")
        "Hello"
    }

    val name1: Lazy<String> = Lazy {
        println("Evaluating name1")
        "Mickey"
    }

    val name2: Lazy<String> = Lazy {
        println("Evaluating name2")
        "Donald"
    }

    val defaultMessage = Lazy {
        println("Evaluating default message")
        "No greetings when time is odd"
    }

    val greetingString = constructMessage(greetings)
    val message1 = greetingString(name1)
    val message2 = greetingString(name2)
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) message1() else defaultMessage())
    println(if (condition) message2() else defaultMessage())
}
```

9.4.2 Преобразование обычных функций в ленивые

Часто бывает желательно преобразовать имеющуюся функцию, вычисляющую свои аргументы, в функцию, которая использует аргументы, не вычисляя их. В конце концов, в этом заключена суть программирования.

Упражнение 9.4

Напишите функцию, которая принимает каррированную функцию с двумя вычисляемыми аргументами и возвращает соответствующую функцию, не вычисляющую свои аргументы, которая возвращает тот же результат, но не вычисленный. Пусть есть функция:

```
val consMessage: (String) -> (String) -> String =
    { greetings ->
        { name ->
            "$greetings, $name!"
        }
    }
```

Напишите функцию `lift2`, которая возвращает следующую функцию (без вычисления аргументов):

```
(Lazy<String>) -> (Lazy<String>) -> Lazy<String>
```

Подсказка

Поместите эту функцию в объект-компаньон `Lazy`, чтобы предотвратить конфликт имен с другой реализацией `lift2`. Также эту функцию можно объявить на уровне пакета, но с другим именем, например `liftLazy2`.

Решение

Сначала определим сигнатуру функции. Функция `lift2` принимает функцию типа `(String) -> (String) -> String`. Аргумент этого типа нужно заключить в фигурные скобки и добавить в конце стрелку вправо:

```
((String) -> (String) -> String) ->
```

Далее запишем тип значения, возвращаемого этой функцией:

```
((String) -> (String) -> String) ->
        (Lazy<String>) -> (Lazy<String>) -> Lazy<String>
```

Добавим `val lift2:` слева и знак равенства справа:

```
val lift2: ((String) -> (String) -> String) ->
        (Lazy<String>) -> (Lazy<String>) -> Lazy<String> =
```

Теперь напишем реализацию. Так как результатом является функция типа

```
(Lazy<String>) -> (Lazy<String>) -> Lazy<String>
```

начнем с:

```
{ f -> { ls1 -> { ls2 -> TODO() } } }
```

где `f` имеет тип `(String) -> (String) -> String` (преобразуемая функция), а `ls1` и `ls2` имеют тип `Lazy<String>`:

```
val lift2: ((String) -> (String) -> String) ->
        (Lazy<String>) -> (Lazy<String>) -> Lazy<String> =
    { f ->
        { ls1 ->
            { ls2 ->
                TODO()
            }
        }
    }
```

Остальное дописать очень просто. У нас есть каррированная функция двух аргументов `String` и два экземпляра `Lazy<String>`. Получим значения и применим к ним функцию. Так как мы должны получить `Lazy<String>`, создадим экземпляр `Lazy` для возврата результата:

```
val lift2: ((String) -> (String) -> String) ->
        (Lazy<String>) -> (Lazy<String>) -> Lazy<String> =
    { f ->
        { ls1 ->
            { ls2 ->
```

```
                Lazy { f(ls1())(ls2()) }
            }
        }
    }
```

Упражнение 9.5

Обобщите функцию из предыдущего примера, чтобы она могла работать с любыми типами. На этот раз реализуйте ее как функцию `fun` на уровне пакета.

Решение

Снова начнем с сигнатуры. Так как функция должна работать с любыми типами, ее следует параметризовать типами `A`, `B` и `C` и определить единственный параметр как функцию типа `(A) -> (B) -> C`. Тип возвращаемого значения – это функция типа `(Lazy<A>) -> (Lazy<B>) -> Lazy<C>`:

```
fun <A, B, C> lift2(f: (A) -> (B) -> C): (Lazy<A>) -> (Lazy<B>) -> Lazy<C>
```

Реализация аналогична приведенной в упражнении 9.4 с той лишь разницей, что первый параметр (функция) теперь не определен, потому что это функция `fun` и типы могут быть разными:

```
fun <A, B, C> lift2(f: (A) -> (B) -> C):
        (Lazy<A>) -> (Lazy<B>) -> Lazy<C> =
    { ls1 ->
        { ls2 ->
            Lazy { f(ls1())(ls2()) }
        }
    }
```

9.4.3 Отображение ленивых значений

Уверен, теперь вы отчетливо видите, что `Lazy` – это еще один контекст вычислений, подобный `List`, `Option` и `Result` (и многим другим, которые вы обнаружите позже), и наверняка испытываете желание написать функции `map` и `flatMap` для него!

Упражнение 9.6

Напишите функцию `map`, которая применяет функцию `(A) -> B` к значению `Lazy<A>` и возвращает `Lazy<B>`.

Подсказка

Определите ее как функцию экземпляра в классе `Lazy`.

Решение

Нам нужно просто применить указанную функцию к ленивому значению. А чтобы предотвратить немедленное вычисление результата, заключим вызов в новый экземпляр `Lazy`:

```
fun <B> map(f: (A) -> B): Lazy<B> = Lazy{ f(value) }
```

Проверить функцию можно с помощью следующей программы:

```
fun main(args: Array<String>) {

    val greets: (String) -> String = { "Hello, $it!" }

    val name: Lazy<String> = Lazy {
        println("Evaluating name")
        "Mickey"
    }

    val defaultMessage = Lazy {
        println("Evaluating default message")
        "No greetings when time is odd"
    }

    val message = name.map(greets)
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) message() else defaultMessage())
    println(if (condition) message() else defaultMessage())
}
```

Если условие выполняется, программа выведет следующие строки, демонстрируя, что вычисления выполнены только один раз, даже притом что функция была вызвана дважды:

```
Evaluating name
Hello, Mickey!
Hello, Mickey!
```

Если условие не выполняется, программа выведет сообщение по умолчанию, которое также вычисляется только один раз:

```
Evaluating default message
No greetings when time is odd
No greetings when time is odd
```

Упражнение 9.7

Напишите функцию `flatMap`, которая применяет функцию `(A) -> Lazy<B>` к значению `Lazy<A>` и возвращает `Lazy<B>`.

Подсказка

Определите ее как функцию экземпляра в классе `Lazy`.

Решение

Нам нужно применить указанную функцию к ленивому значению и инициировать вычисление. А чтобы этого не произошло раньше, чем потребуется результат, заключим вызов в новый экземпляр `Lazy`:

```
fun <B> flatMap(f: (A) -> Lazy<B>): Lazy<B> = Lazy { f(value)() }
```

Проверить функцию можно с помощью следующей программы:

```
fun main(args: Array<String>) {

    // Представьте, что getGreetings -- это функция, выполняющаяся
```

```
    // продолжительное время и с побочным эффектом
    // в виде вывода строки "Evaluating greetings" в консоль

    val greetings: Lazy<String> = Lazy { getGreetings(Locale.US) }

    val flatGreets: (String) -> Lazy<String> =
        { name -> greetings.map { "$it, $name!"} }

    val name: Lazy<String> = Lazy {
        println("computing name")
        "Mickey"
    }

    val defaultMessage = Lazy {
        println("Evaluating default message")
        "No greetings when time is odd"
    }

    val message = name.flatMap(flatGreets)
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) message() else defaultMessage())
    println(if (condition) message() else defaultMessage())
}
```

Если условие выполняется, программа выведет следующие строки, демонстрируя, что вычисление имени и текста приветствия происходит только один раз, даже притом что функция вызвана дважды:

```
Evaluating name
Evaluating greetings
Hello, Mickey!
Hello, Mickey!
```

Если условие не выполняется, программа выведет лишь сообщение по умолчанию, которое также вычисляется только один раз:

```
Evaluating default message
No greetings when time is odd
No greetings when time is odd
```

9.4.4 Комбинирование типов Lazy и List

Тип `Lazy` можно комбинировать с другими типами, созданными нами в предыдущих главах. Одной из наиболее распространенных операций является преобразование `List<Lazy<A>>` в `Lazy<List<A>>`, позволяющее лениво комбинировать список с функциями `A`. Этот вид комбинирования должен выполняться без вычисления данных.

Упражнение 9.8

Напишите функцию `sequence` со следующей сигнатурой:

```
fun <A> sequence(lst: List<Lazy<A>>): Lazy<List<A>>
```

Функция должна быть объявлена на уровне пакета.

Решение

И снова решение выглядит просто. Нам нужно преобразовать список в функцию, вычисляющую значение каждого элемента. Но так как sequence ничего не должна вычислять, ее тело следует заключить в новый экземпляр Lazy:

```
fun <A> sequence(lst: List<Lazy<A>>): Lazy<List<A>> =
    Lazy { lst.map { it() } }
```

Вот пример программы, иллюстрирующей работу нашей функции:

```
fun main(args: Array<String>) {

    val name1: Lazy<String> = Lazy {
        println("Evaluating name1")
        "Mickey"
    }

    val name2: Lazy<String> = Lazy {
        println("Evaluating name2")
        "Donald"
    }

    val name3 = Lazy {
        println("Evaluating name3")
        "Goofy"
    }

    val list = sequence(List(name1, name2, name3))
    val defaultMessage = "No greetings when time is odd"
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) list() else defaultMessage)
    println(if (condition) list() else defaultMessage)
}
```

И снова, в зависимости от внешнего условия, дважды выводится результат или сообщение по умолчанию. Если условие выполняется, дважды выводится результат, при этом элементы вычисляются только один раз:

```
Evaluating name1
Evaluating name2
Evaluating name3
[Mickey, Donald, Goofy, NIL]
[Mickey, Donald, Goofy, NIL]
```

Если условие не выполняется, вычислений не происходит:

```
No greetings when time is odd
No greetings when time is odd
```

9.4.5 Обработка исключений

При работе с собственными функциями можно не опасаться исключений, если вы уверены, что ваш код никогда не сгенерирует их. Однако

при работе с ленивыми данными есть риск получить исключение во время их вычисления.

Исключение при вычислении отдельного элемента данных не является чем-то особенным. Но обезопасить себя от исключений при вычислении списка элементов `Lazy<A>` сложнее. Что делать, если при вычислении одного из элементов будет сгенерировано исключение?

Упражнение 9.9

Напишите функцию `sequenceResult` со следующей сигнатурой:

```
fun <A> sequenceResult(lst: List<Lazy<A>>): Lazy<Result<List<A>>>
```

Функция должна возвращать результат `Result<List<A>>`, который не вычисляется немедленно. Этот результат должен преобразовываться в `Success<List<A>>`, если все вычисления выполнились успешно, и `Failure<List<A>>` – в противном случае. Вот пример тестовой программы:

```
fun main(args: Array<String>) {

    val name1: Lazy<String> = Lazy {
        println("Evaluating name1")
        "Mickey"
    }

    val name2: Lazy<String> = Lazy {
        println("Evaluating name2")
        "Donald"
    }

    val name3 = Lazy {
        println("Evaluating name3")
        "Goofy"
    }

    val name4 = Lazy {
        println("Evaluating name4")
        throw IllegalStateException("Exception while evaluating name4")
    }

    val list1 = sequenceResult(List(name1, name2, name3))
    val list2 = sequenceResult(List(name1, name2, name3, name4))
    val defaultMessage = "No greetings when time is odd"
    val condition = Random(System.currentTimeMillis()).nextInt() % 2 == 0

    println(if (condition) list1() else defaultMessage)
    println(if (condition) list1() else defaultMessage)
    println(if (condition) list2() else defaultMessage)
    println(if (condition) list2() else defaultMessage)
}
```

Если условие выполняется, программа должна вывести:

```
Evaluating name1
Evaluating name2
Evaluating name3
Success([Mickey, Donald, Goofy, NIL])
```

```
Success([Mickey, Donald, Goofy, NIL])
Evaluating name4
Failure(Exception while evaluating name4)
Failure(Exception while evaluating name4)
```

А если условие не выполняется, программа должна вывести:

```
No greetings when time is odd
No greetings when time is odd
No greetings when time is odd
No greetings when time is odd
```


Решение

Как и в предыдущем упражнении, мы должны добавить инструкции, возвращающие желаемый результат, а затем заключить их в новый экземпляр `Lazy`:

```
import com.fpinkotlin.common.sequence

...

fun <A> sequenceResult(lst: List<Lazy<A>>): Lazy<Result<List<A>>> =
    Lazy { sequence(lst.map { Result.of(it) }) }
```

Как видите, нам потребовалось явно импортировать функцию `sequence`. Это другая функция `sequence`, не та, что мы определили на уровне пакета. Также можно использовать функцию `traverse`, которую мы написали в главе 8:

```
fun <A> sequenceResult2(lst: List<Lazy<A>>): Lazy<Result<List<A>>> =
    Lazy { traverse(lst) { Result.of(it) }() }
```


Обращение к результату вызывает вычисление всех элементов списка, даже если для одного из них будет получено значение `Failure`. Причина в том, что функция `traverse` использует функцию `foldRight`. Чтобы прервать вычисления после получения первого же значения `Failure`, нужно использовать специальную реализацию свертки. Реализация на основе `traverse` эквивалентна следующему коду:

```
fun <A> sequenceResult(lst: List<Lazy<A>>): Lazy<Result<List<A>>> =
    Lazy {
        lst.foldRight(Result(List())) { x: Lazy<A> ->
            { y: Result<List<A>> ->
                map2(Result.of(x), y) { a: A ->
                                        { b: List<A> ->
                                            b.cons(a)
                                        }
                                      }
            }
        }
    }
```

У нас нет специальной версии `foldRight`, зато есть специальная версия `foldLeft`, разработанная в главе 8. Используем ее:

```
fun <A> sequenceResult(list: List<Lazy<A>>): Lazy<Result<List<A>>> =
    Lazy {
        val p = { r: Result<List<A>> -> r.map{false}.getOrElse(true) }
        list.foldLeft(Result(List()), p) { y: Result<List<A>> ->
            { x: Lazy<A> ->
                map2(Result.of(x), y) { a: A ->
                                        { b: List<A> ->
                                            b.cons(a)
                                        }
                                    }
            }
        }
    }
```

Чтобы абстрагировать этот процесс, можно определить специальную версию `traverse` в классе `List`.

9.5 Другие способы комбинирования ленивых вычислений

Как вы видите, ленивое комбинирование функций – это самое обычное их комбинирование с последующим заключением в экземпляр `Lazy`. Используя этот прием, можно получать ленивые комбинации чего угодно. Здесь нет никакого волшебства. Вы всегда можете сделать то же самое, заключив любую реализацию в константную функцию:

```
fun <A> lazyComposition(): Lazy<A> =
    Lazy { <код, генерирующий A> }
```

Вы просто пишете программу, которая выполняется только при обращении к экземпляру `Lazy`.

9.5.1 Ленивое применение эффектов

Если вы использовали тестовые программы из предыдущих упражнений, то уже знаете, как применять эффекты к ленивым значениям. Вы можете обратиться к экземпляру `Lazy` и применить эффект к результату, например:

```
val lazyString: Lazy<String> = ...
...
println(lazyString())
```

Но вместо извлечения значения из `Lazy` и применения эффекта можно пойти другим путем и передать эффект в `Lazy`, чтобы применить его к значению:

```
fun forEach(ef: (A) -> Unit) = ef(value)
```

Но этот подход не годится, когда эффект должен применяться по условию, как в предыдущих упражнениях:

```
if (condition) list1.forEach { println(it) } else
println(defaultMessage)
```

На самом деле мы должны передать условие и два эффекта, один из которых должен применяться при выполнении условия, а другой – в противном случае.

Упражнение 9.10

Напишите функцию (метод) `forEach` в классе `Lazy`, принимающую условие и два эффекта. При выполнении условия она должна применять первый эффект, при невыполнении – второй.

Подсказка

Для большей пользы функция должна вычислять значение `Lazy`, только если это необходимо. Следовательно, вам нужно определить три перегруженные версии функции.

Решение

У вас почти наверняка возникнет соблазн написать такую реализацию:

```
fun forEach(condition: Boolean, ifTrue: (A) -> Unit, ifFalse: (A) -> Unit) =
    if (condition) ifTrue(value) else ifFalse(value)
```

Но это не даст желаемого, потому что Kotlin вычислит аргументы функций `ifTrue` и `ifFalse`, как только они будут получены функцией `forEach`, неважно, используют ли значение `value` функции `ifTrue` и `ifFalse`. Обычно условие `ifFalse` не использует его, например:

```
list1.forEach(condition, ::println, { println(defaultMessage)} )
```

Так как обработчик `ifFalse` не использует значение, его не следует вычислять. Но может быть и наоборот, когда значение используется, если условие не выполняется. Конечно, условие можно инвертировать. Но также следует позаботиться о случае, когда оба обработчика должны вычислить `Lazy`. Решить эту проблему можно, написав три версии функции:

```
fun forEach(condition: Boolean,
            ifTrue: (A) -> Unit,
            ifFalse: () -> Unit = {}) =
    if (condition)
        ifTrue(value)
    else
        ifFalse()

fun forEach(condition: Boolean,
            ifTrue: () -> Unit = {},
            ifFalse: (A) -> Unit) =
    if (condition)
        ifTrue()
    else
        ifFalse(value)

fun forEach(condition: Boolean,
```

```
        ifTrue: (A) -> Unit,
        ifFalse: (A) -> Unit) =
    if (condition)
        ifTrue(value)
    else
        ifFalse(value)
```

Но не забывайте о двух проблемах. Чтобы Kotlin смог выбрать правильную версию, нужно явно указать тип каждого обработчика, как показано ниже:

```
val printMessage: (Any) -> Unit = ::println
val printDefault: () -> Unit = { println(defaultMessage)}
list1.forEach(condition, printMessage, printDefault)
```

Причина в том, что обработчик {} можно использовать для аргументов обоих типов: `(A) -> Unit` и `() -> Unit`.

Чтобы можно было использовать значение по умолчанию в аргументе `ifTrue: () -> Unit` (т. е. ничего не делать), нужно передать аргумент `ifFalse` по имени:

```
val printMessage: (Any) -> Unit = ::println
    list1.forEach(condition, ifFalse = printMessage)
```

9.5.2 Вычисления, невозможные без ленивых значений

В настоящий момент может показаться, что отсутствие в Kotlin поддержки по-настоящему ленивых вычислений не имеет большого значения. В конце концов, зачем переписывать булевы функции, если можно использовать булевы операторы? Однако есть множество случаев, когда ленивые вычисления оказываются незаменимыми. Есть даже несколько алгоритмов, которые нельзя реализовать, не прибегая к ленивым вычислениям. Я уже говорил, насколько бесполезна строгая версия `if ... else`. А теперь поразмышляйте над следующим алгоритмом:

1. Взять список всех натуральных чисел.
2. Найти в нем простые числа.
3. Вернуть список с первыми десятью результатами.

Это алгоритм поиска первых десяти простых чисел, но его нельзя реализовать без ленивых вычислений. Если не верите – попробуйте сами. Начните с первой строки. При использовании строгих вычислений вы сначала вычислите список натуральных чисел. Вы не сможете перейти ко второму пункту в строке, потому что список всех натуральных чисел бесконечен, и исчерпаете доступную память, так и не достигнув (несуществующего) конца списка.

Очевидно, что этот алгоритм нельзя реализовать без ленивых вычислений, но его можно заменить другим алгоритмом. Предыдущий алгоритм – это функциональный алгоритм. Чтобы найти результат, не прибегая к ленивым вычислениям, вам придется заменить его императивным алгоритмом, например:

1. Взять первое натуральное число.
2. Проверить, является ли оно простым.
3. Если это простое число, сохранить его в списке с результатами.
4. Проверить, получено ли достаточное количество элементов.
5. Если получено десять элементов, вернуть результат.
6. Иначе увеличить число на 1.
7. Перейти к п. 2.

Да, этот алгоритм работает. Но каким сложным он выглядит! Во-первых, это не лучший подход. Разве не следует увеличивать проверенное целое число на 2, а не на 1, чтобы не проверять *четные* числа? И зачем проверять числа, кратные трем, пяти и т. д.? Но что еще более важно, этот алгоритм не отражает природы задачи. Это всего лишь рецепт вычисления результата.

Это не означает, что детали реализации (например, отказ от проверки четных чисел) не важны для достижения высокой производительности. Но они должны быть четко отделены от определения задачи. Императивное описание не является описанием задачи. Это описание другой задачи, дающей тот же результат. Гораздо более элегантное решение получается при использовании специальной структуры: ленивого списка.

9.5.3 Создание ленивого списка

Теперь, зная, что невычисляемые данные можно представлять как экземпляры `Lazy`, мы легко сможем определить новую структуру данных – ленивый список. Назовем его `Stream`. Он будет похож на односвязный список, разработанный нами в главе 5, с некоторыми тонкими, но важными отличиями. В листинге 9.3 представлена отправная точка для нашего нового типа данных `Stream`.

Листинг 9.3 Тип данных Stream

```
import com.fpinkotlin.common.Result ①

sealed class Stream<out A> { ②

    abstract fun isEmpty(): Boolean

    abstract fun head(): Result<A> ③

    abstract fun tail(): Result<Stream<A>> ④

    private object Empty: Stream<Nothing>() { ⑤

        override fun head(): Result<Nothing> = Result()

        override fun tail(): Result<Nothing> = Result()

        override fun isEmpty(): Boolean = true
    }

    private
    class Cons<out A> (internal val hd: Lazy<A>, ⑥ ⑦
                       internal val tl: Lazy<Stream<A>>):
                                            Stream<A>() { ⑧
```

```
        override fun head(): Result<A> = Result(hd())

        override fun tail(): Result<Stream<A>> = Result(tl())

        override fun isEmpty(): Boolean = false
    }

    companion object {

        fun <A> cons(hd: Lazy<A>,
                     tl: Lazy<Stream<A>>): Stream<A> =
                                    Cons(hd, tl) ⑨

        operator fun <A> invoke(): Stream<A> =
                                    Empty ⑩

        fun from(i: Int): Stream<Int> =
            cons(Lazy { i }, Lazy { from(i + 1) }) ⑪
    }
}
```

① Импортировать класс `Lazy` не требуется, потому что он находится в текущем пакете
② Класс `Stream` объявлен запечатанным (`sealed`), чтобы предотвратить возможность прямого создания его экземпляров
③ Функция `head` возвращает `Result<A>`, чтобы имелась возможность вернуть `Empty`, если поток (т. е. список) пуст
④ По той же причине функция `tail` тоже возвращает `Result<Stream<A>>`
⑤ Подкласс `Empty` в точности повторяет подкласс `List.Nil`
⑥ Подкласс `Cons` представляет непустой поток
⑦ Голова потока (`hd`) не вычисляется, так как передается в форме `Lazy<A>`
⑧ Хвост (`tl`) тоже не вычисляется, потому что представлен типом `Lazy<Stream<A>>`
⑨ Функция `cons` конструирует список вызовом приватного конструктора `Cons`
⑩ Функция-оператор `invoke` возвращает синглтон `Empty`
⑪ Фабричная функция `from` возвращает бесконечный поток последовательных чисел, начиная с заданного значения

Вот пример использования типа `Stream`:

```
fun main(args: Array<String>) {
    val stream = Stream.from(1)
    stream.head().forEach({ println(it) })
    stream.tail().flatMap { it.head() }.forEach({ println(it) })
    stream.tail().flatMap {
        it.tail().flatMap { it.head() }
    }.forEach({ println(it) })
}
```

Эта программа выведет:

```
1
2
3
```

Этот пример не кажется чем-то захватывающим. Чтобы сделать `Stream` ценным инструментом, нужно добавить в него несколько дополнительных функций. Но сначала его следует немного оптимизировать.

ПРИМЕЧАНИЕ Важно понимать разницу между потоком `Stream`, описанным в этой главе, который является списком с поддержкой ленивых вычислений, и потоком `Stream` в Java, который реализован как генератор. *Генераторы* – это функции, вычисляющие следующий элемент, исходя из некоторого текущего состояния, которым может быть предыдущий элемент или что-то еще. В частности, это внешнее состояние может быть уже вычисленной индексированной коллекцией плюс индекс. Это тот случай, когда мы используем Java-конструкцию, такую как:

```
List<A> list = ....
list.stream()...
```

В этом примере на Java функция-генератор использует пару `(list, index)` в качестве изменяемого состояния, из которого генерирует новое значение. Здесь генератор зависит от внешнего изменяемого состояния. Кроме того, после вычисления значения (через терминальную операцию) поток больше нельзя использовать.

Kotlin тоже предлагает генераторы, такие как `Sequence`, но в отличие от Java они не поддерживают параллельную обработку.

Наш поток `Stream` отличается. Данные в нем не вычисляются, но могут вычисляться, и это не мешая повторному использованию потока. В предыдущем примере мы вычислили три первых значения в потоке, но это не мешает повторно использовать его, что наглядно демонстрирует следующий пример:

```
fun main(args: Array<String>) {
    val stream = Stream.from(1)
    stream.head().forEach({ println(it) })
    stream.tail().flatMap { it.head() }.forEach({ println(it) })
    stream.tail().flatMap { it.tail()
            .flatMap { it.head() } }.forEach({ println(it) })
    stream.head().forEach({ println(it) })
    stream.tail().flatMap { it.head() }.forEach({ println(it) })
    stream.tail().flatMap { it.tail()
            .flatMap { it.head() } }.forEach({ println(it) })
}
```

Эта программа выведет:

```
1
2
3
1
2
3
```

При попытке проделать то же самое с потоком в Java вы получите исключение `IllegalStateException` с сообщением: «Поток уже обрабатывается или закрыт»:

```
public class TestStream {
    public static void main(String... args) {
        Stream<Integer> stream = Stream.iterate(0, i -> i + 1);
        stream.findFirst().ifPresent(System.out::println);
        stream.findFirst().ifPresent(System.out::println);
    }
}
```

Конструкция `Sequence` в Kotlin допускает повторное использование, но она не запоминает сгенерированные значения, т. е. не является ленивым списком.

9.6 Работа с потоками

В оставшейся части этой главы вы узнаете, как создавать и комбинировать потоки, с максимальной пользой используя их ленивую природу. Но для просмотра содержимого потоков нам понадобится функция для их вычисления. А чтобы вычислить поток, нужна функция, ограничивающая его длину. Бессмысленно пытаться вычислить бесконечный поток, верно?

Упражнение 9.11

Напишите функцию `repeat`, которая принимает функцию типа `() -> A` и возвращает поток элементов `A`.

Решение

Здесь нет ничего сложного. Нужно просто объединить (`cons`) ленивый вызов указанной функции с ленивым вызовом самой функции `repeat`:

```
fun <A> repeat(f: () -> A): Stream<A> =
    cons(Lazy { f() }, Lazy { repeat(f) })
```

Эта функция не страдает проблемой переполнения стека благодаря ленивому вызову `repeat`.

Упражнение 9.12

Напишите функцию `takeAtMost`, ограничивающую длину потока самое большее `n` элементами. Эта функция должна работать с любыми потоками, даже с потоками, в которых число элементов меньше `n`. Вот ее сигнатура:

```
fun takeAtMost(n: Int): Stream<A>
```

Подсказка

Объявите абстрактную функцию в классе `Stream` и добавьте конкретные реализации в оба подкласса. Если понадобится, используйте рекурсию.

Решение

Реализация в классе `Empty` просто возвращает `this`:

```
override fun takeAtMost(n: Int): Stream<Nothing> = this
```

Реализация в классе `Cons` сравнивает аргумент `n` с числом `0`. Если он больше нуля, возвращает голову потока, к которому «присоединяет» результат рекурсивного применения функции `takeAtMost` к хвосту с аргументом `n - 1`. Если `n` меньше или равно `0`, возвращает пустой поток:

```
override fun takeAtMost(n: Int): Stream<A> = when {
    n > 0 -> cons(hd, Lazy { tl().takeAtMost(n - 1) })
    else -> Empty
}
```

Упражнение 9.13

Напишите функцию `dropAtMost`, удаляющую до `n` элементов из потока. Эта функция должна работать с любыми потоками, даже с потоками, в которых число элементов меньше `n`. Вот ее сигнатура:

```
fun dropAtMost(n: Int): Stream<A>
```

Подсказка

Объявите абстрактную функцию в классе `Stream` и добавьте конкретные реализации в оба подкласса. Если понадобится, используйте рекурсию.

Решение

Реализация в классе `Empty` просто возвращает `this`:

```
override fun dropAtMost(n: Int): Stream<Nothing> = this
```

Реализация в классе `Cons` сравнивает аргумент `n` с числом `0`. Если он больше нуля, возвращает результат рекурсивного применения `dropAtMost` к хвосту потока с аргументом `n - 1`. Если `n` меньше или равно `0`, возвращает текущий поток:

```
override fun dropAtMost(n: Int): Stream<A> = when {
    n > 0 -> tl().dropAtMost(n - 1)
    else -> this
}
```

Упражнение 9.14

Поразмышляйте о функциях `takeAtMost` и `dropAtaMost` с точки зрения рекурсии. Что получится, если вызвать их со слишком большим значением параметра? Если у вас не получится прийти к заключению, взгляните на следующий пример:

```
fun main(args: Array<String>) {
    val stream =
        Stream.repeat(::random).dropAtMost(60000).takeAtMost(60000)
    stream.head().forEach(::println)
}

val rnd = Random()

fun random(): Int {
    val rnd = rnd.nextInt()
    println("evaluating $rnd")
```

```
    return rnd
}
```

Эта программа вызывает исключение `StackOverflowException`, ничего не успев вычислить! Попробуйте решить проблему.

Решение

Функция `takeAtMost` в этом примере не вызывает никаких проблем, потому что она действует лениво; никаких значений не вычисляется после 60 001-го элемента. Но функция `dropAtMost` должна вызывать себя рекурсивно 60 000 раз, чтобы получившийся поток начался с 60 001-го элемента. Рекурсия производится, даже если не вычисляется ни один элемент.

Решение состоит в том, чтобы сделать функцию `dropAtMost` сорекурсивной, что обычно требует использования двух дополнительных аргументов: потока, к которому применяется функция, и аккумулятора для результата. Но в этом конкретном случае аккумулятор не нужен, потому что результат каждого рекурсивного шага игнорируется. Необходим только новый поток (с одним удаленным элементом) и новый параметр `Int` (уменьшающийся на единицу). Вот как выглядит искомая функция в объекте-компаньоне:

```
tailrec fun <A> dropAtMost(n: Int, stream: Stream<A>): Stream<A> =
    when {
        n > 0 -> when (stream) {
        is Empty -> stream
        is Cons -> dropAtMost(n - 1, stream.tl())
    }
    else -> stream
}
```

Как видите, если аргумент `n` достиг 0 или поток пуст, возвращается поток в аргументе `stream`. Иначе функция рекурсивно применяется к хвосту потока после уменьшения параметра `n`. Чтобы упростить использование этой функции, в класс `Stream` добавлена следующая функция экземпляра:

```
fun dropAtMost(n: Int): Stream<A> = dropAtMost(n, this)
```

Упражнение 9.15

В предыдущем упражнении я упомянул, что функция `takeAtMost` не создает проблем со стеком, потому что она ленивая и ничего не вычисляет. Как следствие, возникает вопрос: что случится, когда поток в конечном итоге потребуется вычислить. Для проверки этой ситуации создайте функцию `toList`, превращающую поток в список, которая вычислит все элементы. Используйте эту функцию, чтобы убедиться, что функция `takeAtMost` безвредна, запустив следующую программу:

```
fun main(args: Array<String>) {
    val stream = Stream.from(0).dropAtMost(60000).takeAtMost(60000)
    println(stream.toList())
}
```

Подсказка

Напишите основную функцию в классе `Stream`, которая будет вызывать сорекурсивную вспомогательную функцию в объекте-компаньоне.

Решение

Функция в объекте-компаньоне должна использовать сорекурсивную вспомогательную функцию, принимающую `List<A>` в качестве аккумулятора. Если поток пуст, функция должна вернуть список-аккумулятор. Иначе она должна вызвать себя рекурсивно после добавления головы потока в список и использовать хвост потока в качестве второго параметра. Основная функция должна вызывать вспомогательную функцию с пустым списком в качестве начального аккумулятора и инвертировать получившийся список перед возвратом:

```
fun <A> toList(stream: Stream<A>) : List<A> {
    tailrec
    fun <A> toList(list: List<A>, stream: Stream<A>): List<A> =
        when (stream) {
            Empty -> list
            is Cons -> toList(list.cons(stream.hd()), stream.tl())
        }

    return toList(List(), stream).reverse()
}
```

В класс `Stream` следует добавить основную функцию, чтобы получить возможность вызывать ее с использованием объектной нотации:

```
fun toList(): List<A> = toList(this)
```

Упражнение 9.16

До сих пор мы могли создать только бесконечный поток последовательных целых чисел или поток случайных элементов. Чтобы сделать класс `Stream` более полезным, напишите функцию `iterate`, принимающую начальное число типа `A`, и функцию `(A) -> A`, возвращающую бесконечный поток элементов типа `A`. Затем переопределите функцию `from` в терминах `iterate`.

Решение

Решение заключается в использовании функции `cons` для создания потока из начального значения `seed` в роли головы и ленивого рекурсивного вызова функции `iterate` с `f (seed)` в роли хвоста:

```
fun <A> iterate(seed: A, f: (A) -> A): Stream<A> =
    cons(Lazy { seed }, Lazy { iterate(f(seed), f) })
```

Несмотря на свою рекурсивную природу, эта функция не вызывает переполнения стека, потому что рекурсивный вызов выполняется лениво. Теперь мы можем переписать `from`, использовав в ней эту функцию:

```
fun from(i: Int): Stream<Int> = iterate(i) { it + 1 }
```

Эту функцию можно использовать, чтобы убедиться, что вычисления действительно происходят. Если создать функцию с побочным эффектом, например:

```
fun inc(i: Int): Int = (i + 1).let {
    println("generating $it")
    it
}
```

можно убедиться, что следующая программа вычисляет только значения от 0 до 10 010, прежде чем вывести список с 10 значениями от 10 000 до 10 009:

```
fun main(args: Array<String>) {
    fun inc(i: Int): Int = (i + 1).let {
        println("generating $it")
        it
    }
    val list = Stream
                    .iterate(0, ::inc)
                    .takeAtMost(60000)
                    .dropAtMost(10000)
                    .takeAtMost(10)
                    .toList()
    println(list)
}
```

Также можно написать функцию `iterate`, которая принимает начальное значение типа `Lazy<A>`, если потребуется получить поток, не вычисляя начального значения:

```
fun <A> iterate(seed: Lazy<A>, f: (A) -> A): Stream<A> =
    cons(seed, Lazy { iterate(f(seed()), f) })
```

Упражнение 9.17

Напишите функцию `takeWhile`, которая возвращает поток `Stream`, содержащий все начальные элементы, удовлетворяющие условию. Вот сигнатура функции в родительском классе `Stream`:

```
abstract fun takeWhile(p: (A) -> Boolean): Stream<A>
```

Подсказка

Имейте в виду, что в отличие от `takeAtMost` и `dropAtMost` эта функция вычисляет один элемент, потому что ей необходимо проверить соответствие первого элемента заданному условию, выраженному предикатом. Вы должны гарантировать, что вычисляется только первый элемент потока.

Решение

Эта функция напоминает `takeAtMost`. Основное отличие – условие завершения теперь определяется не выражением `n <= 0`, а функцией, возвращающей `false`:

```
override fun takeWhile(p: (A) -> Boolean): Stream<A> = when {
    p(hd()) -> cons(hd, Lazy { tl().takeWhile(p) })
    else -> Empty
}
```

И снова нет необходимости беспокоиться о безопасности стека, потому что рекурсивный вызов выполняется лениво. Реализация в `Empty` возвращает `this`.

Упражнение 9.18

Напишите функцию `dropWhile`, возвращающую поток, из начала которого удалены элементы, удовлетворяющие условию. Вот сигнатура этой функции в классе `Stream`:

```
fun dropWhile(p: (A) -> Boolean): Stream<A>
```

Подсказка

Если потребуется, напишите сорекурсивную версию, чтобы избежать переполнения стека.

Решение

Как и в случае с предыдущими рекурсивными функциями, функция в классе `Stream` должна вызывать безопасную вспомогательную сорекурсивную функцию в объекте-компаньоне. Вот сорекурсивная функция в объекте-компаньоне:

```
tailrec fun <A> dropWhile(stream: Stream<A>,
                          p: (A) -> Boolean): Stream<A> =
    when (stream) {
        is Empty -> stream
        is Cons -> when {
            p(stream.hd()) -> dropWhile(stream.tl(), p)
            else -> stream
        }
    }
```

А это реализация основной функции в классе `Stream`:

```
fun dropWhile(p: (A) -> Boolean): Stream<A> = dropWhile(this, p)
```

Упражнение 9.19

В главе 8 мы реализовали функцию `exists` в классе `List`:

```
fun exists(p:(A) -> Boolean): Boolean =
    when (this) {
        Nil -> false
        is Cons -> p(head) || tail.exists(p)
    }
```

Эта функция продолжает обход элементов списка, пока не будет найден элемент, удовлетворяющий условию в предикате `p`. Остальная часть списка не проверяется, потому что оператор `||` выполняется лениво и не вычисляет свой второй операнд, если первый вычисляется как `true`.

Напишите аналогичную функцию `exists` для класса `Stream`. Функция должна вычислять элементы, пока не выполнится условие. Если условие никогда не выполнится, должны быть вычислены все элементы.

Решение

Простейшее решение напоминает функцию `exists` в классе `List`:

```
fun exists(p: (A) -> Boolean): Boolean = p(hd()) || tl().exists(p)
```

Но ее следует сделать безопасной с точки зрения переполнения стека. Для этого рекурсию нужно преобразовать в хвостовую рекурсию. Вот возможная реализация в объекте-компаньоне:

```
tailrec fun <A> exists(stream: Stream<A>, p: (A) -> Boolean): Boolean =
    when (stream) {
        Empty -> false
        is Cons -> when {
            p(stream.hd()) -> true
            else           -> exists(stream.tl(), p)
        }
    }
```

А это реализация основной функции в классе `Stream`:

```
fun exists(p: (A) -> Boolean): Boolean = exists(this, p)
```

9.6.1 Свертка потоков

В главе 5 мы узнали, как абстрагировать рекурсию в виде функции свертки и как выполнять свертку списков слева и справа. Свертка потоков выполняется немного иначе, хотя принцип тот же. Основное отличие в том, что потоки не вычисляются.

Вызвать переполнение стека и исключение `StackOverflowException` может рекурсивная операция, но не рекурсивное описание операции. Как следствие, во многих случаях функция `foldRight`, которую нельзя сделать безопасной в классе `List`, не будет вызывать переполнение стека в `Stream`. Но переполнение может возникнуть, если операция, используемая для свертки, предполагает вычисление элементов потока, как, например, суммирование элементов `Stream<Int>`. Однако переполнения можно не опасаться, если вместо вычисления операции использовать невычисляемое описание.

И наоборот, реализацию `foldRight` в классе `List`, основанную на `foldLeft` (чтобы исключить возможность переполнения стека), нельзя использовать с потоками, потому что она требует перевернуть поток. Это вызовет вычисление всех элементов, что может быть невыполнимым для бесконечных потоков. Безопасную версию `foldLeft` тоже нельзя использовать, потому что она производит вычисления в обратном порядке.

Упражнение 9.20

Напишите функцию `foldRight` для потоков. Она будет напоминать функцию `List.foldRight`, но вычисления должны выполняться лениво.

Подсказка

Ленивость можно обеспечить, если использовать элементы `Lazy<A>` вместо `A`. Вот сигнатура функции в классе `Stream`:

```
abstract fun <B> foldRight(z: Lazy<B>,
                           f: (A) -> (Lazy<B>) -> B): B
```

Решение

Реализация в классе `Empty` очевидна:

```
override fun <B> foldRight(z: Lazy<B>,
                           f: (Nothing) -> (Lazy<B>) -> B): B = z()
```

А вот реализация в классе `Cons`:

```
override fun <B> foldRight(z: Lazy<B>,
                           f: (A) -> (Lazy<B>) -> B): B =
    f(hd())(Lazy { tl().foldRight(z, f) })
```

Эта функция может вызвать переполнения стека, поэтому ее не следует использовать, например, для вычисления суммы чисел в потоке с более чем тысячей элементов. Однако, как вы увидите далее, она имеет ряд интересных применений.

Упражнение 9.21

Реализуйте функцию `takeWhile` в терминах `foldRight`. Назовите ее `takeWhileViaFoldRight`. Проверьте ее поведение с длинными списками.

Решение

Начальное значение – это экземпляр `Lazy` с пустым списком. Функция проверяет текущий элемент (`p(a)`). Если проверка завершается успехом (т. е. если элемент соответствует условию, выражаемому предикатом `p`), элемент «встраивается» в текущий поток результата как `Lazy { a }`:

```
fun takeWhileViaFoldRight(p: (A) -> Boolean): Stream<A> =
    foldRight(Lazy { Empty }, { a ->
        { b: Lazy<Stream<A>> ->
            if (p(a))
                cons(Lazy { a }, b)
            else
                Empty
        }
    })
```

Как нетрудно убедиться, запустив тесты, включенные в примеры кода для этой книги (https://github.com/pysaumont/fpinkotlin), эта функция не будет вызывать переполнение стека даже для потоков, длина которых превышает миллион элементов. Это объясняется тем, что сама функция `foldRight` не вычисляет сам результат. Вычисление может выполняться функцией, используемой для свертки. Если эта функция создает новый поток (как в случае `takeWhile`), этот поток не вычисляется.

Упражнение 9.22

Реализуйте функцию `headSafe` на основе `foldRight`. Она должна возвращать `Result.Success` с элементом из головы потока или `Result.Empty`, если поток пуст.

Решение

Начальное значение – это экземпляр `Lazy` с пустым списком. Он будет возвращен, если поток пуст. Функция, используемая для свертки потока, игнорирует второй аргумент, поэтому она вернет `Result(a)` при первом применении к элементу `hd`, и этот результат больше не изменится:

```
fun headSafeViaFoldRight(): Result<A> =
    foldRight(Lazy { Result<A>() }, { a -> { Result(a) } })
```

Упражнение 9.23

Напишите `map` в терминах `foldRight`, причем эта функция не должна вычислять элементы потока.

Решение

Начальное значение – это экземпляр `Lazy` с пустым списком. Функция, используемая для свертки, будет лениво применяться к текущему элементу и объединяться с текущим результатом:

```
fun <B> map(f: (A) -> B): Stream<B> =
    foldRight(Lazy { Empty }, { a ->
        { b: Lazy<Stream<B>> ->
            cons(Lazy { f(a) }, b)
        }
    })
```

Упражнение 9.24

Реализуйте функцию `filter` в терминах `foldRight`. Она не должна вычислять больше элементов потока, чем необходимо.

Решение

И снова начальное значение – это экземпляр `Lazy` с пустым списком. Функция, используемая для свертки, применяет фильтр к текущему аргументу. Если проверка вернет `true`, элемент «встраивается» в текущий поток результата. Иначе текущий поток результата остается без изменений (вызов `b()` не вычисляет никаких элементов):

```
fun filter(p: (A) -> Boolean): Stream<A> =
    foldRight(Lazy { Empty }, { a ->
        { b: Lazy<Stream<A>> ->
            if (p(a)) cons(Lazy { a }, b) else b()
        }
    })
```

Эта функция вычисляет элементы потока, пока не найдет первое совпадение. Дополнительную информацию вы найдете в коде примеров для этой книги.

Упражнение 9.25

Напишите функцию append в терминах foldRight. Она не должна вычислять свой аргумент.

Подсказка

Не забывайте о вариантности!

Решение

Эта функция принимает аргумент Lazy<Stream<A>>, т. е. A находится в позиции in, тогда как параметр класса Stream объявлен в позиции out. По этой причине необходимо запретить проверку вариантности с помощью аннотации @UnsafeVariance.

Начальное значение – это экземпляр Lazy с пустым списком, в конец которого будет добавляться текущий поток. Функция свертки создает новый поток, добавляя текущий элемент в конец текущего результата с помощью функции cons:

```
fun append(stream2: Lazy<Stream<@UnsafeVariance A>>): Stream<A> =
    this.foldRight(stream2) { a: A ->
        { b: Lazy<Stream<A>> ->
            Stream.cons(Lazy { a }, b)
        }
    }
```

Упражнение 9.26

Напишите функцию flatMap в терминах foldRight.

Решение

И снова начальное значение – это экземпляр Lazy с пустым списком. Функция, применяемая к текущему элементу, возвращает поток, добавляя текущий элемент в конец текущего результата. Она производит эффект линеаризации (преобразует поток потоков Stream<Stream<B>> в плоский поток Stream<B>):

```
fun <B> flatMap(f: (A) -> Stream<B>): Stream<B> =
    foldRight(Lazy { Empty as Stream<B> }, { a ->
        { b: Lazy<Stream<B>> ->
            f(a).append(b)
        }
    })
```

9.6.2 Трассировка вычислений и применение функций

Важно особо подчеркнуть следствия, сопровождающие применение ленивых вычислений. Последовательное применение map и filter к строгой коллекции, например к списку, как в примере ниже, подразумевает двукратный обход ее содержимого:

```
import com.fpinkotlin.common.List

private val f = { x: Int ->
```

```
    println("Mapping " + x)
    x * 3
}

private val p = { x: Int ->
    println("Filtering " + x)
    x % 2 == 0
}

fun main(args: Array<String>) {
    val list = List(1, 2, 3, 4, 5).map(f).filter(p)
    println(list)
}
```

Функции `f` и `p` не являются чистыми, потому что выводят сообщения в консоль. Это поможет вам понять происходящее. Данная программа генерирует следующий вывод:

```
Mapping 1
Mapping 2
Mapping 3
Mapping 4
Mapping 5
Filtering 15
Filtering 12
Filtering 9
Filtering 6
Filtering 3
[6, 12, NIL]
```

Как видите, сначала все элементы обрабатываются функцией `f`, что означает обход всех элементов в списке. А затем все элементы обрабатываются функцией `p`, т. е. снова выполняется обход всех элементов в списке. Программа, представленная ниже и использующая `Stream` вместо `List`, действует иначе:

```
fun main(args: Array<String>) {
    val stream = Stream.from(1).takeAtMost(5).map(f).filter(p)
    println(stream.toList())
}
```

Вот ее вывод:

```
Mapping 1
Filtering 3
Mapping 2
Filtering 6
Mapping 3
Filtering 9
Mapping 4
Filtering 12
Mapping 5
Filtering 15
[6, 12, NIL]
```

В данном случае обход содержимого потока выполняется только один раз. Сначала к первому элементу 1 применяется функция f, которая возвращает 3. Затем число 3 отфильтровывается (потому что не является четным). После этого функция f применяется к числу 2 и возвращает 6, которое остается после фильтрации.

Как видите, ленивые потоки позволяют комбинировать описания вычислений, а не их результаты. Объем вычислений элементов сводится к минимуму. Если для конструирования потока использовать невычисляемые значения и убрать вывод содержимого потока в конце, в результате получится следующее:

```
generating 1
Mapping 1
Filtering 3
generating 2
Mapping 2
Filtering 6
```

Как видите, вычисляются только два первых элемента. Остальные вычисления являются результатом окончательного вывода.

Упражнение 9.27

Напишите функцию find, которая принимает предикат (функцию (A) -> Boolean) и возвращает Result<A>. То есть, если будет найден элемент, соответствующий предикату, результатом должен быть экземпляр Success, иначе – экземпляр Empty.

Подсказка

Чтобы решить это упражнение, достаточно объединить две функции, написанные нами в предыдущих разделах.

Решение

Объедините функции filter и head:

```
fun find(p: (A) -> Boolean): Result<A> = filter(p).head()
```

9.6.3 Использование потоков для решения конкретных задач

В следующих упражнениях вам предстоит использовать потоки для решения конкретных задач. Это поможет вам увидеть, как сильно отличаются решения на основе потоков от традиционных подходов.

Упражнение 9.28

Напишите функцию fibs, которая генерирует бесконечный поток чисел Фибоначчи: 1, 1, 2, 3, 5, 8 и т. д.

Подсказка

Возможно, вам пригодится промежуточный поток с парами целых чисел, который можно создать с помощью функции iterate.

Решение

Решение заключается в создании потока пар (`x, y`), где `x` и `y` – два последовательных числа Фибоначчи. После создания этого потока останется только преобразовать его с помощью `map` в поток первых элементов из пар:

```
fun fibs(): Stream<Int> =
    Stream.iterate(Pair(1, 1)) { x ->
        Pair(x.second, x.first + x.second)
    }.map({ x -> x.first })
```

Этот код можно упростить, использовав деструктуризацию:

```
fun fibs(): Stream<Int> = Stream.iterate(Pair(1, 1)) {
    (x, y) -> Pair(y, x + y)
}.map { it.first }
```

В этом примере кортеж (`x, y`) инициализируется значениями первого и второго элементов пары `Pair`. Эта обратная операция по отношению к `Pair(x, y)`, объединяющей в структуру два значения, `x` и `y`; отсюда такое название – *деструктуризация*.

Упражнение 9.29

Функцию `iterate` можно обобщить еще больше. Напишите функцию `unfold`, которая принимает начальное состояние типа `S` и функцию `(S) -> Result<Pair<A, S>>` и возвращает поток `Stream<A>`. Возвращаемое значение `Result` может служить индикатором – будет ли поток продолжаться дальше или уже закончился.

Использование состояния типа `S` означает, что источник, на основе которого генерируются данные, необязательно должен иметь тот же тип, что и генерируемые данные. Для опробования этой функции напишите новые версии функций `fibs` и `from` в терминах `unfold`. Вот сигнатура функции `unfold`:

```
fun <A, S> unfold(z: S, f: (S) -> Result<Pair<A, S>>): Stream<A>
```

Решение

Сначала применим функцию `f` к начальному состоянию `z`. В результате получим `Result<Pair<A, S>>`. Затем применим функцию `map` и объединим в поток первый элемент результата `Pair<A, S>` (значение `A`) с ленивым рекурсивным вызовом `unfold`, передав в него второй элемент пары в качестве исходного состояния. В результате применения `map` получится либо `Success<Stream<A>>`, либо `Empty`. Затем вызовом `getOrElse`, вернем либо поток с элементами, либо пустой поток:

```
fun <A, S> unfold(z: S, f: (S) -> Result<Pair<A, S>>): Stream<A> =
    f(z).map { x ->
        Stream.cons(Lazy { x.first }, Lazy { unfold(x.second, f) })
    }.getOrElse(Stream.Empty)
```

Код можно упростить, использовав деструктуризацию:

```
fun <A, S> unfold(z: S, f: (S) -> Result<Pair<A, S>>): Stream<A> =
    f(z).map { (a, s) ->
```

```
        Stream.cons(Lazy { a }, Lazy { unfold(s, f) })
    }.getOrElse(Stream.Empty)
```


Новая версия `from` использует целое число в качестве начального состояния и функцию `(Int) -> Pair<Int, Int>`. В данном случае состояние имеет тот же тип, что и генерируемые данные:

```
fun from(n: Int): Stream<Int> = unfold(n) { x -> Result(Pair(x, x + 1)) }
```

Функция `fibs` более полно использует возможности функции `unfold`. Состояние имеет тип `Pair<Int, Int>`, а результат – тип `Pair<Int, Pair<Int, Int>>`:

```
fun fibs(): Stream<Int> =
    Stream.unfold(Pair(1, 1)) { x ->
        Result(Pair(x.first, Pair(x.second, x.first + x.second)))
    }
```

Обратите внимание, насколько более компактными и элегантными получились реализации этих функций!

Упражнение 9.30

Использование `foldRight` для реализации различных функций – интересное решение. К сожалению, таким способом нельзя реализовать функцию `filter`, потому что после сопоставления 1000 или 2000 последовательных элементов потока с предикатом произойдет переполнение стека. Напишите версию функции `filter`, не страдающую этим недостатком.

Подсказка

Проблема возникает, когда в потоке имеется длинная последовательность элементов, для которых предикат возвращает `false`. Подумайте, как избавиться от этих элементов.

Решение

Решение состоит в том, чтобы с помощью функции `dropWhile` удалить длинные последовательности элементов, для которых предикат возвращает `false`. Для этого следует инвертировать условие (`!p(x)`), затем проверить – не получился ли пустой поток. Если поток пустой, вернем его. (Можно вернуть любой пустой поток, потому что пустой поток является синглтоном.) Если поток не пустой, создадим новый, объединив голову с отфильтрованным хвостом.

Функция `head` возвращает пару, поэтому в качестве элемента `head` потока мы должны использовать первый элемент из этой пары. Теоретически мы должны были бы использовать правый элемент пары для дальнейшего доступа. Невыполнение этого требования приведет к повторному вычислению головы. Но так как потом мы обращаемся только к хвосту, можно использовать `stream.getTail()`. Это позволит избежать использования локальной переменной для ссылки на результат `stream.head()`:

```
fun filter(p: (A) -> Boolean): Stream<A> =
    dropWhile { x -> !p(x) }.let { stream ->
        when (stream) {
```

```
            is Empty -> stream
            is Cons -> cons(stream.hd, Lazy { stream.tl().filter(p)})
        }
    }
```

Другая возможность – использовать функцию `head`. Эта функция возвращает значение `Result<A>`, которое можно использовать для создания нового потока через рекурсивный вызов. В конечном итоге это даст `Result<Stream<A>>` с пустым списком, когда ни один элемент не удовлетворяет предикату. После этого останется только вызвать `getOrElse`, передав пустой поток в качестве значения по умолчанию:

```
fun filter2(p: (A) -> Boolean): Stream<A> =
    dropWhile { x -> !p(x) }.let { stream ->
        when (stream) {
            is Empty -> stream
            is Cons -> stream.head().map({ a ->
                cons(Lazy { a }, Lazy { stream.tl().filter(p) })
            }).getOrElse(Empty)
        }
    }
```


Итоги

- При использовании строгих вычислений значения вычисляются при любой попытке сослаться на них.
- При использовании ленивых вычислений значения вычисляются, только если это действительно необходимо.
- Некоторые языки по своей природе являются строгими, а другие – ленивыми. Есть языки, использующие ленивые вычисления по умолчанию и строгие при необходимости, а есть языки, использующие строгие вычисления по умолчанию и ленивые при необходимости.
- Kotlin – строгий язык, в котором строгость проявляется в отношении аргументов функций.
- Хотя Kotlin не является ленивым языком, он поддерживает конструкцию `by lazy`, позволяющую реализовать ленивые вычисления с функциями.
- Ленивые вычисления позволяют конструировать и использовать бесконечные структуры данных.
- Свертка справа не требует вычисления элементов потока; вычисления производятся лишь некоторыми функциями, используемыми для свертки.
- Операция свертки позволяет объединить несколько итеративных операций без многократного обхода элементов потока.
- Вы без труда можете определять и конструировать бесконечные потоки.

10 Обработка данных с использованием деревьев

Эта глава охватывает следующие темы:

- понятия размера, высоты и глубины дерева;
- порядок вставки в бинарное дерево поиска;
- обход деревьев в разных направлениях;
- реализация бинарного дерева поиска;
- слияние, свертка и балансировка деревьев.

В главе 5 мы познакомились с односвязным списком – одной из самых широко используемых неизменяемых структур данных. Такой список поддерживает множество эффективных операций, но имеет некоторые ограничения. Основной недостаток – возрастание сложности доступа к элементам с ростом их количества. Например, может потребоваться проверить все элементы, чтобы отыскать нужный, если окажется, что он находится в конце списка. В числе других неэффективных операций можно назвать сортировку, доступ к элементам по индексам и поиск максимального или минимального элемента. Например, чтобы найти максимальный (или минимальный) элемент, необходимо выполнить обход всего списка. В этой главе вы столкнетесь с новой структурой данных, которая решает эти проблемы: бинарным деревом.

В начале главы вы познакомитесь с теоретическими основами бинарных деревьев. Некоторые читатели могут подумать, что бинарные деревья – это общеизвестная тема, хорошо знакомая всем программистам. Если вы один из них и хорошо знакомы с теорией бинарных

деревьев, то можете сразу перейти к упражнениям, которые, вероятно, покажутся вам простыми. Для остальных эти упражнения могут показаться немного сложнее, чем в предыдущих главах. Не сумев решить какое-то упражнение, вы сможете заглянуть в решение. Но я рекомендую в таких случаях просто оставить упражнение, вернуться к нему позже и попробовать решить его еще раз. Имейте в виду, что каждое следующее упражнение обычно основано на предыдущих, поэтому, если вы не поймете решение одного упражнения, вам, вероятно, будет трудно решить следующие.

10.1 Бинарное дерево

Деревья данных – это структуры, в которых в отличие от списков каждый элемент связан с несколькими элементами. В некоторых деревьях каждый элемент (их иногда называют узлами) может быть связан с переменным числом других элементов. Однако чаще они связаны с фиксированным числом элементов. В бинарных деревьях, как следует из названия, каждый элемент связан с двумя элементами. Эти связи называются *ветвями*. В бинарных деревьях это левая и правая ветви. На рис. 10.1 показан пример бинарного дерева.

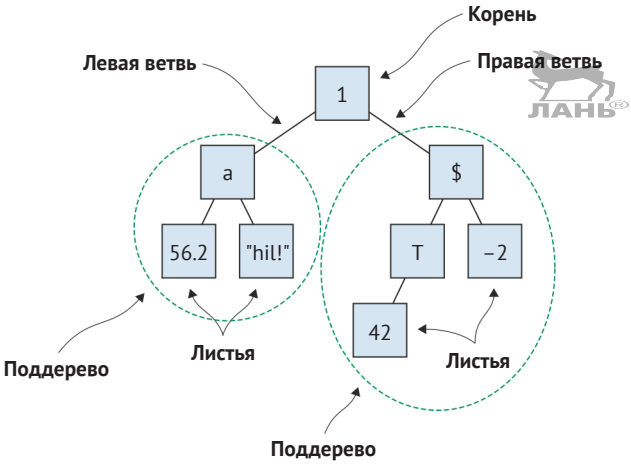


Рис. 10.1 Бинарное дерево – это рекурсивная структура, состоящая из корня и двух ветвей. Левая ветвь является ссылкой на левое поддерево, а правая – на правое. Конечные элементы имеют пустые ветви (не изображены на рисунке) и называются листьями

Дерево, изображенное на рис. 10.1, не является типичным представителем, потому что его элементы имеют разные типы. Это дерево типа `Any`. На практике гораздо чаще приходится иметь дело с деревьями более конкретных типов, например с деревьями целых чисел.

На рис. 10.1 видно, что дерево является рекурсивной структурой. Каждая ветвь ведет к новому дереву (его часто называют *поддеревом*). Также

можно видеть, что некоторые ветви ведут к единственному элементу. Это не проблема, потому что единственный элемент – это то же дерево, но с пустыми ветвями. Такие конечные элементы часто называют *листьями*. Также обратите внимание на элемент `T`: он имеет только одну ветвь – левую, а правая ветвь отсутствует, потому что она пуста и не представлена. Опираясь на рис. 10.1, легко вывести определение бинарного дерева. Деревом является:

- единственный элемент, такой как `56.2`, `"hi"`, `42` и `-2` на рис. 10.1;
- элемент с единственной ветвью (правой или левой), такой как `T` на рис. 10.1;
- элемент с двумя ветвями (правой и левой), такой как `1` или `$`.

Каждая ветвь содержит (под)дерево. Дерево, в котором все элементы имеют либо две ветви, либо ни одной, называется *полным деревом*. Дерево на рис. 10.1 неполное, но его левое поддерево – полное.

10.2 Сбалансированные и несбалансированные деревья

Бинарные деревья могут быть сбалансированы в большей или в меньшей степени. Идеально сбалансированное дерево – это дерево, в котором обе ветви всех поддеревьев содержат одинаковое количество элементов. На рис. 10.2 показаны три примера деревьев с одинаковыми элементами. Первое дерево идеально сбалансировано, а последнее – полностью несбалансированное. Идеально сбалансированные бинарные деревья иногда называют *идеальными деревьями*.

Дерево справа на рис. 10.2 фактически является односвязным списком. Односвязный список можно считать особым случаем полностью несбалансированного дерева.

10.3 Размер, высота и глубина дерева

Дерево можно охарактеризовать числом элементов, содержащихся в нем, и числом листьев, на которые эти элементы распадаются:

- число элементов называется *размером* дерева;
- число уровней, не считая корня, называется *высотой*.

Все деревья, изображенные на рис. 10.2, имеют размер 7. Первое (идеально сбалансированное) дерево имеет высоту 2, второе (плохо сбалансированное) имеет высоту 3 и третье (полностью несбалансированное) имеет высоту 6.

Термин «высота» также используется для характеристики отдельных элементов. Он обозначает самое длинное расстояние от элемента до листа. Высота корня совпадает с высотой дерева, а высота элемента – это высота поддерева, в котором этот элемент является корнем.

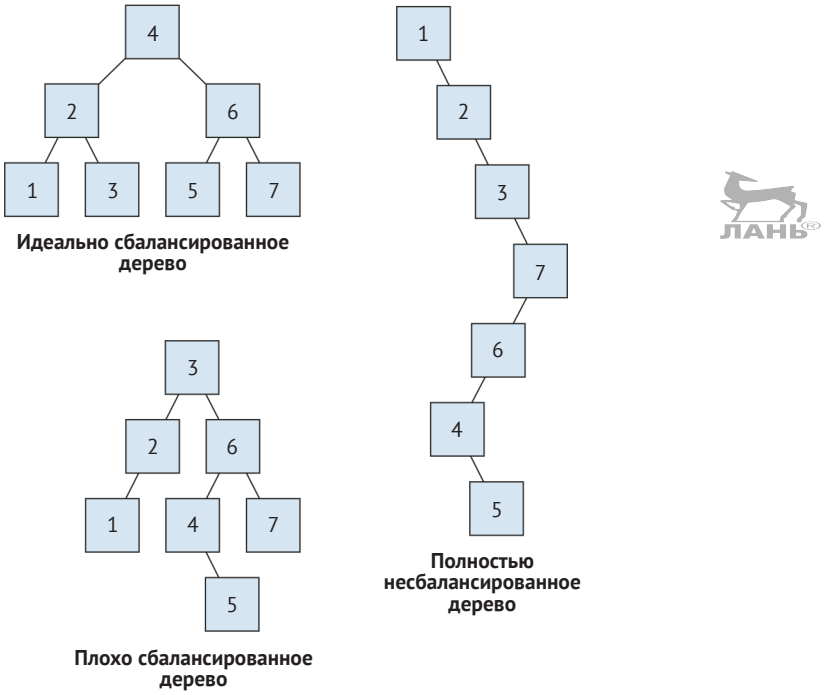


Рис. 10.2 Бинарные деревья могут быть сбалансированы в большей или в меньшей степени. Верхнее дерево сбалансировано идеально, потому что обе ветви во всех поддеревьях содержат одинаковое число элементов. Дерево справа – это односвязный список, особый случай полностью несбалансированного дерева

Глубина элемента – это расстояние от корня до элемента. Первый элемент, также называемый *корнем*, имеет глубину 0. В идеально сбалансированном дереве на рис. 10.2 элементы 2 и 6 имеют глубину 1, а элементы 1, 3, 5 и 7 – глубину 2.

10.4 Пустые деревья и рекурсивное определение

В разделе 10.1 я сказал, что дерево состоит из корневого элемента и ветвей, количество которых может быть 0, 1 или 2. Это упрощенное определение, не учитывающее всех деталей. В частности, оно не определяет пустые деревья. Однако мы можем немного изменить определение, чтобы учесть этот нюанс. Итак, дерево:

- может быть пустым;
- может состоять из корня и двух ветвей, которые сами являются деревьями.

Согласно этому новому рекурсивному определению, каждый элемент, который выглядит так, будто не имеет ветвей (см. рис. 10.2), фактически

имеет два пустых дерева в левой и правой ветвях. Элементы, которые выглядят так, будто имеют одну ветвь, – в действительности пустое дерево во второй ветви. По соглашению пустые ветви не изображаются на схемах. Другое важное соглашение заключается в том, что высота и глубина пустого дерева равны –1. Далее вы узнаете, что это необходимо для некоторых операций, таких как балансировка.

10.5 Лиственные деревья

Иногда бинарные деревья изображают по-другому, как показано на рис. 10.3. В этом представлении дерево изображается в виде набора ветвей без значений. Значения содержат только конечные узлы. Поскольку конечные узлы называются *листьями*, такие деревья называются *лиственными деревьями*.

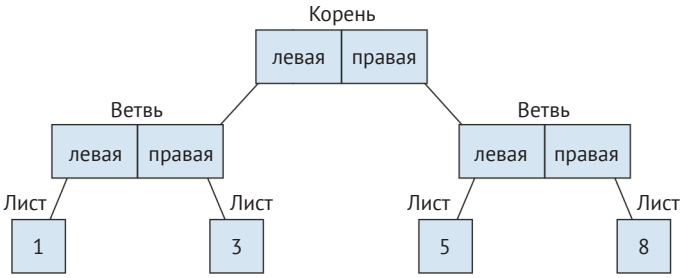


Рис. 10.3 Лиственное дерево хранит значения только в конечных узлах, которые называют листьями

Представление в виде лиственного дерева иногда предпочтительнее, потому что упрощает реализацию некоторых функций. Однако в этой книге мы рассмотрим только классические деревья и не будем касаться лиственных.

10.6 Упорядоченные бинарные деревья, или бинарные деревья поиска

Упорядоченные бинарные деревья также называются *бинарными деревьями поиска* (Binary Search Tree, BST) и характеризуются следующими свойствами:

- содержат элементы, которые можно упорядочить;
- все элементы в одной ветви имеют значения меньше значения корня;
- все элементы в другой ветви имеют значения больше значения корня;
- теми же свойствами обладают все поддеревья.

По соглашению элементы со значениями ниже значения корня находятся в левой ветви, а элементы со значениями выше значения корня – в правой. На рис. 10.4 показан пример упорядоченного дерева.

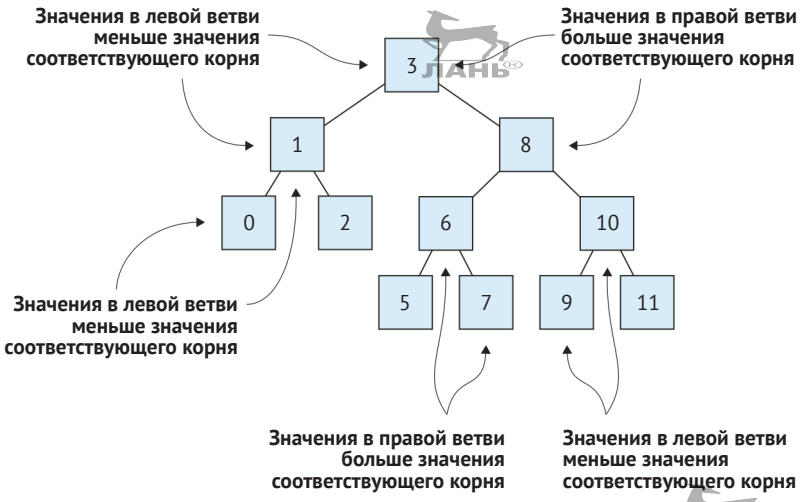


Рис. 10.4 Пример упорядоченного бинарного дерева (или дерева поиска). Все элементы в одной ветви (по соглашению в левой) имеют значения меньше значения корня, а все элементы в другой ветви (по соглашению в правой) имеют значения больше значения корня. Теми же свойствами обладают все поддеревья

ПРИМЕЧАНИЕ Из определения упорядоченных бинарных деревьев вытекает одно важное следствие: они не могут содержать повторяющихся значений.

Упорядоченные деревья особенно интересны тем, что позволяют быстро находить элементы. Чтобы узнать, содержится ли элемент в дереве, нужно:

1. Сравнить искомое значение с корнем. Если они равны, поиск завершен.
2. Если искомое значение меньше корня, продолжить поиск в левой ветви.
3. Если искомое значение больше корня, продолжить поиск в правой ветви.

Как видите, по сравнению с поиском в односвязном списке на поиск в идеально сбалансированном упорядоченном бинарном дереве требуется время, пропорциональное высоте дерева. То есть время поиска пропорционально $\log_2(n)$, где n – размер (количество элементов) дерева. Время поиска в односвязном списке, напротив, пропорционально количеству элементов.

Как следствие, рекурсивный поиск в идеально сбалансированном бинарном дереве никогда не вызовет переполнение стека. Как вы видели в главе 4, стандартный размер стека допускает от 1000 до 3000 рекурсив-

ных шагов. Поскольку идеально сбалансированное бинарное дерево высотой 1000 содержит 2^1000 элементов, у вас просто не хватит основной памяти для хранения такого дерева.

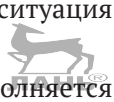
Это были хорошие новости. Однако есть и плохая новость: не все бинарные деревья идеально сбалансированы. Полностью несбалансированное двоичное дерево представляет собой односвязный список, поэтому оно имеет такую же производительность и ту же проблему в отношении рекурсии, что и список. То есть, чтобы получить максимальную отдачу от деревьев, необходим способ сбалансировать их.

10.7 Порядок вставки и структура дерева

Структура дерева (т. е. его сбалансированность) зависит от порядка вставки элементов. Вставка выполняется так же, как поиск:

- если дерево пустое, создается новое дерево, а вставляемый элемент становится его корнем и получает две пустые ветви;
- если дерево непустое, вставляемый элемент сравнивается с корнем. Если они равны, на этом операция вставки завершается и элемент не вставляется, потому что в дерево можно вставить только элемент, который меньше или больше корня.

В реальности ситуация может оказаться сложнее. Объекты, вставляемые в дерево, могут быть равны с точки зрения упорядочения, но различаться с точки зрения других критериев, в таком случае может быть желательно заменить корень на вставляемый элемент. Эта ситуация встречается не так уж редко, как вы вскоре увидите:



- если вставляемый элемент меньше корня, рекурсивно выполняется вставка в левую ветвь;
- если вставляемый элемент больше корня, рекурсивно выполняется вставка в правую ветвь.

Этот процесс имеет одно интересное следствие: сбалансированность дерева зависит от порядка вставки элементов. Очевидно, что вставка упорядоченных элементов приводит к полностью несбалансированному дереву. С другой стороны, попытки организовать вставку в разном порядке могут приводить к созданию одинаковых деревьев. На рис. 10.5 показаны возможные порядки вставки, приводящие к созданию одного и того же дерева.

Справа на рис. 10.5 я попытался изобразить все возможные варианты порядка вставки, дающие в результате это дерево:

1. Первым вставляется элемент `3`. Если первым вставить любой другой элемент, получится другое дерево.
2. Вторым вставляется элемент `1` или `8`. И снова, если вторым вставить другой элемент, получится другое дерево. Если вставить элемент `1`, тогда следующим должен вставляться элемент `0`, `2` или `8`. Если вставить элемент `8`, следующим должен вставляться элемент `6`, `10` или `1`.

Если вставить элемент `6`, следующим должен вставляться элемент `0`, `2`, `10` (если еще не был вставлен), `5` или `7`; и т. д.

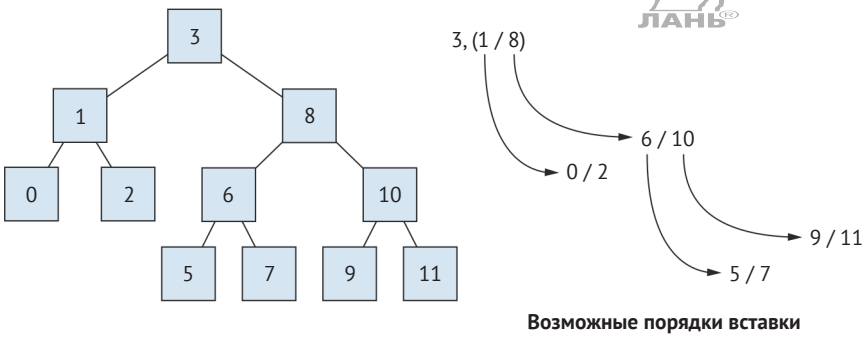


Рис. 10.5 Попытки вставки в разном порядке могут давать одно и то же дерево. Набор из десяти элементов можно вставить в дерево 3 628 800 разными способами, но в результате получается только 16 696 различных деревьев. Эти деревья разнятся по степени сбалансированности – от идеально сбалансированных до полностью несбалансированных. Упорядоченные деревья эффективны для хранения и извлечения случайных данных, но плохо подходят для хранения и извлечения предварительно упорядоченных данных

10.8 Рекурсивный и нерекурсивный обход дерева

При работе с деревьями, такими как на рис. 10.5, часто требуется выполнить обход всех элементов. Обычно это делается с целью преобразования или свертки дерева и реже – с целью найти определенное значение. Знакомясь со списками в главе 5, вы узнали, что есть два пути их обхода: слева направо или справа налево. Деревья предлагают гораздо больше путей, и среди них мы будем различать рекурсивные и нерекурсивные.

10.8.1 Рекурсивный обход дерева

Рассмотрим левую ветвь дерева на рис. 10.5. Эта ветвь сама является деревом с корнем `1`, левой ветвью `0` и правой ветвью `2`. Обход этого дерева можно выполнить в шести разных направлениях:

1. 1, 0, 2
2. 1, 2, 0
3. 0, 1, 2
4. 2, 1, 0
5. 0, 2, 1
6. 2, 0, 1

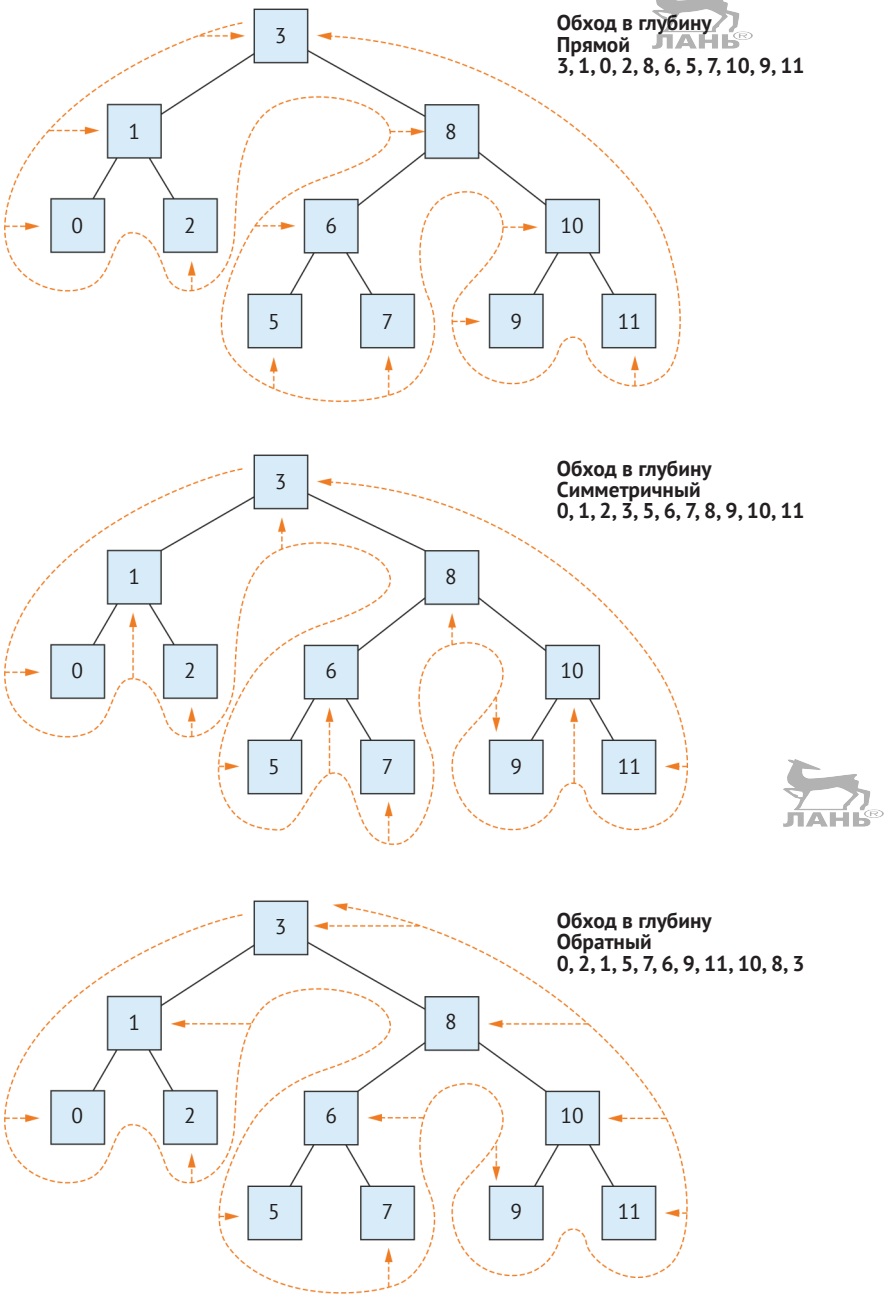


Рис. 10.6 При обходе в глубину движение сначала выполняется по высоте, а с учетом трех основных порядков может быть прямым, симметричным и обратным

Как видите, три из этих направлений симметричны другим трем: 1, 0, 2 и 1, 2, 0 симметричны. Обход начинается с корня, и затем выполняется обход ветвей слева направо или справа налево. То же верно для вариантов 0, 1, 2 и 2, 1, 0, которые отличаются только порядком выбора ветвей, и для вариантов 0, 2, 1 и 2, 0, 1. Если ограничиться только направлением слева направо (потому что противоположное направление является простым зеркальным отражением), у нас останется три варианта, которые принято называть по позиции корня:

- прямой (1, 0, 2 или 1, 2, 0);
- симметричный (0, 1, 2 или 2, 1, 0);
- обратный (0, 2, 1 или 2, 0, 1).

При рекурсивном обходе всего дерева, когда перемещение в первую очередь выполняется по высоте, мы получаем траектории обхода, показанные на рис. 10.6. Такой способ обхода обычно называют *обходом в глубину*, вместо более логичного *в высоту*. В терминологии деревьев высота и глубина дерева обозначают высоту корня и глубину самого глубокого листа, как мы узнали в разделе 10.3. И эти два значения равны.

10.8.2 Нерекурсивный обход дерева

Другой способ обхода дерева – сначала выполнить обход одного уровня, а затем перейти на следующий. И снова обход можно выполнять слева направо или справа налево. Этот вид обхода называют *поуровневым*. (Когда речь идет о поиске элемента, а не об обходе дерева, обычно используют термин «поиск в ширину».) Один и вариантов поуровневого обхода показан на рис. 10.7.

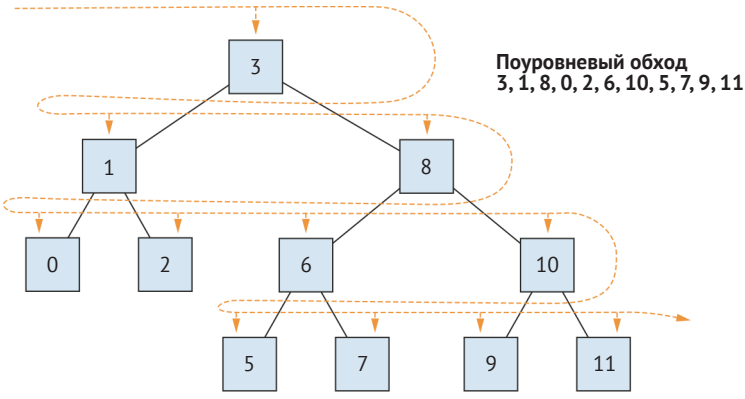


Рис. 10.7 При поуровневом обходе сначала выполняется обход всех элементов на одном уровне, а потом происходит переход на другой уровень

10.9 Реализация бинарного дерева поиска

Мы можем реализовать бинарное дерево, следуя за аналогией с односвязным списком, как структуру, которая имеет голову (или значение) и два хвоста (две ветви: левая `left` и правая `right`). В этом разделе мы определим абстрактный класс `Tree` с двумя подклассами: `T` и `Empty`. Класс `T` будет представлять непустое дерево, а класс `Empty`, как легко догадаться, – пустое.

Как и в случае со списком, пустое дерево можно представить объектом-синглтоном благодаря поддержке вариантности типов в Kotlin. Этот объект-синглтон будет иметь тип `Tree<Nothing>`. В листинге 10.1 представлена минимальная реализация `Tree`.

Листинг 10.1 Минимальная реализация Tree

```
sealed class Tree<out A: Comparable<@UnsafeVariance A>> { ① ②

    abstract fun isEmpty(): Boolean

    internal object Empty : Tree<Nothing>() { ③

        override fun isEmpty(): Boolean = true

        override fun toString(): String = "E" ④
    }

    internal class T<out A: Comparable<@UnsafeVariance A>>(
                            internal val left: Tree<A>, ⑤ ⑥
                            internal val value: A,
                            internal val right: Tree<A>) : Tree<A>() {

        override fun isEmpty(): Boolean = false

        override fun toString(): String =
                    "(T $left $value $right)" ④
    }

    companion object {

        operator fun <A: Comparable<A>> invoke(): Tree<A> =
                                        Empty ⑧
    }
}
```

① Класс `Tree` параметризован, при этом тип параметра должен реализовать интерфейс `Comparable`
② Класс `Tree` – ковариантный к типу `A`, но интерфейс `Comparable` сам является ковариантным, поэтому необходимо использовать аннотацию `@UnsafeVariance`
③ Пустое дерево представлено объектом-синглтоном, параметризованным типом `Nothing`
④ Минимальные реализации функции `toString` помогут увидеть содержимое дерева
⑤ Подкласс `T` представляет непустое дерево
⑥ Все свойства объявлены внутренними (`internal`), чтобы сделать их недоступными из других модулей
⑧ Функция `invoke` возвращает синглтон `Empty`

Это простой, но бесполезный класс, потому что не предусматривает возможности конструировать деревья. Однако, прежде чем добавить в него эту возможность, рассмотрим одну проблему, которую нужно решить до этого.

10.9.1 Деревья и вариантность

В главе 5 вы узнали, как связаны вариантность и списки. Та же проблема с вариантностью наблюдается и в деревьях. Имеет смысл сделать дерево ковариантным по своему параметру. В конце концов, должна иметься возможность использовать `Tree<Int>` там, где ожидается `Tree<Number>`. Этого легко добиться, объявив дерево ковариантным с помощью ключевого слова `out`. С другой стороны, вы не сможете добавить `Number` в дерево `Tree<Int>`. Если бы это было возможно, класс `Tree` был бы контрвариантным по своему параметру типа.

Проблема заключается в том, что параметр типа класса `Tree` должен реализовать интерфейс `Comparable`. Это не Java-интерфейс `Comparable`, а отдельный интерфейс для Kotlin:

```
public interface Comparable<in T> {
    public operator fun compareTo(other: T): Int
}
```

Как видите, `Comparable` является контрвариантным к `T`, т. е. параметр типа `T` встречается только в позиции `in`. Это делает невозможным объявить класс `Tree` ковариантным. Проблему можно решить, но для этого придется явно приводить объект-синглтон `Empty` к типу `Tree<T>` каждый раз, когда потребуется его использовать. Это приведение можно преобразовать в функцию:


```
operator fun <A: Comparable<A>> invoke() = Empty as Tree<A>
```

Но тогда компилятор будет выводить предупреждение о неконтролируемом приведении. Какое решение использовать – выбирать вам, но применение аннотации `@UnsafeVariance` выглядит проще, поскольку позволяет объявить класс `Tree` ковариантным, что намного удобнее.

Теперь займемся реализацией возможности конструировать деревья путем добавления элементов.

Упражнение 10.1

Напишите функцию `plus` для вставки значения в дерево. Как обычно, структура `Tree` является неизменяемой и постоянной, поэтому при добавлении элемента должно создаваться новое дерево, а прежнее – оставаться нетронутым. Выбор имени `plus` для этой функции позволит добавлять элементы с использованием оператора `+`.

Подсказка

Если добавляемый элемент равен корню, функция должна вернуть новое дерево с добавленным значением в качестве корня, а ветви исходного дерева должны остаться без изменений. Иначе, если значение меньше

корня, оно должно быть добавлено в левую ветвь, а значение больше корня – в правую. Определите функцию в родительском классе `Tree` со следующей сигнатурой:

```
operator fun plus(element: @UnsafeVariance A): Tree<A>
```

Если хотите, можете определить абстрактную функцию в классе `Tree` и конкретные реализации в обоих подклассах.

Решение

Для выбора реализации на основе типа ссылки `this` функция использует сопоставление с образцом. Если она пустая, возвращается новое непустое дерево `T`, состоящее из добавляемого элемента и двух пустых деревьев в качестве ветвей. Если дерево непустое, возможны три случая:

1. *Добавляемый элемент меньше корня.* В этом случае создается новое дерево с тем же корнем и той же правой ветвью, но с новой левой ветвью, получающейся в результате вставки нового элемента в исходную левую ветвь.
2. *Добавляемый элемент больше корня.* В этом случае создается новое дерево с тем же корнем и той же левой ветвью, но с новой правой ветвью, получающейся в результате вставки нового элемента в исходную правую ветвь.
3. *Добавляемый элемент равен корню.* В этом случае функция должна вернуть новое дерево с добавляемым элементом в качестве корня и двумя исходными ветвями:

```
operator fun plus(element: @UnsafeVariance A): Tree<A> = when (this) {
    Empty -> T(Empty, element, Empty)
    is T -> when {
        element < this.value -> T(left + element, this.value, right)
        element > this.value -> T(left, this.value, right + element)
        else -> T(this.left, element, this.right)
    }
}
```

Этот алгоритм отличается от происходящего во многих реализациях (например, в Java-классе `TreeSet`), которые оставляют исходное дерево нетронутым при попытке вставить элемент, равный элементу, уже имеющемуся в дереве. Такое поведение может быть допустимо для изменяемых элементов, но оно менее приемлемо, когда элементы являются неизменяемыми. Может показаться, что создание нового экземпляра `T` с той же левой и правой ветвью и корнем и с новым корнем, содержащим новый элемент, является пустой тратой времени и памяти, поскольку можно было бы просто вернуть оригинальное дерево – ссылку `this`, – что, по сути, эквивалентно возврату:

```
T(this.left, this.value, this.right)
```

Если это отвечает вашим намерениям, тогда возврат оригинального дерева будет отличной оптимизацией. Но получить тот же результат с изменяемыми элементами будет сложнее. Вам придется удалить эле-

мент перед вставкой равного ему элемента, чтобы изменить некоторые свойства. Мы рассмотрим этот случай, когда займемся реализацией ассоциативного массива в главе 11.

10.9.2 Об абстрактных функциях в классе Tree

Определение абстрактной функции в родительском классе `Tree` и конкретных реализаций в подклассах может показаться неплохим решением программистам, с опытом объектно-ориентированного программирования. Но на самом деле это не так. Вот как могла бы выглядеть такая реализация в объекте `Empty`:

```
override fun <A: Comparable<A>> plus(element: Nothing): Tree<Nothing> =
            T(Empty, element, Empty)
```

Такое решение не будет работать, потому что использование параметра `element` с типом `Nothing` (в `T(Empty, element, Empty)`) всегда будет вызывать ошибку – вы не сможете создать экземпляр `Nothing`. Да, мы знаем, что фактический элемент `element` никогда не будет иметь тип `Nothing`, но мы не можем указать тип `A`, потому что параметры типа не разрешены для объектов. По этой причине придется определить функцию в классе `Tree` и использовать сопоставление с образцом. В случае `Empty` параметр по-прежнему имеет тип `A`, и никаких проблем не возникает.

10.9.3 Перегрузка операторов

Kotlin поддерживает перегрузку операторов. Использовав ключевое слово `operator` и имя `plus` для функции, мы добавляем возможность вызывать эту функцию с помощью оператора `+`:

```
val tree = Tree<Int>() + 5 + 2 + 8
```

В этом случае не получится избежать явного объявления типов аргументов, даже если явно указать тип слева от оператора присваивания. Этот код не скомпилируется:

```
val tree: Tree<Int> = Tree() + 5 + 2 + 8
```

10.9.4 Рекурсия в деревьях

Возможно, вам интересно узнать, следует ли позаботиться о проблеме переполнения стека и использовать вспомогательную функцию в рекурсивной функции `plus`. Как я уже говорил выше, для сбалансированных деревьев в этом нет необходимости, потому что высота дерева (которая определяет максимальное количество шагов рекурсии) обычно намного меньше его размера. Но вы видели, что это не всегда так, особенно когда выполняется вставка упорядоченных элементов. В конечном счете можно получить дерево, содержащее единственную ветвь, высота которого будет на единицу меньше его размера, что может вызвать переполнение стека при рекурсивном обходе.

Однако пока мы не будем беспокоиться об этом. Вместо этого мы пойдем другим путем и найдем способ автоматической балансировки де-

ревьев. Простое дерево, над которым мы работаем сейчас, – это всего лишь учебный образец. Оно никогда не будет использоваться в промышленном коде. Но сбалансированные деревья сложнее в реализации, поэтому нам проще будет начать с простого несбалансированного дерева.

Упражнение 10.2

Часто бывает полезно иметь возможность построить дерево из коллекции другого типа, например из списка или из массива. Напишите функции для каждого из этих случаев. Используйте тип `List`, который мы определили в предыдущих главах вместо стандартного типа `List` в Kotlin.

Решение

Решение с использованием `vararg` напоминает решение для класса `List`:

```
operator fun <A: Comparable<A>> invoke(vararg az: A): Tree<A> =
    az.foldRight(Empty, { a: A, tree: Tree<A> -> tree.plus(a) })
```

Чтобы реализовать то же для стандартного типа `List` в Kotlin, достаточно изменить только тип аргумента и оставить ту же реализацию:

```
operator fun <A: Comparable<A>> invoke(az: List<A>): Tree<A> =
    az.foldRight(Empty, { a: A, tree: Tree<A> -> tree.plus(a) })
```

Тип `List`, который мы определили в главах 5 и 8, имеет небольшое отличие от стандартного типа: функция `foldRight` в нем каррирована, и, кроме того, функция `List.foldRight` небезопасна для стека. Поэтому в данном случае лучше использовать `foldLeft`:

```
operator fun <A: Comparable<A>> invoke(list: List<A>): Tree<A> =
    list.foldLeft(Empty as Tree<A>, { tree: Tree<A> ->
        { a: A ->
            tree.plus(a)
        }
    })
```

Функции `foldLeft` и `foldRight` производят разные деревья, потому что порядок обхода элементов в них не совпадает.

Упражнение 10.3

При работе с деревьями часто бывает нужно проверить присутствие некоторого конкретного элемента в дереве. Напишите функцию `contains`, выполняющую такую проверку. Вот ее сигнатура:

```
fun contains(a: @UnsafeVariance A): Boolean
```

Решение

Итак, мы должны сравнить параметр со значением `value` дерева (значением в корне дерева):

- если значение параметра меньше, рекурсивно выполнить сравнение с левой ветвью;

- если значение параметра больше, рекурсивно выполнить сравнение с правой ветвью;
- если значение параметра равно значению корня, вернуть `true`.

Как обычно, определим функцию в классе `Tree` и добавим аннотацию `@UnsafeVariance` в объявление параметра, чтобы запретить проверку вариантности:

```
fun contains(a: @UnsafeVariance A): Boolean = when (this) {
    is Empty -> false
    is T -> when {
        a < value -> left.contains(a)
        a > value -> right.contains(a)
        else -> value == a
    }
}
```

Также это упражнение можно решить, написав следующую реализацию:

```
fun <A: Comparable<@UnsafeVariance A>> contains(a: A): Boolean =
    when (this) {
        is Empty -> false
        is T -> a == value || left.contains(a) || right.contains(a)
    }
```

Она проще и действует правильно, но немного медленнее, если поиск предполагает рекурсивный обход правой ветви. Но есть еще одно важное отличие: эта реализация позволяет проверять элементы с типами, отличающимися от типа элементов в дереве, потому что параметр типа `A` в этой реализации не совпадает с параметром типа `A` класса `Tree`. Он затеняет параметр типа в объявлении `Tree`. Это легко заметить, если переименовать параметр типа:

```
fun <B: Comparable<@UnsafeVariance B>> contains(b: B): Boolean =
    when (this) {
        is Empty -> false
        is T -> b == value || left.contains(b) || right.contains(b)
    }
```

Эта реализация позволяет написать, например, такой код:

```
Tree(1, 2, 3).contains1("2")
```

который не вызовет ошибку при компиляции. Какой вариант выбрать – решать вам.

Упражнение 10.4

Напишите две функции, `size` и `height`, для вычисления размера и высоты дерева соответственно. Вот их сигнатуры в классе `Tree`:

```
abstract fun size(): Int
abstract fun height(): Int
```

Попробуйте найти лучшее решение по аналгии с определением длины списка в главе 8.

Решение

Реализация `size` в `Empty` должна просто возвращать `0`. А реализация `height` в `Empty`, как я уже говорил выше, должна возвращать `-1`. Реализация `size` в классе `T` должна возвращать `1` плюс размер каждой ветви. Реализация функции `height` должна возвращать `1` плюс наибольшую высоту из двух ветвей:

```
import kotlin.math.max

...

override fun size(): Int = 1 + left.size() + right.size()
override fun height(): Int = 1 + max(left.height(), right.height())
```

Исходя из этого, очевидно, почему высота пустого дерева должна быть равна –1. Если бы она была равна 0, она указывала бы число элементов в пути, а не число сегментов.

Но эти функции действуют неэффективно, потому что в каждом вызове заново вычисляют размер и высоту. Полученные ими результаты следует запомнить. Но самое неприятное в том, что эти функции могут вызвать исключение `StackOverflowException`, если дерево окажется достаточно большим и несбалансированным. Более удачное решение: добавить два абстрактных свойства в родительский класс `Tree`:

```
abstract val size: Int
abstract val height: Int
```


и затем реализовать их в подклассах:

```
internal object Empty : Tree<Nothing>() {

    override val size: Int = 0

    override val height: Int = -1

    ...
}

internal class T<out A: ...>(override val left: Tree<A>,
                             override val value: A,
                             override val right: Tree<A>) : Tree<A>() {

    override val size: Int = 1 + left.size + right.size

    override val height: Int = 1 + max(left.height, right.height)

    ...
}
```

Напомню, что для общедоступных свойств Kotlin автоматически создает функции доступа. Единственная их особенность заключается в том, что эти функции получают то же имя, что и свойство, и функция чтения вызывается без круглых скобок.

Упражнение 10.5

Напишите функции `max` и `min` для поиска максимального и минимального значений в дереве.

Подсказка

Подумайте, что должны возвращать эти функции в классе `Empty`.

Решение

В пустом дереве нет ни максимального, ни минимального значения. Учитывая это, функции должны возвращать `Result<A>`, и в классе `Empty` они должны возвращать `Result.empty()`.

Реализации в классе `T` немного сложнее. Функция `max` должна вернуть максимальное значение из правой ветви. Если правая ветвь непустая, получится рекурсивный вызов. Если правая ветвь пустая, поиск в ней вернет `Result.Empty`, и тогда максимальным значением в дереве будет значение его корня, а значит, можно вызвать `orElse` для значения, возвращаемого вызовом `right.max()`:

```
override fun max(): Result<A> = right.max().orElse { Result(value) }
```

Напомню, что функция `orElse` вычисляет свой аргумент лениво, т. е. она принимает `{ () -> Result<A>> }`, где часть `() ->` можно опустить. Функция `min` действует симметрично:

```
override fun min(): Result<A> = left.min().orElse { Result(value) }
```

10.9.5 Удаление элементов из дерева

В отличие от односвязных списков деревья позволяют получить конкретный элемент, как мы видели в упражнении 10.3, когда реализовали функцию `contains`. Это свойство позволяет также удалить конкретный элемент из дерева.

Упражнение 10.6

Напишите функцию `remove`, которая удаляет элемент из дерева. Эта функция должна принимать параметр с элементом для удаления. Если элемент присутствует в дереве, она должна удалить его и вернуть новое дерево без этого элемента. Это новое дерево должно соответствовать требованию: все элементы в левой ветви должны быть меньше корня, а все элементы в правой ветви – больше корня. Если элемент отсутствует в дереве, функция должна вернуть исходное дерево без изменений. Вот сигнатура функции `remove`:

```
Tree<A> remove(A a)
```

Подсказка

Вы должны также реализовать функцию слияния двух деревьев, для которых выполняется условие: все элементы в одном дереве больше или меньше любого элемента в другом. Не забывайте о вариантности.

Решение

Если дерево пустое, функция не сможет ничего удалить и вернет this. Иначе мы должны реализовать следующий алгоритм:

- если a < this.value, выполнить удаление в ветви left;
- если a > this.value, выполнить удаление в ветви right;
- иначе объединить левую и правую ветви, отбросив корень, и вернуть результат.

Слияние реализуется просто, потому что известно, что все элементы в левой ветви меньше любого элемента в правой. Сначала напишем функцию removeMerge со следующей сигнатурой:

```
fun removeMerge(ta: Tree<@UnsafeVariance A>): Tree<A>
```

Если дерево пустое, функция должна вернуть свой параметр. Иначе выполняется следующий алгоритм:

- если ta пустое дерево, вернуть this (которое не может быть пустым деревом);
- если ta.value < this.value, объединить ta с левой ветвью;
- если ta.value > this.value, объединить ta с правой ветвью.

Вот сама реализация:

```
fun removeMerge(ta: Tree<@UnsafeVariance A>): Tree<A> = when (this) {
    Empty -> ta
    is T -> when (ta) {
        Empty -> this
        is T -> when {
            ta.value < value -> T(left.removeMerge(ta), value, right)
            else             -> T(left, value, right.removeMerge(ta))
        }
    }
}
```

Обратите внимание, что корни двух деревьев не могут быть равны, потому что объединяются деревья, которые прежде были левой и правой ветвями исходного дерева. Теперь можно написать саму функцию remove:

```
fun remove(a: @UnsafeVariance A): Tree<A> = when(this) {
    Empty -> this
    is T -> when {
        a < value -> T(left.remove(a), value, right)
        a > value -> T(left, value, right.remove(a))
        else -> left.removeMerge(right)
    }
}
```

10.9.6 Слияние произвольных деревьев

В предыдущем разделе мы написали функцию слияния, условившись, что она будет объединять только такие деревья, в которых все элементы в одном дереве меньше любого элемента в другом. Слияние деревьев по-

добно операции конкатенации списков. Поэтому хорошо было бы иметь универсальную функцию слияния, объединяющую произвольные деревья.

Упражнение 10.7 (сложное)

Мы написали функцию слияния деревьев, в которых все элементы в одном дереве меньше любого элемента в другом. Напишите функцию `merge`, которая объединяет произвольные деревья. Вот ее сигнатура:

```
abstract fun merge(tree: Tree<@UnsafeVariance A>): Tree<A>
```

Решение

Реализация в `Empty` должна просто вернуть свой параметр:

```
override fun merge(tree: Tree<Nothing>): Tree<Nothing> = tree
```

Реализация в подклассе `T` выполняет следующий алгоритм, где «this» означает дерево, для которого вызвана эта функция:

- если дерево в параметре пустое, вернуть `this`;
- если корень дерева в параметре больше корня в `this`, удалить левую ветвь из дерева в параметре и объединить результат с правой ветвью в `this`. Затем объединить результат с левой ветвью дерева в параметре;
- если корень дерева в параметре меньше корня в `this`, удалить правую ветвь из дерева в параметре и объединить результат с левой ветвью в `this`. Затем объединить результат с правой ветвью дерева в параметре;
- если корень дерева в параметре равен корню в `this`, объединить левую ветвь дерева в параметре с левой ветвью в `this` и правую ветвь дерева в параметре с правой ветвью в `this`.

Вот реализация в подклассе `T`:

```
override fun merge(tree: Tree<@UnsafeVariance A>): Tree<A> = when (tree) {
    Empty -> this
    is T -> when {
        tree.value > this.value ->
            T(left, value, right.merge(T(Empty, tree.value, tree.right)))
                .merge(tree.left)
        tree.value < this.value ->
            T(left.merge(T(tree.left, tree.value, Empty)), value, right)
                .merge(tree.right)
        else ->
            T(left.merge(tree.left), value, right.merge(tree.right))
    }
}
```

Эта реализация не замещает корень дерева в `this` корнем из дерева в параметре, если они равны. Это может не соответствовать вашим требованиям. Если вы хотите, чтобы корень из дерева в параметре заменил корень исходного дерева, измените последнюю строку реализации, как показано ниже:

```
T(left.merge(tree.left), tree.value, right.merge(tree.right))
```

Работу этого алгоритма иллюстрируют рис. 10.8–10.17. Пустые ветви на этих рисунках обозначены явно. Напомню, что «this» означает дерево, для которого вызвана эта функция.

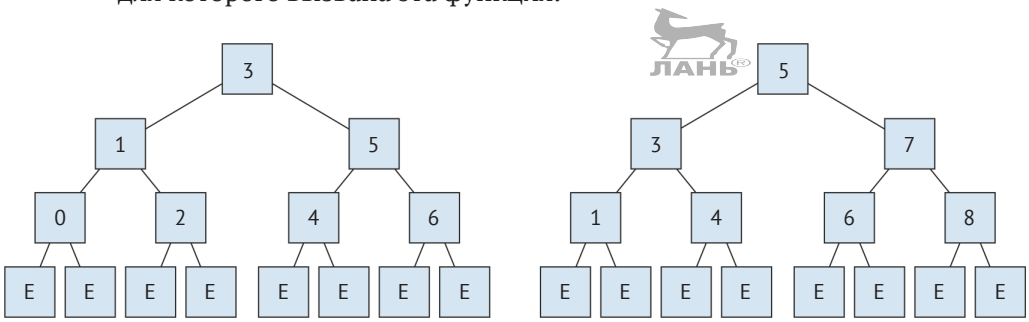


Рис. 10.8 Два дерева для слияния. Слева – дерево this, а справа – дерево в параметре

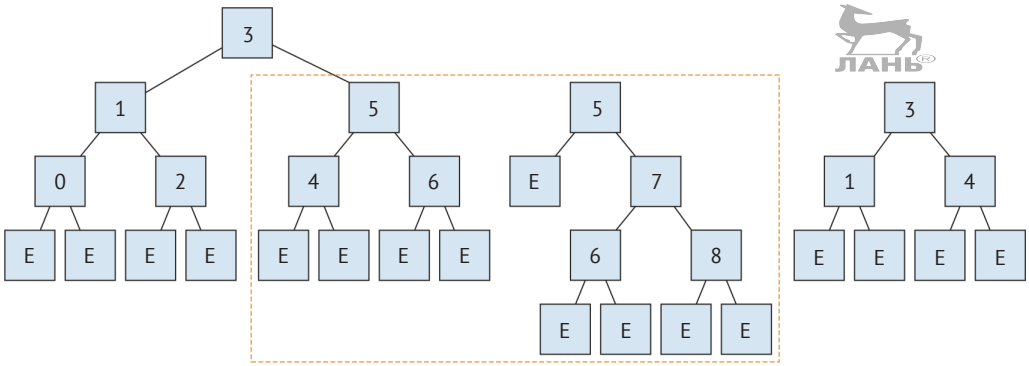


Рис. 10.9 Корень дерева в параметре больше корня дерева в this. Объединить правую ветвь из дерева в this с деревом в параметр, из которого удалена левая ветвь (операция объединения обозначена прямоугольником с пунктирными границами)

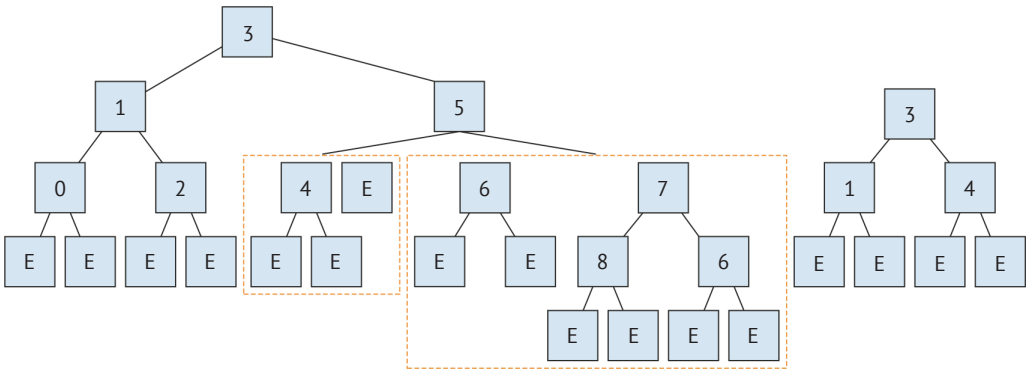


Рис. 10.10 Корни объединяемых деревьев равны, в дереве с результатом используется корень из this. Левая ветвь является результатом слияния двух левых ветвей, а правая ветвь – результатом слияния двух правых

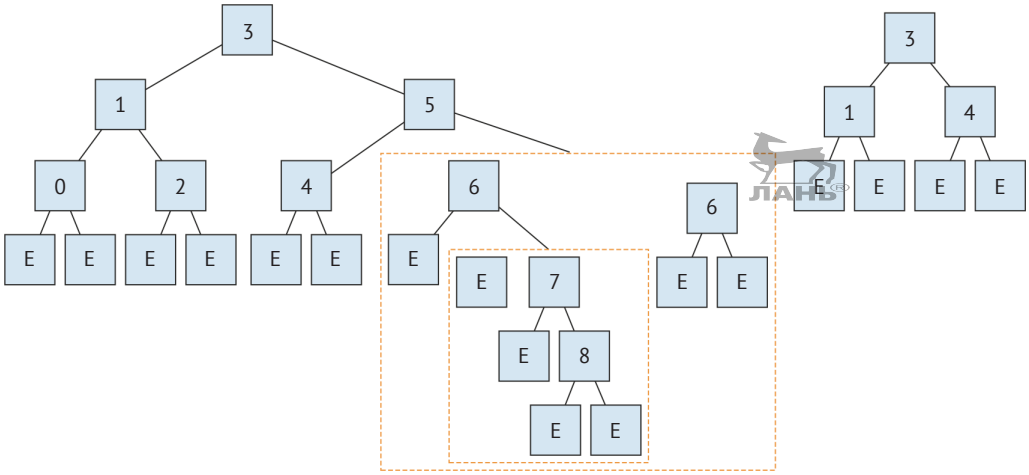


Рис. 10.11 Для левой ветви операция слияния с пустым деревом просто возвращает исходное дерево (корень 4 и две пустые ветви). Для правой ветви первое дерево имеет корень 6 и пустые ветви, а второе дерево имеет корень 7, поэтому удаляем левую ветвь из дерева с корнем 7 и используем результат для слияния с пустой правой ветвью в дереве с корнем 6. Удаленная левая ветвь объединяется с результатом предыдущего слияния. Дерево с корнем 6 справа – это ветвь из дерева с корнем 7, где она была заменена пустым деревом

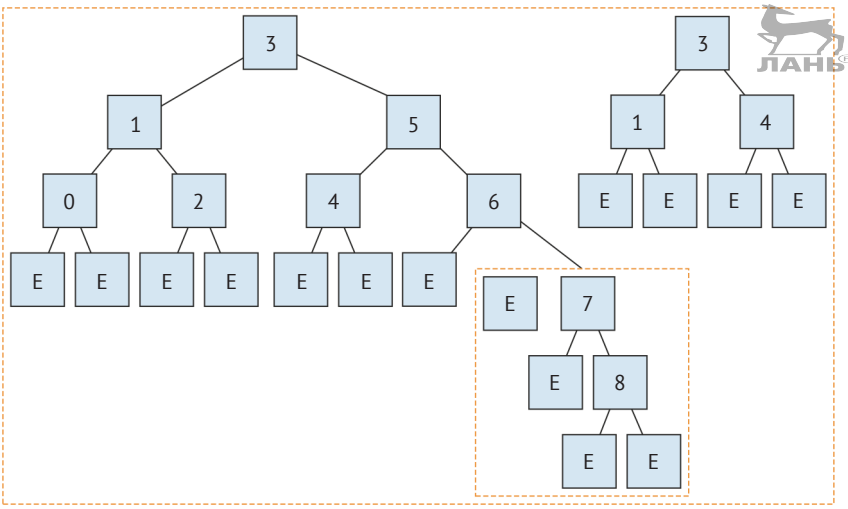


Рис. 10.12 Объединяемые деревья имеют равные корни (6), поэтому выполняется слияние ветвей (левой с левой и правой с правой). Так как в присоединяемом дереве обе ветви пустые, ничего делать не нужно

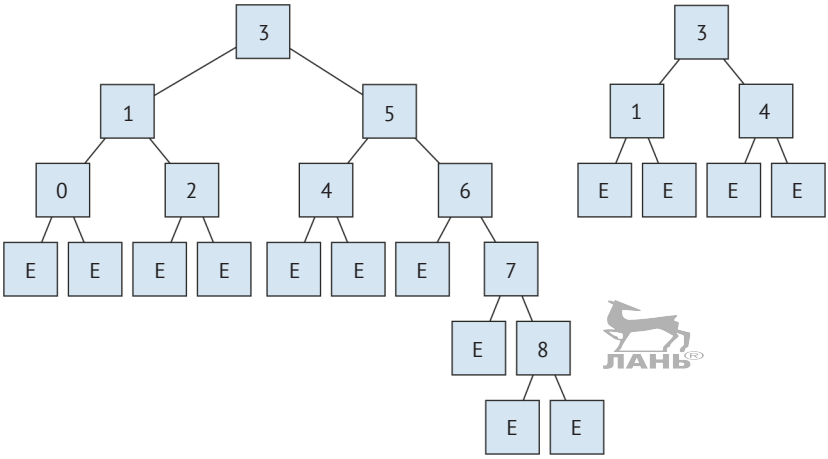


Рис. 10.13 Объединение с пустым деревом дает в результате добавляемое дерево. Теперь предстоит объединить два дерева с одинаковыми корнями

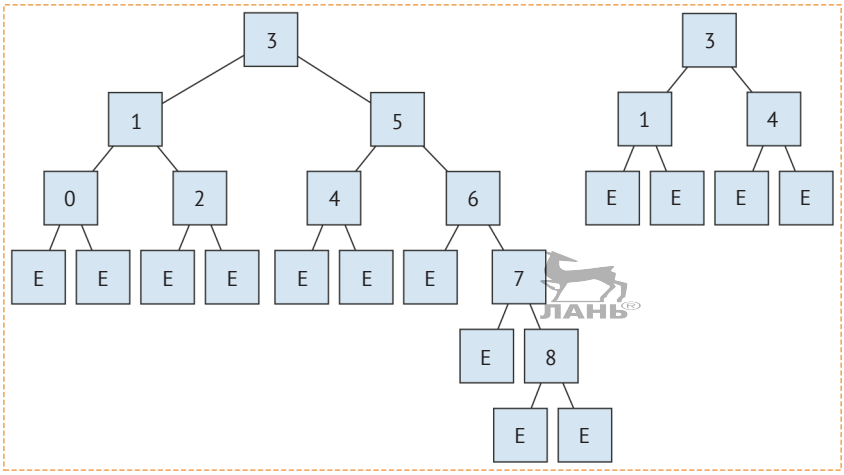


Рис. 10.14 Слияние деревьев с одинаковыми корнями осуществляется просто: правая ветвь объединяется с правой, левая – с левой, а результаты используются как новые ветви

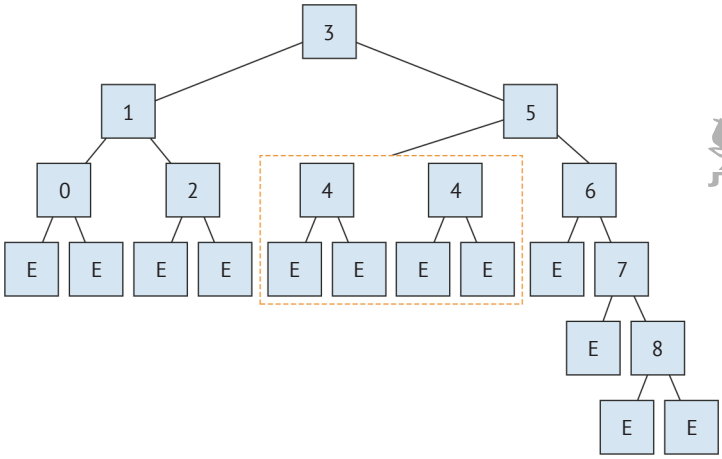


Рис. 10.15 Слияние левых ветвей осуществляется тривиально просто, потому что их корни равны, а обе ветви добавляемого дерева – пустые. Для правых ветвей: корень добавляемого дерева меньше (4), поэтому удаляем правую ветвь (E) и объединяем остаток с левой ветвью оригинального дерева

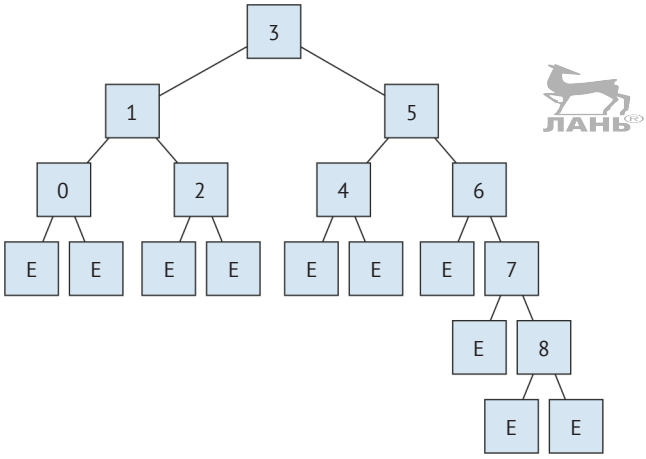


Рис. 10.16 Слияние двух идентичных деревьев (не требует пояснений)

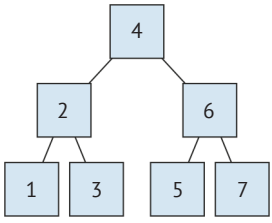


Рис. 10.17 Окончательный результат после объединения с последним пустым деревом

Как можно видеть на этих рисунках, слияние двух деревьев дает в результате дерево, размер которого (число элементов) меньше суммы раз-

меров двух исходных деревьев, потому что повторяющиеся элементы автоматически удаляются.

С другой стороны, высота результата получилась больше, чем можно было бы ожидать. Слияние двух деревьев с высотой 3 может дать в итоге дерево с высотой 5. Нетрудно подсчитать, что оптимальная высота не должна быть больше, чем log2(размер). Оптимальная высота – это наименьшая степень двойки, которая больше размера получившегося дерева. В этом примере исходные деревья имеют размер 7 и высоту 3. Размер объединенного дерева равен 9, а оптимальная высота равна 4, а не 5. В таком небольшом примере это не так важно. Но при объединении больших деревьев можно получить плохо сбалансированное дерево, что приведет к ухудшению производительности и даже к переполнению стека при использовании рекурсивных функций.

10.10 О свертке деревьев

Свертка дерева напоминает свертку списка; она заключается в преобразовании дерева в единственное значение. Например, подсчет суммы всех элементов в числовом дереве можно реализовать через свертку. Но свертка дерева выполняется сложнее, чем свертка списка.

Подсчитать сумму всех элементов в дереве с целыми числами тривиально просто, потому что операция сложения коммутативна и ассоциативна в обоих направлениях. Следующие выражения дают один и тот же результат:

```
1  (((1 + 3) + 2) + ((5 + 7) + 6)) + 4.
2  4 + ((2 + (1 + 3)) + (6 + (5 + 7))).
3  (((7 + 5) + 6) + ((3 + 1) + 2)) + 4.
4  4 + ((6 + (7 + 5)) + (2 + (3 + 1))).
5  (1 +(2 + 3)) + (4 + (5 + (6 + (7)))).
6  (7 + (6 + 5)) + (4 + (3 + (2 + 1))).
```

Исследовав эти выражения, можно догадаться, что они представляют результат свертки следующего дерева с использованием операции сложения:

Об этих выражениях можно сказать, что операции в них выполняются:

- в обратном направлении слева;
- в прямом направлении слева;
- в обратном направлении справа;
- в прямом направлении справа;

- симметрично слева;
- симметрично справа.

Слева и справа в данном случае означает, что суммирование начинается слева или справа соответственно. Мы можем проверить, вычислив результат каждого выражения. Например, первое выражение:

```
(((1 + 3) + 2) + ((5 + 7) + 6)) + 4 =
(( 4 + 2) + ((5 + 7) + 6)) + 4 =
( 6 + ((5 + 7) + 6)) + 4 =
( 6 + ( 12 + 6)) + 4 =
( 6 + 18 ) + 4 =
24 + 4 =
28
```

Есть и другие возможные варианты, но наибольший интерес представляют эти шесть. Для операции сложения все они эквивалентны, но для других операций это может быть далеко не так, например для операции объединения символов в строку или объединения элементов в список.

10.10.1 Свертка с двумя функциями

Проблема свертки дерева заключается в том, что рекурсивный подход на самом деле будет бирекурсивным. Можно свернуть каждую ветвь с помощью одной операции и затем объединить результаты с помощью другой.

Не находите, что эта ситуация напоминает пример параллельной свертки сегментов списка, который мы рассмотрели в главе 8? Все верно, нам нужна дополнительная операция. Если представить операцию свертки `Tree<A>` как функцию `(B) -> (A) -> B`, тогда нам нужна дополнительная функция `(B) -> (B) -> B`, объединяющая результаты свертки левой и правой ветвей.

Упражнение 10.8

Напишите функцию `foldLeft`, выполняющую свертку дерева, которая принимает две функции, описанные выше. Вот ее сигнатура в классе `Tree`:

```
abstract fun <B> foldLeft(identity: B,
                          f: (B) -> (A) -> B,
                          g: (B) -> (B) -> B): B
```

Решение

Реализация в подклассе `Empty` тривиально проста. Она должна вернуть `identity`. Реализация в подклассе `T` намного сложнее. Она должна выполнить свертку обеих ветвей и затем объединить результаты с корнем. Проблема в том, что в результате свертки каждой ветви получается значение типа `B`, а корень имеет значение типа `A`, но у нас нет функции `(A) -> B`. Вот как могло бы выглядеть решение:

- рекурсивно свернуть левую и правую ветви и получить два значения `B`;
- объединить два значения `B` с помощью функции `g`, затем объединить результат с корнем и вернуть получившееся значение.

Это решение можно реализовать так:

```
override
fun <B> foldLeft(identity: B, f: (B) -> (A) -> B, g: (B) -> (B) -> B): B =
    g(right.foldLeft(identity, f, g))(f(left.foldLeft(identity, f, g))
(this.value))
```

Просто? Не совсем. Проблема в том, что `g` – это функция `(B) -> (B) -> B`, поэтому легко можно поменять аргументы местами:

```
override
fun <B> foldLeft(identity: B, f: (B) -> (A) -> B, g: (B) -> (B) -> B): B =
    g(*left*.foldLeft(identity, f, g))(f(*right*.foldLeft(identity, f, g))
                                       (this.value))
```

В чем, собственно, проблема? Если использовать для свертки коммутативную операцию, например сложение, результат не изменится. Но если использовать некоммутативную операцию, возникнет проблема. Эти два решения будут давать разные результаты, например следующая функция:

```
fun main(args: Array<String>) {
    val result = Tree(4, 2, 6, 1, 3, 5, 7)
                    .foldLeft(List(),
                        { list: List<Int> -> { a: Int -> list.cons(a) } })
                    { x -> { y -> y.concat(x) } }
    println(result)
}
```

в первой реализации даст результат:

```
[7, 5, 3, 1, 2, 4, 6, NIL]
```

а во второй реализации – другой результат:

```
[7, 5, 6, 3, 4, 1, 2, NIL]
```

Какой из них правильный? На самом деле оба списка представляют одно и то же дерево, хотя порядок следования элементов в них отличается. Эти две ситуации представлены на рис. 10.18.

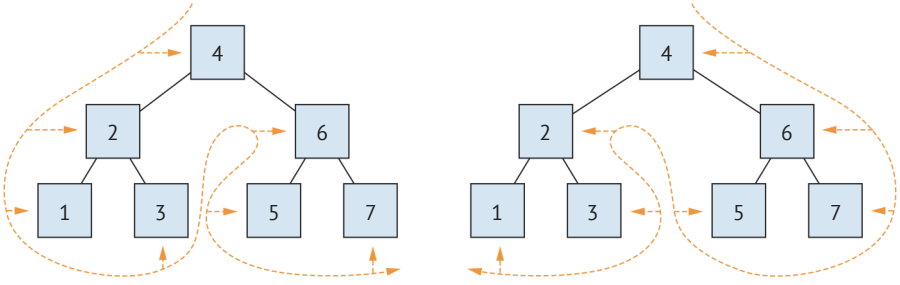


Рис. 10.18 Обход дерева слева направо и справа налево. Порядок следования элементов в обоих списках отличается, но они оба представляют одно и то же дерево

Это совсем другое, нежели разница между `foldLeft` и `foldRight` в классе `List`. Свертка справа – это фактически свертка перевернутого списка слева. Свертку справа можно реализовать так:

```
override
fun <B> foldRight(identity: B, f: (A) -> (B) -> B, g: (B) -> (B) -> B): B =
    g(f(this.value)(left.foldRight(identity, f, g)))
(right.foldRight(identity, f, g))
```

Так как обход может осуществляться несколькими способами, можно написать несколько разных реализаций, дающих разные результаты при использовании некоммутативной операции.

10.10.2 Свертка с одной функцией

Свертку также можно выполнить с одной функцией, принимающей дополнительный параметр – функцию `(B) -> (A) -> (B) -> B`. И снова возможно несколько реализаций по числу вариантов обхода.

Упражнение 10.9

Напишите три функции свертки, выполняющие обход дерева в симметричном, прямом и обратном порядке: `foldInOrder`, `foldPreOrder` и `foldPostOrder`. Применительно к дереву на рис. 10.18 они должны обрабатывать элементы в следующем порядке:

- `foldInOrder`: 1 2 3 4 5 6 7;
- `foldPreOrder`: 4 2 1 3 6 5 7;
- `foldPostOrder`: 1 3 2 5 7 6 4.

Вот сигнатуры этих функций:

```
abstract fun <B> foldInOrder(identity: B, f: (B) -> (A) -> (B) -> B): B
abstract fun <B> foldPreOrder(identity: B, f: (A) -> (B) -> (B) -> B): B
abstract fun <B> foldPostOrder(identity: B, f: (B) -> (B) -> (A) -> B): B
```

Решение

Все реализации в `Empty` возвращают `identity`. Далее приводятся реализации в классе `T`:

```
override fun <B> foldInOrder(identity: B, f: (B) -> (A) -> (B) -> B): B =
    f(left.foldInOrder(identity, f))(value)(right.foldInOrder(identity, f))

override fun <B> foldPreOrder(identity: B, f: (A) -> (B) -> (B) -> B): B =
    f(value)(left.foldPreOrder(identity, f))(right.foldPreOrder(identity, f))

override fun <B> foldPostOrder(identity: B, f: (B) -> (B) -> (A) -> B): B =
    f(left.foldPostOrder(identity, f))(right.foldPostOrder(identity, f))(value)
```

10.10.3 Выбор реализации свертки

Теперь у нас есть несколько реализаций свертки. Возникает вопрос: какую выбрать? Чтобы ответить на этот вопрос, давайте перечислим свойства, которыми должна обладать функция свертки.

Между порядком, в каком выполняется свертка структуры данных, и порядком ее конструирования есть прямая связь. Конструирование дерева начинается с создания пустого элемента, к которому затем последовательно добавляются новые элементы. То есть конструирование выполняется в порядке, обратном свертке. В идеале хорошо было бы иметь возможность свернуть структуру с использованием определенных параметров, позволяющих превратить функцию свертки в функцию тождественного отображения. Для списка такую функцию можно реализовать так:

```
list.foldRight(List()) { i -> { l -> l.cons(i) } }
```

То же можно сделать с функцией `foldLeft`, но реализация будет выглядеть немного сложнее:

```
list.foldLeft(List()) { l -> { i -> l.reverse().cons(i).reverse() } }
```

В этом нет ничего удивительного, если заглянуть в реализацию `foldRight`, – `foldRight` можно реализовать через `foldLeft` и `reverse`.

Возможно ли то же самое для свертки дерева? Чтобы добиться этого, нужен новый способ конструирования дерева, который сначала конструирует левую ветвь, потом добавляет корень и затем – правую ветвь. В этом случае можно использовать любую из трех функций свертки, принимающую единственную функцию.

Упражнение 10.10 (сложное)

Напишите функцию, которая принимает два дерева и корень и создает новое дерево. Вот ее сигнатура в объекте-компаньоне:

```
operator fun <A: Comparable<A>> invoke(left: Tree<A>,
                                       a: A, right: Tree<A>): Tree<A>
```

Эта функция должна позволять реконструировать дерево, идентичное оригинальному, с использованием любой из трех функций свертки: `foldPreOrder`, `foldInOrder` и `foldPostOrder`.

Подсказка

Вам придется обрабатывать два случая по отдельности. Если деревья, предназначенные для слияния, упорядочены, это означает, что максимальное значение в первом из них меньше корня во втором, а минимальное значение во втором больше корня в первом, и тогда объединенное дерево можно создать с помощью конструктора `T`. Иначе следует использовать другой способ конструирования результата.

Вам также понадобится добавить в класс `Result` дополнительную функцию (`mapEmpty`), которая возвращает `Success`, если экземпляр `Result` является экземпляром `Empty`, и `Failure` - в противном случае. Вы найдете эту функцию в классе `com.fpinkotlin.common.Result`.

Решение

Реализовать эту функцию можно несколькими способами. Один из них – определить функцию, которая проверит упорядоченность исход-

ных деревьев. Но сначала определим функцию, возвращающую результат сравнения:

```
fun <A: Comparable<A>> lt(first: A, second: A): Boolean = first < second

fun <A: Comparable<A>> lt(first: A, second: A, third: A): Boolean =
    lt(first, second) && lt(second, third)
```

Теперь напишем функцию ordered, сравнивающую деревья:

```
fun <A: Comparable<A>> ordered(left: Tree<A>,
                               a: A,
                               right: Tree<A>): Boolean =
    (left.max().flatMap { lMax ->
        right.min().map { rMin ->
            lt(lMax, a, rMin)
        }
    }.getOrElse(left.isEmpty() && right.isEmpty()) ||
        left.min()
            .mapEmpty()
            .flatMap {
                right.min().map { rMin -> lt(a, rMin) }
            }.getOrElse(false) ||
        right.min()
            .mapEmpty()
            .flatMap {
                left.max().map { lMax -> lt(lMax, a) }
            }.getOrElse(false))
```

Первая проверка (перед первым оператором ||) возвращает true, если оба дерева не пустые, а левое значение max и правое min упорядочены. Вторая и третья проверки обрабатывают случаи, когда одно из деревьев (слева или справа, но не оба) пустое. Функция Result.mapEmpty возвращает Success<Any> для экземпляра Empty и Failure в противном случае. С помощью функции ordered легко написать функцию invoke:

```
operator
fun <A: Comparable<A>> invoke(left: Tree<A>,
                              a: A, right: Tree<A>): Tree<A> =
    when {
        ordered(left, a, right) -> T(left, a, right)
        ordered(right, a, left) -> T(right, a, left)
        else                    -> Tree(a).merge(left).merge(right)
    }
```

Если деревья не упорядочены, проверяем – не указаны ли они в обратном порядке, и только потом переходим к обычному алгоритму вставки/слияния. Теперь мы можем свернуть дерево и получить то же дерево, что и оригинальное (при условии использования правильной функции). В примерах кода, прилагаемых к книге, вы найдете следующие примеры:

```
tree.foldInOrder(Tree<Int>(), { t1 -> { i -> { t2 -> Tree(t1, i, t2) } } })
tree.foldPostOrder(Tree<Int>(), { t1 -> { t2 -> { i -> Tree(t1, i, t2) } } })
tree.foldPreOrder(Tree<Int>(), { i -> { t1 -> { t2 -> Tree(t1, i, t2) } } })
```

Также можно определить функцию свертки, принимающую одну функцию с двумя параметрами, как в классе `List`. Хитрость заключается в том, чтобы сначала преобразовать дерево в список, как показано в следующем примере `foldLeft`:

```
// абстрактная версия в Tree и эта конкретная реализация в T:
override fun <B> foldLeft(identity: B, f: (B) -> (A) -> B): B =
    toListPreOrderLeft().foldLeft(identity, f)

// B Tree:
override fun toListPreOrderLeft(): List<A> =
    left.toListPreOrderLeft().concat(right.toListPreOrderLeft()).cons(value)
```

Это не самая быстрая реализация, но она все еще может пригодиться.

10.11 О преобразовании элементов деревьев

Элементы деревьев можно преобразовывать, подобно элементам списков, но реализация функции `map` для деревьев немного сложнее аналогичной функции в списке. Применение функции к каждому элементу дерева может показаться простой задачей, но это не так. Проблема в том, что в результате преобразования элементов в дереве их порядок может измениться. Увеличение всех элементов в дереве целых чисел на одно и то же значение не вызовет никаких проблем, но при использовании функции $f(x) = x * x$ могут возникнуть сложности, если дерево содержит отрицательные значения, потому что после применения функции дерево перестанет быть бинарным деревом поиска.

Упражнение 10.11

Напишите функцию `map` для деревьев. Попробуйте обеспечить сохранность структуры дерева, если возможно. Например, преобразование элементов дерева целых чисел с использованием функции возведения в квадрат может изменить структуру дерева, но, если используется функция увеличения на константу, структура дерева не изменится.

Подсказка

Использование одной из функций свертки упростит задачу.

Решение

Есть несколько вариантов реализации с использованием разных функций свертки. Вот вариант, который можно реализовать в классе `Tree`:

```
fun <B: Comparable<B>> map(f: (A) -> B): Tree<B> =
    foldInOrder(Empty) { t1: Tree<B> ->
        { i: A ->
            { t2: Tree<B> ->
                Tree(t1, f(i), t2)
            }
        }
    }
```

10.12 О балансировке деревьев

Как я уже говорил выше, деревья показывают хорошую производительность, только если они сбалансированы, т. е. когда все пути от корня до листового элемента имеют почти одинаковую длину. В идеально сбалансированном дереве разница в длинах не превышает 1, что может произойти, если самый нижний уровень будет заполнен не полностью. (Только в идеально сбалансированных деревьях с размером $2^n + 1$ все пути от корня до любого листового элемента имеют одинаковую длину.)

Несбалансированность дерева может привести к существенному снижению производительности, потому что для обхода такого дерева операциям может потребоваться время, пропорциональное размеру дерева, а не log2(размер). Более того, несбалансированные деревья могут вызвать переполнение стека при выполнении рекурсивных операций. Вот два способа избежать этой проблемы:

- сбалансировать несбалансированное дерево;
- использовать самобалансирующиеся деревья.

Имея возможность балансировать деревья, легко создать самобалансирующиеся деревья, которые автоматически запускают процесс балансировки после каждой операции, способной изменить его структуру.

10.12.1 Вращение деревьев

Прежде чем заняться проблемой балансировки деревьев, необходимо понять, как можно постепенно менять структуру дерева. Для этого используем прием, который называется *поворотом* дерева и показан на рис. 10.19 и 10.20.

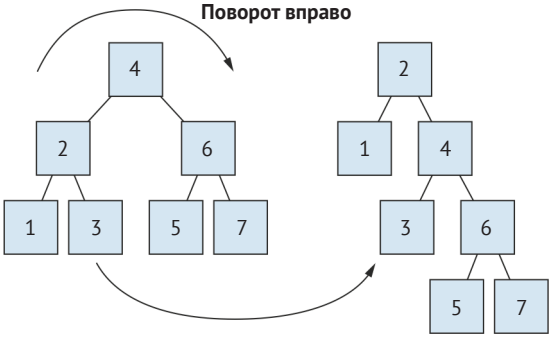


Рис. 10.19 Поворот дерева вправо. При повороте связь между 2 и 3 замещается связью между 2 и 4, а элемент 3 превращается в левый элемент элемента 4

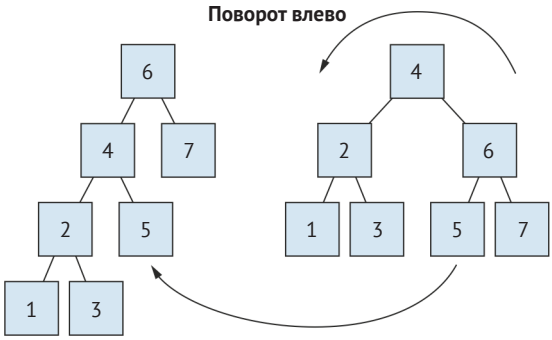


Рис. 10.20 Поворот дерева влево. Элемент 4, бывший родитель элемента 6, становится его левым элементом, а элемент 5 становится правым элементом элемента 4

Упражнение 10.12

Напишите функции `rotateRight` и `rotateLeft`, чтобы получить возможность вращать дерево в обоих направлениях. Не забывайте о сохранении порядка следования элементов в ветвях: элементы слева всегда должны быть меньше корня, а элементы справа – всегда больше. Объявите абстрактные функции в родительском классе. Сделайте их защищенными (`protected`), так как они будут использоваться только внутри класса `Tree`. Вот сигнатуры функций в родительском классе:

```
protected abstract fun rotateRight(): Tree<A>
protected abstract fun rotateLeft(): Tree<A>
```

Решение

Реализация в `Empty` должна просто вернуть `this`. В классе `T` поворот дерева вправо выполняется в три приема:

1. Проверить левую ветвь – не пустая ли.
2. Если левая ветвь пустая, вернуть `this`, потому что при повороте вправо корень левой ветви становится корнем дерева, но мы не можем вернуть пустое дерево.
3. Если левая ветвь непустая, ее корень становится корнем дерева, т. е. создается новый экземпляр `T` со значением `left.value` в корне. Левая ветвь левого элемента становится левой ветвью нового дерева. Чтобы получить правую ветвь, конструируется новое дерево, корнем которого является корень исходного дерева, левой ветвью – правая ветвь левого элемента в оригинальном дереве, а правой ветвью – правая ветвь оригинального дерева.

Поворот влево осуществляется симметрично:

```
override fun rotateRight(): Tree<A> = when (left) {
    Empty -> this
    is T -> T(left.left, left.value,
              T(left.right, value, right))
```

```
}

override fun rotateLeft(): Tree<A> = when (right) {
    Empty -> this
    is T -> T(T(left, value, right.left),
              right.value, right.right)
}
```

Описание выглядит сложным, но в действительности все просто. Сравните код с рис. 10.19 и 10.20, чтобы лучше понять суть происходящего. Попытавшись повернуть дерево еще несколько раз, вы достигнете момента, когда одна из ветвей окажется пустой и продолжать поворачивать дерево станет невозможно.

Упражнение 10.13

Для балансировки дерева нужна функция, которая преобразует его в упорядоченный список. Напишите функцию, превращающую дерево в упорядоченный список, выполняя симметричный обход справа налево (т. е. элементы в списке должны быть упорядочены по убыванию).

> **ПРИМЕЧАНИЕ** Желающие поупражняться побольше могут написать функцию для упорядочивания слева направо, а также функции, которые составляют список, выполняя обход дерева в прямом и обратном порядке.

Вот сигнатура функции `toListInOrderRight`:

```
fun toListInOrderRight(): List<A>
```

Решение

Это упражнение решается просто и в большей степени относится к спискам, чем к деревьям. Реализация в `Empty` должна вернуть пустой список. Может показаться, что реализация в классе `T` должна выглядеть как-то так:

```
override fun toListInOrderRight(): List<A> =
    right.toListInOrderRight()
         .concat(List(value))
         .concat(left.toListInOrderRight())
```

К сожалению, эта функция вызовет переполнение стека, если дерево окажется достаточно большим и плохо сбалансированным. Так как эта функция нужна нам для балансировки деревьев, будет досадно, если она не сможет работать с несбалансированными деревьями.

Чтобы избежать переполнения стека, нам нужна сорекурсивная версия. Эта версия, объявленная в классе `Tree`, использует сорекурсивную вспомогательную функцию для балансировки деревьев, которую можно поместить в объект-компаньон. Вот вспомогательная функция:

```
private tailrec
fun <A: Comparable<A>> unBalanceRight(acc: List<A>,
                                      tree: Tree<A>): List<A> =
```

```
    when (tree) {
        Empty -> acc
        is T -> when (tree.left) {
            Empty -> unBalanceRight(acc.cons(tree.value),
                                    tree.right) ①
            is T -> unBalanceRight(acc,
                                   tree.rotateRight()) ②
        }
    }
```

① Добавляет дерево в список-аккумулятор
② Поворачивает дерево, пока левая ветвь не опустеет

А вот основная функция в классе `Tree`:

```
fun toListInOrderRight(): List<A> = unBalanceRight(List(), this)
```

Функция `unBalancedRight` поворачивает дерево вправо, пока его левая ветвь не опустеет. Затем добавляет значение дерева (корня) в список-аккумулятор и рекурсивно вызывает саму себя, чтобы проделать то же самое со всеми поддеревьями справа. В конце концов она получит пустое дерево в параметре `tree` и вернет накопленный список.

10.12.2 Алгоритм Дея/Стоута/Уоррен

Алгоритм Дея/Стоута/Уоррен (Day/Stout/Warren)[1] – это простой и эффективный метод балансировки бинарных деревьев поиска. Суть его заключается в следующем:

1. Превратить дерево в полностью несбалансированное.
2. Затем применять к нему операцию поворота, пока оно не станет идеально сбалансированным.

Чтобы сделать дерево полностью несбалансированным, достаточно превратить его в упорядоченный список и на его основе создать новое дерево. Так как дерево должно создаваться в порядке возрастания, мы должны сначала создать список, упорядоченный по убыванию, а затем поворачивать получившийся результат влево.

Вот алгоритм получения идеально сбалансированного дерева:

1. Поворачивать дерево влево, пока в результате все его ветви не получатся максимально одинаковой длины. То есть, если размер дерева является нечетным числом, ветви должны иметь одинаковые размеры, а если размер дерева – четное число, размеры ветвей могут отличаться не более чем на 1. В результате получится дерево с полностью несбалансированными ветвями равного или почти равного размера.

[1] Этот алгоритм был разработан Квентином Ф. Стоутом (Quentin F. Stout) и Бетти Уоррен (Bette Warren) на основе работы Колина Дея (Colin Day). – *Прим. перев.*

2. Тот же процесс применить рекурсивно к правой ветви. Применить симметричный процесс (с поворотом вправо) к левой ветви.
3. Прекратить, когда высота результата станет равна log2(*размер_списка*). Для этого понадобится следующая вспомогательная функция:

```
fun log2nlz(n: Int): Int = when (n) {
    0 -> 0
    else -> 31 - Integer.numberOfLeadingZeros(n)
}
```

Вот описание метода `numberOfLeadingZeros` из Javadoc:

> Возвращает количество нулевых битов, предшествующих самому старшему (левому) единичному биту в дополнительном коде для указанного значения типа `int`. Если указанное значение не имеет единичных битов в представлении с дополнением до двух, т. е. если оно равно нулю, возвращает 32.
>
> Этот метод тесно связан с вычислением логарифма по основанию 2. Для всех положительных `x` типа `int`:
>
> - `floor(log2(x)) = 31 - numberOfLeadingZeros(x);`
> - `ceil(log2(x)) = 32 - numberOfLeadingZeros(x - 1).`

Упражнение 10.14

Напишите функцию `balance`, выполняющую идеальную балансировку любого дерева. Эта функция должна быть объявлена в объекте-компаньоне и принимать дерево для балансировки в параметре.

Подсказка

Эта реализация основана на нескольких вспомогательных функциях: сначала функция создает полностью несбалансированное дерево вызовом `toListInOrderRight`. Получившийся список свертывается влево в (полностью несбалансированное) дерево, которое затем легко будет сбалансировать.

Также нам понадобится функция, которая определит, является ли дерево идеально сбалансированным, и функция, которая выполнит рекурсивный поворот дерева. Вот как можно было бы реализовать функцию поворота дерева в объекте-компаньоне:

```
fun <A> unfold(a: A, f: (A) -> Result<A>): A {
    tailrec fun <A> unfold(a: Pair<Result<A>, Result<A>>,
                           f: (A) -> Result<A>): Pair<Result<A>, Result<A>> {
        val x = a.second.flatMap { f(it) }
        return x.map { unfold(Pair(a.second, x), f) }.getOrElse(a)
    }
    return Result(a).let { unfold(Pair(it, it), f).second.getOrElse(a) }
}
```

К сожалению, она не будет работать, потому что рекурсия во вспомогательной функции не является хвостовой. Чтобы получить хвостовую рекурсию, нужен какой-то способ, который позволит определить, явля-

ется ли экземпляр Result экземпляром Success. Сделать это можно двумя путями:

- использовать сопоставление с образцом, включив классы Result и Tree в один модуль (потому что подклассы в Result объявлены внутренними);
- добавить функцию isSuccess в класс Result.

Выбирайте решение, какое вам больше нравится. Лично я скопировал класс Result в модуль с примерами для этой главы (вместе с остальными необходимыми классами). Вот действующая реализация:

```
fun <A> unfold(a: A, f: (A) -> Result<A>): A {
    tailrec fun <A> unfold(a: Pair<Result<A>, Result<A>>,
                           f: (A) -> Result<A>): Pair<Result<A>, Result<A>> {
        val x = a.second.flatMap { f(it) }
        return when (x) {
            is Result.Success -> unfold(Pair(a.second, x), f)
            else -> a
        }
    }
    return Result(a).let { unfold(Pair(it, it), f).second.getOrElse(a) }
}
```

Эта функция получила имя unfold по аналогии с List.unfold и Stream.unfold. Она выполняет ту же работу, но тип ее результата совпадает с типом входного параметра. Она отбрасывает большинство результатов и оставляет только два последних. Благодаря этому она работает быстрее и использует меньше памяти. Также нужно определить внутренние функции для доступа к значению дерева и его ветвям.

Решение

Сначала определим вспомогательную функцию, определяющую несбалансированность дерева. Сбалансированным считается дерево, если разность высоты его ветвей равна 0, когда общий размер ветвей является четным числом, и 1, когда общий размер ветвей является нечетным числом:

```
fun <A : Comparable<A>> isUnBalanced(tree: Tree<A>): Boolean =
    when (tree) {
        Empty -> false
        is T -> Math.abs(tree.left.height - tree.right.height) >
                                                (tree.size - 1) % 2
    }
```

Затем определим функции для доступа к значению дерева и его ветвям. Для этого можно применить тот же прием, что использовался для определения высоты и размера. Определим абстрактные свойства в классе Tree:

```
internal abstract val value: A
internal abstract val left: Tree<A>
internal abstract val right: Tree<A>
```

Реализации в `Empty` должны возбуждать исключение, потому что это свойства, а не функции. Мы не можем написать:

```
override val value: Nothing =
    throw IllegalStateException("No value in Empty")
override val left: Tree<Nothing> =
    throw IllegalStateException("No left in Empty")
override val right: Tree<Nothing> =
    throw IllegalStateException("No right in Empty")
```

потому что исключения будут генерироваться немедленно, на этапе инициализации свойств, т. е. в момент создания объекта. В главе 9 мы узнали, как осуществлять ленивую инициализацию, воспользуемся этим знанием:

```
override val value: Nothing by lazy {
    throw IllegalStateException("No value in Empty")
}

override val left: Tree<Nothing> by lazy {
    throw IllegalStateException("No left in Empty")
}

override val right: Tree<Nothing> by lazy {
    throw IllegalStateException("No right in Empty")
}
```

ПРИМЕЧАНИЕ Ответственность за то, чтобы эти функции никогда не вызывались, целиком и полностью лежит на вас.

В классе `T` нужно немного изменить конструктор:

```
internal
class T<out A: Comparable<@UnsafeVariance A>>(override val left: Tree<A>,
                                              override val value: A,
                                              override val right: Tree<A>): Tree<A>
() {
```

Переопределяя свойства в конструкторе, мы автоматически назначаем им ту же область видимости (`internal`), которую имеют переопределяемые свойства. Теперь можно написать основную функцию балансировки:

```
fun <A: Comparable<A>> balance(tree: Tree<A>): Tree<A> =
    balanceHelper(tree.toListInOrderRight().foldLeft(Empty) {
        t: Tree<A> -> { a: A -> T(Empty, a, t) }
    })

fun <A: Comparable<A>> balanceHelper(tree: Tree<A>): Tree<A> = when {
    !tree.isEmpty() && tree.height > log2nlz(tree.size) -> when {
        Math.abs(tree.left.height - tree.right.height) > 1 ->
            balanceHelper(balanceFirstLevel(tree))
        else -> T(balanceHelper(tree.left), tree.value,
                  balanceHelper(tree.right))
    }
```

```
    else -> tree
}

private fun <A: Comparable<A>> balanceFirstLevel(tree: Tree<A>):Tree<A> =
    unfold(tree) { t: Tree<A> -> when {
        isUnBalanced(t) -> when {
            tree.right.height > tree.left.height -> Result(t.rotateLeft())
            else -> Result(t.rotateRight())
        }
        else -> Result()
    }
}
```

10.12.3 Самобалансирующиеся деревья

Функция `balance` прекрасно справляется с подавляющим большинством деревьев, но ее нельзя применить к большим несбалансированным деревьям, потому что это может вызвать переполнение стека. Эту проблему можно решить, применяя `balance` только к небольшим, полностью несбалансированным деревьям или к частично несбалансированным деревьям любого размера. Это означает, что дерево должно быть сбалансировано, до того как оно станет слишком большим. Возникает вопрос: можно ли организовать автоматическую балансировку после каждого изменения дерева?

Упражнение 10.15

Доработайте класс `Tree`, созданный в этой главе, чтобы он автоматически выполнял балансировку после каждой операции вставки, слияния или удаления.

Решение

Само собой напрашивается простое решение: вызывать `balance` после каждой операции, изменяющей дерево, например:

```
operator fun plus(a: @UnsafeVariance A): Tree<A> =
    balance(plusUnBalanced(a))

private fun plusUnBalanced(a: @UnsafeVariance A): Tree<A> = plus(this, a)
```

Это решение хорошо подходит для маленьких деревьев (которые, по большому счету, не особо нуждаются в балансировке), но не подходит для больших деревьев из-за слишком больших затрат времени. Смягчить проблему можно, выполняя только частичную балансировку. Например, функцию балансировки можно запустить, когда высота дерева в 100 раз превысит его идеальную высоту (высоту идеально сбалансированного дерева):

```
operator fun plus(a: @UnsafeVariance A): Tree<A> {

    fun plusUnBalanced(a: @UnsafeVariance A, t: Tree<A>): Tree<A> =
        when (t) {
            Empty -> T(Empty, a, Empty)
            is T -> when {
```

```
                a < t.value -> T(plusUnBalanced(a, t.left), t.value, t.right)
                a > t.value -> T(t.left, t.value, plusUnBalanced(a, t.right))
                else        -> T(t.left, a, t.right)
            }
        }

    return plusUnBalanced(a, this).let {
        when {
            it.height > log2nlz(it.size) * 100 -> balance(it)
            else -> it
        }
    }
}
```

Производительность этого решения может показаться далекой от идеала, но это всего лишь компромисс. Создание дерева из упорядоченного списка со 100 000 элементов может занять, например, 2,5 секунды и дать в результате идеально сбалансированное дерево с высотой 16. Замена значения 100 на 20 в функции `plusUnBalanced` удвоит это время без явной выгоды, а замена на 1000 даст ускорение в пять раз.

Итоги

- Деревья – это рекурсивные структуры данных, в которых каждый элемент связан с одним или несколькими поддеревьями. В некоторых деревьях каждый узел может быть связан с произвольным количеством поддеревьев. Однако на практике чаще используются деревья, в которых узлы связаны с фиксированным количеством поддеревьев.
- В бинарных деревьях каждый узел связан с двумя поддеревьями. Эти связи называют *ветвями*, а сами ветви называются левой и правой. Бинарные деревья поиска позволяют быстро находить искомые элементы.
- Деревья могут быть сбалансированы в той или иной степени. Полностью сбалансированные деревья обеспечивают максимальную производительность, тогда как полностью несбалансированные имеют ту же производительность, что и списки.
- Размер дерева – это количество элементов, содержащихся в нем; высота дерева – самая длинная ветвь в дереве.
- Структура дерева зависит от порядка вставки элементов.
- Обход деревьев может выполняться в разных порядках (прямом, симметричном или обратном) и направлениях (слева направо или справа налево).
- Деревья можно объединять, не выполняя полный обход их элементов.
- Деревья можно преобразовывать, вращать и применять к ним операцию свертки несколькими разными способами.
- Деревья можно балансировать, чтобы повысить производительность и избежать переполнения стека в рекурсивных операциях.

11 Решение задач с использованием усовершенствованных деревьев

Эта глава охватывает следующие темы:

- как избежать переполнения стека с использованием самобалансирующихся деревьев;
- реализация красно-черного дерева;
- реализация ассоциативных массивов;
- реализация приоритетных очередей.

В предыдущей главе вы познакомились с бинарными деревьями и основными операциями с ними. Но, как вы видели, чтобы получить от деревьев максимальную выгоду, они должны использоваться в особых условиях, например для обработки случайно упорядоченных данных или ограниченных наборов данных, чтобы избежать переполнения стека. Сделать деревья безопасными с точки зрения переполнения стека намного сложнее, чем списки, потому что каждый шаг вычислений включает два рекурсивных вызова. Это делает невозможным создание сорекурсивных версий операций. В этой главе вы познакомитесь с двумя конкретными видами деревьев:

- красно-черные деревья – это самобалансирующиеся деревья общего назначения, обладающие высокой производительностью; они подходят для представления наборов данных любого размера;
- левосторонняя куча – это специальное дерево, прекрасно подходящее для реализации приоритетных очередей.

Вы также узнаете, как на основе дерева реализовать ассоциативный массив, хранящий кортежи ключ/значение, и приоритетную очередь для хранения элементов, не поддерживающих операции сравнения.

11.1 Улучшение производительности и безопасности деревьев добавлением самобалансировки

Алгоритм балансировки Дея/Стоута/Уоррен (Day/Stout/Warren), о котором рассказывалось в предыдущей главе, плохо подходит для балансировки неизменяемых деревьев, потому что проектировался для случаев, когда есть возможность производить изменения на месте. Чтобы программы получались максимально безопасными, таких изменений на месте следует избегать, если это возможно.

При любой попытке изменить неизменяемую структуру данных создается новая структура. Следовательно, нам нужно определить процесс балансировки, который не нуждается в преобразовании дерева в список, перед преобразованием полностью несбалансированного дерева в идеально сбалансированное. Вот два способа оптимизировать этот процесс:

- выполнять поворот исходного дерева (опустив этап преобразования дерева в список);
- допустить возможность некоторой степени несбалансированности.

Вы можете попробовать придумать свое решение, но другие уже давно это сделали. Одной из наиболее эффективных конструкций самобалансирующихся деревьев являются красно-черные деревья. Эта структура была придумана в 1978 году Гибасом (Guibas) и Седжвиком (Sedgewick)[1]. В 1999 году Крис Окасаки (Chris Okasaki) опубликовал функциональную версию алгоритма красно-черного дерева[2]. Описание было проиллюстрировано реализацией на Standard ML, а позднее появилась реализация на Haskell. Именно этот алгоритм мы реализуем на Kotlin.

Если вам интересны неизменяемые структуры данных, я настоятельно рекомендую купить и прочитать книгу Окасаки[3]. Вы также можете прочитать его диссертацию 1996 года с таким же названием. Она менее полная, чем книга, зато бесплатно доступна для загрузки (http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf).

---

[1] *Лео Дж. Гибас* (Leo J. Guibas) и *Роберт Седжвик* (Robert Sedgewick). A dichromatic framework for balanced trees // Foundations of Computer Science (1978), http://www.computer.org/csdl/proceedings/focs/1978/5428/00/542800008-abs.html.

[2] *Крис Окасаки* (Chris Okasaki). Purely Functional Data Structures. Cambridge University Press, 1999.

[3] *Окасаки Крис.* Чисто функциональные структуры данных. М.: ДМК Пресс, 2016. ISBN: 978-5-97060-233-1. – *Прим. перев.*

11.1.1 Структура красно-черных деревьев

Красно-черное дерево – это бинарное дерево поиска с некоторыми дополнениями в структуре и усовершенствованным алгоритмом вставки, который одновременно балансирует результат. К сожалению, Окасаки не описал алгоритм удаления, который оказался гораздо сложнее. Но в 2014 году Герман (Germane) и Майт (Might) описали этот недостающий метод[1].

Каждое красно-черное дерево (включая поддеревья) имеет дополнительное свойство, представляющее его цвет. Обратите внимание, что это может быть любой цвет или даже любое свойство, представляющее выбор из двух характеристик. Кроме того, структура красно-черного дерева ничем не отличается от структуры бинарного дерева, как показано в листинге 11.1.

Листинг 11.1 Базовая структура красно-черного дерева

```
package com.fpinkotlin.advancedtrees.listing01

import com.fpinkotlin.advancedtrees.listing01.Tree.Color.B ①
import com.fpinkotlin.advancedtrees.listing01.Tree.Color.R
import kotlin.math.max

sealed class Tree<out A: Comparable<@UnsafeVariance A>> {

    abstract val size: Int

    abstract val height: Int ②

    internal abstract val color: Color

    internal abstract val isTB: Boolean ③

    internal abstract val isTR: Boolean

    internal abstract val right: Tree<A>

    internal abstract val left: Tree<A>

    internal abstract val value: A

    internal abstract
    class Empty<out A: Comparable<@UnsafeVariance A>>: Tree<A>() { ④

        override val isTB: Boolean = false

        override val isTR: Boolean = false

        override val right: Tree<Nothing> by lazy { ⑤
            throw IllegalStateException("right called on Empty tree")
        }
```

[1] *Кимбалл Герман* (Kimball Germane) и *Мэттью Майт* (Matthew Might). Functional Pearl: Deletion, The curse of the red-black tree. JFP 24, 4 (2014): 423–433; http://matt.might.net/papers/germane2014deletion.pdf.

```
        override val left: Tree<Nothing> by lazy {
            throw IllegalStateException("left called on Empty tree")
        }

        override val value: Nothing by lazy {
            throw IllegalStateException("value called on Empty tree")
        }

        override val color: Color = B ⑥

        override val size: Int = 0

        override val height: Int = -1

        override fun toString(): String = "E"
    }

    internal object E: Empty<Nothing>() ⑦

    internal
    class T<out A: Comparable<@UnsafeVariance A>>(
                            override val color: Color, ⑧
                            override val left: Tree<A>,
                            override val value: A,
                            override val right: Tree<A>) : Tree<A>() {
        override val isTB: Boolean = color == B

        override val isTR: Boolean = color == R

        override val size: Int = left.size + 1 + right.size

        override val height: Int = max(left.height, right.height) + 1

        override fun toString(): String = "(T $color $left $value $right)"
    }

    companion object {

        operator
        fun <A: Comparable<A>> invoke(): Tree<A> = E ⑨
    }

    sealed class Color {

        // Красный
        internal object R: Color() { ⑩
            override fun toString(): String = "R"
        }

        // Черный
        internal object B: Color() {
            override fun toString(): String = "B"
        }
    }
}
```

① **Цвета импортируются, чтобы упростить код**
② **В родительском классе объявляются абстрактные свойства**
③ **Методы `isTR` и `isTB` проверяют, является ли дерево непустым и красным или непустым и черным соответственно**

④ **Абстрактный класс `Empty` позволяет вынести реализацию функций в этот класс и не использовать сопоставление с образцом в родительском классе `Tree`**
⑤ **Свойства, не имеющие смысла для класса `Empty`, инициализируются лениво и возбуждают исключение**
⑥ **Пустое дерево является черным**
⑦ **Пустое дерево представлено синглтоном `E`**
⑧ **Непустое дерево может быть красным или черным**
⑨ **Эта функция возвращает пустое дерево**
⑩ **Цвета – это объекты-синглтоны**

Функция `contains` здесь не представлена, как и некоторые другие функции, такие как `fold...`, `map` и др., потому что они не отличаются от версий в стандартном дереве. Как вы увидите далее, отличаются только функции `plus` и `minus`.

11.1.2 Добавление элемента в красно-черное дерево

Основной характеристикой красно-черного дерева являются инварианты, которые нужно проверить. При изменении дерево проверит, не нарушены ли эти инварианты, и, если потребуется, восстановит их поворотом и сменой цвета. Вот эти инварианты:

- пустое дерево всегда черное (это условие не меняется, поэтому проверять его не нужно);
- левая и правая ветви красного дерева – черные; при спуске вниз по дереву не должно встречаться двух красных ветвей, следующих друг за другом;
- все пути от корня до пустых поддеревьев имеют одинаковое число черных элементов.

Добавление элемента в красно-черное дерево – это довольно сложный процесс, включающий проверку инвариантов после вставки (и перебалансировку, если необходимо). Вот алгоритм этого процесса:

- пустое дерево всегда черное;
- вставка выполняется как в обычном дереве, но за ней следует перебалансировка;
- в результате вставки элемента в пустое дерево получается красное дерево;
- после балансировки корень окрашивается в черный.

На рис. 11.1–11.7 показан процесс вставки целых чисел от 1 до 7 в первоначально пустое дерево. Это худший из возможных случаев, когда элементы вставляются по порядку. Если производить подобную вставку в обычное бинарное дерево, в результате получится полностью несбалансированное дерево. На рис. 11.1 показана вставка элемента 1 в пустое дерево. Поскольку вставка происходит в пустое дерево, ему присваивается начальный красный цвет. После вставки элемента корень окрашивается в черный.

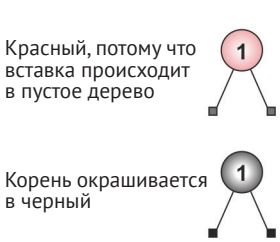


Рис. 11.1 Шаг 1: вставка числа 1 в первоначально пустое дерево

На рис. 11.2 показана вставка элемента 2. Вставляемый элемент окрашивается в красный, корень остается черным, и это дерево не требуется балансировать.

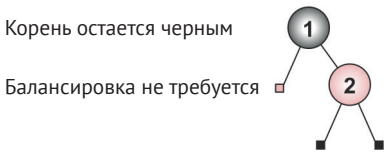


Рис. 11.2 Шаг 2: вставка числа 2 в пустое поддерево

На рис. 11.3 показана вставка элемента 3. Вставляемый элемент окрашивается в красный. Дерево требует балансировки, потому что подряд идут два красных элемента. Так как красный элемент теперь имеет два дочерних элемента, они окрашиваются в черный. (Дочерние элементы красного элемента всегда окрашиваются в черный.) В заключение корень окрашивается в черный.

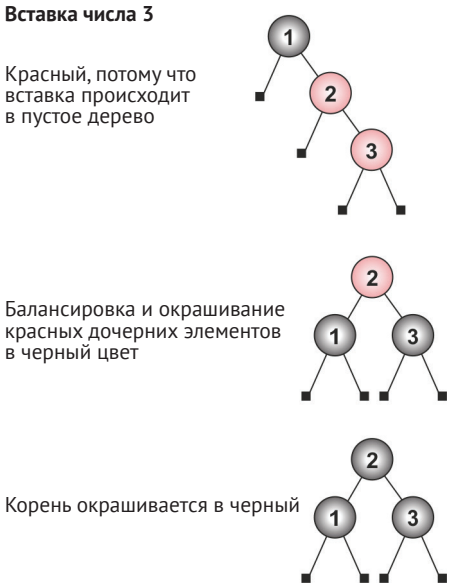


Рис. 11.3 Шаг 3: вставка числа 3 в пустое поддерево

На рис. 11.4 показана вставка элемента 4. Никаких дополнительных манипуляций с деревом не требуется.

Вставка числа 4

Балансировка
и окрашивание корня
в черный цвет
не требуются

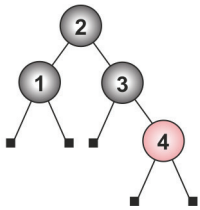
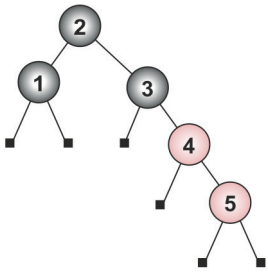


Рис. 11.4 Шаг 4: вставка числа 4 в пустое поддерево

На рис. 11.5 показана вставка элемента 5. Так как теперь появилось два красных элемента, следующих друг за другом, дерево требуется сбалансировать. В результате элемент 3 становится левым дочерним элементом элемента 4, а элемент 4 становится правым дочерним элементом элемента 2.

Вставка числа 5



Балансировка

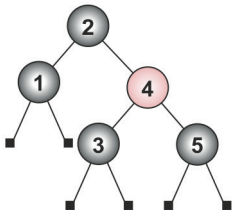
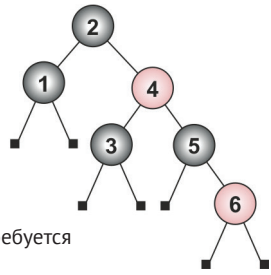


Рис. 11.5 Шаг 5: вставка числа 5 в пустое поддерево

На рис. 11.6 показана вставка элемента 6. Никаких дополнительных манипуляций с деревом не требуется.



Вставка числа 6

Балансировка не требуется

Рис. 11.6 Шаг 6: вставка числа 6 в пустое поддерево

На рис. 11.7 показана вставка элемента 7. Так как теперь появились два красных элемента, 6 и 7, следующих друг за другом, дерево требуется сбалансировать. На первом шаге элемент 5 становится левым дочерним элементом элемента 6, а элемент 6 становится правым дочерним элементом элемента 4. В результате снова получаем два красных элемента, 4 и 6, следующих друг за другом. Дерево снова нужно сбалансировать. В результате балансировки элемент 4 становится корнем, элемент 2 – левым дочерним элементом элемента 4, и элемент 3 – правым дочерним элементом элемента 2. Элемент 6 окрашивается в черный, потому что все пути до пустых поддеревьев должны иметь одинаковое число черных элементов. На последнем этапе корень окрашивается в черный.

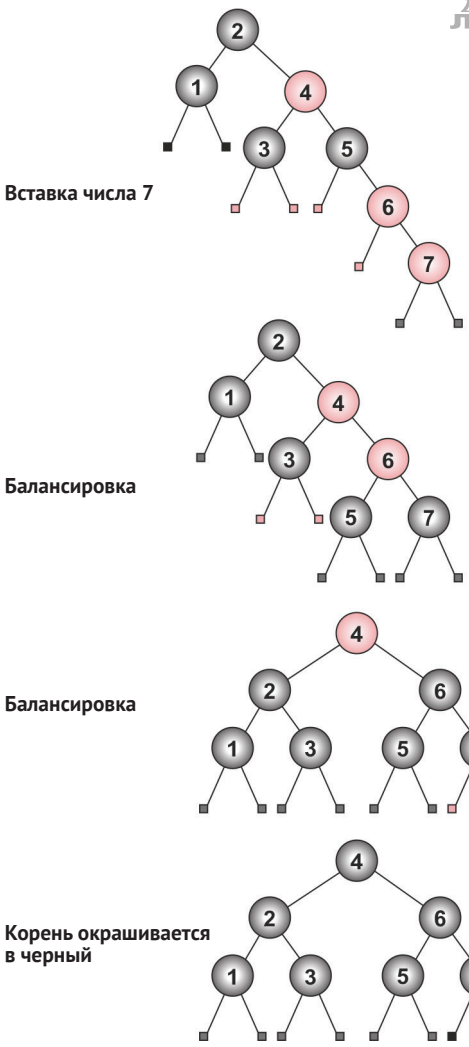


Рис. 11.7 Шаг 7: вставка числа 7 в пустое поддерево

Функция `balance` принимает те же аргументы, что и конструктор дерева: `color`, `left`, `value` и `right`. Эти четыре параметра проверяются на

соответствие различным шаблонам, и выполняются соответствующие действия. Функция `balance` заменяет конструктор дерева. Везде, где используется конструктор, мы должны заменить его этой функцией. В следующем списке пыеречислены все шаблоны аргументов и результаты их преобразования этой функцией:

1. (T B (T R (T R a x b) y c) z d) → (T R (T B a x b) y (T B c z d));
2. (T B (T R a x (T R b y c)) z d) → (T R (T B a x b) y (T B c z d));
3. (T B a x (T R (T R b y c) z d)) → (T R (T B a x b) y (T B c z d));
4. (T B a x (T R b y (T R c z d))) → (T R (T B a x b) y (T B c z d));
5. (T color a x b) → (T color a x b).

Каждое выражение в скобках соответствует дереву. Буква **T** обозначает непустое дерево, а буквы **B** и **R** – черный и красный цвета. Строчные буквы представляют любые значения, допустимые в этом месте. Шаблоны слева (слева от стрелки →) применяются в порядке убывания, т. е., если найдено соответствие шаблону слева, применяется соответствующий правый шаблон. Этот способ представления вещей можно сравнить с выражением `when`: последний шаблон является вариантом по умолчанию.

Упражнение 11.1

Напишите функции `plus`, `balance` и `blacken` для добавления элемента в красно-черное дерево. Вот их сигнатуры:

```
operator fun plus(value: @UnsafeVariance A): Tree<A>

fun balance(color: Color, left: Tree<A>, value: A, right: Tree<A>):Tree<A>

fun <A: Comparable<A>> blacken(): Tree<A>
```


Подсказка

Напишите защищенную функцию `add`, которая выполняет обычное добавление элемента, а затем замените вызовы конструктора вызовами функции `balance`. Затем напишите функцию `blacken` и, наконец, функцию `plus` в родительском классе, вызывающую `blacken` для балансировки результата, возвращаемого функцией `add`. Все функции должны быть приватными или защищенными, за исключением функции `plus`, которая должна быть общедоступной.

Решение

В функции `balance`, реализованной в классе `Tree`, шаблоны можно представить в виде выражений `when`, и использовать псевдонимы типов, чтобы сократить код:

```
protected fun balance(color: Color,
                      left: Tree<@UnsafeVariance A>,
                      value: @UnsafeVariance A,
                      right: Tree<@UnsafeVariance A>): Tree<A> = when {

    // balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
    color == B && left.isTR && left.left.isTR ->
        T(R, left.left.blacken(), left.value, T(B, left.right, value, right))
```

```
    // balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
    color == B && left.isTR && left.right.isTR ->
        T(R, T(B, left.left, left.value, left.right.left), left.right.value,
          T(B, left.right.right, value, right))

    // balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
    color == B && right.isTR && right.left.isTR ->
        T(R, T(B, left, value, right.left.left), right.left.value,
          T(B, right.left.right, right.value, right.right))

    // balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
    color == B && right.isTR && right.right.isTR ->
        T(R, T(B, left, value, right.left), right.value, right.right.blacken())

    // balance color a x b = T color a x b
    else -> T(color, left, value, right)
}
```

Каждая ветка в `when` реализует один из шаблонов, перечисленных выше (они показаны в комментариях). Если вы решите сравнить их, сделать это будет проще в текстовом редакторе, чем перелистывать страницы в книге.

Функция `add` напоминает аналогичную ей в стандартном бинарном дереве поиска и отличается только тем, что вместо конструктора `T` вызывает функцию `balance`. Функцию `add` можно объявить как абстрактную в классе `Tree` и добавить конкретные реализации в `Empty` и `T`. Вот реализация в классе `Empty`:

```
override fun add(newVal: @UnsafeVariance A): Tree<A> = T(R, E, newVal, E)
```

А это реализация в классе `T`:

```
override fun add(newVal: @UnsafeVariance A): Tree<A> = when {
    newVal < value -> balance(color, left.add(newVal), value, right)
    newVal > value -> balance(color, left, value, right.add(newVal))
    else           -> when (color) {
        B -> T(B, left, newVal, right)
        R -> T(R, left, newVal, right)
    }
}
```

В предыдущих реализациях класса `Tree` мы столкнулись с невозможностью реализовать какую-либо функцию в синглтоне `Empty`, потому что параметр `A` был недоступен (синглтон параметризован типом `Nothing`). Объявив класс `Empty` абстрактным, параметризовав его типом `A` и унаследовав объект-синглтон `E<Nothing>`, мы сможем реализовать функции в классе `Empty` и сохранить возможность представлять пустые деревья объектом-синглтоном.

Функция `blacken` тоже объявлена в классе `Tree` как абстрактная и имеет конкретные реализации в подклассах. Вот ее реализация в `Empty`:

```
override fun blacken(): Tree<A> = E
```

А вот реализация в `T`:

```
override fun blacken(): Tree<A> = T(B, left, value, right)
```

Наконец, функция `plus` определяется в классе `Tree` с ключевым словом `operator`. Это позволит вызывать ее с помощью оператора +. Она возвращает сбалансированный результат функции `add`:

```
operator fun plus(value: @UnsafeVariance A): Tree<A> =
    add(value).blacken()
```

11.1.3 Удаление элементов из красно-черного дерева

Удаление элемента из красно-черного дерева обсуждается в работе уже знакомых нам авторов – Германа (Germane) и Майта (Might)[1]. Реализация в Kotlin слишком длинная, чтобы включить ее в эту книгу, но вы найдете ее в примерах, сопровождающих книгу (http://github.com/pysaumont/fpinKotlin). Мы используем ее в следующем упражнении.

11.2 Практические примеры использования красно-черных деревьев: ассоциативные массивы

Деревья целых чисел редко используются на практике. Более востребованным применением бинарных деревьев поиска являются ассоциативные массивы, которые также называют словарями. Ассоциативные массивы – это наборы пар ключ/значение, которые позволяют вставлять, удалять и быстро находить любые пары.

Ассоциативные массивы хорошо знакомы программистам, и Kotlin предлагает несколько реализаций, среди которых наибольшей популярностью пользуются типы `Map` и `MutableMap`. Но `MutableMap` нельзя использовать в многопоточном окружении без дополнительных механизмов защиты, которые сложно правильно спроектировать и использовать. Тип `Map`, напротив, защищен от подобных проблем. Но он неэффективен, потому что не поддерживает совместного использования данных, поэтому каждая операция вставки или удаления создает новый ассоциативный массив.

11.2.1 Реализация Map

Функциональные деревья, такие как красно-черное дерево, разработанное нами выше, обладают преимуществом неизменности, что позволяет использовать их в многопоточных окружениях, не заботясь о блокировках и синхронизации. Они также обладают хорошей производительностью, поскольку при добавлении или удалении элемента большая часть данных совместно используется старым и новым деревом. В листинге 11.3 показан интерфейс `Map`, который можно реализовать с применением красно-черного дерева.

---

[1] *Кимбалл Герман* (Kimball Germane) и *Мэттью Майт* (Matthew Might). The missing method: Deleting from Okasaki's red-black trees (http://matt.might.net/articles/red-black-delete/).

Листинг 11.3 Функциональный ассоциативный массив

```
class Map<out K: Comparable<@UnsafeVariance K>, V> {

    operator fun plus(entry: Pair<@UnsafeVariance K, V>): Map<K, V> =
        TODO()

    operator fun minus(key: @UnsafeVariance K): Map<K, V> =
        TODO()

    fun contains(key: @UnsafeVariance K): Boolean = TODO()

    fun get(key: @UnsafeVariance K): Result<MapEntry<@UnsafeVariance K, V>> =
        TODO()

    fun isEmpty(): Boolean = TODO()

    fun size(): Int = TODO("size")

    companion object {
        operator fun invoke(): Map<Nothing, Nothing> = Map()
    }
}
```


Упражнение 11.2

Закончите класс `Map`, реализовав все функции.

Подсказка

Используйте прием делегирования. При таком подходе вы сможете реализовать любую функцию одной строкой кода. Единственная проблема – выбор способа хранения данных в ассоциативном массиве. Но она не должна вызвать у вас сложностей. Возможно, вам придется изменить тип аргумента функции `plus` и, может быть, тип значения, возвращаемого функцией `get`. Вам также придется добавить функцию `get` в класс `Tree`, возвращающую элемент. Вот сигнатура этой функции:

```
fun operator get(element: @UnsafeVariance A): Result<A>
```

Эта функция возвращает не свой параметр, а экземпляр `Result` с элементом, если такой элемент присутствует в дереве, или пустой результат. Также определите функцию `isEmpty` в классе `Tree`. Затем определите класс `MapEntry` для представления пар ключ/значение и организуйте хранение его экземпляров в дереве.

Решение

Класс `MapEntry` напоминает класс `Pair`, но имеет одно важное отличие: он должен поддерживать возможность сравнения экземпляров по ключу. Функции `equals` и `hashCode` также будут основаны на сравнении и хешкодах ключей. Вот одна из возможных реализаций:

```
class MapEntry<K: Comparable<@UnsafeVariance K>, V>
        private constructor(private val key: K, val value: Result<V>):
        Comparable<MapEntry<K, V>> {

    override fun compareTo(other: MapEntry<K, V>): Int =
```

```
        this.key.compareTo(other.key)

    override fun toString(): String = "MapEntry($key, $value)"

    override fun equals(other: Any?): Boolean =
        this === other || when (other) {
            is MapEntry<*, *> -> key == other.key
            else -> false
        }

    override fun hashCode(): Int = key.hashCode()

    companion object {

        fun <K: Comparable<K>, V> of(key: K, value: V): MapEntry<K, V> =
            MapEntry(key, Result(value))

        operator
        fun <K: Comparable<K>, V> invoke(pair: Pair<K, V>): MapEntry<K, V> =
            MapEntry(pair.first, Result(pair.second))

        operator
        fun <K: Comparable<K>, V> invoke(key: K): MapEntry<K,V> =
            MapEntry(key, Result())
    }
}
```

В реализации класса `Map` остается только делегировать все операции экземпляру `Tree<MapEntry<Key, Value>>`. Вот одна из возможных реализаций:

```
class Map<out K: Comparable<@UnsafeVariance K>, V>(
        private val delegate: Tree<MapEntry<@UnsafeVariance K, V>> = Tree()){

    operator
    fun plus(entry: Pair<@UnsafeVariance K, V>): Map<K, V> =
        Map(delegate + MapEntry(entry))

    operator
    fun minus(key: @UnsafeVariance K): Map<K, V> =
        Map(delegate - MapEntry(key))

    fun contains(key: @UnsafeVariance K): Boolean =
        delegate.contains(MapEntry(key))

    operator
    fun get(key: @UnsafeVariance K): Result<MapEntry<@UnsafeVariance K, V>> =
        delegate[MapEntry(key)]

    fun isEmpty(): Boolean = delegate.isEmpty

    fun size() = delegate.size

    override fun toString() = delegate.toString()

    companion object {
        operator
        fun <K: Comparable<@UnsafeVariance K>, V> invoke(): Map<K, V> =
            Map()
    }
}
```

11.2.2 Расширение ассоциативных массивов

Мы делегировали не все операции, потому что некоторые из них не имеют смысла в текущих условиях. Но в некоторых особых случаях нам могут потребоваться дополнительные операции. Реализовать их просто: нужно расширить класс `Map` и добавить делегирующие функции.

Например, иногда может понадобиться найти объект с максимальным или минимальным ключом. Также иногда может понадобиться выполнить свертку ассоциативного массива, например чтобы получить список хранящихся в нем значений. Вот пример делегирующей функции `foldLeft`:

```
fun <B> foldLeft(identity: B, f: (B) ->
        (MapEntry<@UnsafeVariance K, V>) -> B, g: (B) -> (B) -> B): B =
    delegate.foldLeft(identity, { b ->
        { me: MapEntry<K, V> ->
            f(b)(me)
        }
    }, g)
```

Обратите внимание, что некоторые варианты свертки используются настолько часто, что заслуживают их абстрагирования внутри класса `Map`.

Упражнение 11.3

Напишите функцию `values` в классе `Map`, которая возвращает список значений, хранящихся в ассоциативном массиве, в порядке возрастания ключей.

Подсказка

Возможно, вам придется добавить новую функцию свертки в класс `Tree` и делегировать ей выполнение операции из класса `Map`.

Решение

Реализовать функцию `values` можно несколькими способами. Можно было бы делегировать работу функции `foldInOrder`, но она выполняет обход дерева в порядке возрастания элементов, и, если использовать ее для конструирования списка, элементы в таком списке будут расположены в порядке убывания.

Можно, конечно, перевернуть получившийся список, но это очень неэффективно. Лучше будет добавить в класс `Tree` функцию `foldInReverseOrder`. Напомню, как выглядит функция `foldInOrder`:

```
override fun <B> foldInOrder(identity: B, f: (B) -> (A) -> (B) -> B): B =
    f(left.foldInOrder(identity, f))(value)(right.foldInOrder(identity, f))
```

Нам нужно только изменить порядок на обратный:

```
override fun <B> foldInReverseOrder(identity: B,
                                    f: (B) -> (A) -> (B) -> B): B =
    f(right.foldInReverseOrder(identity, f))(value)(left
        .foldInReverseOrder(identity, f))
```

Как обычно, реализация в Empty просто возвращает identity. Теперь мы можем делегировать работу этой функции в классе Map:

```
fun values(): List<V> =
    sequence(delegate.foldInReverseOrder(List<Result<V>>()) { lst1 ->
        { me ->
            { lst2 ->
                lst2.concat(lst1.cons(me.value))
            }
        }
    }).getOrElse(List())
```

11.2.3 Использование ключей, не поддерживающих сравнение

Класс Map удобен и относительно эффективен, но имеет большой недостаток по сравнению с ассоциативными массивами, к которым мы привыкли: ключи должны поддерживать возможность сравнения. В большинстве случаев в роли ключей используются сопоставимые типы, например целые числа или строки, но что, если потребуется использовать ключи несопоставимого типа?

Упражнение 11.4

Реализуйте версию Map, которая может работать с ключами, не поддерживающими возможность сравнения.

Подсказка

Вам, во-первых, потребуется изменить класс MapEntry, добавив в него возможность сравнения его экземпляров; и во-вторых, поскольку неодинаковые значения могут давать в результате одинаковые элементы MapEntry, конфликты следует разрешать, сохраняя оба элемента.

Решение

Сначала изменим класс MapEntry, убрав требование к сопоставимости ключей:

```
class MapEntry<K: Any, V> private constructor(val key: K,
                                              val value: Result<V>):
        Comparable<MapEntry<K, V>> {
```

Обратите внимание, что сам класс MapEntry по-прежнему реализует интерфейс Comparable, хотя мы убрали это требование к типу K.

Затем изменим реализацию для функции CompareTo. Как вариант, можно сравнивать хеш-коды ключей в экземплярах MapEntry. Обратите внимание, что нам пришлось немного изменить параметр типа класса, чтобы класс ключа расширял Any. По умолчанию он расширяет Any? – тип с поддержкой значения null, – поэтому мы должны предусмотреть обработку пустых значений в реализации CompareTo. Вызов other.hashCode() может сгенерировать исключение NullPointerException:

```
class MapEntry<K: Any, V> private constructor(val key: K,
                                              val value: Result<V>):
```

```
        Comparable<MapEntry<K, V>> {

    override fun compareTo(other: MapEntry<K, V>): Int =
        hashCode().compareTo(other.hashCode())
    ...
```

Затем обработаем конфликты, возникающие, когда два экземпляра `MapEntry` имеют разные ключи с одинаковым хеш-кодом. В таких случаях их оба следует оставить. Самое простое решение – сохранить экземпляры `MapEntry` в списке. Для этого необходимо изменить класс `Map`. Во-первых, делегат дерева (в конструкторе) будет иметь другой тип:

```
private val delegate: Tree<MapEntry<Int, List<Pair<K, V>>>> = Tree()
```

Далее нам понадобится функция для извлечения списка кортежей ключ/значение, соответствующих заданному хеш-коду:

```
private fun getAll(key: @UnsafeVariance K): Result<List<Pair<K, V>>> =
    delegate[MapEntry(key.hashCode())]
        .flatMap { x ->
            x.value.map { lt ->
                lt.map { t -> t }
            }
        }
```

Теперь определим функции `plus`, `contains`, `minus` и `get` в терминах `getAll`. Вот функция `add`:

```
operator fun plus(entry: Pair<@UnsafeVariance K, V>): Map<K, V> {
    val list = getAll(entry.first).map { lt ->
        lt.foldLeft(List(entry)) { lst ->
            { pair ->
                if (pair.first == entry.first) lst else lst.cons(pair)
            }
        }
    }.getOrElse { List(entry) }
    return Map(delegate + MapEntry.of(entry.first.hashCode(), list))
}
```

Функция `minus`:

```
operator fun minus(key: @UnsafeVariance K): Map<K, V> {
    val list = getAll(key).map { lt ->
        lt.foldLeft(List()) { lst: List<Pair<K, V>> ->
            { pair ->
                if (pair.first == key) lst else lst.cons(pair)
            }
        }
    }.getOrElse { List() }
    return when {
        list.isEmpty() -> Map(delegate - MapEntry(key.hashCode()))
        else -> Map(delegate + MapEntry.of(key.hashCode(), list))
    }
}
```

Функция `contains`:

```
fun contains(key: @UnsafeVariance K): Boolean =
    getAll(key).map { list ->
        list.exists { pair ->
            pair.first == key
        }
    }.getOrElse( false)
```

И функция `get`:

```
fun get(key: @UnsafeVariance K): Result<Pair<K, V>> =
    getAll(key).flatMap { list ->
        list.filter { pair ->
            pair.first == key
        }.headSafe()
    }
```

Обратите внимание, что теперь функции `values` и `foldLeft` перестали компилироваться. Исправьте эту проблему самостоятельно. Задача может показаться сложной, но это не так. Внимательно проследите типы. Если у вас ничего не получится, загляните в решение в примерах кода в репозитории GitHub (https://github.com/pysaumont/fpinkotlin).

С этими модификациями класс `Map` можно использовать с ключами, не поддерживающими сравнение. Использование списка для хранения кортежей ключ/значение может быть не самой эффективной реализацией, поскольку поиск в списке занимает время, пропорциональное количеству элементов. Но в большинстве случаев этот список будет содержать только один элемент, поэтому поиск будет выполняться очень быстро. По этой же причине нет смысла оптимизировать поиск первого вхождения, соответствующего ключу в функции `get`. Если вы решите все-таки оптимизировать реализацию, можете использовать специальную реализацию свертки (с «нулевым» параметром) вместо пары `filter` и `headSafe`, которые выполняют обход всего списка, перед тем как взять первый элемент. Если вы забыли, как это делается, – обратитесь к упражнению 8.13.

Следует заметить, что функция `minus` в этой реализации проверяет, является ли полученный список пар пустым. Если это так, она вызывает функцию `minus` делегата. Иначе вызывается функция `plus` для повторной вставки нового списка, из которого удален соответствующий элемент. Вспомните упражнение 10.1 из главы 10. Это возможно только потому, что мы решили реализовать `plus` таким образом, чтобы вместо исходного элемента вставлялся найденный элемент, равный элементу, присутствующему ассоциативном массиве. Если этого не сделать, придется сначала удалить элемент, а затем вставить новый с измененным списком.

11.3 Реализация функциональной приоритетной очереди

Как вы уже знаете, очередь – это список с определенным протоколом доступа. Очереди могут быть односторонними, как односвязный список, который мы так часто использовали в предыдущих главах. В этом случае используется протокол доступа «Последним пришел – первым ушел» (Last In First Out, LIFO). Очереди могут быть двухсторонними с протоколом доступа «Первым пришел – первым ушел» (First In First Out, FIFO). Но также есть структуры данных с более специализированными протоколами, и среди них – *приоритетная очередь*.

11.3.1 Протоколы доступа к приоритетной очереди

Значения могут вставляться в приоритетную очередь в любом порядке, но извлекаться должны в строго определенном порядке. Все значения имеют приоритет, и первыми извлекаться из очереди должны элементы с наивысшим приоритетом. *Приоритет* представлен порядком элементов, а значит, элементы должны поддерживать возможность сравнения.

Приоритет соответствует положению элемента в очереди. Самый высокий приоритет принадлежит элементу с самой меньшей позицией (первый элемент). По соглашению самый высокий приоритет представлен самым низким значением.

Поскольку приоритетная очередь содержит сопоставимые элементы, ее можно реализовать на основе дерева. Но с точки зрения пользователя, приоритетная очередь выглядит как список с головой (элемент с наивысшим приоритетом, т. е. с наименьшим значением) и хвостом (остальная часть очереди).

11.3.2 Варианты использования приоритетных очередей

Приоритетная очередь имеет множество вариантов практического применения. На ум сразу приходит идея быстрой сортировки. Вы можете вставлять элементы в приоритетную очередь в произвольном порядке и извлекать их отсортированными. Но это не основной способ использования этой структуры, хотя может пригодиться для сортировки небольших наборов данных.

Другой распространенный вариант использования – переупорядочение элементов после асинхронной параллельной обработки. Представьте, что у вас есть несколько страниц данных для обработки. Чтобы ускорить обработку, вы можете распределить данные между несколькими потоками выполнения, действующими параллельно. Но в этом случае нет никакой гарантии, что потоки вернут свои результаты в том же порядке, в каком получили исходные данные. Для повторного упорядочения страниц можно поместить их в очередь с приоритетами. Процесс, использующий результаты обработки, должен затем обратиться к очереди, чтобы проверить, является ли доступный элемент (начало очереди)

ожидаемым. Например, если страницы 1, 2, 3, 4, 5, 6, 7 и 8 сгенерированы восемью параллельными потоками, потребитель обращается к очереди, чтобы узнать, доступна ли страница 1. Если она доступна, он извлекает ее и обрабатывает. Если нет, то ждет ее появления.

В этом сценарии очередь действует как буфер и одновременно как механизм переупорядочения элементов. Обычно это подразумевает небольшое изменение размера очереди, потому что элементы удаляются из нее более или менее с той же скоростью, с которой добавляются. Это верно, если потребитель извлекает элементы примерно в том же темпе, что и восемь потоков. Если это не так, возможен вариант с несколькими потребителями.

Как я уже говорил выше, выбор реализации, как правило, зависит от компромисса между объемом памяти и временем или временем обработки и временем потребления. В данном случае вам придется выбирать между временем вставки и временем извлечения. В общем случае время извлечения должно быть меньше времени вставки, потому что операций извлечения, как правило, выполняется больше, чем операций вставки. (Иногда голова будет извлекаться без удаления из очереди.)

11.3.3 Требования к реализации

Реализовать приоритетную очередь можно на основе красно-черного дерева, которое обеспечивает быстрый поиск минимального значения. Но извлечение (чтение) элемента не означает удаление. Прочитав минимальное значение и обнаружив, что это не то, что вам нужно, вам придется вернуться позже и повторить поиск, на что требуется время. Решить эту проблему можно, запоминая минимальное значение при вставке. Другое дополнение, которое можно внести, касается удаления. Удаление элемента происходит относительно быстро, но, поскольку всегда будет удаляться наименьший элемент, можно попытаться оптимизировать структуру данных для этой операции.

Еще одно важное замечание касается дубликатов. Красно-черное дерево не допускает появления дубликатов, но приоритетная очередь должна их поддерживать, потому что наличие нескольких элементов с одинаковым приоритетом вполне допустимо. Решить эту проблему можно по аналогии с ассоциативными массивами – вместо отдельных элементов хранить списки элементов с одинаковым приоритетом. Но это, вероятно, будет не самое оптимальное решение.

11.3.4 Левосторонняя куча

Для удовлетворения требований к приоритетной очереди мы используем структуру данных с названием *левосторонняя куча*, описанную Окасаки[1]. Окасаки определяет левостороннюю кучу как пирамидально-упо-

[1] *Крис Окасаки* (Chris Okasaki). Purely Functional Data Structures. Cambridge University Press, 1999 (*Окасаки К.* Чисто функциональные структуры данных. М.: ДМК Пресс, 2016. ISBN: 978-5-97060-233-1. – *Прим. перев.*).

рядоченное (heap-ordered) дерево с дополнительным левосторонним свойством:

1. Пирамидально-упорядоченное дерево – это дерево, в котором каждая ветвь, исходящая из элемента, больше или равна самому элементу. Это гарантирует, что корнем такого дерева всегда будет наименьший элемент, что обеспечивает его мгновенную доступность.
2. Согласно левостороннему свойству, ранг левой ветви любого элемента больше или равен рангу правой ветви.
3. *Ранг* элемента – это длина правого пути (также называется правым спином) до пустого элемента. Левостороннее свойство гарантирует, что кратчайшим путем из любого элемента до пустого элемента является правый путь. Как следствие, элементы всегда располагаются в порядке возрастания в любом нисходящем пути. На рис. 11.8 показан пример левостороннего дерева.

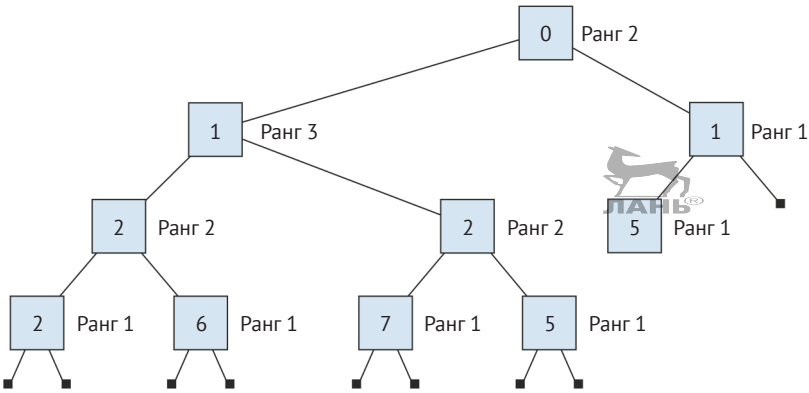


Рис. 11.8 Пирамидально-упорядоченное левостороннее дерево. Ветвь любого элемента больше или равна самому элементу, а ранг левой ветви больше или равен рангу соответствующей правой ветви

Как видите, эта структура позволяет получить элемент с наивысшим приоритетом за постоянное время, потому что он всегда будет корнем дерева. Этот элемент называется *головой* структуры. Удаление элемента – по аналогии со списком – заключается в возврате остальной части дерева после удаления корня. Это возвращаемое значение называется *хвостом* структуры.

11.3.5 Реализация левосторонней кучи

Основной класс левосторонней кучи называется `Heap` и должен реализовать свойства и методы дерева. В листинге 11.4 показана основная структура. Главное отличие от деревьев, которые мы разрабатывали до сих пор, заключается в том, что такие функции, как `right`, `left` и `head` (которая в предыдущих примерах называлась `value`), возвращают `Result` вместо фактического значения. Также обратите внимание, что ранг `rank`

вычисляется кодом, вызывающим конструктор, а не самим конструктором. Такой подход не обусловлен какими-то требованиями, я просто хотел показать другой способ организации вычислений. Поскольку конструкторы являются приватными, это отличие не просочится за пределы класса `Heap`.

Листинг 11.4 Структура левосторонней кучи

```
sealed class Heap<out A: Comparable<@UnsafeVariance A>> {

    internal abstract val left: Result<Heap<A>> ①
    internal abstract val right: Result<Heap<A>> ①
    internal abstract val head: Result<A> ①
    protected abstract val rank: Int
    abstract val size: Int ②
    abstract val isEmpty: Boolean

    abstract class Empty<out A: Comparable<@UnsafeVariance A>>: Heap<A>() { ③

        override val isEmpty: Boolean = true

        override val left: Result<Heap<A>> = Result(E)

        override val right: Result<Heap<A>> = Result(E)

        override val head: Result<A> =
            Result.failure("head() called on empty heap")

        override val rank: Int = 0

        override val size: Int = 0
    }

    internal object E: Empty<Nothing>() ④

    internal class H<out A: Comparable<@UnsafeVariance A>>(
                            override val rank: Int, ⑤
                            private val lft: Heap<A>,
                            private val hd: A,
                            private val rght: Heap<A>): Heap<A>() {

        override val isEmpty: Boolean = false

        override val left: Result<Heap<A>> = Result(lft)

        override val right: Result<Heap<A>> = Result(rght)

        override val head: Result<A> = Result(hd)

        override val size: Int = lft.size + rght.size + 1
    }

    companion object {
        operator fun <A: Comparable<A>> invoke(): Heap<A> = E ⑥
    }
}
```

① Функции `left`, `right` и `head` возвращают `Result`
② Свойство `size` – размер дерева, т. е. число элементов в нем
③ Класс `Empty` объявлен абстрактным

④ Объект-синглтон E представляет все пустые деревья
⑤ Свойство rank вычисляется за пределами подкласса H и передается в конструктор как аргумент
⑥ Эта функция возвращает пустое дерево

Упражнение 11.5

Первым делом вы должны реализовать в классе Heap возможность добавления новых элементов. Напишите функцию plus. Объявите ее функцией-оператором экземпляра в классе Heap со следующей сигнатурой:

```
operator fun plus(element: @UnsafeVariance A): Heap<A>
```

Главное требование заключается в том, что, если значение меньше любого элемента в куче, оно должно стать корнем новой кучи. Иначе корень кучи не должен измениться. Также должны соблюдаться другие требования о ранге и длине правого пути.

Подсказка

Определите функцию в объекте-компаньоне для создания кучи Heap из элемента и еще одну – для создания кучи путем объединения двух куч. Вот их сигнатуры:

```
operator fun <A: Comparable<A>> invoke(element: A): Heap<A>

fun <A: Comparable<A>> merge(first: Heap<A>, second: Heap<A>): Heap<A>
```

Затем определите функцию plus в терминах этих двух.

Решение

Функция для создания кучи из единственного элемента реализуется просто. Она создает новое дерево с рангом 1, элементом из аргумента в качестве головы и двумя пустыми кучами в левой и правой ветвях:

```
operator fun <A: Comparable<A>> invoke(element: A): Heap<A> =
    H(1, E, element, E)
```

Создание кучи объединением двух других куч немного сложнее. Для этого потребуется дополнительная вспомогательная функция, создающая кучу из одного элемента и двух куч:

```
protected
fun <A : Comparable<A>> merge(head: A,
                              first: Heap<A>,
                              second: Heap<A>): Heap<A> =
    when {
        first.rank >= second.rank -> H(second.rank + 1, first, head,
                                       second)
        else -> H(first.rank + 1, second, head, first)
    }
```

Этот код сначала сравнивает ранг первой и второй кучи. Если ранг первой кучи больше или равен рангу второй кучи, новый ранг устанавливается равным рангу второй кучи + 1, и затем кучи используются в указанном порядке (first, second). Иначе новый ранг устанавливается

равным рангу первой кучи + 1, и затем кучи используются в обратном порядке (second, first). Функцию слияния двух куч можно реализовать следующим образом:

```
fun <A: Comparable<A>> merge(first: Heap<A>, second: Heap<A>): Heap<A> =
    first.head.flatMap { fh ->
        second.head.flatMap { sh ->
            when {
                fh <= sh -> first.left.flatMap { fl ->
                    first.right.map { fr ->
                        merge(fh, fl, merge(fr, second))
                    }
                }
                else -> second.left.flatMap { sl ->
                    second.right.map { sr ->
                        merge(sh, sl, merge(first, sr))
                    }
                }
            }
        }
    }.getOrElse(when (first) {
        E -> second
        else -> first
    })
```

Если одна из двух куч окажется пустой, в качестве результата возвращается другая. Иначе вычисляется результат слияния. Определив эти функции, легко написать функцию plus:

```
operator fun plus(element: @UnsafeVariance A): Heap<A> =
    merge(this, Heap(element))
```

11.3.6 Реализация интерфейса, характерного для очередей

Наша куча реализована в виде дерева, но, с точки зрения пользователя, она похожа на приоритетную очередь, т. е. на своеобразный список, голова которого всегда содержит наименьший элемент. Как мы уже знаем, корень дерева тоже называется головой, а все остальное – хвостом.

Упражнение 11.6

Напишите функцию tail, возвращающую хвост кучи. Эта функция, как и функция head, должна возвращать Result, чтобы обеспечить безопасность при вызове для пустой очереди. Вот ее сигнатура в родительском классе Heap:

```
abstract fun tail(): Result<Heap<A>>
```

Решение

Реализация в Empty очевидна и возвращает Failure:

```
override fun tail(): Result<Heap<A>> =
    Result.failure(IllegalStateException("tail() called on empty heap"))
```

Реализация в `H` ничуть не сложнее. Она использует функции, которые мы написали в предыдущем упражнении, и возвращает результат слияния левой и правой ветвей:

```
override fun tail(): Result<Heap<A>> = Result(merge(lft, rght))
```

Упражнение 11.7

Напишите функцию `get`, которая принимает параметр `int` и возвращает *n*-й элемент в порядке приоритета. Она должна возвращать `Result`, чтобы обеспечить безопасность, когда элемент не найден. Вот ее сигнатура в родительском классе `Heap`:

```
fun get(index: Int): Result<A>
```

Решение

Реализация в `Empty` очевидна и возвращает `Failure`:

```
override fun get(index: Int): Result<A> =
    Result.failure(NoSuchElementException("Index out of bounds"))
```

Реализация в `H` выглядит так же просто. Сначала она проверяет индекс. Если он равен 0, возвращает `Success` со значением головы. Иначе выполняет рекурсивный поиск элемента с индексом *index* - 1 в хвосте. Поскольку самого хвоста не существует, а есть только значение, возвращаемое функцией `tail` (`Result`), ее результат преобразуется в плоский список с рекурсивным вызовом `get`:

```
override fun get(index: Int): Result<A> = when (index) {
    0 -> Result(hd)
    else -> tail().flatMap { it.get(index - 1) }
}
```

У вас может появиться желание выводить более конкретное сообщение в случае, когда искомый элемент не найден. Вы не можете использовать значение `index` в реализации в `Empty`, потому что оно уже уменьшено. Преодолеть эту проблему можно множеством способов. Реализуйте свое решение самостоятельно.

11.4 Элементы и сортированные списки

Иногда может понадобиться преобразовать кучу в отсортированный список. На первый взгляд, в этом нет ничего сложного – достаточно просто извлекать элементы из кучи по одному и добавлять в список. Но это частный случай более общей операции свертки.

Упражнение 11.8

Напишите функцию `pop`, которая выталкивает элемент из кучи, возвращая необязательную (`Option`) пару, содержащую голову и хвост кучи. Если куча пустая, функция должна вернуть `None`. Затем напишите функцию, создающую отсортированный список из кучи.

Подсказка

Вот сигнатура функции pop:

```
fun pop(): Option<Pair<A, Heap<A>>>
```

Решение

Вот одна из возможных реализаций pop. В классе Heap она объявляется как абстрактная:

```
abstract fun pop(): Option<Pair<A, Heap<A>>>
```

Реализация в классе Empty возвращает пустое значение:

```
override fun pop(): Option<Pair<A, Heap<A>>> = Option()
```

В классе H функция pop возвращает Option с парой, содержащей голову и хвост кучи:

```
override fun pop(): Option<Pair<A, Heap<A>>> =
    Option(Pair(hd, merge(lft, rght)))
```

Тип значения, возвращаемого функцией pop, позволяет использовать ее в функции unfold в типе List. Вот как можно реализовать функцию toList в родительском классе Heap:

```
fun toList(): List<A> = unfold(this) { it.pop() }
```

Упражнение 11.9

В предыдущем упражнении мы использовали функцию unfold, которая создает List<A>. Но мы можем обобщить ее для получения любого типа B, если List<A> является одной из возможных реализаций B.

Напишите функцию unfold в классе Heap, возвращающую B вместо List<A>, и перепишите функцию toList в терминах этой функции. Затем напишите функцию foldLeft с той же сигнатурой, что и функция List.foldLeft.

Подсказка

Для начала скопируйте реализацию функции unfold из класса List в класс Heap и внесите необходимые изменения, чтобы заменить тип List<A> на более общий тип B. Вот реализация функции unfold, создающая List:

```
fun <A, S> unfold(z: S, getNext: (S) -> Option<Pair<A, S>>): List<A> {
    tailrec fun unfold(acc: List<A>, z: S): List<A> {
        val next = getNext(z)
        return when (next) {
            is Option.None -> acc
            is Option.Some ->
                unfold(acc.cons(next.value.first), next.value.second)
        }
    }
    return unfold(List.Nil, z).reverse()
}
```

Решение

Чтобы обобщить эту функцию, нужно заменить все ссылки `List<A>` на `B`:

- `List.Nil` следует заменить на единичное значение типа `B`, которое должно передаваться в функцию в дополнительном параметре;
- `acc.cons(next.value.first)` – это реализация функции типа `(List<A>) -> (A) -> List<A>`, которая используется для создания списка; в обобщенной версии реализация этой функции неизвестна на этапе компиляции, поэтому она должна передаваться в дополнительном параметре;
- вызов `reverse` перед возвратом списка – это операция, характерная для списков `List`, и его следует убрать:

```
fun <A, S, B> unfold(z: S,
                     getNext: (S) -> Option<Pair<A, S>>,
                     identity: B,
                     f: (B) -> (A) -> B): B {
    tailrec fun unfold(acc: B, z: S): B {
        val next = getNext(z)
        return when (next) {
            is Option.None -> acc
            is Option.Some ->
                unfold(f(acc)(next.value.first), next.value.second)
        }
    }
    return unfold(identity, z)
}
```

Теперь функцию `foldLeft` можно переписать так:

```
fun <B> foldLeft(identity: B, f: (B) -> (A) -> B): B =
    unfold(this, { it.pop() }, identity, f)
```

А функцию `toList` можно переписать так:

```
fun toList(): List<A> =
    foldLeft(List<A>()) { list -> { a -> list.cons(a) } }.reverse()
```

11.5 Приоритетная очередь для несопоставимых элементов

Для вставки элементов в приоритетную очередь нужна возможность сравнивать их приоритеты. Но приоритет не всегда является свойством элементов; не все элементы реализуют интерфейс `Comparable`. Элементы, которые не реализуют этот интерфейс, все еще можно сравнить с помощью реализации класса `Comparator`. Что вы могли бы предложить для приоритетной очереди с этой точки зрения?

Упражнение 11.10

Измените класс `Heap` так, чтобы его можно было использовать с элементами, реализующими интерфейс `Comparable`, и в случаях, когда предлагается своя реализация `Comparator`.

Решение

Сначала нужно изменить объявление класса `Heap`, заменив

```
sealed class Heap<out A: Comparable<@UnsafeVariance A>>
```

на

```
sealed class Heap<out A>
```

Затем нужно аналогично изменить объявления подклассов и добавить в класс `Heap` свойство для хранения экземпляра `Comparator`. Поскольку компаратор является необязательным, это свойство должно иметь тип `Result<Comparator>` и может хранить пустое значение. Вот абстрактное свойство в классе `Heap`:

```
internal abstract val comparator: Result<Comparator<@UnsafeVariance A>>
```

Объект-синглтон `E` нужно удалить, а класс `Empty` объявить внутренним, а не абстрактным, и в его конструктор добавить свойство типа `Result<Comparator>` со значением по умолчанию `Empty`:

```
internal class Empty<out A>(
        override val comparator: Result<Comparator<@UnsafeVariance A>> =
            Result.Empty): Heap<A>()
{
```

То же самое нужно проделать с классом `H`, но использовать существующий конструктор (если имеется) для получения значения по умолчанию:

```
internal class H<out A>(override val rank: Int,
        internal val lft: Heap<A>,
        internal val hd: A,
        internal val rght: Heap<A>,
        override val comparator: Result<Comparator<@UnsafeVariance A>> =
    lft.comparator.orElse { rght.comparator }): Heap<A>() {
```


В объекте-компаньоне есть несколько версий функций для создания пустой кучи: одна не требует компаратора, а другая принимает `Result<Comparator>`. Для простоты добавьте версию, принимающую `Comparator`. Также нужно реализовать конкретную версию функции, создающей кучу из одного элемента:

```
operator fun <A: Comparable<A>> invoke(): Heap<A> = Empty()

operator fun <A> invoke(comparator: Comparator<A>): Heap<A> =
    Empty(Result(comparator))

operator fun <A> invoke(comparator: Result<Comparator<A>>): Heap<A> =
    Empty(comparator)

operator fun <A> invoke(element: A, comparator: Result<Comparator<A>>):
    Heap<A> = H(1, Empty(comparator), element, Empty(comparator), comparator)
```

Обратите внимание, что функция `invoke()` без аргумента может вызываться только для типа `Comparable`. Как следствие, нельзя создать пустую кучу с элементами несопоставимого типа, не предоставив компаратор.

Чтобы решить проблему, нужно изменить функцию, принимающую элемент в качестве параметра, чтобы она создавала компаратор или принимала его в дополнительном параметре:

```
operator fun <A: Comparable<A>> invoke(element: A): Heap<A> =
    invoke(element, Comparator { o1: A, o2: A -> o1.compareTo(o2) })

operator fun <A> invoke(element: A, comparator: Comparator<A>): Heap<A> =
    H(1, Empty(Result(comparator)), element,
      Empty(Result(comparator)), Result(comparator))
```

Функцию `merge`, принимающую элемент и две кучи, тоже нужно изменить, но на этот раз компаратор должен извлекаться из куч в аргументах:

```
protected fun <A> merge(head: A, first: Heap<A>, second: Heap<A>):Heap<A> =
    first.comparator.orElse { second.comparator }.let {
        when {
            first.rank >= second.rank -> H(second.rank + 1,
                                           first, head, second, it)
            else -> H(first.rank + 1, second, head, first, it)
        }
    }
```

В реализации функции `merge`, принимающей две кучи, можно использовать компаратор из любой из объединяемых куч. Если ни в одной нет компаратора, можно использовать `Result.Empty`. Чтобы не извлекать компаратор из аргументов в каждом рекурсивном вызове, функцию можно разбить на две:

```
fun <A> merge(first: Heap<A>, second: Heap<A>,
              comparator: Result<Comparator<A>> =
                  first.comparator.orElse { second.comparator }): Heap<A> =
    first.head.flatMap { fh ->
        second.head.flatMap { sh ->
            when {
                compare(fh, sh, comparator) <= 0 ->
                    first.left.flatMap { fl ->
                        first.right.map { fr ->
                            merge(fh, fl, merge(fr, second, comparator))
                        }
                    }
                else -> second.left.flatMap { sl ->
                    second.right.map { sr ->
                        merge(sh, sl, merge(first, sr, comparator))
                    }
                }
            }
        }
    }.getOrElse(when (first) {
        is Empty -> second
        else -> first
    })
```

Вторая функция использует вспомогательную функцию `compare`:

```
private fun <A> compare(first: A, second: A,
                        comparator: Result<Comparator<A>>): Int =
    comparator.map { comp ->
        comp.compare(first, second)
    }.getOrElse { (first as Comparable<A>).compareTo(second) }
```

Эта функция выполняет приведение одного из своих аргументов, но мы знаем, что в данном случае нет риска получить `ClassCastException`, потому что мы гарантировали, что куча не может быть создана без компаратора, если параметр типа не реализует интерфейс `Comparable`. Функцию `plus` также следует изменить, как показано ниже:


```
operator fun plus(element: @UnsafeVariance A): Heap<A> =
    merge(this, Heap(element, comparator))
```

Наконец, функции `left` и `right` в классе `Empty` следует изменить так:

```
override val left: Result<Heap<A>> = Result(this)
override val right: Result<Heap<A>> = Result(this)
```

Итоги

- Деревья можно балансировать для улучшения производительности и чтобы избежать переполнения стека в рекурсивных операциях.
- Красно-черное дерево балансируется автоматически, что освобождает нас от необходимости беспокоиться о его балансировки.
- Ассоциативный массив можно реализовать, делегируя дереву все операции по сохранению кортежей ключ/значение.
- Ассоциативные массивы с несопоставимыми ключами должны обрабатывать конфликты при упорядочении элементов с одинаковыми ключами.
- Приоритетные очереди – это структуры, позволяющие извлекать элементы в порядке их приоритетов.
- Приоритетные очереди можно реализовать на основе левосторонней кучи – бинарного пирамидально-упорядоченного дерева.
- Приоритетные очереди с несопоставимыми элементами можно конструировать с использованием дополнительных компараторов.

12 Функциональный ввод/вывод

Эта глава охватывает следующие темы:

- безопасное применение эффектов внутри контекста;
- комбинирование эффектов;
- безопасное чтение данных;
- тип `IO` и императивные структуры управления;
- комбинирование операций `IO`.

До сих пор вы учились писать безопасные программы, не дающие никаких полезных результатов. Вы узнали, как писать истинные функции и конструировать из них более мощные. Также – что особенно интересно – вы узнали, как безопасным и функциональным способом использовать нефункциональные операции. *Нефункциональные операции* – это операции, создающие побочные эффекты, такие как исключения, изменяющие внешний мир или зависящие от внешнего мира. Например, вы узнали, как целочисленное деление, которое является потенциально небезопасной операцией, превратить в безопасную операцию, поместив его в вычислительный контекст. Вот несколько примеров вычислительных контекстов, которые мы создали в предыдущих главах:

- в главе 7 мы реализовали тип `Result`, позволяющий безопасно использовать функции, которые могут генерировать ошибки;
- тип `Option`, разработанный нами в главе 6, тоже является вычислительным контекстом; он позволяет безопасно применять функции, которые иногда (для некоторых комбинаций аргументов) могут ничего не возвращать;

- в главах 5 и 8 мы исследовали класс `List` – еще один вычислительный контекст, но не для обработки и предотвращения ошибочных ситуаций, а для организации функций, которые работают с отдельными элементами в контексте коллекции элементов. Он также обрабатывает отсутствующие данные, представляя их как пустой список;
- типы `Lazy` и `Stream` из главы 9 тоже вычислительные контексты. Первый предназначен для представления данных, которые могут оставаться неинициализированными, пока не потребуются, а второй решает аналогичную задачу в отношении коллекций.

Занимаясь этими типами, мы не ставили себе задачу получить полезный результат. В этой главе мы познакомимся с методами получения практических результатов из наших программ, такими как отображение результатов на экране или передача результатов в другие программы.

12.1. Что означает «эффект внутри контекста»?

Вспомним, как мы поступали, когда требовалось применить функцию к результату целочисленной операции. Представьте, что нам потребовалось написать функцию `inverse`, которая вычисляет обратное число для заданного целочисленного значения:

```
val inverse: (Int) -> Result<Double> = { x ->
    when {
        x != 0 -> Result(1.toDouble() / x)
        else -> Result.failure("Division by 0")
    }
}
```

Эту функцию можно применить к целочисленному значению, но при комбинировании с другими функциями это значение будет выводиться другой функцией! Обычно это происходит уже в контексте, часто в том же типе контекста. Например:

```
val ri: Result<Int> = ...
val rd: Result<Double> = ri.flatMap(inverse)
```

Обратите внимание, что мы не извлекаем значение `ri` из контекста, чтобы применить функцию. Это обратная ситуация, когда функция передается в контекст (тип `Result`), чтобы применить ее внутри контекста. В результате создается новый контекст, который обертывает полученное значение. В предыдущем фрагменте мы передали функцию в контекст `ri` и создали новый контекст `rd`.

Этот код действует аккуратно и безопасно. Ничего плохого в нем произойти не может, и не может возникнуть никаких исключений. В этом прелесть программирования с чистыми функциями: у нас есть программа, которая работает всегда независимо от того, какие данные передаются на вход. Но теперь возникает вопрос: как использовать полученный результат? Допустим, нам нужно вывести результат в консоль; как это сделать?

12.1.1 Обработка эффектов

Чистые функции определяются как функции без видимых побочных эффектов. Эффект – это все, что можно наблюдать за пределами программы. Роль функции – возвращать значение, а *побочные эффекты* – это может быть все что угодно, кроме возвращаемого значения, что можно наблюдать снаружи функции. Эти явления называют побочными эффектами, потому что они дополняют возвращаемое значение. Просто эффект, напротив, похож на побочный эффект, но является основной (и, как правило, единственной) ролью программы. Безопасность наших программ достигается разработкой программ с использованием чистых функций (без побочных эффектов) и чистых эффектов (не возвращающих никаких значений) функциональным способом.

Встает вопрос: что означает «функциональная обработка эффектов»? Вот наиболее точное определение, которое я могу дать на данном этапе: обработка эффектов таким способом, чтобы они не препятствовали применению принципов функционального программирования, наиболее важным из которых является принцип ссылочной прозрачности.

Достичь этой цели порой очень непросто, но часто бывает достаточно хотя бы приблизиться к ней. Есть разные способы, помогающие в этом, и вам решать, какие из них вы будете использовать. Применение эффектов внутри контекста – это самый простой способ заставить функциональные программы создавать наблюдаемые эффекты.

12.1.2 Реализация эффектов

Как я уже отмечал, эффект – это все, что можно наблюдать вне программы. Чтобы представлять какую-либо ценность, эффект должен, как правило, отражать результат программы. В таком случае обычно нужно взять результат и сделать с ним что-то заметное.

Обратите внимание, что *заметное* не всегда означает наблюдаемое человеком-оператором. Часто результат может наблюдать другая программа, которая может преобразовать этот эффект в нечто, наблюдаемое человеком-оператором, – синхронно или асинхронно.

Вывод на экран компьютера может видеть оператор. Часто именно это и требуется. С другой стороны, запись в базу данных не всегда видна пользователю непосредственно. Конечно, пользователь может сам искать результаты, но обычно они будут читаться позже другой программой. В главе 13 вы узнаете, как такие эффекты можно использовать для организации взаимодействий между программами.

Поскольку эффект обычно применяется к значению, чистый эффект можно смоделировать в виде особой функции, не возвращающей значения. В Kotlin такие функции представлены типом:

```
(T) -> Unit
```

Такую функцию можно определить с использованием типа `Any`, который является родителем всех типов:

```
val display = { x: Any -> println(x) }
```

Или, что еще лучше, использовать ссылку на функцию:

```
val display: (Any) -> Unit = ::println
```

Но чаще для создания эффектов используются анонимные функции, как в следующем примере:


```
val ri: Result<Int> = ...
val rd: Result<Double> = ri.flatMap(inverse)
rd.map { it * 1.35 }
```

Здесь функция { `it * 2.35` } не имеет имени. Но мы можем дать ей имя, чтобы получить возможность многократного использования:

```
val ri: Result<Int> = ...
val rd: Result<Double> = ri.flatMap(inverse)
val function: (Double) -> Double = { it * 2.35 }
val result = rd.map(function)
```

Для применения эффектов необходимо что-то эквивалентное, например:


```
val ri: Result<Int> = ...
val rd: Result<Double> = ri.flatMap(inverse)
val function: (Double) -> Double = { it * 2.35 }
val result = rd.map(function)
val ef: (Double) -> Unit = ::println
result.map(ef)
```

Догадались? Этот прием работает! Такое возможно, потому что `ef` – это функция, возвращающая `Unit` (эквивалент `void` в Java, точнее – эквивалент `Void`). Предыдущий код можно записать иначе:

```
val ri: Result<Int> = ...
val rd: Result<Double> = ri.flatMap(inverse)
val function: (Double) -> Double = { it * 2.35 }
val result = rd.map(function)
val ef: (Double) -> Unit = ::println
val x: Result<Unit> = result.map(ef)
```

Если требуется только применить эффект, можно использовать этот прием: отобразить эффект с помощью `map` и игнорировать результат. Но есть способ лучше. В главе 7 мы написали функцию `forEach` для класса `Result`, которая получает эффект и применяет его к своему значению. Вот реализация этой функции в подклассе `Empty`:

```
override fun forEach(onSuccess: (Nothing) -> Unit,
                     onFailure: (RuntimeException) -> Unit,
                     onEmpty: () -> Unit) {
    onEmpty()
}
```

В подклассе `Success`:

```
override fun forEach(onSuccess: (A) -> Unit,
                     onFailure: (RuntimeException) -> Unit,
                     onEmpty: () -> Unit) {
```

```
    onSuccess(value)
}
```

И в подклассе `Failure`:

```
override fun forEach(onSuccess: (A) -> Unit,
                     onFailure: (RuntimeException) -> Unit,
                     onEmpty: () -> Unit) {
    onFailure(exception)
}
```

Функция `forEach` принимает три параметра с эффектами: один для значения `Success`, один для `Failure` и один для `Empty`. Кроме того, абстрактное объявление в родительском классе `Result` определяет значение по умолчанию для каждого эффекта:

```
abstract fun forEach(onSuccess: (A) -> Unit = {},
                     onFailure: (RuntimeException) -> Unit = {},
                     onEmpty: () -> Unit = {})
```

Мы не можем написать модульные тесты для этой функции. Чтобы убедиться в ее работоспособности, можно запустить программу, показанную в листинге 12.1, и посмотреть результат на экране. (Можно написать несколько тестов, изменяющих некоторые глобальные переменные или параметры и затем проверяющих эти изменения, но это не модульное тестирование!)

Листинг 12.1 Вывод данных

```
fun main(args: Array<String>) {
    val ra = Result(4) ①
    val rb = Result(0) ①
    val inverse: (Int) -> Result<Double> = { x ->
        when {
            x != 0 -> Result(1.toDouble() / x)
            else -> Result.failure("Division by 0")
        }
    }
    val showResult: (Double) -> Unit = ::println
    val showError: (RuntimeException) -> Unit =
        { println("Error - ${it.message}")}

    val rt1 = ra.flatMap(inverse)
    val rt2 = rb.flatMap(inverse)

    print("Inverse of 4: ")
    rt1.forEach(showResult, showError) ②

    System.out.print("Inverse of 0: ")
    rt2.forEach(showResult, showError) ③
}
```

① Имитация данных, возвращаемых другими функциями, которые могут вызвать ошибку
② Выведет полученный результат
③ Выведет сообщение об ошибке

Эта программа выведет следующий результат:

```
Inverse of 4: 0.25
Inverse of 0: Error - Division by 0
```

Упражнение 12.1

Напишите функцию `forEach` в классе `List`, которая получает эффект и применяет его ко всем элементам списка. Вот ее сигнатура:

```
fun forEach(ef: (A) -> Unit)
```


Решение

Эту функцию можно реализовать в родительском классе `List` или объявить ее как абстрактную и добавить конкретные реализации в оба подкласса. В отличие от `Result.Empty` нет никакой причины применять эффект, если список пуст (хотя это возможно, если необходимо для вашего варианта использования). Поэтому реализация в `Nil` будет иметь вид:

```
override fun forEach(ef: (Nothing) -> Unit) {}
```

А вот простейшая рекурсивная реализация в классе `Cons`:

```
override fun forEach(ef: (A) -> Unit) {
    ef(head)
    tail.forEach(ef)
}
```

К сожалению, эта реализация может вызвать переполнение стека при работе со списками, насчитывающими больше нескольких тысяч элементов. Чтобы решить проблему, нужно преобразовать эту функцию в сорекурсивную. Для этого достаточно определить сорекурсивную вспомогательную функцию внутри функции `forEach`:

```
override fun forEach(ef: (A) -> Unit) {
    tailrec fun forEach(list: List<A>) {
        when (list) {
            Nil -> {}
            is Cons -> {
                ef(list.head)
                forEach(list.tail)
            }
        }
    }
    forEach(this)
}
```

12.2 Чтение данных

До сих пор мы заостряли свое внимание только на выводе. Обычно вывод данных производится в конце программы после вычисления результатов. Это позволяет написать большую часть программы без эффектов, используя все преимущества парадигмы функционального программи-

рования. В этом случае нефункциональной оказывается только часть программы, осуществляющая вывод.

Но мы не рассматривали, как исходные данные попадают в программу. Именно этим мы и займемся сейчас. Позднее вы увидите более функциональный способ ввода данных, но прежде посмотрим, как сделать это чисто и аккуратно (пусть и императивно), чтобы такой способ ввода хорошо сочетался с остальными функциональными компонентами программы.

12.2.1 Чтение данных с клавиатуры

Чтение данных с клавиатуры должно производиться так, чтобы сохранилась возможность тестирования и программа оставалась детерминированной. Для начала рассмотрим пример ввода целых чисел и строк. В листинге 12.2 показан интерфейс, который мы должны реализовать.

Листинг 12.2 Интерфейс для ввода данных

```
interface Input: Closeable { ①

    fun readString(): Result<Pair<String, Input>> ②

    fun readInt(): Result<Pair<Int, Input>> ②

    fun readString(message: String): Result<Pair<String, Input>> =
        readString() ③

    fun readInt(message: String): Result<Pair<Int, Input>> =
        readInt() ③
}
```

① Наследует интерфейс `Closeable`
② Читают строки и целые числа соответственно
③ Принимают параметр `message` с сообщением

Наследование интерфейса `Closeable` в листинге 12.2 может пригодиться для автоматического закрытия ресурса. Возможность передачи сообщения в параметре можно использовать для вывода строки приглашения.

Но представленные здесь реализации по умолчанию игнорируют сообщения. Обратите внимание, что функции возвращают `Result<Pair<String, Input>>`, а не `Result<String>`, что позволяет прозрачно комбинировать функции.

Мы могли бы сразу написать конкретную реализацию для этого интерфейса, но сначала напишем абстрактную (потому что впоследствии может потребоваться возможность прочитать данные из какого-то другого источника, такого как файл). Поместим общий код в абстрактный класс и расширим его для каждого конкретного источника ввода. Эта реализация показана в листинге 12.3.

Листинг 12.3 Реализация AbstractReader

```
abstract class AbstractReader (
        private val reader: BufferedReader): Input { ①
```

```
    override fun readString():
            Result<Pair<String, Input>> = try { ②
        reader.readLine().let {
            when {
                it.isEmpty() -> Result()
                else         -> Result(Pair(it, this))
            }
        }
    } catch (e: Exception) {
        Result.failure(e)
    }

    override fun readInt(): Result<Pair<Int, Input>> = try {
        reader.readLine().let {
            when {
                it.isEmpty() -> Result()
                else         -> Result(Pair(it.toInt(), this))
            }
        }
    } catch (e: Exception) {
        Result.failure(e)
    }

    override fun close(): Unit = reader.close() ③
}
```

① Объявляет класс, принимающий объект, который выполняет чтение, что позволит использовать его для чтения данных из разных источников
② Читает строку из объекта `reader` и возвращает `Result.Empty`, если строка пустая; `Result.Success`, если прочитаны какие-то данные; и `Result.Failure`, если возникла какая-то ошибка
③ Делегирует закрытие ресурса функции `close` объекта `BufferedReader`

Теперь реализуем конкретный класс, осуществляющий чтение с клавиатуры (листинг 12.4). Этот класс отвечает за инициализацию объекта чтения `reader`. Также он переопределяет две функции из интерфейса, чтобы вывести приглашение к вводу.

Листинг 12.4 Реализация ConsoleReader

```
import com.fpinkotlin.common.Result

import java.io.BufferedReader
import java.io.InputStreamReader

class ConsoleReader(reader: BufferedReader): AbstractReader(reader) {

    override
    fun readString(message: String): Result<Pair<String, Input>> { ①
        print("$message ")
        return readString()
    }

    override
    fun readInt(message: String): Result<Pair<Int, Input>> { ①
        print("$message ")
```

```
        return readInt()
    }

    companion object {
        operator fun invoke(): ConsoleReader =
            ConsoleReader(BufferedReader(
                InputStreamReader(System.`in`))) ②
    }
}
```

① Переопределенные версии двух функций по умолчанию, которые выводят приглашение к вводу
② Для ссылки на поле `in` в Java-классе `System` следует использовать обратные апострофы, потому что имя `in` – это зарезервированное слово в Kotlin

Теперь можно использовать класс `ConsoleReader` и написать законченную программу, осуществляющую ввод и вывод. Эта программа показана в листинге 12.5.

Листинг 12.5 Законченная программа, осуществляющая ввод и вывод

```
fun main(args: Array<String>) {

    val input = ConsoleReader() ①

    val rString = input.readString("Enter your name:") ②
                                   .map { t -> t.first }

    val nameMessage = rString.map { "Hello, $it!" } ③

    nameMessage.forEach(::println, onFailure =
                            { println(it.message)}) ④

    val rInt = input.readInt("Enter your age:").map { t -> t.first }

    val ageMessage = rInt.map { "You look younger than $it!" }

    ageMessage.forEach(::println, onFailure = ⑤
        { println("Invalid age. Please enter an integer")})
}
```

① Создание объекта чтения
② Вызов `readString` с приглашением к вводу и преобразование результата `Result<Tuple<String, Input>>` в `Result<String>`
③ Бизнес-логика программы (что программа должна делать, с точки зрения пользователя). Эта часть может быть чисто функциональной
④ Применяется шаблон, описанный в предыдущем разделе для вывода результата или сообщения об ошибке
⑤ Выводит сообщение, отличное от указанного в исключении

Обратите внимание, что в эффекте `ageMessage.forEach` нет никакой возможности сослаться на входное значение. Чтобы получить такую возможность, вам придется использовать какой-то другой контекст с проверкой, отличный от `Result`.

Эта программа совершенно не впечатляет. Она является эквивалентом вездесущей программы «Hello», которая в большинстве руководств

по программированию обычно служит вторым примером (сразу после «Hello, world»). Конечно, это всего лишь пример. Но обратите внимание, как легко можно превратить эту программу в нечто более полезное.

Упражнение 12.2

Напишите программу, которая в цикле предлагает пользователю ввести целое число, имя и фамилию и потом выводит список людей в консоль. Ввод данных завершается, когда пользователь введет пустое значение вместо целого числа.

Подсказка

Для хранения каждой введенной записи вам потребуется класс. Используйте следующий класс данных `Person`:

```
data class Person (val id: Int, val firstName: String, val lastName: String)
```

Реализуйте решение в функции `main` на уровне пакета. Используйте функцию `Stream.unfold`, чтобы создать поток записей. Возможно, вам будет проще создать отдельную функцию для ввода данных об одном человеке и использовать ссылку на эту функцию как аргумент в вызове `unfold`. Такая функция могла бы иметь следующую сигнатуру:

```
fun person(input: Input): Result<Pair<Person, Input>>
```

Решение

Это упражнение решается просто. При наличии отдельной функции для ввода данных об одном человеке можно создать поток персон и вывести результаты, как показано ниже (эта реализация игнорирует любые ошибки и не заботится о закрытии ресурсов):

```
import com.fpinkotlin.common.List
import com.fpinkotlin.common.Stream

fun readPersonsFromConsole(): List<Person> =
    Stream.unfold(ConsoleReader(), ::person).toList()

fun main(args: Array<String>) {
    readPersonsFromConsole().forEach(::println)
}
```

Теперь осталось только написать функцию ввода данных об одном человеке. Эта функция предлагает ввести числовой идентификатор, имя и фамилию и создает три экземпляра `Result`, которые можно объединить с помощью шаблона включения, с которым мы познакомились в главе 7:

```
fun person(input: Input): Result<Pair<Person, Input>> =
    input.readInt("Enter ID:").flatMap { id ->
        id.second.readString("Enter first name:")
            .flatMap { firstName ->
                firstName.second.readString("Enter last name:")
                    .map { lastName ->
                        Pair(Person(id.first,
```

```
                                firstName.first,
                                lastName.first), lastName.second)
                }
            }
        }
```

Шаблон включения – это, пожалуй, один из самых важных шаблонов в функциональном программировании, поэтому вам обязательно следует овладеть им. В других языках, таких как Scala и Haskell, есть даже синтаксический сахар, упрощающий его реализацию, но, к сожалению, в Kotlin его нет. В псевдокоде этот шаблон можно выразить так:

```
for {
    id in input.readInt("Enter ID:")
    firstName in id.second.readString("Enter first name:")
    lastName in firstName.second.readString("Enter last name:")
} return Pair(Person(id.first, firstName.first, lastName.first),
              lastName.second))
```

Но отсутствие синтаксического сахара не может служить основанием для отказа от этого шаблона. Идиома `flatMap` в первое время будет казаться сложнее в освоении, но она наглядно иллюстрирует происходящее. Многим программистам этот шаблон известен в виде:

```
a.flatMap { b ->
    flatMap { c ->
        map { d ->
            getSomething(a, b, c, d)
        }
    }
}
```

Они часто думают, что он всегда реализуется как последовательность вызовов `flatMap`, заканчивающаяся вызовом `map`. Однако это не так. Будет ли конечный вызов вызовом `map` или `flatMap` – зависит исключительно от типа возвращаемого значения. Часто последняя функция (здесь `getSomething`) возвращает голое значение. Вот почему шаблон заканчивается вызовом `map`. Но если `getSomething` вернет `Result`, шаблон будет выглядеть следующим образом:

```
a.flatMap { b ->
    flatMap { c ->
        flatMap { d ->
            getSomething(a, b, c, d)
        }
    }
}
```

12.2.2 Чтение из файла

Программу, созданную выше, легко адаптировать для чтения файлов. Класс `FileReader` похож на `ConsoleReader`. Единственное отличие – функция вызова `invoke` в объекте-компаньоне должна обрабатывать исключение,

которое может возникнуть при создании `BufferedReader`, и поэтому возвращать `Result<Input>` вместо голого значения, как показано в листинге 2.6.

Листинг 12.6 Реализация FileReader

```
class
FileReader private constructor(private val reader: BufferedReader) :
        AbstractReader(reader), AutoCloseable {

    override fun close() {
        reader.close()
    }

    companion object {

        operator fun invoke(path: String): Result<Input> = try {
            Result(FileReader(File(path).bufferedReader()))
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

Упражнение 12.3

Напишите программу `ReadFile`, похожую на `ReadConsole`, но читающую данные из файла с записями, каждая из которых находится в отдельной строке. Пример файла можно взять в примерах, сопровождающих эту книгу (http://github.com/pysaumont/fpinkotlin).

Подсказка

Несмотря на сходство с программой `ReadConsole`, вам придется учесть, что функция `invoke` возвращает `Result`. Попробуйте использовать одну и ту же функцию `person`. Также обратите внимание, что нужно будет позаботиться о закрытии ресурсов. Для этого следует использовать функцию `use` из стандартной библиотеки Kotlin.

Решение

Обратите внимание, как в следующем решении используется функция `use`, чтобы гарантировать корректное закрытие файла в любом случае:

```
fun readPersonsFromFile(path: String): Result<List<Person>> =
    FileReader(path).map {
        it.use {
            Stream.unfold(it, ::person).toList()
        }
    }

fun main(args: Array<String>) {
    val path = "<path>/data.txt"
    readPersonsFromFile(path).forEach({ list: List<Person> ->
        list.forEach(::println)
    }, onFailure = ::println)
}
```

Ссылка it используется дважды в функции readPersonsFromFile и каждый раз представляет параметр текущего лямбда-выражения. Если вас это смущает, можете использовать конкретные имена, например:

```
fun readPersonsFromFile(path: String): Result<List<Person>> =
    FileReader(path).map { input1 ->
        input1.use { input2 ->
            Stream.unfold(input2, ::person).toList()
        }
    }
```

Но это особый случай – оба имени представляют один и тот же объект.

12.3 Тестирование программ с вводом

Одним из преимуществ показанного подхода является простота тестирования программ. Такие программы можно тестировать, использовав файлы вместо пользовательского ввода с клавиатуры, а кроме того, такую программу легко связать с другой программой, которая генерирует сценарий с командами ввода. В листинге 12.7 показан пример класса ScriptReader, который можно использовать для тестирования.

Листинг 12.7 Класс ScriptReader, позволяющий использовать список команд ввода

```
class ScriptReader : Input {

    constructor(commands: List<String>) : super() { ①
        this.commands = commands
    }

    constructor(vararg commands: String) : super() { ②
        this.commands = List(*commands)
    }

    private val commands: List<String>

    override fun close() { }

    override fun readString(): Result<Pair<String, Input>> = when {
        commands.isEmpty() ->
            Result.failure("Not enough entries in script")
        else -> Result(Pair(commands.headSafe().getOrElse(""),
            ScriptReader(commands.drop(1))))
    }

    override fun readInt(): Result<Pair<Int, Input>> = try {
        when {
            commands.isEmpty() ->
                Result.failure("Not enough entries in script")
            Integer.parseInt(commands.headSafe().getOrElse("")) >= 0 ->
                Result(Pair(Integer.parseInt(
                    commands.headSafe().getOrElse("")),
                    ScriptReader(commands.drop(1))))
```

```
            else -> Result()
        }
    } catch (e: Exception) {
        Result.failure(e)
    }
}
```

① Экземпляр ScriptReader можно создать со списком команд...
② ...или с аргументом vararg

В листинге 12.8 показан пример использования класса ScriptReader. Примеры модульного тестирования вы найдете в коде, сопровождающем эту книгу.

Листинг 12.8 Использование ScriptReader для ввода данных

```
fun readPersonsFromScript(vararg commands: String): List<Person> =
    Stream.unfold(ScriptReader(*commands), ::person).toList()

fun main(args: Array<String>) {
    readPersonsFromScript("1", "Mickey", "Mouse",
                          "2", "Minnie", "Mouse",
                          "3", "Donald", "Duck").forEach(::println)
}
```


12.4 Полностью функциональный ввод/вывод

Большинству программистов на Kotlin вполне достаточно того, что мы узнали к данному моменту. Разделения программы на функциональную и нефункциональную части является необходимым, а также достаточным. Но было бы интересно посмотреть, как можно сделать программы на Kotlin еще более функциональными.

Использовать или не использовать следующие методы в программах на Kotlin – решать только вам. Иногда дополнительное усложнение может не стоить затраченных усилий, но все же полезно знать и понимать, как работают эти методы, чтобы вы могли сделать осознанный выбор.

12.4.1 Как сделать ввод/вывод полностью функциональным

Есть несколько ответов на вопрос в заголовке этого раздела. Самый короткий: это невозможно. Согласно определению парадигмы функционального программирования, функциональная программа не производит никаких других наблюдаемых эффектов, кроме возврата значения, т. е. она не может вводить и выводить данные.

Но многим программам не требуется ничего вводить или выводить. В эту категорию попадают некоторые библиотеки. Библиотеки – это программы, предназначенные для использования другими программами. Они получают значения в аргументах и возвращают результаты, полученные в ходе вычислений с использованием аргументов. В первых двух разделах этой главы вы разделили свои программы на три части:

одна выполняет ввод, вторая – вывод, и третья действует как библиотека и полностью функциональна.

Другой способ решить проблему – написать библиотечную часть, которая производит в качестве окончательного возвращаемого значения другую (нефункциональную) программу, обрабатывающую весь ввод и вывод. Это сродни идее ленивых вычислений. Вы можете обрабатывать ввод/вывод, поскольку то, что происходит позже в отдельной программе, будет возвращаемым значением нашей чисто функциональной программы.

12.4.2 Реализация чисто функционального ввода/вывода

В этом разделе вы увидите, как реализовать чисто функциональный ввод/вывод. Начнем с вывода. Представьте, что нам нужно вывести приветственное сообщение в консоль. Вместо такого кода:

```
fun sayHello(name: String) = println("Hello, $name!")
```

мы можем написать функцию `sayHello`, возвращающую программу, которая имеет тот же самый эффект:

```
fun sayHello(name: String): () -> Unit = { println("Hello, $name!") }
```

Вот как можно использовать такую функцию:

```
fun main(args: Array<String>) {
    val program = sayHello("Georges")
}
```

Этот чисто функциональный код. Он не производит никаких видимых эффектов, и это правда. Он создает программу, которую можно запустить, чтобы получить желаемый эффект. Эту программу можно запустить, вычислив ее аргумент. Результатом является нефункциональная программа, но для нас это неважно. Основная программа осталась функциональной.

Как я уже говорил несколько раз, лучший способ сделать программу безопасной – отделить функциональные части от эффектов. Методика, показанная здесь, не самая простая в реализации, но, безусловно, является исчерпывающим решением в смысле отделения функций от эффектов.

Не является ли это самообманом? Нет. Поразмышляйте над программой, написанной на любом функциональном языке. В конечном итоге она компилируется в выполняемую программу, которая абсолютно нефункциональна и может быть запущена на вашем компьютере. Здесь вы делаете то же самое, за исключением того, что генерируемая вами программа выглядит как написанная на Kotlin. На самом деле это не так. Она написана на некотором предметно-ориентированном языке (Domain Specific Language, DSL), который определяет ваша программа. Чтобы выполнить эту программу, вы можете написать:

```
program()
```

В этом примере созданная программа имеет тип `() -> Unit`. Она работает, но было бы интересно сделать нечто большее с результатом, чем просто вычислить его. Например, можно объединить несколько таких результатов разными способами. Для этого нам понадобится нечто более мощное, поэтому давайте создадим новый тип с именем `IO`. Начнем с единственной функции `invoke`. На этом этапе он не сильно отличается от `() -> Unit`:

```
class IO(private val f: () -> Unit) {
    operator fun invoke() = f()
}
```

Допустим, у нас есть еще три функции:

```
fun show(message: String): IO = IO { println(message) }

fun <A> toString(rd: Result<A>): String =
    rd.map { it.toString() }.getOrElse { rd.toString() }

fun inverse(i: Int): Result<Double> = when (i) {
    0 -> Result.failure("Div by 0")
    else -> Result(1.0 / i)
}
```

Мы можем написать следующую чисто функциональную программу:

```
val computation: IO = show(toString(inverse(3)))
```

Эта программа производит другую программу, которую позднее можно выполнить инструкцией:

```
computation()
```

12.4.3 Комбинирование ввода/вывода

Используя новый интерфейс `IO`, можно сконструировать любую программу, действующую как единое целое. Было бы интересно иметь возможность комбинировать такие программы. Самая простая комбинация – группировка двух программ в одну. Именно ее вы реализуете в следующем упражнении.

Упражнение 12.4

Напишите функцию в классе `IO`, объединяющую два экземпляра `IO` в один. Дайте этой функции имя `plus`. Вот ее сигнатура:

```
operator fun plus(io: IO): IO
```

Решение

Идея состоит в том, чтобы вернуть новый экземпляр `IO` с реализацией `run`, которая сначала выполняет текущий экземпляр `IO`, а затем – экземпляр в аргументе:

```
operator fun plus(io: IO): IO = IO {
    f()
    io.f()
}
```

Позднее нам понадобится «ничего не делающий» IO, который будет служить нейтральным элементом в некоторых способах комбинирования экземпляров IO. Мы легко можем создать его в объекте-компаньоне:

```
companion object {
    val empty: IO = IO {}
}
```

Используя эти новые функции, можно создавать более сложные программы, комбинируя экземпляры IO:

```
fun getName() = "Mickey"

val instruction1 = IO { print("Hello, ") } ①
val instruction2 = IO { print(getName()) } ①
val instruction3 = IO { print("!\n") } ①

val script: IO = instruction1
                    + instruction2
                    + instruction3 ②
script() ③
```

① Эти три строки ничего не выводят. Это всего лишь DSL-подобные инструкции
② Объединяет три инструкции и создает программу
③ Выполняет программу

Если хотите, можете использовать вызовы функций вместо операторов:

```
instruction1.plus(instruction2).plus(instruction3)()
```

Аналогично можно создать программу из списка инструкций:

```
val script = List(
    IO { print("Hello, ") },
    IO { print(getName()) },
    IO { print("!\n") }
)
```

Не напоминает ли этот код императивную программу? Да, так и есть. Чтобы выполнить его, мы должны сначала скомпилировать его в один IO. Сделать это можно с помощью правой свертки:

```
val program: IO = script.foldRight(IO.empty) { io -> { io + it } }
```

или левой свертки:

```
val program: IO = script.foldLeft(IO.empty) { acc -> { acc + it } }
```

Теперь должно быть понятно, зачем нам понадобилась «ничего не делающая» реализация. Наконец, мы можем запустить программу как обычно:

```
program()
```

Имейте в виду, что левая свертка помещает единичный (пустой) IO в первую позицию, а правая свертка – в последнюю. Выбор позиции для IO.empty не имеет значения. Но при использовании в комбинации

с каким-то другим IO (например, выполняющим некоторую задачу инициализации), его, вероятно, нужно выполнить первым, поэтому следует использовать левую свертку, а правую – для выполнения некоторых завершающих задач.

12.4.4 Обработка ввода с IO

К данному моменту мы использовали наш класс IO только для организации вывода. Чтобы с его помощью организовать ввод, необходимо параметризовать этот класс типом входного значения. Вот новый параметризованный тип IO:


```
class IO<out A>(private val f: () -> A) { ①

    operator fun invoke() = f()

    companion object {
        val empty: IO<Unit> = IO { } ②

        operator fun <A> invoke(a: A): IO<A> = IO { a } ③
    }
}
```


① **Параметризованный класс IO и функция, которая конструирует и возвращает экземпляр указанного типа**

② **Пустой экземпляр параметризован типом Unit и создается с помощью функции, которая ничего не возвращает (имейте в виду, что это не то же самое, что возврат Nothing)**

③ **Функция invoke объекта-компаньона принимает голое значение и возвращает его в контексте IO**

Как видите, интерфейс IO создает контекст для вычислений, как это делают Option, Result, List, Stream, Lazy и т. д. Он также имеет функцию, возвращающую пустой экземпляр, и функцию, которая помещает голое значение в контекст.

Теперь для вычислений с использованием значений IO нам нужны такие функции, как map и flatMap, чтобы получить возможность связывать другие функции с контекстом IO. Но, чтобы упростить тестирование этих функций, прежде определим объект, представляющий консоль компьютера.

Упражнение 12.5

Определите объект Console с тремя функциями:

```
object Console {

    fun readln(): IO<String> = TODO("")

    fun println(o: Any): IO<Unit> = TODO("")

    fun print(o: Any): IO<Unit> = TODO("")
}
```

Решение

Функции print и println вызывают эквивалентные функции Kotlin, например:

```
fun println(o: Any): IO<Unit> = IO {
    kotlin.io.println(o.toString())
}

fun print(o: Any): IO<Unit> = IO {
    kotlin.io.print(o.toString())
}
```

Мы должны использовать полные имена функций, чтобы избежать рекурсивных вызовов функций самого объекта. Впрочем, функциям объекта можно дать любые другие имена.

Функция `readln` вызывает функцию `realLine` экземпляра `BufferedReader`, обертывающего `System.in`. Напомню, что обратные апострофы необходимы, потому что имя `in` – это зарезервированное слово в Kotlin:

```
private val br = BufferedReader(InputStreamReader(System.`in`))

fun readln(): IO<String> = IO {
    try {
        br.readLine()
    } catch (e: IOException) {
        throw IllegalStateException(e)
    }
}
```

Если что-то пойдет не так, можно просто возбудить исключение. Вы можете попробовать решить проблему, использовав более функциональный подход.

На данном этапе объект `Console` ничего не дает, кроме дополнительной сложности. Чтобы получить возможность создавать экземпляры `IO`, нужно добавить функцию `map`.

Упражнение 12.6

Напишите функцию `map` в `IO<A>`, которая принимает функцию `(A) -> B` и возвращает `IO<B>`.

Решение

Вот реализация, которая применяет функцию к значению `this` и возвращает результат в новом контексте `IO`:

```
fun <B> map (g: (A) -> B): IO<B> = IO {
    g(this())
}
```

А вот так можно использовать эту функцию:

```
fun main(args: Array<String>) {
    val script = sayHello()
    script()
}

private fun sayHello(): IO<Unit> = Console.print("Enter your name: ")
    .map { Console.readln()() }
```

```
    .map { buildMessage(it) }
    .map { Console.println(it)() }

private fun buildMessage(name: String): String = "Hello, $name!"
```

Упражнение 12.7

Многократные вызовы IO в виде Console.readln()() и Console.println(it)() выглядят довольно громоздко. Это необходимо, потому что функции readln и println возвращают экземпляры IO, а не фактические значения. Напишите функцию flatMap, абстрагирующую этот процесс. Она должна принимать функцию от (A) -> IO<B> и возвращать IO<B>.

Решение

Решение очевидно. Нужно вызвать экземпляр IO<IO<B>>, который возвращает функция map, чтобы преобразовать его в IO<B>:

```
fun <B> flatMap (g: (A) -> IO<B>): IO<B> = IO {
    g(this())()
}
```

Как видите, здесь имеет место рекурсия. Пока это не является проблемой, потому что выполняется только один шаг рекурсии, но это может стать проблемой при попытке объединить в цепочку огромное количество вызовов flatMap. Теперь можно скомбинировать ввод/вывод функциональным способом:

```
fun main(args: Array<String>) {
    val script = sayHello()
    script()
}

private fun sayHello(): IO<Unit> = Console.print("Enter your name: ")
    .flatMap { Console.readln() }
    .map { buildMessage(it) }
    .flatMap { Console.println(it) }

private fun buildMessage(name: String): String = "Hello, $name!"
```

Функция sayHello абсолютно безопасна. Он никогда не сгенерирует исключения IOException, потому что не выполняет никаких операций ввода/вывода. Он просто возвращает программу, которая выполняет эти операции после запуска путем вызова возвращаемого значения (script()). Исключение может возникнуть только в этом вызове.

12.4.5 Расширение типа IO

С помощью типа IO можно создавать нефункциональные программы (программы с эффектами) чисто функциональным способом. Но на данном этапе эти программы могут только читать и выводить в элемент, такой как класс Console. Мы можем расширить свой предметный язык DSL, добавив инструкции для создания управляющих структур, таких как циклы и условные выражения.

Для начала реализуем цикл, аналогичный циклу `for`, в виде функции `repeat`, которая принимает число итераций и экземпляр `IO` для повторения.

Упражнение 12.8

Напишите функцию `repeat` в объекте-компаньоне `IO` со следующей сигнатурой:

```
fun <A> repeat(n: Int, io: IO<A> ): IO<List<A>>
```

Подсказка

Чтобы упростить код в классе `IO`, добавьте в объект-компаньон `Stream` вспомогательную функцию `fill`. Эта функция должна создавать поток из `n` невычисляемых элементов:

```
fun <A> fill(n: Int, elem: Lazy<A>): Stream<A> {
    tailrec
    fun <A> fill(acc: Stream<A>, n: Int, elem: Lazy<A>): Stream<A> =
        when {
            n <= 0 -> acc
            else -> fill(Cons(elem, Lazy { acc }), n - 1, elem)
        }
    return fill(Empty, n, elem)
}
```


Создайте коллекцию `IO`, представляющую каждую итерацию, а затем выполните свертку этой коллекции, чтобы объединить экземпляры `IO`. Для этого вам понадобится нечто более мощное, чем функция `plus`. Начнем с реализации функции `map2` со следующей сигнатурой:

```
fun <A, B, C> map2(ioa: IO<A>, iob: IO<B>, f: (A) -> (B) -> C): IO<C>
```

Решение

Вот как можно реализовать функцию `map2`:

```
fun <A, B, C> map2(ioa: IO<A>, iob: IO<B>, f: (A) -> (B) -> C): IO<C> =
    ioa.flatMap { a ->
        iob.map { b ->
            f(a)(b)
        }
    }
```

Это пример применения вездесущего шаблона включения. Благодаря этой функции мы легко можем реализовать `repeat`:

```
fun <A> repeat(n: Int, io: IO<A> ): IO<List<A>> =
    Stream.fill(n, Lazy { io })
        .foldRight( Lazy { IO { List<A>() } }) { ioa ->
            { sioLa ->
                map2(ioa, sioLa()) { a ->
                    { la: List<A> -> cons(a, la) }
                }
            }
        }
```

Выглядит немного замысловато, но отчасти это обусловлено необходимостью обертывания строки, предназначенной для вывода, а отчасти потому, что она написана в виде однострочной версии для оптимизации. Вот эквивалентная реализация:

```
fun <A> repeat(n: Int, io: IO<A> ): IO<List<A>> {
    val stream: Stream<IO<A>> = Stream.fill(n, Lazy { io })
    val f: (A) -> (List<A>) -> List<A> =
        { a ->
            { la: List<A> -> cons(a, la) }
        }
    val g: (IO<A>) -> (Lazy<IO<List<A>>>) -> IO<List<A>> =
        { ioa ->
            { sioLa ->
                map2(ioa, sioLa(), f)
            }
        }
    val z: Lazy<IO<List<A>>> = Lazy { IO { List<A>() } }
    return stream.foldRight(z, g)
}
```

СОВЕТ Если вы используете интегрированную среду разработки, то без труда найдете типы. Например, чтобы узнать тип в IntelliJ, нужно навести указатель мыши на ссылку, удерживая нажатой клавишу **Ctrl**.

С помощью этих функций можно написать, например, такой код:

```
val program = IO.repeat(3, sayHello())
```

Он создаст программу, эквивалентную вызову следующей функции как `sayHello(3)`:

```
fun sayHello(n: Int) {

    val br = BufferedReader(InputStreamReader(System.`in`))

    for (i in 0 until n) {
        print("Enter your name: ")
        val name = br.readLine()
        println(buildMessage(name))
    }
}
```

Однако она имеет одно важное отличие: вызов `sayHello(3)` выполняет эффект трижды и немедленно, а `IO.repeat(3, sayHello())` возвращает (не вычисляя) программу, которая делает то же самое только при вызове функции `invoke`.

Аналогичным образом можно определить много других управляющих структур. Примеры вы найдете в коде, сопровождающем книгу, который доступен по адресу http://github.com/pysaumont/fpinkotlin.

В листинге 12.9 показан пример использования функций `condition` и `doWhile`, которые действуют подобно инструкциям `if` и `while` во многих других языках.

Листинг 12.9 Использование IO для обертывания императивных инструкций

```
private val buildMessage = { name: String ->
    IO.condition(name.isNotEmpty(), Lazy {
        IO("Hello, $name!").flatMap { Console.println(it) }
    })
}

fun program(f: (String) -> IO<Boolean>, title: String): IO<Unit> {
    return IO.sequence(Console.println(title),
        IO.doWhile(Console.readln(), f),
        Console.println("bye!")
    )
}

fun main(args: Array<String>) {
    val program = program(buildMessage,
                    "Enter the names of the persons to welcome: ")
    program()
}
```

Этот пример не означает, что вы должны программировать именно так. Конечно, лучше использовать тип `IO` только для ввода/вывода, выполняя все вычисления в функциональном стиле. Реализация императивного DSL в функциональном коде – не самое эффективное решение, независимо от задачи. Но это упражнение поможет вам понять принцип работы этого подхода.

12.4.6 Добавление в IO защиты от переполнения стека

В предыдущих упражнениях вы, возможно, не заметили, что некоторые функции ввода/вывода использовали стек так же, как рекурсивные функции. Например, функция `repeat` может вызвать переполнение стека, если количество повторений окажется слишком большим. Сколько это «слишком большое число повторений» – зависит от размера стека и насколько он заполнен перед запуском программы, возвращаемой функцией. (На данный момент, я думаю, вы понимаете, что сам вызов функции `repeat` не приведет к переполнению стека; это может произойти только после запуска программы, которую она возвращает.)

Упражнение 12.9

В порядке эксперимента создайте функцию `forever`, которая принимает аргумент типа `IO` и возвращает новый экземпляр `IO`, который выполняет аргумент в бесконечном цикле. Вот сигнатура этой функции в объекте-компаньоне `IO`:

```
fun <A, B> forever(ioa: IO<A>): IO<B>
```

Решение

Это настолько же простая функция, как и бесполезная! Чтобы реализовать ее, нужно сделать сконструированную программу бесконечно ре-

курсивной. Имейте в виду, что сама функция `forever` не должна быть рекурсивной – она должна возвращать рекурсивную программу.

Решение заключается в использовании вспомогательной функции типа `() -> IO` и преобразовании аргумента `IO` функции `forever` с помощью `flatMap` и функции, вызывающей эту вспомогательную функцию:

```
fun <A, B> forever(ioa: IO<A>): IO<B> {
    val t: () -> IO<B> = { forever(ioa) }
    return ioa.flatMap { t() }
}
```

Вот как можно вызвать эту функцию:

```
fun main(args: Array<String>) {
    val program = IO.forever<String, String>(IO { "Hi again!" })
        .flatMap { Console.println(it) }
    program()
}
```

Эта программа переполняет стек после нескольких тысяч итераций и эквивалентна следующему вызову:

```
IO.forever<Unit, String>(Console.println("Hi again!"))()
```

Если вам непонятно, почему он переполняет стек, взгляните на следующий псевдокод (сразу замечу, что он не компилируется!), где переменная `t` в реализации функции `forever` заменяется соответствующим выражением:

```
fun <A, B> forever(ioa: IO<A>): IO<B> {
    return ioa.flatMap { { forever(ioa) }() }
}
```

Теперь заменим рекурсивный вызов соответствующим кодом из реализации функции `forever`:

```
fun <A, B> forever(ioa: IO<A>): IO<B> {
    return ioa.flatMap { { ioa.flatMap { { forever<A, B>(ioa) }() } }() }
}
```

Вы можете продолжить рекурсию до бесконечности. Но обратите внимание, что вызовы `flatMap` являются вложенными, в результате при каждом вызове в стек будет помещаться текущее состояние. Это действительно вызовет переполнение стека после нескольких тысяч итераций. В отличие от императивного кода, где инструкции выполняются одна за другой, функция `flatMap` вызывается рекурсивно.

Чтобы устранить опасность переполнения стека в типе `IO`, можно использовать прием под названием *трамполайнинг* (trampolining). Во-первых, нужно определить три состояния программы:

- `Return` представляет готовый результат, который нужно просто вернуть;
- `Suspend` представляет приостановленное вычисление, перед возобновлением которого требуется применить некоторый эффект;

- `Continue` представляет состояние, оказавшись в котором программа должна перед продолжением выполнить некоторые дополнительные вычисления.

Эти состояния представлены в листинге 12.10 тремя классами.

ПРИМЕЧАНИЕ Листинги с 12.10 по 12.12 – это части одного целого. Они могут использоваться только вместе.

Листинг 12.10 Три класса, необходимые для устранения опасности переполнения стека в IO

```
sealed class IO<out A> { ①

    internal
    class Return<out A>(val value: A): IO<A>() ②

    internal
    class Suspend<out A>(val resume: () -> A): IO<A>() ③

    internal
    class Continue<A, out B>(val sub: IO<A>, ④
                             val f: (A) -> IO<B>): IO<A>() ⑤
```

① Теперь `IO` определен как запечатанный (`sealed`) класс, чтобы предотвратить создание его экземпляров за пределами класса
② Это значение будет возвращаться вычислениями
③ Эта функция без аргументов применяет (побочный) эффект и возвращает значение
④ Этот экземпляр `IO` выполняется первым и производит значение
⑤ Вычисления продолжаются применением этой функции к возвращаемому значению

Также в объемлющий класс `IO` необходимо внести некоторые изменения, как показано в листингах 12.11 и 12.12.

Листинг 12.11 Изменения, устраняющие опасность переполнения стека в IO

```
sealed class IO<out A> { ①

    operator fun invoke(): A = invoke(this) ②

    operator fun invoke(io: IO<@UnsafeVariance A>): A { ③
        tailrec fun invokeHelper(io: IO<A>): A =
            when (io) { ④
                ...
            }
        return invokeHelper(io)
    }

    fun <B> map(f: (A) -> B): IO<B> =
        flatMap { Return(f(it)) } ⑤

    fun <B> flatMap(f: (A) -> IO<B>): IO<B> =
        Continue(this, f) as IO<B> ⑥

    class IORef<A>(private var value: A) {

        fun set(a: A): IO<A> {
```

```
            value = a
            return unit(a)
        }

        fun get(): IO<A> = unit(value)

        fun modify(f: (A) -> A): IO<A> = get().flatMap({ a -> set(f(a)) })
    }

    internal class Return<out A>(val value: A): IO<A>()

    internal class Suspend<out A>(val resume: () -> A): IO<A>()

    internal class Continue<A, out B>(val sub: IO<A>,
                                      val f: (A) -> IO<B>): IO<A>()

    companion object {

        val empty: IO<Unit> = IO.Suspend { Unit } ⑦

        internal fun <A> unit(a: A): IO<A> =
            IO.Suspend { a } ⑧

        // остальная часть класса опущена
    }
}
```

① Теперь IO определен как запечатанный (sealed) класс
② Функция invoke теперь вызывает вспомогательную функцию invoke(this)
③ Функция invoke(this) в свою очередь вызывает функцию invokeHelper, реализующую хвостовую рекурсию
④ Функция invokeHelper показана в листинге 12.12
⑤ Функция map теперь определена в терминах применения flatMap к композиции f и конструктора Return
⑥ Функция flatMap возвращает экземпляр Continue, который приводится к типу IO<B>
⑦ Пустой IO теперь Suspend
⑧ Функция unit возвращает Suspend

Листинг 12.12 Вспомогательная функция invokeHelper

```
tailrec fun invokeHelper(io: IO<A>): A = when (io) {
    is Return  -> io.value ①
    is Suspend -> io.resume() ②
    else       -> {
        val ct = io as Continue<A, A> ③
        val sub = ct.sub
        val f = ct.f
        when (sub) {
            is Return  -> invokeHelper(f(sub.value)) ④
            is Suspend -> invokeHelper(f(sub.resume())) ⑤
            else       -> {
                val ct2 = sub as Continue<A, A> ⑥
                val sub2 = ct2.sub
                val f2 = ct2.f
                invokeHelper(sub2.flatMap { f2(it).flatMap(f) })
            }
        }
    }
}
```

① Если входящий экземпляр IO – это Return, вычисления завершаются
② Если входящий экземпляр IO – это Suspend, перед возвратом значения применяется содержащийся эффект
③ Если входящий экземпляр IO – это Continue, выполняется чтение вложенного экземпляра IO
④ Если sub – это Return, выполняется рекурсивный вызов функции с результатом применения указанной функции
⑤ Если sub – это Suspend, к нему применяется указанная функция, возможно, производящая соответствующий эффект
⑥ Если sub – это Continue, извлекается содержащийся в нем экземпляр it типа IO (sub2) и объединяется с sub в цепочку

Листинг 12.13 демонстрирует, как использовать эту новую безопасную версию IO.

Листинг 12.13 Новая безопасная версия класса Console

```
object Console {

    private val br = BufferedReader(InputStreamReader(System.`in`))

    /**
     * Реализация readLine в виде val-функции
     */
    val readLine2: () -> IO<String> = {
        IO.Suspend {
            try {
                br.readLine()
            } catch (e: IOException) {
                throw IllegalStateException(e)
            }
        }
    }

    /**
     * Более простая реализация readLine. Ссылка на функцию невозможна
     * из-за конфликта имен. Если вам понадобятся ссылки на функции,
     * используйте для fun- и val-функций разные имена
     */
    val readLine = { readLine() }

    /**
     * fun-версия readLine
     */
    fun readLine(): IO<String> = IO.Suspend {
        try {
            br.readLine()
        } catch (e: IOException) {
            throw IllegalStateException(e)
        }
    }

    /**
     * val-версия printLine
     */
    val printLine: (String) -> IO<Unit> = { s: Any ->
```

```
        IO.Suspend {
            println(s)
        }
    }

    /**
     * fun-версия printLine
     */
    fun printLine(s: Any): IO<Unit> = IO.Suspend { println(s) }

    /**
     * Функция print. Обратите внимание на полностью квалифицирванный
     * вызов kotlin.io.print.
     */
    fun print(s: Any): IO<Unit> = IO.Suspend { kotlin.io.print(s) }
}
```

Теперь вы можете использовать `forever` или `doWhile`, не опасаясь переполнения стека. Также можете переписать функцию `repeat`, чтобы сделать ее безопасной для стека. Я не буду показывать здесь новую реализацию, вы найдете ее в примерах кода (http://github.com/pysaumont/fpinkotlin).

Напомню еще раз, что я не могу рекомендовать такой способ написания функциональных программ. Это был лишь пример, как в принципе можно сделать, а не практическая рекомендация. Также обратите внимание, что «в принципе» здесь относится к программированию на Kotlin. В других, более дружественных функциональных языках можно создавать гораздо более мощные программы.

Итоги

- Вместо извлечения значений из `List`, `Result` и других контекстов и применения к ним внешних эффектов, что может привести к ошибкам при отсутствии значений, можно, наоборот, передавать эффекты в эти контексты.
- Применение двух разных эффектов в случае успеха и неудачи можно абстрагировать внутри типа `Result`.
- Чтение из файлов выполняется точно так же, как чтение ввода с клавиатуры или из памяти через абстракцию `Reader`.
- Ввод/вывод можно реализовать более функциональным способом с помощью типа `IO`.
- Тип `IO` можно расширить до более общего типа, который позволяет функционально выполнять любые императивные задачи путем создания программы, выполняемой позже.
- Тип `IO` можно сделать безопасным для стека, использовав примем, известный как «трамполайнинг» (trampolining).

13 Общее изменяемое состояние и акторы

Эта глава охватывает следующие темы:

- обзор модели акторов;
- асинхронный обмен сообщениями;
- реализация инфраструктуры акторов;
- включение акторов в работу;
- оптимизация акторов.

В предыдущих главах вы познакомились с множеством приемов, помогающих писать безопасные программы. Большинство из этих приемов заимствовано из функционального программирования. Один из них заключается в использовании неизменяемых данных, чтобы избежать мутаций состояния. Программы без изменяемого состояния безопаснее, надежнее, проще в разработке и масштабировании.

Вы узнали, как можно обрабатывать изменяемые состояния функциональным образом, передавая состояние функциям через аргументы. Вы увидели несколько примеров реализации этого приема, научились генерировать потоки данных, передающие состояние генератора вместе с каждым новым значением. (Если вы этого не помните, загляните в упражнение 9.29, где мы реализовали функцию `unfold`, которая принимает каждое сгенерированное значение с новым состоянием генератора.) В главе 12 вы узнали, как передавать консоль в виде параметра, чтобы отправить текст на экран и получить ввод с клавиатуры. Эти методы могут широко применяться во многих областях. Однако многие

думают, что методы функционального программирования помогают обезопасить использование общего изменяемого состояния, а это в корне неверно.

Использование неизменяемых структур данных, например, не помогает при совместном использовании изменяемого состояния. Они предотвращают непреднамеренное совместное использование изменяемого состояния, устраняя мутации этого состояния. Передача состояния в функции в виде параметров и возврат нового (неизменяемого) состояния в составе результата (в форме пары, содержащей сам результат и новое состояние) прекрасно подходит для работы в однопоточной среде. Но когда возникает потребность обмениваться мутациями состояний между потоками выполнения, что далеко не редкость в современных приложениях, неизменяемые структуры данных оказываются бессильными. Для обмена данными такого типа необходима изменяемая ссылка на них, чтобы новые неизменяемые данные могли заменить предыдущие.

Представьте, что вам нужно подсчитать количество вызовов некоторой функции. В однопоточном приложении для этого достаточно увеличить счетчик, передаваемый функции в виде аргумента, и вернуть его как часть результата. Но большинство императивных программистов предпочитают наращивать счетчик как побочный эффект. Это решение не вызывает проблем, поскольку существует только один поток выполнения и для предотвращения одновременного доступа к счетчику не требуется использовать сложные механизмы синхронизации. Это напоминает жизнь на необитаемом острове. Если вы единственный житель, вам не нужны запоры на дверях. Но как в многопоточной программе безопасно увеличить счетчик и избежать одновременного доступа из нескольких потоков? Обычно для этой цели используются блокировки или атомарные операции или и то, и другое.

В функциональном программировании совместное использование ресурсов должно быть реализовано как эффект, т. е., каждый раз обращаясь к общему ресурсу, вы вынуждены оставить функциональную безопасность и обращаться с этим ресурсом, как мы делали это с операциями ввода/вывода в главе 12. Значит ли это, что мы вынуждены использовать блокировки и механизмы синхронизации при каждом обращении к общему изменяемому состоянию? Конечно, нет!

Как вы узнали в предыдущих главах, функциональное программирование также подразумевает максимальное расширение абстракций. Использование общего изменяемого состояния тоже можно абстрагировать так, чтобы получить возможность использовать его, не заботясь о деталях. Один из способов, позволяющих добиться этого, заключается в применении инфраструктуры акторов.

В этой главе мы не будем разрабатывать полновесную инфраструктуру акторов. Это требует настолько много усилий, что вам лучше использовать уже существующую. Здесь мы создадим минимальную инфраструктуру, которая поможет нам понять, что такие инфраструктуры привносят в функциональное программирование.

13.1 Модель акторов

Согласно модели акторов, многопоточное приложение делится на однопоточные компоненты, которые называют *акторами*. Так как каждый актор является однопоточным, ему не нужно использовать блокировки или синхронизацию для обмена данными.

Акторы взаимодействуют между собой посредством эффектов. Это означает, что акторы полагаются на механизм сериализации получаемых сообщений. (В данном случае под *сериализацией* подразумевается последовательная обработка сообщений. Не путайте с сериализацией объектов – операцией преобразования объектов в последовательное представление.) Благодаря этому механизму акторы могут обрабатывать сообщения по одному, не беспокоясь о возможности параллельного доступа к своим внутренним ресурсам. В результате систему акторов можно рассматривать как серию функциональных программ, взаимодействующих друг с другом посредством эффектов. Каждый актор является однопоточным, поэтому одновременного доступа к внутренним ресурсам не происходит. Параллельность абстрагируется внутри инфраструктуры.

13.1.1 Асинхронный обмен сообщениями

Обрабатывая сообщения, акторы могут отправлять сообщения другим акторам. Сообщения отправляются *асинхронно*, т. е. актору не нужно ждать ответа – его просто нет. Сразу после отправки сообщения отправитель может продолжить выполнять свою работу, которая в основном состоит из обработки получаемых сообщений по очереди. Обработка очереди сообщений означает, что должен поддерживаться некоторый параллельный доступ к очереди. Но это управление абстрагируется в инфраструктуре акторов, поэтому вам, как программисту, не придется беспокоиться об этом.

В некоторых случаях может понадобиться дождаться ответа на сообщение. Предположим, что актор выполняет длительные вычисления. Клиент может воспользоваться преимуществами асинхронных взаимодействий, послать сообщение актору, и пока тот обрабатывает его, продолжить выполнять свою работу. Но клиент должен иметь возможность получить результат, как только актор завершит вычисления. Для этого актор должен выполнить обратный вызов клиента и отправить ему результаты, причем снова в асинхронном режиме. Обратите внимание, что клиент может быть отправителем исходного сообщения, но это совершенно не обязательно.

13.1.2 Параллельное выполнение

Модель акторов позволяет распараллеливать задачи с помощью актора-диспетчера, отвечающего за деление задачи на подзадачи и распределение их между действующими акторами. Каждый раз, когда актор-обработчик возвращает результат диспетчеру, ему дается новая подзадача. Эта модель обладает важным преимуществом перед другими моделями параллельного выполнения: ни один актор-обработчик не будет проста-

ивать, пока список подзадач не опустеет. Однако у нее есть и свой недостаток: актор-диспетчер не участвует в вычислениях. Но обычно это не оказывает существенного влияния на общую производительность.

В некоторых случаях результаты выполнения подзадач требуется переупорядочить после получения. В таком случае актор-диспетчер, вероятно, отправит результаты специальному актору, ответственному за эту работу. Вы увидите пример такой организации работы в разделе 13.3. В небольших программах эту задачу может решать сам диспетчер. На рис. 13.1 этот актор называется актором-получателем.

13.1.3 Управление изменением состояния актора

Акторы могут не иметь состояния (быть неизменяемыми) или иметь его, т. е. изменять свое состояние в соответствии с получаемыми сообщениями. Например, актор-диспетчер может получать результаты вычислений, которые необходимо переупорядочить перед использованием.

Представьте, например, что у вас есть список данных, которые необходимо обработать с использованием сложных вычислений, чтобы получить список результатов, проще говоря, применить операцию отображения. Ее можно распараллелить, разбив список на несколько подсписков и передав подсписки акторам-обработчикам для выполнения вычислений. Но акторы-обработчики могут закончить работу не в том же порядке, в каком они получали свои задания.

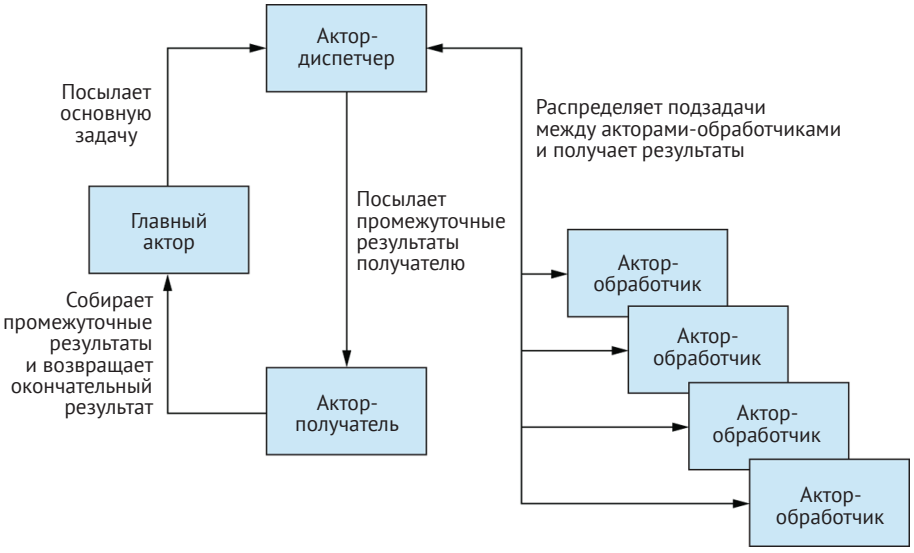


Рис. 13.1 Главный актор создает основную задачу и отправляет ее диспетчеру, который делит ее на подзадачи и передает их для параллельной обработки несколькими акторам-обработчикам. Промежуточные результаты подзадач возвращаются диспетчеру, который пересылает их получателю. После получения всех промежуточных результатов получатель отправляет окончательный результат главному актору

Как одно из решений, для синхронизации промежуточных результатов можно использовать нумерацию задач. Когда обработчик возвращает свой результат, он добавляет соответствующий номер задачи, чтобы получатель мог поместить результаты в очередь с приоритетами. Это не только обеспечит автоматическую сортировку, но также позволит обрабатывать результаты, поступающие в форме асинхронного потока. Каждый раз, когда получатель принимает промежуточный результат, он сравнивает номер задачи с ожидаемым. Если они совпадают, он передает результат клиенту и затем проверяет очередь на наличие в ней следующего промежуточного результата с новым ожидаемым номером задачи. Если такой результат присутствует, извлекает его из очереди и пересылает клиенту. Процесс проверки очереди и отправки соответствующих промежуточных результатов продолжается, пока будут обнаруживаться совпадения с ожидаемым номером задачи. Если полученный результат не соответствует ожидаемому номеру задачи, он добавляется в очередь с приоритетами.

При таком подходе актор-получатель должен поддерживать два изменяемых элемента данных: приоритетную очередь и ожидаемый номер результата. Означает ли это, что актор должен использовать изменяемые свойства? Это не имеет большого значения, но, поскольку акторы являются однопоточными, в таких свойствах нет необходимости. Как вы увидите далее, изменение свойств можно включить и обобщить в общий процесс изменения состояния, что позволит программисту использовать только неизменные данные.

13.2 Реализация инфраструктуры акторов

В этом разделе вы узнаете, как создать минимальную, но полностью функциональную инфраструктуру акторов. В процессе создания этой инфраструктуры вы увидите, как она обеспечивает безопасность при совместном использовании изменяемого состояния, простоту и безопасность распараллеливания и повторной сериализации, а также модульную архитектуру приложений. В конце этой главы вы увидите некоторые обобщенные примеры использования инфраструктуры акторов.

Наша инфраструктура акторов будет включать следующие четыре компонента:

- интерфейс `Actor`, определяющий поведение акторов;
- класс `AbstractActor`, реализующий обобщенную основу для всех акторов; он будет наследоваться специализированными версиями акторов;
- класс `ActorContext`, реализующий функции доступа к акторам; в нашей реализации этот компонент будет предлагать лишь самый минимум возможностей, в основном для доступа к состоянию акторов; (Данный компонент не нужен в такой минимальной реализации инфраструктуры, как наша, но может использоваться в более серьезных реализациях. Этот контекст позволяет, например, искать доступные акторы.)

- интерфейс `MessageProcessor`, который должны реализовать все компоненты, принимающие и обрабатывающие сообщения.

13.2.1 Обзор ограничений

Как я уже говорил, здесь мы реализуем не больше, чем нужно для знакомства и опробования модели акторов. В нашей инфраструктуре не будет многих функций, имеющихся в настоящих системах акторов, особенно связанных с контекстом актора. Также мы пойдем на еще одно упрощение: каждый актор будут действовать в одном потоке выполнения. В настоящих системах акторы запускаются с использованием пулов потоков, что позволяет тысячам или даже миллионам акторов действовать в нескольких десятках потоков.

Другое ограничение нашей реализации касается удаленных акторов. Большинство систем акторов позволяют прозрачно запускать и взаимодействовать с удаленными акторами, т. е. дают возможность запускать акторы на разных машинах и взаимодействовать с ними как с локальными акторами. Это превращает инфраструктуры акторов в идеальный способ создания масштабируемых приложений. Впрочем, эту сторону акторов я не буду рассматривать в книге.

13.2.2 Интерфейсы инфраструктуры акторов

Сначала определим интерфейсы, которые будут составлять основу нашей инфраструктуры акторов. Наиболее важным является интерфейс `Actor`, определяющий несколько функций. Вот главная функция этого интерфейса:

```
fun tell(message: T, sender: Result<Actor<T>>)
```

Эта функция используется для отправки сообщения актору `this` (имеется в виду актор, владеющий функцией). Это означает, что для отправки сообщения актору необходимо иметь ссылку на него. (Этим наша инфраструктура отличается от многих настоящих систем акторов, в которых сообщения отправляются не акторам, а ссылкам на акторов, прокси-компонентам или другим аналогичным объектам. Без этого усовершенствования невозможно было бы отправлять сообщения удаленным акторам.) Эта функция (которая фактически является эффектом) принимает `Result<Actor>` во втором параметре. Предполагается, что он представляет отправителя, но иногда ему присваивается пустое значение или ссылка на другого актора.

Другие функции, показанные в листинге 13.1, используются для управления жизненным циклом актора. Этот код использует тип `Result`, не разработанный в упражнениях из предыдущих глав, а из модуля `fpinkotlin-common`, доступного в примерах кода, сопровождающих эту книгу (https://github.com/pysaumont/fpinkotlin). В своей основе это тот же тип, что и в решениях упражнений, но с некоторыми дополнительными функциями.

Листинг 13.1 Интерфейс Actor

```
interface Actor<T> {

    val context: ActorContext<T> ①

    fun self(): Result<Actor<T>> = Result(this) ②

    fun tell(message: T, sender: Result<Actor<T>> = self()) ③

    fun shutdown() ④

    fun tell(message: T, sender: Actor<T>) =
        tell(message, Result(sender)) ⑤

    companion object {

        fun <T> noSender(): Result<Actor<T>> = Result() ⑥
    }
}
```

① Свойство `context` открывает доступ к контексту актора
② Функция `self` возвращает экземпляр `Result` с текущим актором
③ По умолчанию аргумент `sender` получает результат вызова `self()`, что помогает упростить отправку сообщений, не указывая отправителя явно
④ Функция `shutdown` сообщает актору, что тот должен завершиться. В нашем минималистичном фреймворке это будет приводить к завершению потока выполнения
⑤ Это вспомогательная функция, которая отправляет сообщение актору по ссылке, вместо `Result<Actor>`
⑥ Функция `noSender` – это вспомогательная функция, возвращающая `Result.Empty` в виде типа `Result<Actor>`

В листинге 13.2 показаны два других интерфейса: `ActorContext` и `MessageProcessor`.

Листинг 13.2 Интерфейсы ActorContext и MessageProcessor

```
interface ActorContext<T> {

    fun behavior(): MessageProcessor<T> ①

    fun become(behavior: MessageProcessor<T>) ②
}

interface MessageProcessor<T> {

    fun process(message: T,
                sender: Result<Actor<T>>) ③
}
```

① Открывает доступ к поведению актора
② Позволяет актору менять свое поведение путем регистрации нового экземпляра `MessageProcessor`
③ Интерфейс `MessageProcessor` определяет только одну функцию, реализующую обработку одного сообщения

Наиболее важным в этой паре является интерфейс `ActorContext`. Функция `become` позволяет актору изменить свое поведение – способ обработ-

ки сообщений. Как видите, поведение актора выглядит как эффект, принимающий два аргумента: сообщение для обработки и отправителя.

В процессе работы приложения поведение каждого актора может меняться. Обычно изменение поведения обусловлено изменением состояния актора. Как и почему это происходит, вы поймете, когда увидите реализацию.

13.3 Реализация AbstractActor

Класс `AbstractActor` определяет часть реализации, общую для всех акторов. Все операции, связанные с управлением сообщениями, являются общими для всех акторов и определяются инфраструктурой акторов. При таком подходе вам останется только реализовать бизнес-логику. Реализация класса `AbstractActor` показана в листинге 13.3.

Листинг 13.3 Реализация AbstractActor

```
abstract class AbstractActor<T>(protected val id: String) : Actor<T> {

    override val context: ActorContext<T> = object: ActorContext<T> { ①
        var behavior: MessageProcessor<T> = object: MessageProcessor<T> { ②
            override fun process(message: T, sender: Result<Actor<T>>)
            {
                onReceive(message, sender)
            }
        }

        @Synchronized
        override
        fun become(behavior: MessageProcessor<T>) { ③
            this.behavior = behavior
        }

        override fun behavior() = behavior
    }

    private val executor: ExecutorService = ④
        Executors.newSingleThreadExecutor(DaemonThreadFactory())

    abstract fun onReceive(message: T, sender: Result<Actor<T>>) ⑤

    override fun self(): Result<Actor<T>> {
        return Result(this)
    }

    override fun shutdown() {
        this.executor.shutdown()
    }

    @Synchronized
    override fun tell(message: T, sender: Result<Actor<T>>) { ⑥
        executor.execute {
            try {
                context.behavior()
```

```
                    .process(message, sender) ⑦
            } catch (e: RejectedExecutionException) {
                /*
                 * Это нормальное явление, означающее, что все ожидающие
                 * задания были отменены из-за остановки актора.
                 */
            } catch (e: Exception) {
                throw RuntimeException(e)
            }
        }
    }
}
```

① Инициализация свойства `context` новым экземпляром `ActorContext`
② Делегирует поведение по умолчанию функции `onReceived`
③ Чтобы изменить поведение, `ActorContext` регистрирует новое поведение. Это точка, где происходит мутация (изменение) состояния, но она скрыта внутри инфраструктуры
④ Инициализация механизма выполнения `ExecutorService`
⑤ Содержит бизнес-логику, реализованную пользователем инфраструктуры
⑥ Функция `tell` определяет, как актор получит сообщение. Она объявлена синхронной, чтобы гарантировать последовательную обработку сообщений
⑦ После приема сообщения оно обрабатывается функцией, реализующей поведение актора

Свойство `executor` инициализируется однопоточным механизмом выполнения, для чего вызывается фабричная функция, возвращающая поток-демон, что обеспечивает автоматическое его завершение при завершении главного потока. Благодаря этому акторы не будут препятствовать завершению приложения при остановке основного потока. (Код с реализацией `DaemonThreadFactory` вы найдете в хранилище для этой книги.)

На этом реализацию нашей инфраструктуры акторов можно считать завершенной, хотя, как я уже говорил, она весьма ограничена в своих возможностях. Цель этого минимального примера – показать принцип работы системы акторов.

13.4 Включение акторов в работу

Теперь, получив инфраструктуру акторов, задействуем ее для решения некоторых конкретных задач. Акторы особенно удобны, когда предполагается совместное использование некоторого общего изменяемого состояния из нескольких потоков выполнения, например когда один поток производит некоторые вычисления и результаты этих вычислений нужно передать другому потоку для дальнейшей обработки.

Обычно такое использование общего изменяемого состояния подразумевает сохранение значений в общих изменяемых свойствах с обязательным применением механизмов блокировки и синхронизации. Сначала мы опробуем минимальный пример, который можно рассматривать как аналог программы «Hello, world!» с акторами. А затем создадим более полное приложение, в котором один актор будет распределять задания между другими акторами, действующими параллельно.

Первый пример – это традиционная демонстрация работы акторов. Он состоит из двух игроков в пинг-понг и судьи. Игра начинается с передачи мячика, представленного целым числом, одному из игроков. Затем игроки посылают мячик друг другу, пока не будет зафиксирована десятая передача, после чего мяч возвращается судье.

13.4.1 Реализация примера игры в пинг-понг

Первым реализуем судью. Для этого достаточно создать экземпляр актора и реализовать его функцию onReceive, которая будет выводить сообщение, как показано в листинге 13.4.

Листинг 13.4 Создание актора-судьи

```
val referee = object : AbstractActor<Int>("Referee") {
    override fun onReceive(message: Int, sender: Result<Actor<Int>>) {
        println("Game ended after $message shots")
    }
}
```

Далее создадим двух игроков. Так как у нас их два, мы на выбор имеем два варианта реализации. Первый вариант, объектно-ориентированный, – объявить класс Player, как показано в листинге 13.5.

Листинг 13.5 Класс Player

```
private class Player(id: String, ①
                     private val sound: String, ②
                     private val referee: Actor<Int>):
        AbstractActor<Int>(id) { ③

    override fun onReceive(message: Int, sender: Result<Actor<Int>>) {
        println("$sound - $message") ④
        if (message >= 10) {
            referee.tell(message, sender) ⑤
        } else {
            sender.forEach(
                { actor: Actor<Int> -> ⑥
                    actor.tell(message + 1, self())
                },
                { referee.tell(message, sender) } ⑦
            )
        }
    }
}
```

① Класс объявлен приватным (private), потому что определяется на уровне пакета. Также его можно было бы объявить локальным внутри функции main в программе

② Строка sound – это сообщение, которое выводит игрок, когда получает мяч ("Ping" или "Pong")

③ При создании каждому игроку передается ссылка на судью (referee), чтобы он мог вернуть мяч судье, когда игра завершится. В этом не было бы необходимости, если бы класс Player определялся локально в той же функции, где создается экземпляр судьи referee (например, в функции main)

④ Бизнес-логика актора, т. е. логика, определяемая пользователем
⑤ Если игра завершена, возвращает мяч судье
⑥ В противном случае посылает мяч другому игроку, если тот существует
⑦ Если второй игрок отсутствует, об этой проблеме сообщается судье

Второй вариант, функциональный, – написать функцию, возвращающую `Actor`, как показано в листинге 13.6.

Листинг 13.6 Функция player

```
fun player(id: String,
           sound: String, ①
           referee: Actor<Int>) =
               object : AbstractActor<Int>("id") { ②

    override fun onReceive(message: Int, sender: Result<Actor<Int>>) {
        println("$sound - $message") ③
        if (message >= 10) {
            referee.tell(message, sender) ④
        } else {
            sender.forEach(
                { actor: Actor<Int> -> ⑤
                    actor.tell(message + 1, self())
                },
                { referee.tell(message, sender) } ⑥
            )
        }
    }
}
```

① Строка sound – это сообщение, которое выводит игрок, когда получает мяч ("Ping" или "Pong")
② При создании каждому игроку передается ссылка на судью (referee), чтобы он мог вернуть мяч судье, когда игра завершится
③ Бизнес-логика актора
④ Если игра завершена, возвращает мяч судье
⑤ В противном случае посылает мяч другому игроку, если тот существует
⑥ Если второй игрок отсутствует, об этой проблеме сообщается судье

Как видите, оба решения почти идентичны. Это показывает, что объекты в действительности являются функциями.

Написав функцию `player` (или класс `Player`), мы можем дописать программу до конца. Но нам еще нужно как-то предотвратить завершение приложения, пока игра не закончится. Без этого основной поток приложения завершится сразу после запуска игры, и у игроков не будет возможности поиграть. Эту проблему легко решить с помощью семафора, как показано в листинге 13.7.

Листинг 13.7 Пример игры в пинг-понг

```
private val semaphore = Semaphore(1) ①

fun main(args: Array<String>) {
    val referee = object : AbstractActor<Int>("Referee") {

        override fun onReceive(message: Int, sender: Result<Actor<Int>>) {
```

```
            println("Game ended after $message shots")
            semaphore.release() ②
        }
    }

    val player1 = player("Player1", "Ping", referee) ③
    val player2 = player("Player2", "Pong", referee)

    semaphore.acquire() ④
    player1.tell(1, Result(player2))
    semaphore.acquire() ⑤
    // здесь главный поток завершается ⑥
}
```

① Создается семафор, который можно приобрести только один раз
② Когда игра завершается, семафор освобождается, что позволяет главному потоку приобрести его вторично и продолжить выполнение (т. е. завершиться)
③ Если вы решите выбрать объектно-ориентированный вариант, просто замените имя `player` на `Player`
④ Текущий поток приобретает семафор и запускает игру
⑤ Главный поток пытается приобрести семафор вторично. Так как данный семафор может приобрести только кто-то один, поток блокируется до момента освобождения семафора
⑥ Когда семафор освободится по окончании игры, главный поток продолжит выполнение и тут же завершится. Все потоки акторов – это потоки-демоны, поэтому они автоматически завершатся вместе с главным потоком

Эта программа выведет следующее:

```
Ping - 1
Pong - 2
Ping - 3
Pong - 4
Ping - 5
Pong - 6
Ping - 7
Pong - 8
Ping - 9
Pong - 10
Game ended after 10 shots
```

13.4.2 Параллельное выполнение вычислений

Пришло время взглянуть на более серьезный пример использования модели акторов: параллельное выполнение вычислений. Чтобы сымитировать длительные вычисления, возьмем список случайных чисел от 0 до 30 и для каждого вычислим значение Фибоначчи с соответствующим номером, используя медленный алгоритм.

Приложение состоит из трех типов акторов: диспетчер `Manager`, отвечающий за создание заданного числа акторов-обработчиков и распределение заданий между ними; несколько экземпляров акторов-обработчиков; и клиент, реализованный в классе основной программы как анонимный актор. В листинге 13.8 показан самый простой из этих классов – актор `Worker`.

Листинг 13.8 Актор Worker, получающий задание и выполняющий его

```
class Worker(id: String) : AbstractActor<Int>(id) {

    override fun onReceive(message: Int, sender: Result<Actor<Int>>) {
        sender.forEach (onSuccess = { a: Actor<Int> ->
            a.tell(slowFibonacci(message), self()) ①
        })
    }

    private fun slowFibonacci(number: Int): Int {
        return when (number) {
            0    -> 1
            1    -> 1
            else -> slowFibonacci(number - 1)
                    + slowFibonacci(number - 2) ②
        }
    }
}
```

① Когда Worker получает число, он вычисляет число Фибоначчи с соответствующим порядковым номером и возвращает его вызывающему

② Неэффективная реализация алгоритма, имитирующая продолжительные вычисления

Как видите, этот актор не имеет состояния. Он вычисляет результат и посылает его обратно отправителю, ссылку на которого он получил вместе с заданием. Это может быть другой актор, отличный от вызывающего.

Так как числа выбираются случайно из диапазона от 0 до 35, для вычисления каждого результата требуется разное время. Этот список имитирует перечень задач, решение которых занимает разное время. В отличие от примера автоматического распараллеливания, представленного в главе 8, все потоки/акторы остаются занятыми, пока не будут завершены все вычисления.

Класс Manager выглядит немного сложнее. В листинге 13.9 показан конструктор класса и свойства, которые он инициализирует.

Листинг 13.9 Конструктор и свойства класса Manager

```
class Manager(id: String, list: List<Int>,
              private val client: Actor<Result<List<Int>>>, ①
              private val workers: Int) : AbstractActor<Int>(id) { ②

    private val initial: List<Pair<Int, Int>> ③
    private val workList: List<Int> ④
    private val resultList: List<Int> ⑤
    private val managerFunction:
        (Manager) -> (Behavior) -> (Int) -> Unit ⑥

    init {
        val splitLists = list.splitAt(this.workers) ⑦
        this.initial = splitLists.first.zipWithPosition() ⑧
        this.workList = splitLists.second ⑨
        this.resultList = List() ⑩
```

```
    managerFunction = { manager -> ⑪
        { behavior ->
            { i ->
                val result = behavior.resultList.cons(i) ⑫
                if (result.length == list.length) {
                    this.client.tell(Result(result)) ⑬
                } else {
                    manager.context
                           .become(Behavior(behavior.workList ⑭
                           .tailSafe()
                           .getOrElse(List()),result))
                }
            }
        }
    }
}
...
```

① Экземпляр `Manager` хранит ссылку на клиента, которому должен переслать результаты вычислений
② Число акторов-обработчиков
③ Начальный список пар целых чисел; каждая пара хранит число для обработки (`.first`) и его позицию в списке (`.second`)
④ `workList` – это список еще не выполненных заданий, оставшихся после того, как все акторы-обработчики получат свое первое задание
⑤ `resultList` хранит результаты вычислений
⑥ `managerFunction` – основа диспетчера `Manager`; эта функция определяет, что тот должен делать. Она вызывается каждый раз, когда диспетчер получает очередной результат от актора-обработчика
⑦ Список чисел для обработки разбивается на две части: первая часть становится списком начальных заданий с длиной, равной количеству акторов-обработчиков, а вторая – списком невыполненных заданий
⑧ Список начальных заданий (с числами для вычисления значений Фибоначчи) объединяется с позициями элементов. Позиции (от 0 до n) используются для создания имен акторов-обработчиков
⑨ `workList` – список пока невыполненных заданий
⑩ `resultList` инициализируется пустым списком
⑪ Функция `managerFunction`, определяющая действия диспетчера, – это каррированная функция, самого диспетчера, его поведения и полученного сообщения (`i`), которая возвращает результат выполнения задания
⑫ Получив результат, она добавляет его в список результатов, который извлекается из поведения диспетчера
⑬ Если длина `resultList` равна длине входного списка, вычисления завершаются и результаты посылаются клиенту
⑭ В противном случае вызывается функция `become` контекста, чтобы изменить поведение диспетчера. Это изменение соответствует изменению состояния. Новое поведение создается на основе хвоста `workList` и текущего списка результатов (в который было добавлено только что полученное значение)

Как видите, если вычисления закончены, результат добавляется в список результатов и отправляется клиенту. Иначе результат просто добавляется в текущий список результатов. В традиционном программировании это делается путем изменения списка результатов, хранящегося

диспетчером. Именно это и происходит здесь, но есть два важных отличия:

- список результатов хранится в поведении;
- ни поведение, ни список не изменяются. Вместо этого создается новое поведение и изменяется контекст, в котором сохраняется новое поведение, замещающее старое. Но мы не имеем дела с этой мутацией. Снаружи все выглядит неизменным, потому что мутация абстрагируется инфраструктурой акторов.

В листинге 13.10 показан класс `Behavior`, реализованный как внутренний. Такой подход позволяет скрыть мутации в недрах инфраструктуры.

Листинг 13.10 Внутренний класс Behavior

```
internal inner class Behavior
    internal constructor(
        internal val workList: List<Int>, ①
        internal val resultList: List<Int>) : MessageProcessor<Int> {

    override fun process(message: Int,
                         sender: Result<Actor<Int>>) { ②
        managerFunction(this@Manager)(this@Behavior)(message)
        sender.forEach(onSuccess = { a: Actor<Int> ->
            workList.headSafe()
                    .forEach({ a.tell(it, self()) }) { a.shutdown() }
        })
    }
}
```

① Экземпляр `Behavior` создается на основе `workList` (из которого перед вызовом конструктора удаляется голова) и `resultList` (куда был добавлен результат)

② Функция `process`, которая вызывается при получении сообщения, сначала применяет функцию `managerFunction` к полученному сообщению, затем отправляет следующее задание (голову `workList`) отправителю (актору-обработчику, который должен его выполнить) или, если список `worklist` опустел, дает актору-обработчику команду завершить работу

Это была основная часть класса `Manager`. Оставшуюся часть составляют служебные функции, которые используются в основном для начала работы. Эти функции показаны в листинге 13.11.

Листинг 13.11 Служебные функции в классе Manager

```
class Manager(id: String, list: List<Int>, . . .

    ...

    fun start() {
        onReceive(0, self()) ①
        sequence(initial.map { this.initWorker(it) })
            .forEach(onSuccess = { this.initWorkers(it) },
                     onFailure = ②
                        { this.tellClientEmptyResult(
                            it.message ?: "Unknown error") })
```

```
    }

    private fun initWorker(t: Pair<Int, Int>): Result<() -> Unit> = ③
        Result({ Worker("Worker " + t.second).tell(t.first, self()) })

    private fun initWorkers(lst: List<() -> Unit>) {
        lst.forEach { it() } ④
    }

    private
    fun tellClientEmptyResult(string: String) { ⑤
        client.tell(Result.failure("$string caused by empty input list."))
    }

    override fun onReceive(message: Int,
                           sender: Result<Actor<Int>>) { ⑥
        context.become(Behavior(workList, resultList))
    }

    ...
}
```

① Для начала диспетчер посылает сообщение самому себе. Что это за сообщение, не имеет значения, потому что поведение еще не инициализировано
② Затем создаются и инициализируются акторы-обработчики
③ Эта функция создает функцию типа `() -> Unit`, которая в свою очередь создает актор-обработчик
④ Эта функция выполняет функцию создания актора
⑤ Если возникла ошибка, сообщить о ней клиенту
⑥ Начальное поведение диспетчера. В процессе инициализации он переключает поведение, начав со списка `workList`, содержащего оставшиеся задания, и пустого списка с результатами

Важно понимать, что функция `onReceive` представляет действия, которые актор будет выполнять, когда получит первое сообщение. Эта функция не будет вызываться, когда акторы-обработчики начнут посылать результаты диспетчеру.

Последняя часть программы показана в листинге 13.12. Это реализация клиента. В отличие от диспетчера и обработчиков это не актор, а функция `main`, *использующая* актор. Это произвольный выбор для конкретной реализации, потому что ни один из других вариантов не дает видимых преимуществ. Но сам актор-клиент необходим, потому что иначе невозможно будет получить результат.

Листинг 13.12 Приложение-клиент

```
import com.fpinkotlin.common.List
import com.fpinkotlin.common.Result
import com.fpinkotlin.common.range
import java.util.concurrent.Semaphore

private val semaphore = Semaphore(1) ①
private const val listLength = 20_000 ②
private const val workers = 8 ③
private val rnd = java.util.Random(0)
private val testList = ④
```

```
    range(0, listLength).map { rnd.nextInt(35) }

fun main(args: Array<String>) {
    semaphore.acquire() ⑤
    val startTime = System.currentTimeMillis()
    val client = ⑥
        object: AbstractActor<Result<List<Int>>>("Client") {
            override fun onReceive(message: Result<List<Int>>,
                                   sender: Result<Actor<Result<List<Int>>>>) {
                message.forEach({ processSuccess(it) }, ⑦
                    { processFailure(it.message ?: "Unknown error") })
                println("Total time: "
                    + (System.currentTimeMillis() - startTime))
                semaphore.release() ⑧
            }
        }

    val manager = ⑨
        Manager("Manager", testList, client, workers)
    manager.start()
    semaphore.acquire() ⑩
}

private fun processFailure(message: String) {
    println(message)
}

fun processSuccess(lst: List<Int>) {
    println("Input: ${testList.splitAt(40).first}")
    println("Result: ${lst.splitAt(40).first}")
}
```

① Семафор создается, чтобы приостановить главный поток выполнения, пока акторы не выполнят свою работу
② Число заданий
③ Число акторов-обработчиков
④ Список заданий заполняется случайными числами в диапазоне от 0 до 35
⑤ В момент запуска главный поток приобретает семафор
⑥ Создается клиентский актор как анонимный объект-синглтон
⑦ Клиент отвечает только за обработку окончательного результата или сообщения об ошибке
⑧ Клиент освобождает семафор после получения списка с результатами
⑨ Создается и запускается диспетчер
⑩ Главный поток пытается приобрести семафор и блокируется, пока клиент не освободит его

Попробуйте запустить эту программу с разной длиной списка заданий и разным числом акторов-обработчиков. На моем восьмиядерном компьютере с Linux на борту я получил следующие результаты для списка с 20 000 заданий:

- один актор-обработчик: 73 с;
- два актора-обработчика: 37 с;
- четыре актора-обработчика: 19 с;
- восемь акторов-обработчиков: 12 с;
- шестнадцать акторов-обработчиков: 12 с.

Эти цифры не точные, но характер их изменения показывает, что бессмысленно создавать акторов-обработчиков больше, чем имеется ядер процессора. Вот как выглядит результат, отображаемый программой (здесь приведены только первые 11 результатов):

```
Input: [5, 23, 4, 2, 25, 28, 16, 1, 34, 9, 22, ..., NIL]
Result: [8, 5, 2, 1597, 46368, 121393, 2, 55, 28657, 1, 2, ..., NIL]
Total time: 12558
```

Как видите, у нас есть проблема!

13.4.3 Переупорядочение результатов

Как видите, результат получился неверным. Это особенно заметно на втором случайном значении (23) и соответствующем ему результате (5). Если желаете, можете также сравнить другие исходные значения и результаты. Если вы попробуете запустить программу у себя, каждый раз у вас будут получаться разные последовательности результатов.

Причина происходящего в том, что не все задания выполняются одинаковое время. Я намеренно настроил вычисления так, чтобы одни задачи (с маленькими значениями аргументов) выполнялись быстро, а другие (с большими значениями аргументов) – намного дольше. В итоге получаемые результаты возвращаются в неправильном порядке.

Чтобы устранить проблему, результаты нужно отсортировать в порядке следования аргументов. Одно из решений – использовать тип `Heap` из главы 11. Мы можем пронумеровать задания и использовать их номера в качестве приоритетов в приоритетной очереди.

С этой целью нужно изменить класс акторов-обработчиков. Теперь они должны обрабатывать не целые числа, а кортежи с двумя целыми числами: одно число будет представлять аргумент для вычислений, а другое – номер задания. Соответствующие изменения в классе `Worker` показаны в листинге 13.13.

Листинг 13.13 Актор Worker сохраняет номер задания

```
class Worker(id: String) :
    AbstractActor<Pair<Int, Int>>(id) { ①

    override fun onReceive(message: Pair<Int, Int>,
            sender: Result<Actor<Pair<Int, Int>>>) { ②
        sender.forEach(onSuccess =
            { a: Actor<Pair<Int, Int>> ->
              a.tell(Pair(slowFibonacci(message.first), ③
                          message.second) , self())
            })
    }
    ...
}
```

① Параметр типа изменился с `Int` на `Pair<Int, Int>`
② Сигнатура функции `onReceive` изменилась в соответствии с новым типом актора
③ Возвращаемое сообщение теперь включает номер задания

Номер задания передается во втором элементе кортежа. Такой код трудно читать, потому что номер задания и аргумент имеют один и тот же тип (`Int`). В реальной жизни такого кода следует избегать и для нумерации заданий использовать конкретный тип. Также вместо `Pair` можно использовать специализированный тип и использовать его для обертывания заданий им их порядковых номеров, например тип `Task` со свойством `number`.

Изменения в классе `Manager` более обширные. Мы должны изменить тип класса и типы свойств `workList` и результата:


```
class Manager(id: String, list: List<Int>,
              private val client: Actor<Result<List<Int>>>,
              private val workers: Int) : AbstractActor<Pair<Int, Int>>(id) {
    private val initial: List<Pair<Int, Int>>
    private val workList: List<Pair<Int, Int>>
    private val resultHeap: Heap<Pair<Int, Int>>
    private val managerFunction: (Manager) -> (Behavior) -> (Int) -> Unit
```

Вот как инициализируются эти свойства в конструкторе:

```
init {
    val splitLists = list.zipWithPosition().splitAt(this.workers)
    this.initial = splitLists.first
    this.workList = splitLists.second
    this.resultHeap = Heap(Comparator {
        p1: Pair<Int, Int>, p2: Pair<Int, Int> ->
            p1.second.compareTo(p2.second)
    })
```

Свойство `workList` теперь хранит список пар (как и список `initial` в предыдущем примере), а результаты сохраняются в приоритетной очереди (`Heap`) пар. Это свойство типа `Heap` инициализируется компаратором `Comparator`, который сравнивает вторые элементы пар.

Если бы мы определили тип `Task`, включающий само задание и его номер, мы могли бы реализовать в нем интерфейс `Comparable` и избавиться от необходимости определять класс `Comparator`. (Я предлагаю вам реализовать такой вариант самостоятельно.) Функция `managerFunction` также изменилась:

```
private val managerFunction:
    (Manager) -> (Behavior) -> (Pair<Int, Int>) -> Unit
```

Она инициализируется в конструкторе, как показано ниже:

```
managerFunction = { manager ->
    { behavior ->
        { p ->
            val result = behavior.resultHeap + p ①
            if (result.size == list.length) {
                this.client.tell(Result(result.toList()
                            .map { it.first })) ②
            } else {
                ...
```

```
            }
        }
    }
}
```

① Теперь полученный результат вставляется в очередь Heap
② По завершении вычислений очередь Heap преобразуется в список и возвращается клиенту

Внутренний класс `Behavior` тоже нужно изменить, чтобы привести его в соответствие с новым типом акторов:

```
internal inner class Behavior
        internal
        constructor(internal
                val workList: List<Pair<Int, Int>>, ①
                internal val resultHeap: Heap<Pair<Int, Int>>): ②
                    MessageProcessor<Pair<Int, Int>> { ③
    override
    fun process(message: Pair<Int, Int>,
                sender: Result<Actor<Pair<Int, Int>>>) { ④
        managerFunction(this@Manager)(this@Behavior)(message)
        sender.forEach(onSuccess = { a: Actor<Pair<Int, Int>> ->
            workList.headSafe()
                    .forEach({ a.tell(it, self()) }) { a.shutdown() }
        })
    }
}
```

① Теперь свойство `workList` имеет тип `List<Pair<Int, Int>>`
② Результат получил тип `Heap<Pair<Int, Int>>`
③ Параметр типа класса `Behavior` теперь `Pair<Int, Int>`
④ Сигнатура функции `process` тоже изменилась в соответствии с параметром типа

Осталось внести небольшие изменения в остальные функции класса `Manager`. В функцию `start`:

```
fun start() {
    onReceive(Pair(0, 0), self()) ①
    sequence(initial.map { this.initWorker(it) })
        .forEach({ this.initWorkers(it) },
                 { this.tellClientEmptyResult(
                       it.message ?: "Unknown error") })
}
```

① Тип сообщения должен соответствовать параметру типа актора `Manager`

В процедуру инициализации экземпляров `Worker`:

```
private fun initWorker(t: Pair<Int, Int>): Result<() -> Unit> =
    Result({ Worker("Worker " + t.second)
        .tell(Pair(t.first, t.second), self()) })
```

И наконец, в функцию `onReceive`:

```
override fun onReceive(message: Pair<Int, Int>,
                       sender: Result<Actor<Pair<Int, Int>>>) {
```

```
    context.become(Behavior(workList, resultHeap))
}
```

Теперь результаты будут отображаться в правильном порядке.

13.4.4 Оптимизация производительности

Теперь наш пример работает правильно, но действует далеко не оптимально. Основная проблема: все результаты предварительно помещаются в приоритетную очередь для сортировки. Я уже говорил в главе 11, что это неправильный способ использования приоритетных очередей.

Приоритетная очередь предназначена для размещения элементов, которые должны обрабатываться в определенном порядке (согласно их приоритету). Предполагается, что элементы будут потребляться примерно с той же скоростью, с какой они создаются, и в очереди никогда не будет храниться больше нескольких элементов одновременно. В данном случае элементы должны храниться, только пока существуют необработанные элементы с более высоким приоритетом. Это не единственный способ использования очередей с приоритетами, но один из лучших.

Чтобы увидеть проблему на практике, попробуйте заменить функцию `slowFibonnacci` в классе `Worker` более эффективной, такой как:

```
private fun fibonacci(number: Int): Int {
    tailrec fun fibonacci(acc1: Int, acc2: Int, x: Int): Int = when (x)
    {
        0 -> 1
        1 -> acc1 + acc2
        else -> fibonacci(acc2, acc1 + acc2, x - 1)
    }
    return fibonacci(0, 1, number)
}
```

Присвойте переменной `listLength` в клиентской программе число 500 000 и опробуйте ее с одним, двумя, четырьмя и восемью акторами (если процессор вашего компьютера имеет достаточное число ядер). Вот результаты, которые получились у меня:

```
1 actor: 40567 ms
2 actors: 24399 ms
4 actors: 22394 ms
8 actors: 22389 ms
```

Здесь можно заметить интересную особенность: для коротких задач наличие нескольких акторов дает определенный выигрыш, причем наибольший выигрыш получается при переходе от одного к двум акторам. Использование большего количества акторов не дает увеличения производительности. Это может быть неочевидно, потому приведенные цифры зависят от конкретного компьютера, но они довольно наглядно демонстрируют тенденцию. Для сравнения можете использовать следующий код, воспользовавшись `testList` из программы `WorkersExample`:

```
println(testList.map { fibonacci(it) }.splitAt(40).first)
```

Он выполняется 700 мс, что примерно в 30 раз быстрее, чем версия с двумя, четырьмя или восемью акторами. Одна из причин в том, что узким местом является очередь `Heap`. Структура данных `Heap` не предназначена для сортировки. Она показывает хорошую производительность, пока число элементов в ней невелико, но, так как после вставки каждого элемента происходит сортировка всей структуры, при наличии 200 000 результатов в очереди ее производительность резко падает.

Очевидно, что эта неэффективность не является проблемой реализации. Это проблема выбора неправильного инструмента. Гораздо лучшую производительность можно получить, если сохранить все результаты и отсортировать их один раз по окончании вычислений, но для сортировки нужно использовать правильный инструмент.

Другой вариант – изменить дизайн программы. Одна из проблем текущего дизайна: вставка в очередь не только занимает много времени, но и выполняется актором-диспетчером. Вместо того чтобы выдавать новые задания акторам-обработчикам, как только они заканчивают вычисления, диспетчер заставляет их ждать, пока завершится вставка в очередь. Одно из возможных решений: использовать отдельного актора для вставки результатов в очередь.

Но иногда лучше правильно использовать имеющийся инструмент. Синхронное использование результата не всегда является обязательным требованием. Если это не так, появляется неявное требование, которое затрудняет решение проблемы. Одним из возможных решений может быть передача результатов клиенту по отдельности. При таком подходе очередь будет заполняться, только когда результаты поступают не по порядку, и не будет сильно разрастаться в размерах.

Именно так должна использоваться приоритетная очередь. Чтобы реализовать это решение, добавим в программу актор `Receiver`, представленный в листинге 13.14.

Листинг 13.14 Актор Receiver осуществляет прием результатов асинхронным способом

```
import com.fpinkotlin.common.List
import com.fpinkotlin.common.Result

class Receiver(id: String,
               private val client: Actor<List<Int>>): ①
                                AbstractActor<Int>(id) { ②

    private val receiverFunction: (Receiver) -> (Behavior) -> (Int) -> Unit

    init {
        receiverFunction = { receiver -> ③
            { behavior ->
                { i ->
                    if (i == -1) {
                        this.client.tell(behavior.resultList.reverse())
                        shutdown()
                    } else {
                        receiver.context ④
```

```
                            .become(Behavior(behavior.resultList.cons(i)))
                    }
                }
            }
        }
    }

    override fun onReceive(i: Int, sender: Result<Actor<Int>>) {
        context.become(Behavior(List(i))) ⑤
    }

    internal inner class Behavior internal constructor(
        internal val resultList: List<Int>) : ⑥
                        MessageProcessor<Int> {

        override fun process(i: Int, sender: Result<Actor<Int>>) {
            receiverFunction(this@Receiver)(this@Behavior)(i)
        }
    }
}
```

① Клиент в классе `Receiver` параметризован типом `List<Int>`
② Класс `Receiver` – это актор, параметризуемый типом данных, которые он должен принимать: `Int`
③ Функция `receiverFunction` принимает аргумент `Int`. Если он равен `–1`, значит, вычисления завершились и он должен послать результаты своему клиенту и завершиться
④ В противном случае он изменяет свое поведение, добавляя результат в список
⑤ Начальная реализация `onReceive` устанавливает поведение актора, которое использует новый список с первым результатом
⑥ Поведение хранит текущий список с результатами

Главная программа (`WorkersExample.kt`) мало отличается от предыдущего примера. Единственное отличие – появление экземпляра `Receiver`:

```
fun main(args: Array<String>) {
    semaphore.acquire()
    val startTime = System.currentTimeMillis()
    val client =
        object: AbstractActor<List<Int>>("Client") {
            override fun onReceive(message: List<Int>,
                                   sender: Result<Actor<List<Int>>>) {
                println("Total time: "
                        + (System.currentTimeMillis() - startTime))
                println("Input: ${testList.splitAt(40).first}")
                println("Result: ${message.splitAt(40).first}")
                semaphore.release()
            }
        }

    val receiver = Receiver("Receiver", client) ①
    val manager = Manager("Manager", testList, receiver, workers) ②
    manager.start()
    semaphore.acquire()
}
```

① Создается экземпляр `Receiver`, и в роли клиента ему передается главный актор
② Теперь в роли клиента актору-диспетчеру передается экземпляр `Receiver`

Класс `Worker` остался прежним, как в предыдущем примере. А наиболее существенные изменения претерпел класс `Manager`. Первое изменение: диспетчер получает клиента типа `Actor<Int>` и хранит длину списка заданий:

```
class Manager(
        id: String, list: List<Int>,
        private val client: Actor<Int>, ①
        private val workers: Int) : AbstractActor<Pair<Int, Int>> (id) {

    private val initial: List<Pair<Int, Int>>
    private val workList: List<Pair<Int, Int>>
    private val resultHeap: Heap<Pair<Int, Int>>
    private val managerFunction: (Manager) -> (Behavior) -> (Pair<Int, Int>) -> Unit
    private val limit: Int ②
```

① **Класс `Manager` получает клиента типа `Actor<Int>`**
② **Диспетчер хранит длину списка заданий**

Клиент `Receiver` теперь получает результаты асинхронно по одному. Функция `managerFunction` изменилась:

```
managerFunction = { manager ->
    { behavior ->
        { p ->
            val result =
                streamResult(behavior.resultHeap + p,
                             behavior.expected, List()) ①
                result.third.forEach { client.tell(it) }
                if (result.second > limit) {
                    this.client.tell(-1) ②
                } else {
                    manager.context
                        .become(Behavior(behavior.workList
                            .tailSafe()
                            .getOrElse(List()), result.first,
                                       result.second))
            }
        }
    }
}
```

① **Вызов функции `streamResult`**
② **Отправка признака завершения**

Эта функция теперь вызывает функцию `streamResult`, возвращающую `Triple`. Первый элемент – очередь, куда был добавлен полученный результат. Второй элемент – следующий ожидаемый номер результата, а третий – это список результатов, которые находятся в ожидаемом порядке. Если все задания выполнены, клиенту отправляется признак завершения. Как видите, основная работа выполняется в функции `streamResult`:

```
private fun streamResult(result: Heap<Pair<Int, Int>>,
                         expected: Int, list: List<Int>):
            Triple<Heap<Pair<Int, Int>>, Int, List<Int>> {
    val triple = Triple(result, expected, list)
    val temp = result.head
        .flatMap { head ->
            result.tail().map { tail ->
                if (head.second == expected)
                    streamResult(tail, expected + 1,
                                 list.cons(head.first))
                else
                    triple
            }
        }

    return temp.getOrElse(triple)
}
```

Функция `streamResult` получает аргумент `Heap` с результатами, следующий ожидаемый номер задания и список целых чисел, который первоначально пуст:

- если голова очереди с результатами отличается от ожидаемого номера задания, ничего не делается и три параметра возвращаются как `Triple`;
- если голова очереди совпадает с ожидаемым номером задания, она удаляется из очереди и добавляется в список. Затем функция вызывается рекурсивно, пока голова очереди не перестанет совпадать с очередным номером задания, чтобы получить список с результатами, следующими в ожидаемом порядке.

При такой организации обработки очередь всегда остается небольшой. Например, когда я попробовал выполнить 200 000 заданий, максимальный размер очереди составил 121. В 12 случаях он превысил 100, а в 95 % случаев оставался равным двум или меньше. Общий процесс приема результатов с точки зрения `Manager` показан на рис. 13.2.

Функция `onReceive` изменилась, потому что теперь первоначально ожидается результат с номером 0:

```
override fun onReceive(message: Pair<Int, Int>,
                       sender: Result<Actor<Pair<Int, Int>>>) {
    context.become(Behavior(workList, resultHeap, 0))
}
```

Класс `Behavior` тоже изменился. Теперь он должен хранить ожидаемый номер задания:

```
internal inner class Behavior
    internal constructor(internal val workList: List<Pair<Int, Int>>,
                         internal val resultHeap: Heap<Pair<Int, Int>>,
                         internal val expected: Int) :
                             MessageProcessor<Pair<Int,Int>> {
    ...
```

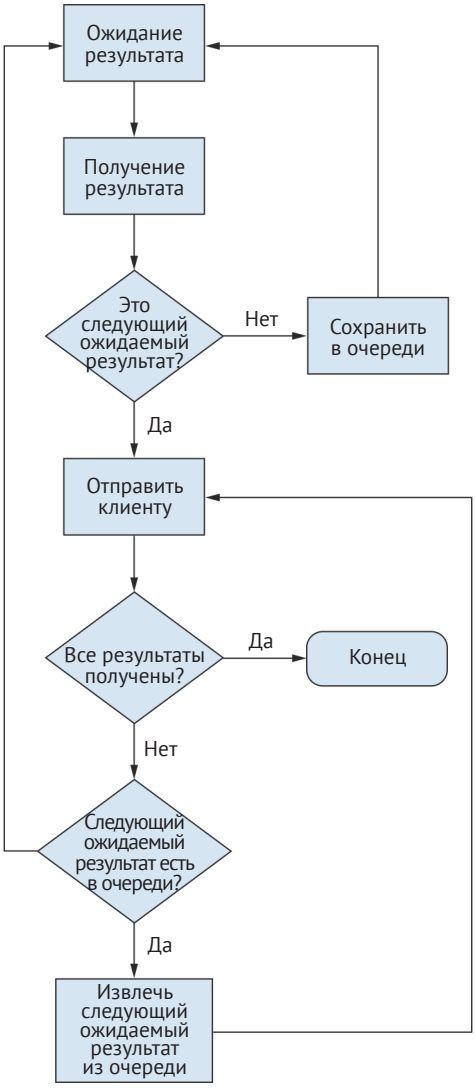


Рис. 13.2 Актор-диспетчер принимает результаты и сохраняет их в очереди (если их номера не соответствуют ожидаемым) или посылает клиенту. В последнем случае он дополнительно проверяет содержимое очереди, чтобы уточнить – был ли получен ранее следующий по порядку результат

Последнее изменение коснулось функции `Manager.start`, потому что теперь клиент имеет тип `Actor<Int>`:

```
fun start() {
    onReceive(Pair(0, 0), self())
    sequence(initial.map { this.initWorker(it) })
        .forEach({ this.initWorkers(it) },
                 { client.tell(-1) })
}
```

После внесения всех этих изменений приложение стало работать намного быстрее. Например, при тех же начальных условиях, что и в предыдущем примере, для обработки 1 000 000 чисел с одним, двумя, четырьмя и восемью акторами-обработчиками потребовалось:

```
1 actor: 40567 ms
2 actors: 12251 ms
4 actors: 11055 ms
8 actors: 11043 ms
```

Конечно, эта версия выполняется медленнее, чем отображение функции Фибоначчи на список чисел. Но не забывайте, что мы переключились на быструю реализацию заданий: распараллеливание задач обычно не дает увеличения производительности с короткими и быстро выполняющимися заданиями. Но если переключиться обратно на медленную версию функции Фибоначчи, результаты будут совсем другими. Вот результаты для списка с 200 000 чисел:

```
simple mapping: 12 mn 46 s
1 actor: 12 mn 2 s
2 actors: 6 mn 2 s
4 actors: 3 mn 3 s
8 actors: 1 mn 40 s
```

Теперь, когда каждое задание выполняется довольно долго, распараллеливание вычислений с использованием акторов значительно увеличило общую производительность. Целью этого примера было лишь показать, как пользоваться акторами. Подобные же задачи лучше решать с использованием других средств, таких как автоматическое распараллеливание списков (как рассказывалось в главе 8).

Главное предназначение акторов – не распараллеливание обработки, а абстракция совместного использования изменяемого состояния. В этих примерах мы применяли списки, которые использовались несколькими заданиями. Без акторов нам пришлось бы синхронизировать доступ к `workList` и `resultHeap`. Акторы позволяют абстрагировать синхронизацию и изменение общего состояния.

Заглянув в бизнес-код, который мы написали (т. е. код, не имеющий отношения к инфраструктуре акторов), вы не найдете в нем изменяемых данных, благодаря чему нам не пришлось заботиться о синхронизации и проблемах нехватки потоков выполнения или взаимоблокировки. Несмотря на то что акторы основаны на эффектах, они предлагают хороший способ заставить функциональные части кода работать вместе, используя изменяемое состояние абстрактным образом.

Наша инфраструктура акторов содержит лишь минимальный набор инструментов и не предназначена для использования в серьезных приложениях. Для этих целей лучше использовать одну из инфраструктур акторов, доступных для Java, таких как Akka. Несмотря на то что фреймворк Akka написан на Scala, его с успехом можно использовать в программах на Kotlin. Пользуясь фреймворком Akka, вы не увидите ни строчки кода на Scala, если не захотите. Чтобы узнать больше об акторах в целом и об Akka в частности, обращайтесь к книге Раймонда Ростенбурга (Raymond Roestenburg), Роба Баккера (Rob Bakker) и Роба Уильямса (Rob Williams) «Akka in Action» (Manning, 2016)[1].

[1] *Рестенбург Р., Баккер Р., Уильямс Р.* Akka в действии. М.: ДМК-Пресс, 2018. ISBN: 978-5-94074-616-4. – *Прим. перев.*

Итоги

- Акторы – это компоненты, которые асинхронно получают и обрабатывают сообщения по очереди.
- Общее изменяемое состояние можно абстрагировать в акторах.
- Абстрагирование общего изменяемого состояния освобождает от проблем синхронизации и параллельной обработки.
- Модель акторов основана на асинхронном обмене сообщениями и является хорошим дополнением к функциональному программированию.
- Модель акторов обеспечивает простое и безопасное распараллеливание.
- Изменения абстрагируются инфраструктурой в акторах и невидимы, с точки зрения программиста.
- Программистам на Kotlin доступны несколько инфраструктур поддержки акторов.
- Akka – один из наиболее часто используемых фреймворков акторов, доступных программистам на Kotlin.

14 Решение типичных проблем функциональным способом

Эта глава охватывает следующие темы:

- использование утверждений;
- автоматические повторные вызовы функций или эффектов в случае неудачи;
- чтение файлов свойств;
- адаптация императивных библиотек;
- преобразование императивных программ;
- повторное выполнение эффектов.

Теперь в вашем арсенале имеется много инструментов, способных упростить вашу жизнь программиста и предлагающих безопасные методы функционального программирования. Но знания инструментов недостаточно. Чтобы добиться максимального эффекта от использования функциональных приемов, они должны стать вашей второй натурой. Вы должны научиться думать функционально. Так же как объектно-ориентированные (ОО) программисты думают в терминах классов, функциональные программисты должны думать в терминах функций.

Приступая к решению задачи, ОО-программисты ищут подходящие классы и пытаются выразить задачу в виде комбинации классов. После этого им остается только реализовать классы и объединить их.

Функциональные программисты делают то же самое с функциями, но с одним отличием: выяснив, какую функцию можно использовать для решения проблемы, они не переопределяют ее. Ее можно использовать

повторно, потому что функции, в отличие от классов, являются кодом многократного использования.

Функциональные программисты пытаются свести каждую задачу к комбинации ранее реализованных функций. Это не всегда возможно, поэтому иногда им приходится писать новые функции. Но эти новые функции становятся частью их арсенала. Основное отличие здесь состоит в том, что функциональные программисты всегда ищут абстракции, потому что именно абстракции делают функции многократно используемыми.

Все программисты используют библиотеки, предлагающие более или менее одинаковые возможности. Библиотеки обобщают решение задач и позволяют повторно использовать код, а не изобретать колесо снова и снова, когда требуется решить новую задачу. Разница заключается лишь в уровне абстракции. Преждевременная абстракция считается грехом в объектно-ориентированном программировании, тогда как в функциональном программировании это один из фундаментальных инструментов. Абстракции позволяют программисту не только повторно использовать функции, но и лучше понимать истинную природу возникающих проблем.

В этой главе мы рассмотрим некоторые типичные проблемы, которые программистам приходится решать в повседневной работе. Вы увидите, как можно решать эти проблемы, используя парадигму функционального программирования. Кроме изучения функциональных способов решения повседневных проблем, вам также часто придется использовать императивный код. Но как лучше использовать такой код в функциональных программах? Чтобы ответить на этот вопрос, мы возьмем простую императивную программу и изменим ее, сделав более эффективной и полезной.

14.1 Утверждения и проверка данных

Утверждения используются для проверки таких инвариантов, как предусловия, постусловия, условия управления потоком выполнения и условия в классах. В функциональном программировании непосредственное управление потоком выполнения обычно не используется, а классы часто неизменяемы, поэтому единственными условиями для проверки остаются предусловия и постусловия. По тем же причинам (неизменяемость и отсутствие управления потоком выполнения) эти проверки заключаются в проверке аргументов методов и функций и их результатов перед возвратом. Проверки значений аргументов, например, необходимы в таких частичных функциях, как эта:

```
fun inverse(x: Int): Double = 1.0 / x
```

Эта функция возвращает действительное значение для любого аргумента, кроме `0`, для которого она возвращает бесконечность. Поскольку с этим значением ничего нельзя сделать, вы, вероятно, предпочтете об-

рабатывать его каким-то особым образом. В императивном программировании это можно реализовать так:

```
fun inverse(x: Int): Double {
    assert(x == 0)
    return 1.0 / x
}
```

В этом фрагменте используется стандартное Java-утверждение `assert`, которое доступно и в Kotlin. Но проверку утверждений можно запретить во время выполнения, и тогда, чтобы запретить запуск программы с отключенными проверками утверждений, можно использовать такой код:

```
if (!Thread.currentThread().javaClass.desiredAssertionStatus()) {
    throw RuntimeException("Asserts must be enabled!!!")
}
```

ПРИМЕЧАНИЕ Если вы пользуетесь старой версией IntelliJ, утверждения могут быть включены по умолчанию. В таких случаях, чтобы сымитировать нормальное выполнение, можно явно отключить проверку утверждений, добавив параметр `-da` виртуальной машины в вашей конфигурации. В этом конкретном случае проще написать:

```
fun inverse(x: Int): Double = when (x) {
    0 -> throw IllegalArgumentException("div. By 0")
    else -> 1.0 / x
}
```

Для большей безопасности эту функцию следует сделать более обобщенной:

```
fun inverse(x: Int): Result<Double> = when (x) {
    0 -> Result.failure("div. By 0")
    else -> Result(1.0 / x)
}
```

Еще более обобщенная форма утверждения состоит из проверки соответствия аргумента определенному условию и возврата `Result.Failure`, если условие не соблюдается, и `Result.Success` – в противном случае. Возьмем в качестве примера функцию-оператор `invoke` из типа `Person`:

```
class Person private constructor(val id: Int,
                                 val firstName: String,
                                 val lastName: String) {

    companion object {
        operator
        fun invoke(id: Int?,
                   firstName: String?,
                   lastName: String?): Person =
            Person(id, firstName, lastName)
    }
}
```

Эту функцию можно использовать с данными, извлекаемыми из базы данных:

```
val person = Person(rs.getInt("personId"),
                    rs.getString("firstName"),
                    rs.getString("lastName"))
```

В таких случаях желательно проверить данные перед вызовом функции. Например, можно потребовать, чтобы числовой идентификатор `id` был положительным числом, а имя `firstName` и фамилия `lastName` не были пустыми строками или `null` и начинались с заглавной буквы. В императивном программировании такие проверки можно выполнить с помощью специальных функций-утверждений:

```
class Person private constructor(val id: Int,
                                 val firstName: String,
                                 val lastName: String) {

    companion object {
        operator
        fun invoke(id: Int?, firstName: String?, lastName: String?) =
            Person(assertPositive(id, "null or negative id"),
                   assertValidName(firstName, "invalid first name"),
                   assertValidName(lastName, "invalid last name"))

        private fun assertPositive(i: Int?,
                                   message: String): Int = when (i) {
            null -> throw IllegalStateException(message)
            else -> i
        }

        private fun assertValidName(name: String?,
                                    message: String): String = when {
            name == null
                || name.isEmpty()
                || name[0].toInt() < 65
                || name[0].toInt() > 91 ->
                                throw IllegalStateException(message)
            else -> name
        }
    }
}
```

Но чтобы программа была безопасной, код не должен возбуждать исключений. Вместо этого для обработки ошибок следует использовать специальные контексты, такие как `Result`. Такие проверки абстрагированы в типе `Result`. Вам остается только написать проверяющие функции, т. е. функции, использующие ссылки на функции. Обобщенные функции проверки можно определить на уровне пакета:

```
fun isPositive(i: Int?): Boolean = i != null && i > 0

fun isValidName(name: String?): Boolean =
    name != null && name[0].toInt() >= 65 && name[0].toInt() <= 91
```

И затем использовать их для проверки данных:

```
class Person private constructor(val id: Int,
                                 val firstName: String,
                                 val lastName: String) {

    companion object {
        fun of(id: Int, firstName: String, lastName: String) =
            Result.of(::isPositive, id, "Negative id").flatMap { validId ->
                Result.of(::isValidName, firstName, "Invalid first name")
                    .flatMap { validFirstName ->
                        Result.of(::isValidName, lastName, "Invalid last name")
                            .map { validLastName ->
                                Person(validId, validFirstName, validLastName)
                            }
            }
        }
    }
}
```

Этот код можно упростить еще больше, добавив еще более общие функции:

```
fun assertPositive(i: Int, message: String): Result<Int> =
    Result.of(::isPositive, i, message)

fun assertValidName(name: String, message: String): Result<String> =
    Result.of(::isValidName, name, message)
```

и затем использовать их:

```
fun of(id: Int, firstName: String, lastName: String) =
    assertPositive(id, "Negative id")
        .flatMap { validId ->
            assertValidName(firstName, "Invalid first name")
                .flatMap { validFirstName ->
                    assertValidName(lastName, "Invalid last name")
                        .map { validLastName ->
                            Person(validId, validFirstName, validLastName)
                        }
                }
        }
```

В листинге 14.1 представлено еще несколько примеров функций, реализующих проверки.

Листинг 14.1 Примеры проверки условий функциональным способом

```
fun <T> assertCondition(value: T, f: (T) -> Boolean): Result<T> =
    assertCondition(value, f,
        "Assertion error: condition should evaluate to true")

fun <T> assertCondition(value: T, f: (T) -> Boolean,
                        message: String): Result<T> =
    if (f(value))
        Result(value)
```

```
    else
        Result.failure(IllegalStateException(message))

fun assertTrue(condition: Boolean,
        message: String = "Assertion error: condition should be true"):
        Result<Boolean> =
    assertCondition(condition, { x -> x }, message)

fun assertFalse(condition: Boolean,
        message: String = "Assertion error: condition should be false"):
        Result<Boolean> =
    assertCondition(condition, { x -> !x }, message)

fun <T> assertNotNull(t: T): Result<T> =
    assertNotNull(t, "Assertion error: object should not be null")

fun <T> assertNotNull(t: T, message: String): Result<T> =
    assertCondition(t, { x -> x != null }, message)

fun assertPositive(value: Int,
        message: String = "Assertion error: value $value must be positive"):
        Result<Int> =
    assertCondition(value, { x -> x > 0 }, message)

fun assertInRange(value: Int, min: Int, max: Int): Result<Int> =
    assertCondition(value, { x -> x in min..(max - 1) },
        "Assertion error: value $value should be > $min and < $max")

fun assertPositiveOrZero(value: Int,
        message: String = "Assertion error: value $value must not be < 0"):
        Result<Int> =
    assertCondition(value, { x -> x >= 0 }, message)

fun <A: Any> assertType(element: A, clazz: Class<*>): Result<A> =
    assertType(element, clazz,
               "Wrong type: ${element.javaClass}, expected: ${clazz.name}")

fun <A: Any> assertType(element: A, clazz: Class<*>,
        message: String): Result<A> =
    assertCondition(element, { e -> e.javaClass == clazz }, message)
```

14.2 Повторный вызов функций и эффектов

Нечистые функции и эффекты часто требуется вызывать повторно, если они потерпели неудачу при первом вызове. Обычно неудача сопровождается возбуждением исключения, что очень осложняет повторные попытки выполнить функцию.

Представьте, что мы читаем данные из некоторого устройства, которое может возбудить исключение `IOException`, если окажется не готово к операции чтения. Допустим, что нам нужно выполнить три попытки с задержкой 100 мс перед каждой. Вот как может выглядеть императивное решение:

```
fun get(path: String): String = ①
    Random().nextInt(10).let {
```

```
        when {
            it < 8 -> throw IOException("Error accessing file $path")
            else -> "content of $path"
        }
    }

var retries = 0
var result: String? = null
(0 .. 3).forEach rt@ { ②
    try {
        result = get("/my/path")
        return@rt ③
    } catch(e: IOException) {
        if (retries < 3) {
            Thread.sleep(100) ④
            retries += 1
        } else {
            throw e ⑤
        }
    }
} println(result)
```

① Эта функция имитирует устройство, которое генерирует исключение в 80 % случаев
② Используется как параметр функции `forEach`; `rt@` определяет точку, куда должно вернуться управление из внутренней функции
③ `return@rt` выполняет локальный возврат из внутренней функции, переданной в параметре функции `forEach`, если функция `get` завершилась успехом
④ Если `get` сгенерировала исключение и число попыток `retries` меньше трех, выполняется повторная попытка прочитать значение через 100 мс
⑤ Если `get` сгенерировала исключение и число попыток `retries` не меньше трех, повторно сгенерировать исключение

Это не лучшее решение, потому что:

- вынуждает использовать `var`-ссылки;
- вынуждает использовать тип с поддержкой `null` для представления результата;
- не поддерживает возможность повторного использования, хотя идея повторных попыток используется довольно широко.

Хотелось бы иметь функцию `retry`, принимающую следующие параметры:

- функцию, вызов которой может потребоваться повторить;
- максимальное число попыток;
- задержка между попытками.

Эта функция не должна генерировать никаких исключений. Вместо этого она должна возвращать `Result`. Вот ее сигнатура:

```
fun <A, B> retry(f: (A) -> B,
                 times: Int,
                 delay: Long = 10): (A) -> Result<B>
```

Наличие этой функции позволит писать такой код:

```
val functionWithRetry = retry(::get, 10, 100)
functionWithRetry("/my/file.txt")
    .forEach({ println(it) }, { println(it.message) })
```

Реализовать такую функцию можно разными способами. Один из них – использовать операцию свертки по диапазону от 0 до максимального числа попыток, которая использует вычисления по короткой схеме, т. е. прерывает цикл, как только вызов `get` завершится успехом. Это легко реализовать, использовав метку и стандартную функцию `fold` с диапазоном:

```
fun <A, B> retry(f: (A) -> B, times: Int, delay: Long = 10) = rt@
    { a: A ->
        (0 .. times).fold("Not executed") { _, n ->
            try {
                print("Try $n: ")
                return@rt "Success $n: ${f(a)}"
            } catch (e: Exception) {
                Thread.sleep(delay)
                "${e.message}"
            }
        }
    }
```

Однако эта версия не будет работать с функцией `range`, которую мы разработали на основе класса `List` в упражнении 8.19. Это, скорее всего, связано с ошибкой в Kotlin, в любом случае она не будет компилироваться[1]. Решить проблему можно с использованием явный сорекурсии, как рассказывалось в главе 4. Это, как несложно догадаться, подразумевает определение вспомогательной локальной функции:

```
fun <A, B> retry(f: (A) -> B,
                 times: Int,
                 delay: Long = 10): (A) -> Result<B> {
    fun retry(a: A, result: Result<B>, e: Result<B>, tms: Int): Result<B> =
        result.orElse {
            when (tms) {
                0 -> e
                else -> {
                    Thread.sleep(delay)
                    // вывести номер попытки
                    println("retry ${times - tms}")
                    retry(a, Result.of { f(a) }, result, tms - 1)
                }
            }
        }
    return { a -> retry(a, Result.of { f(a) }, Result(), times - 1)}
}
```

[1] Подробнее об этой ошибке можно прочитать в отчете о проблеме KT-24055, «Incorrect use of label with local return cause internal exception in the compiler»: https://youtrack.jetbrains.com/issue/KT-24055.

Эта реализация использует локальную функцию, которая рекурсивно вызывает себя с уменьшенным числом повторных попыток, пока это число не станет равным 0 или вызов функции `f` не завершится успехом. Мы не можем использовать здесь ключевое слово `tailrec`, потому что Kotlin не считает эту функцию рекурсивной. Однако это не проблема, потому что количество повторных попыток едва ли будет настолько большим, чтобы вызвать переполнение стека. Инструкция `println` включена в реализацию, только чтобы вы могли видеть, что происходит.

Локальная функция первоначально вызывается с `Result.of {f (a)}`, что несколько необычно. Обычно локальная функция вызывается с теми же параметрами, что и основная, плюс дополнительные параметры. Здесь мы имеем дело с особым случаем использования – задержка перед первой попыткой не должна выполняться.

С помощью этой функции вы сможете преобразовать любую функцию в функцию, автоматически выполняющую повторные попытки. Ее также можно использовать с чистыми эффектами (возвращающими `Unit`), как в следующем примере:


```
fun show(message: String) =
    Random().nextInt(10).let {
        when {
            it < 8 -> throw IllegalStateException("Failure !!!")
            else -> println(message)
        }
    }

fun main(args: Array<String>) {
    retry(::show, 10, 20)("Hello, World!").forEach(onFailure =
        { println(it.message) })
}
```

14.3 Чтение свойств из файла

Большинство приложений настраивается с использованием файлов свойств, которые читаются в момент запуска. Свойства – это пары ключ/значение, где ключи и значения записываются в виде строк. Независимо от выбранного формата определения свойств (ключ = значение, XML, JSON, YAML и т. д.) программист всегда должен читать строки и преобразовывать их в объекты. Этот процесс утомителен и сопряжен с ошибками.

Для чтения файлов свойств можно использовать специализированную библиотеку, но, если что-то пойдет не так, вы столкнетесь с исключениями. Чтобы определить более функциональное поведение, придется написать свою библиотеку. Это позволит вам:

- читать свойства как строки;
- читать свойства как числовые значения разных типов;
- читать свойства как перечисления или даже как значения произвольных типов;

- читать свойства как коллекции перечисленных выше типов;
- определять значения по умолчанию для свойств и осмысленные сообщения об ошибках, если что-то пойдет не так;
- читать свойства без генерации исключений.

14.3.1 Загрузка файла со свойствами

Независимо от формата файла процесс загрузки всегда будет выглядеть одинаково: чтение файла и обработка любых исключений, которые могут возникнуть в этом процессе. Первым делом нужно прочитать файл и вернуть `Result<Properties>`, как показано в листинге 14.2.

Листинг 14.2 Чтение файла со свойствами

```
class PropertyReader(configFileName: String) {

    internal val properties: Result<Properties> = ①
        Result.of { ②
            MethodHandles.lookup().lookupClass() ③
                .getResourceAsStream(configFileName) ④
                .use { inputStream -> ⑤
                    Properties().let {
                        it.load(inputStream) ⑥
                        it ⑦
                    }
                }
        }
}

fun main(args: Array<String>) {
    val propertyReader =
        PropertyReader("/config.properties") ⑧
    propertyReader.properties.forEach(onSuccess =
            { println(it) }, onFailure = { println(it) })
}
```

① Класс `PropertyReader` хранит `Result<Properties>`, откуда потом будут читаться свойства и их значения
② Функция `Result.of` вернет `Success`, если все прошло успешно, и `Failure`, если возникла какая-то ошибка
③ Этот вызов позволяет получить ссылку на сгенерированный класс, хотя функция определена на уровне пакета
④ Загрузит файл, находящийся в пути поиска классов (classpath)
⑤ Загрузит файл со свойствами; может вызвать исключение `IOException`
⑥ Загрузит свойства из `InputStream`
⑦ Последняя строка в блоке возвращает значение. Для ссылки на значение необходимо использовать имя параметра по умолчанию «it», потому что предыдущая строка вернет `Boolean`
⑧ Здесь указывается, что файл находится в корневом каталоге в пути поиска классов (classpath)

Если файл не будет найден, функция `use` не сгенерирует `IOException`, а вернет `null` вместо `inputStream`, что приведет к исключению `NullPointerException`. Эта функция гарантирует, что поток будет закрыт в любом случае.

ПРИМЕЧАНИЕ Если вы используете Intellij, вам потребуется пересобрать проект перед запуском примера. Если запустить его без перекомпиляции, классы будут созданы, но ресурсы не будут скопированы в выходной каталог. Причина проблемы в том, что мы загружаем файл свойств из пути поиска классов. Однако его можно загрузить из любого места на диске, из удаленного URL-адреса или из любого другого источника.

14.3.2 Чтение свойств как строк

При работе с файлами свойств самый простой случай – чтение свойств в виде строк. Это кажется просто, но помните, что следующий код не будет работать:

```
properties.map { it.getProperty("name") }
```

Если свойства с таким именем не существует, функция `getProperty` вернет `null`, что в результате даст `Success("null")`. Класс `Properties` можно сконструировать со списком значений свойств по умолчанию, и сама функция `getProperty` может вызываться со значением по умолчанию. Но не все свойства имеют значения по умолчанию. Чтобы решить эту проблему, нужно использовать функцию `flatMap` вместе с `Result.of`:

```
fun readProperty(name: String) =
    properties.flatMap {
        Result.of {
            it.getProperty(name)
        }
    }
```

Теперь представим, что у нас есть файл свойств со следующим содержимым:

```
host=acme.org
port=6666
name=
temp=71.3
price=$45
list=34,56,67,89
person=id:3;firstName:Jeanne;lastName:Doe
id=3
type=SERIAL
```

Пусть этот файл имеет имя `config.properties` и находится в корневом каталоге в пути поиска классов. Мы можем безопасно читать эти свойства, как показано ниже:

```
fun main(args: Array<String>) {
    val propertyReader = PropertyReader("/config.properties")
    propertyReader.readProperty("host")
        .forEach(onSuccess = { println(it) }, onFailure = { println(it) })

    propertyReader.readProperty("name")
        .forEach(onSuccess = { println(it) }, onFailure = { println(it) })
```

```
    propertyReader.readProperty("year")
        .forEach(onSuccess = { println(it) }, onFailure = { println(it) })
}
```

Он выведет следующее:

```
acme.org

java.lang.NullPointerException
```

Первая строка соответствует свойству `host` и содержит верное значение. Вторая строка соответствует свойству `name`. Это свойство в файле содержит пустую строку, что может быть правильно или неправильно – этого мы не знаем. Возможно, с точки зрения бизнес-логики имя является необязательным свойством.

Третья строка соответствует отсутствующему свойству `year`, но сообщение мало о чем говорит. Это сообщение содержится в значении типа `Result<String>`, которое можно присвоить переменной `year`, чтобы знать, какое свойство отсутствует. Но было бы лучше вставить имя свойства в сообщение. Давайте сделаем это сообщение об ошибке более полезным.

14.3.3 Вывод более информативных сообщений об ошибках

Проблема, с которой мы сталкиваемся, – наглядный пример того, чего никогда не должно происходить. Kotlin использует стандартную библиотеку Java, поэтому мы уверены, что все пойдет так, как ожидалось. В частности, мы ожидаем, что, если файл не найден или его нельзя прочитать, мы получим исключение `IOException`. Можно даже надеяться, что нам сообщат полный путь к файлу, потому что часто ошибка обусловлена тем, что файл был помещен не в тот каталог. Хорошим сообщением об ошибке в таком случае было бы: «Поиск файла ‘abc’ в папке ‘xyz’ потерпел неудачу». Теперь посмотрим, что делает Java-метод `getResourceAsStream`:

```
public InputStream getResourceAsStream(String name) {
    URL url = getResource(name);
    try {
        return url != null ? url.openStream() : null;
    } catch (IOException e) {
        return null;
    }
}
```

Да, вот так этот метод написан на Java. Урок на будущее: никогда не используйте методы из стандартной библиотеки Java, не посмотрев, как они реализованы!

В документации Javadoc говорится, что метод возвращает «поток ввода для чтения ресурса или `null`, если ресурс не найден». Это означает, что многое может пойти не так. Метод может сгенерировать исключение `IOException`, если не найдет файл или не сможет прочитать его. Или имя файла может оказаться значением `null`. Или метод `getResource` может возбудить исключение или вернуть `null`. (Загляните в реализацию метода, и вы поймете, о чем я говорю.)

Мы как минимум должны предусмотреть отдельное сообщение для каждого из этих случаев. И даже если появление исключения `IOException` весьма маловероятно, мы должны предусмотреть этот вариант развития событий, а также вариант появления любого другого исключения, как показано в листинге 14.3.

Листинг 14.3 Создание информативных сообщений об ошибках

```
// теперь properties можно объявить приватным
private val properties: Result<Properties> =
    try {
        MethodHandles.lookup().lookupClass()
            .getResourceAsStream(configFileName)
            .use { inputStream ->
                when (inputStream) {
                    null ->
                        Result.failure(
                            "File $configFileName not found in classpath")
                    else -> Properties().let {
                        it.load(inputStream)
                        Result(it)
                    }
                }
            }
    } catch (e: IOException) {
        Result.failure(
            "IOException reading classpath resource $configFileName")
    } catch (e: Exception) {
        Result.failure("Exception: ${e.message} " +
                " while reading classpath resource $configFileName")
    }
```

Если файл не будет найден, мы получим сообщение:

```
File /config.properties not found in classpath
```

Нам также нужно предусмотреть сообщения об ошибках, имеющие отношение к свойствам. Например, если следующий код:

```
val year: Result<String> = propertyReader.readProperty(properties, "year")
```

получит сообщение `NullPointerException`, мы поймем, какое свойство отсутствует в файле. Но в следующем примере мы не сможем этого понять:

```
data class Person(val id: Int, val firstName: String, val lastName:String)

fun main(args: Array<String>) {
    val propertyReader = PropertyReader("/config.properties")
    val person = propertyReader.readProperty("id")
        .map(String::toInt)
        .flatMap { id ->
            propertyReader.readProperty("firstName")
                .flatMap { firstName ->
                    propertyReader.readProperty("lastName")
                        .map { lastName -> Person(id, firstName, lastName) }
```

```
                }
            }
        person.forEach(onSuccess = { println(it) },
                       onFailure = { println(it) })
    }
```

Решить эту проблему можно несколькими способами. Самый простой – отобразить ошибку во вспомогательной функции `readProperty` в классе `PropertyReader`:


```
fun readProperty(name: String) =
    properties.flatMap {
        Result.of {
            it.getProperty(name)
        }.mapFailure("Property \"$name\" no found")
    }
```

Предыдущий пример сгенерирует следующее сообщение об ошибке, явно указывающее, что свойство `id` отсутствует в файле:

```
java.lang.RuntimeException: Property "firstName" not found
```

Другой потенциальный источник ошибок – операция преобразования строкового свойства `id` в числовое значение. Например, если свойство определить так:

```
id=three
```

тогда мы получим сообщение об ошибке:

```
java.lang.NumberFormatException: For input string: "three"
```

Это стандартное для Java сообщение об ошибке парсинга недостаточно информативно для нас. Так выглядит большинство стандартных сообщений об ошибках в Java. Это похоже на `NullPointerException`. Оно говорит, что ссылка признана недействительной, но не указывает, какая именно. Нам нужно указать имя свойства, вызвавшего исключение, например:

```
propertyReader.readProperty("id")
    .map(String::toInt)
    .mapFailure("Invalid format for property \"id\": ???")
```

Здесь мы написали имя свойства дважды, а кроме того, было бы хорошо заменить `???` значением, прочитанным из файла. (Здесь это невозможно, так как значение уже потеряно.) Поскольку нам придется выполнять парсинг для всех нестроковых значений свойств, мы должны абстрагировать эту операцию внутри класса `PropertyReader`. Для этого сначала переименуем функцию `readProperty`:

```
fun readAsString(name: String) =
    properties.flatMap {
        Result.of {
            it.getProperty(name)
        }.mapFailure("Property \"$name\" no found")
    }
```

Затем добавим функцию readAsInt:

```
fun readAsInt(name: String): Result<Int> =
    readAsString(name).flatMap {
        try {
            Result(it.toInt())
        } catch (e: NumberFormatException) {
            Result.failure<Int>(
                "Invalid value while parsing property '$name' to Int: '$it'")
        }
    }
```

Теперь нам не придется волноваться об ошибках преобразования строк в целые числа:

```
val person = propertyReader.readAsInt("id")
    .flatMap { id ->
        propertyReader.readAsString("firstName")
            .flatMap { firstName ->
                propertyReader.readAsString("lastName")
                    .map { lastName -> Person(id, firstName, lastName) }
            }
    }
    person.forEach(onSuccess = { println(it) }, onFailure = { println(it)
})
```

Если во время парсинга свойства id возникнет исключение, мы получим сообщение:

```
java.lang.IllegalStateException:
Invalid value while parsing property 'id' to Int: 'three'")
```

14.3.4 Чтение свойств как списков

Все, что можно делать с целыми числами, можно делать и с другими числовыми типами, такими как Long или Double. Но мы можем читать не только свойства с единственными значениями, но и с целыми списками, например:

```
list=34,56,67,89
```

Для этого достаточно добавить специализированную функцию. Например, вот функция, которая читает свойство как список целых чисел:

```
fun readAsIntList(name: String): Result<List<Int>> =
    readAsString(name).flatMap {
        try {
            Result(fromSeparated(it, ",").map(String::toInt))
        } catch (e: NumberFormatException) {
            Result.failure<List<Int>>(
            "Invalid value while parsing property '$name' to List<Int>: '$it'")
        }
    }
```

Здесь используется функция fromSeparated, объявленная в классе List, в модуле com.fpinkotlin.common. Этот модуль доступен в репозитории

с примерами для книги (https://github.com/pysaumont/fpinkotlin). Код можно изменить и задействовать в нем функцию из стандартного класса `List` в Kotlin:

```
fun readAsIntList(name: String): Result<List<Int>> =
    readAsString(name).flatMap {
        try {
            // В следующей строке используется класс List из Kotlin
            Result(it.split(",").map(String::toInt))
        } catch (e: NumberFormatException) {
            Result.failure<List<Int>>(
                "Invalid value while parsing property '$name' to List<Int>: $it")
        }
    }
```

Но и это еще не все! Мы можем читать свойства как списки с числовыми значениями любого типа, предусмотрев передачу соответствующей функции преобразования:

```
fun <T> readAsList(name: String, f: (String) -> T): Result<List<T>> =
    readAsString(name).flatMap {
        try {
            Result(fromSeparated(it, ",").map(f))
        } catch (e: Exception) {
            Result.failure<List<T>>(
                "Invalid value while parsing property '$name' to List: $it")
        }
    }
```

и определив специализированные функции для всех числовых типов, использующие `readAsList`:

```
fun readAsIntList(name: String): Result<List<Int>> =
    readAsList(name, String::toInt)

fun readAsDoubleList(name: String): Result<List<Double>> =
    readAsList(name, String::toDouble)

fun readAsBooleanList(name: String): Result<List<Boolean>> =
    readAsList(name, String::toBoolean)
```

На практике нередки случаи, когда приходится читать свойства, инициализированные значениями из перечислений. Это частный случай чтения свойства произвольного типа.

14.3.5 Чтение значений перечислений

Прежде всего нужно определить функцию преобразования свойства в произвольный тип `T`, которая принимает функцию `(String) -> Result<T>`:

```
fun <T> readAsType(f: (String) -> Result<T>, name: String) =
    readAsString(name).flatMap {
        try {
            f(it)
        } catch (e: Exception) {
```

```
            Result.failure<T>(
                "Invalid value while parsing property '$name': '$it'")
        }
    }
```

Теперь можно реализовать функцию `readAsEnum` в терминах `readAsType`:

```
inline
fun <reified T: Enum<T>> readAsEnum(name: String,
                                    enumClass: Class<T>): Result<T> {
    val f: (String) -> Result<T> = {
        try {
            val value = enumValueOf<T>(it)
            Result(value)
        } catch (e: Exception) {
            Result.failure("Error parsing property '$name': " +
                "value '$it' can't be parsed to ${enumClass.name}.")
        }
    }
    return readAsType(f, name)
}
```

Обратите внимание: ключевое слово `reified` означает, что тип `T` должен быть доступен во время выполнения. В отличие от Java Kotlin позволяет получить доступ к параметрам типа во время выполнения с использованием ключевого слова `reified`, предотвращающего стирание. Эта возможность доступна только в функциях, объявленных как `inline`, что означает, что вместо ссылки на функцию компилятор будет копировать в точку вызова код функции. Это увеличивает объем скомпилированного кода.

Для следующего свойства:

```
type=SERIAL
```

и определения перечисления:

```
enum class Type {SERIAL, PARALLEL}
```

мы можем прочитать значение свойства, как показано ниже:

```
val type = propertyReader.readAsEnum("type", Type::class.java)
```

До сих пор вы читали свойства как `String`, `Int`, `Double`, `Boolean`, списки и перечисления. Однако точно так же можно читать свойства, представляющие произвольные объекты. Для этого нужно записать свойства объекта в сериализованной форме, а затем загрузить их и десериализовать.

14.3.6 Чтение свойств произвольных типов

Для чтения свойств произвольного типа можно использовать функцию `getAsType`. Например, можно прочитать такое свойство и получить экземпляр класса `Person`:

```
person=id:3,firstName:Jane,lastName:Doe
```

Для этого требуется только представить функцию, преобразующую `String` в `Result<Person>`. Она должна уметь создавать экземпляры `Person` из строк вида `id:3,firstName:Jane,lastName:Doe`. Чтобы упростить ее использование, можно написать функцию `readAsPerson`. Но так как она выполняет преобразование в пользовательский тип, ее не следует помещать в класс `PropertyReader`. Лучше добавить функцию, принимающую `PropertyReader` и имя свойства, в класс `Person`.

Реализовать эту функцию можно несколькими способами. Один из них: получить свойство в виде списка, разбить его на элементы и поместить пары ключ/значение в ассоциативный массив, из которого без труда можно создать экземпляр `Person`. Другой способ: создать второй экземпляр `PropertyReader`, который будет читать свойства из строки после замены запятых символами перевода строки. В листинге 14.4 показан класс `Person` с двумя конкретными функциями для создания экземпляров из строки со свойствами.

Листинг 14.4 Методы чтения свойств с объектами и списками объектов

```
data class Person(val id: Int,
                  val firstName: String,
                  val lastName: String) {

  companion object {
    fun readAsPerson(propertyName: String,
                     propertyReader: PropertyReader): Result<Person> {
      val rString = propertyReader.readAsPropertyString(propertyName)
      val rPropReader = rString.map { stringPropertyReader(it) }
      return rPropReader.flatMap { readPerson(it) }
    }

    fun readAsPersonList(propertyName: String,
                         propertyReader: PropertyReader):
            Result<List<Person>> =
      propertyReader.readAsList(propertyName, { it }).flatMap { list ->
        sequence(list.map { s ->
          readPerson(PropertyReader
            .stringPropertyReader(PropertyReader.toPropertyString(s)))
        })
    }

    private fun readPerson(propReader: PropertyReader): Result<Person> =
      propReader.readAsInt("id")
        .flatMap { id ->
          propReader.readAsString("firstName")
            .flatMap { firstName ->
              propReader.readAsString("lastName")
                .map { lastName -> Person(id, firstName, lastName) }
            }
        }
  }
}
```

С помощью функции `readAsPersonList` можно читать свойства-векторы:

```
employees=\
  id:3;firstName:Jane;lastName:Doe,\
  id:5;firstName:Paul;lastName:Smith,\
  id:8;firstName:Mary;lastName:Winston
```

Эти функции требуют немного изменить класс `PropertyReader`, как показано в листинге 14.5.

Листинг 14.5 Функции, добавленные в класс PropertyReader

```
class PropertyReader(
        private val properties: Result<Properties>, ①
        private val source: String) { ②

    ...

    fun readAsPropertyString(propertyName: String): Result<String> = ③
        readAsString(propertyName).map { toPropertyString(it) }

    companion object {

        fun toPropertyString(s: String): String =
            s.replace(";", "\n") ④

        private
        fun readPropertiesFromFile(configFileName: String):
                Result<Properties> = ⑤
            try {
                MethodHandles.lookup().lookupClass()
                    .getResourceAsStream(configFileName)
                    .use { inputStream ->
                        when (inputStream) {
                            null -> Result.failure(
                                "File $configFileName not found in classpath")

                            else -> Properties().let {
                                it.load(inputStream)
                                Result(it)
                            }
                        }
                    }
            } catch (e: IOException) {
                Result.failure(
                    "IOException reading classpath resource $configFileName")
            } catch (e: Exception) {
                Result.failure("Exception: ${e.message}reading classpath"
                               + " resource $configFileName")
            }

        private fun readPropertiesFromString(propString: String):
                Result<Properties> = ⑥
            try {
                StringReader(propString).use { reader ->
                    val properties = Properties()
```

```
                properties.load(reader)
                Result(properties)
            }
        } catch (e: Exception) {
            Result.failure("Exception reading property string " +
                           "$propString: ${e.message}")
        }

    fun filePropertyReader(fileName: String): PropertyReader = ⑦
        PropertyReader(readPropertiesFromFile(fileName),
                       "File: $fileName")

    fun stringPropertyReader(propString: String): PropertyReader = ⑧
        PropertyReader(readPropertiesFromString(propString),
                       "String: $propString")
}
}
```

① Создается `PropertyReader` с `Result<Properties>`
② Источник `source` используется в сообщениях об ошибках
③ Преобразует значение свойства в строку со свойствами, которую потом можно передать вложенному экземпляру `PropertyReader`
④ Читает свойство и преобразует его значение в строку со свойствами
⑤ Оригинальная функция для чтения свойств из файла
⑥ Новая функция для чтения свойств из строки со свойствами
⑦ Создает экземпляр `PropertyReader` и передает ему имя файла
⑧ Создает экземпляр `PropertyReader` и передает ему строку со свойствами

Аналогично можно обрабатывать файлы свойств в формате XML, JSON или YAML.

14.4 Преобразование императивной программы: чтение файлов XML

Разработка новых функциональных программ для решения любых задач – увлекательное занятие, но у многих разработчиков нет на это времени. Им часто приходится использовать в своем коде существующие императивные решения. Это относится и к случаям использования существующих библиотек.

Возможно, вам будет интереснее начать с нуля и написать совершенно новое, на 100 % функциональное решение. Но мы должны быть реалистами. Часто у вас не будет времени или средств, чтобы воплотить такое решение, и вам придется использовать существующие нефункциональные библиотеки функций, возвращающих `null` и/или возбуждающих исключения, а также нечистых функций, изменяющих свои параметры во внешнем мире.

Вы очень скоро заметите, что, освоившись с приемами функционального программирования, вам будет трудно вернуться к старому императивному стилю. Обычно решение этой проблемы заключается в создании тонкой функциональной оболочки вокруг императивных библиотек.

В качестве примера рассмотрим универсальную библиотеку для чтения XML-файлов: JDOM 2.0.6. Эта библиотека широко используется в программах на Java и идеально подходит для Kotlin.

14.4.1 Шаг 1: императивное решение

Для начала рассмотрим пример программы в листинге 14.6. Эта программа взята из руководства по использованию библиотеки JDOM (http://www.mkyong.com/java/how-to-read-xml-file-in-java-jdom-example/). Я выбрал этот пример, потому что он содержит не так много кода и идеально умещается на книжной странице. Это программа на Java, использующая библиотеку на Java.

Листинг 14.6 Чтение XML-файла с помощью JDOM: версия на Java

```
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;
import java.io.File;
import java.io.IOException;
import java.util.List;

public class ReadXmlFile {

    public static void main(String[] args) {
        SAXBuilder builder = new SAXBuilder();
        File xmlFile = new File("path_to_file");
        try {
            Document document = (Document) builder.build(xmlFile);
            Element rootNode = document.getRootElement();
            List list = rootNode.getChildren("staff");
            for (int i = 0; i < list.size(); i++) {
                Element node = (Element) list.get(i);
                System.out.println("First Name : " +
                                   node.getChildText("firstname"));
                System.out.println("\tLast Name : " +
                                   node.getChildText("lastname"));
                System.out.println("\tNick Name : " +
                                   node.getChildText("email"));
                System.out.println("\tSalary : " +
                                   node.getChildText("salary"));
            }
        } catch (IOException io) {
            System.out.println(io.getMessage());
        } catch (JDOMException jdomex) {
            System.out.println(jdomex.getMessage());
        }
    }
}
```

Программу на Java легко можно переписать на Kotlin в том же императивном стиле, как показано в листинге 14.7.

Листинг 14.7 Чтение XML-файла с помощью JDOM: императивная версия на Kotlin

```
import org.jdom2.JDOMException
import org.jdom2.input.SAXBuilder
import java.io.File
import java.io.IOException

/**
 * Не тестируется, генерирует исключения.
 */
fun main(args: Array<String>) {

    val builder = SAXBuilder()
    val xmlFile = File("/path/to/file.xml") // Исправьте путь

    try {
        val document = builder.build(xmlFile)
        val rootNode = document.rootElement
        val list = rootNode.getChildren("staff")

        list.forEach {
            println("First Name: ${it.getChildText("firstName")}")
            println("\tLast Name: ${it.getChildText("lastName")}")
            println("\tEmail: ${it.getChildText("email")}")
            println("\tSalary: ${it.getChildText("salary")}")
        }
    } catch (io: IOException) {
        println(io.message)
    } catch (e: JDOMException) {
        println(e.message)
    }
}
```

Для опробования этого примера будем использовать XML-файл из листинга 14.8.

Листинг 14.8 XML-файл для опробования примера

```
<?xml version="1.0"?>
<company>
    <staff>
        <firstName>Paul</firstName>
        <lastName>Smith</lastName>
        <email>paul.smith@acme.com</email>
        <salary>100000</salary>
    </staff>
    <staff>
        <firstName>Mary</firstName>
        <lastName>Colson</lastName>
        <email>mary.colson@acme.com</email>
        <salary>200000</salary>
    </staff>
</company>
```

Давайте посмотрим, какие преимущества можно получить, переписав этот пример в функциональном стиле. Первая проблема, которую мы видим, – ни одну часть этой программы нельзя использовать повторно. Это всего лишь пример, но даже он должен быть написан с прицелом на возможность повторного использования, чтобы его можно было хотя бы протестировать. Единственный способ проверить эту программу – внимательно наблюдать за сообщениями в консоли, которые покажут либо ожидаемый результат, либо сообщение об ошибке. Как вы увидите далее, такой подход может привести к ошибочному результату.

14.4.2 Шаг 2: превращаем императивную программу в функциональную

Чтобы сделать эту программу более функциональной:

- перечислим основные необходимые нам функции;
- напишем эти функции как автономные модули, доступные для тестирования и повторного использования;
- перепишем пример с использованием этих функций.

Вот основные функции, которые нам потребуются:

- для чтения файла и возврата содержимого в формате XML в виде строки;
- для преобразования XML-строки в список элементов;
- для преобразования списка элементов в список строковых представлений элементов.

Нам также понадобится эффект для вывода списка строк на экран.

ПРИМЕЧАНИЕ Это описание подходит только для небольших файлов, которые могут уместиться в памяти.

Первая функция, которая читает файл и возвращает его содержимое в формате XML в виде строки, должна иметь следующую сигнатуру:

```
fun readFile2String(path: String): Result<String>
```

Эта функция не должна возбуждать исключений; она возвращает `Result<String>`.

Вторая функция должна преобразовать XML-строку в список элементов, поэтому ей необходимо знать имя корневого элемента XML. Вот ее сигнатура:

```
fun readDocument(rootElementName: String,
                 stringDoc: String): Result<List<Element>>
```

Третья функция принимает список элементов и возвращает список строковых представлений элементов; она будет иметь следующую сигнатуру:

```
fun toStringList(list: List<Element>, format: String): List<String>
```

Наконец, нам потребуется применить эффект к данным, который определяется следующей сигнатурой:

```
fun <A> processList(list: List<A>)
```

Эта декомпозиция на функции мало отличается от того, что мы могли бы реализовать в императивном стиле. В конце концов, считается хорошей практикой разбивать императивные программы на функции, каждая из которых отвечает за что-то одно. Но на самом деле эта декомпозиция дает больше, чем может показаться.

Обратите внимание, что функция `readDocument` принимает в первом параметре строку, возвращаемую функцией, которая может (в императивном мире) вызвать исключение. По этой причине нам понадобится дополнительная функция:

```
fun getRootElementName(): Result<String>
```

Аналогично нам требуется функция, возвращающая путь к файлу:

```
fun getXmlFilePath(): Result<String>
```

Важно отметить, что типы аргументов и возвращаемых значений в этих функциях не совпадают! Это явно свидетельствуют о том, что императивные версии этих функций могли быть частичными функциями, т. е. они могут генерировать исключения. Функции, генерирующие исключения, плохо объединяются друг с другом. Наши функции, напротив, объединяются идеально.

Объединение функций и применение эффекта

Несмотря на то что типы аргументов и возвращаемых значений наших функций не совпадают, мы легко можем объединить их, использовав шаблон включения:

```
const val format = "First Name : %s\n" +
    "\tLast Name : %s\n" +
    "\tEmail : %s\n" +
    "\tSalary : %s"

fun main(args: Array<String>) {
    val path = getXmlFilePath()
    val rDoc = path.flatMap(::readFile2String)
    val rRoot = getRootElementName()
    val result = rDoc.flatMap { doc ->
        rRoot.flatMap { rootElementName ->
            readDocument(rootElementName, doc)
        }.map { list ->
            toStringList(list, format)
        }
    }
    ...
}
```

Применим соответствующий эффект, чтобы вывести результат:

```
result.forEach(onSuccess =
    { processList(it) }, onFailure = { it.printStackTrace() })
```

Эта функциональная версия программы намного чище и легко поддается тестированию. По крайней мере, так будет, когда мы реализуем все необходимые функции.

Реализация функций

Программа получилась элегантной, но нам еще нужно реализовать функции и эффекты, чтобы она заработала. Самое замечательное, что все функции выглядят просто и легко поддаются тестированию.

Сначала напишем функции `getXmlFilePath` и `getRootElementName`. В нашем примере их роль играют константы, которые желательно заменить конкретными реализациями в действующем приложении:

```
fun getRootElementName(): Result<String> =
    // Имитация вычислений, которые могут потерпеть неудачу.
    Result.of { "staff" }

fun getXmlFilePath(): Result<String> =
    Result.of { "file.xml" } // <- подставьте свой путь
```

Затем добавим функцию `readFile2String`. Вот одна из возможных реализаций:

```
fun readFile2String(path: String): Result<String> =
    Result.of { File(path).readText()
}
```

Теперь реализуем `readDocument`. Эта функция принимает строку с данными в формате XML и имя корневого элемента:

```
fun readDocument(rootElementName: String,
                 stringDoc: String): Result<List<Element>> =
    SAXBuilder().let { builder ->
        try {
            val document =
                builder.build(StringReader(stringDoc)) ①
            val rootElement = document.rootElement ②
            Result(List(*rootElement.getChildren(rootElementName)
                                    .toTypedArray())) ③
        } catch (io: IOException) {
            Result.failure("Invalid root element name '$rootElementName' "
                        + "or XML data $stringDoc: ${io.message}")
        } catch (jde: JDOMException) {
            Result.failure("Invalid root element name '$rootElementName' "
                        + "or XML data $stringDoc: ${jde.message}")
        } catch (e: Exception) {
            Result.failure("Unexpected error while reading XML data "
                        + "$stringDoc: ${e.message}")
        }
    }
```

① Может возбудить `NullPointerException`
② Может возбудить `IllegalStateException`
③ Символ звездочки (*) в начале выражения указывает, что получившийся в результате массив должен использоваться как список аргументов (vararg), а не как единственный объект

Первыми перехватываются исключения `IOException` (которое вряд ли будет сгенерировано, потому что чтение происходит из строки) и `JDOMException`, и в ответ на каждое из них возвращается признак ошибки с соответствующим сообщением. Но, заглянув в код с реализацией JDOM (я уже говорил, что не следует вызывать библиотечные методы, не посмотрев, как они реализованы), можно заметить, что он также может вызвать исключение `IllegalStateException` или `NullPointerException`. Поэтому далее мы перехватим также исключение `Exception`. Функция `toStringList` отображает список в функцию, отвечающую за преобразование:

```
fun toStringList(list: List<Element>, format: String): List<String> =
    list.map { e -> processElement(e, format) }

fun processElement(element: Element, format: String): String =
    String.format(format, element.getChildText("firstName"),
        element.getChildText("lastName"),
        element.getChildText("email"),
        element.getChildText("salary"))
```

И наконец, реализуем эффект, который будет применяться к результату:

```
fun <A> processList(list: List<A>) = list.forEach(::println)
```

14.4.3 Шаг 3: делаем программу еще более функциональной

Наша программа стала намного более модульной и простой в тестировании, и ее составляющие теперь можно повторно использовать в других проектах. Но мы можем сделать ее еще лучше. У нас все еще остаются четыре нефункциональных элемента:

- путь к файлу;
- имя корневого элемента;
- формат, используемый для преобразования элементов в строку;
- эффект, применяемый к результату.

Под словом «нефункциональные» я подразумеваю прямой доступ к этим элементам из реализаций наших функций, что делает их ссылочно-непрозрачными. Чтобы сделать программу полностью функциональной, мы должны превратить эти элементы в параметры программы.

Кроме того, функция `processElement` использует конкретные данные – имена элементов, соответствующие параметрам в строке формата, используемой для их отображения. Мы можем заменить строковый параметр `format` парой `Pair` (со строкой формата и списком параметров). В этом случае функция `processElement` будет выглядеть так:

```
fun toStringList(list: List<Element>,
                 format: Pair<String, List<String>>): List<String> =
```

```
    list.map { e -> processElement(e, format) }

fun processElement(element: Element,
                   format: Pair<String, List<String>>): String {
    val formatString = format.first
    val parameters = format.second.map { element.getChildText(it) }
    return String.format(formatString, *parameters.toArrayList().toArray())
}
```

Теперь мы можем превратить программу в чистую функцию, принимающую четыре аргумента и возвращающую новую (нефункциональную) выполняемую программу. Эта версия программы представлена в листинге 14.9.

Листинг 14.9 Полностью функциональная программа для чтения файлов XML

```
import com.fpinkotlin.common.List
import com.fpinkotlin.common.Result
import org.jdom2.Element
import org.jdom2.JDOMException
import org.jdom2.input.SAXBuilder
import java.io.File
import java.io.FileInputStream
import java.io.IOException
import java.io.StringReader

fun readXmlFile(sPath: () -> Result<String>,
                sRootName: () -> Result<String>,
                format: Pair<String, List<String>>,
                effect: (List<String>) -> Unit): () -> Unit { ①
    val path = sPath() ②
    val rDoc = path.flatMap(::readFile2String)
    val rRoot = sRootName() ③
    val result = rDoc.flatMap { doc ->
        rRoot.flatMap { rootElementName ->
            readDocument(rootElementName, doc) }
                .map { list -> toStringList(list, format) }
    }
    return {
        result.forEach(onSuccess = { effect(it) },
                       onFailure = { it.printStackTrace() }) ④
    }
}

fun readFile2String(path: String): Result<String> =
    Result.of { File(path).readText() }

fun readDocument(rootElementName: String,
                 stringDoc: String): Result<List<Element>> =
    SAXBuilder().let { builder ->
        try {
            val document = builder.build(StringReader(stringDoc))
            val rootElement = document.rootElement
            Result(List(*rootElement.getChildren(rootElementName)
```

```
                .toTypedArray()))
        } catch (io: IOException) {
            Result.failure("Invalid root element name '$rootElementName' "
                    + "or XML data $stringDoc: ${io.message}")
        } catch (jde: JDOMException) {
            Result.failure("Invalid root element name '$rootElementName' "
                    + "or XML data $stringDoc: ${jde.message}")
        } catch (e: Exception) {
            Result.failure("Unexpected error while reading XML data "
                    + "$stringDoc: ${e.message}")
        }
    }

fun toStringList(list: List<Element>,
                 format: Pair<String, List<String>>): List<String> =
    list.map { e -> processElement(e, format) }

fun processElement(element: Element,
                   format: Pair<String, List<String>>):
    String { ⑤
        val formatString = format.first
        val parameters = format.second.map { element.getChildText(it) }
        return String.format(formatString,
                             *parameters.toArrayList().toArray())
    }
```

① Путь к файлу и имя корневого элемента теперь принимаются как константные функции. Параметр `format` включает имена элементов. Наконец, функция принимает необязательный эффект типа `(List<String>) -> Unit`
② Вызов функций для получения значений параметров
③ Параметризует применяемый эффект
④ Функция возвращает программу, т. е. функцию типа `() -> Unit`, применяющую эффект из параметра к результату. Эта функция может возбудить исключение. Лучшего решения не существует, потому что это эффект, который ничего не возвращает
⑤ Функция `processElement` теперь стала универсальной

Теперь мы можем протестировать эту программу с помощью клиентского кода, представленного в листинге 14.10.

Листинг 14.10 Клиентская программа для тестирования программы чтения XML-файлов

```
import com.fpinkotlin.common.List
import com.fpinkotlin.common.Result

fun <A> processList(list: List<A>) = list.forEach(::println)

fun getRootElementName(): Result<String> =
    // Имитация вычислений, которые могут потерпеть неудачу.
    Result.of { "staff" }

fun getXmlFilePath(): Result<String> =
    Result.of { "/path/to/file.xml" } // <- подставьте свой путь

private val format = Pair("First Name : %s\n" +
                          "\tLast Name : %s\n" +
```

```
                        "\tEmail : %s\n" +
                        "\tSalary : %s",
                        List("firstName", "lastName", "email", "salary"))

fun main(args: Array<String>) {
    val program = readXmlFile( { getXmlFilePath() },
                               { getRootElementName() },
                               format, { processList(it) })
    program()
}
```

Эта программа не идеальна, потому что не обрабатывает потенциальные ошибки, которые могут возникнуть из-за неверных имен элементов. Например, если передать программе следующий фрагмент XML с неверным именем элемента:

```
<company>
    <staff>
        <firstname></firstname>
        <lastName>Smith</lastName>
        <email>paul.smith@acme.com</email>
        <salary>100000</salary>
    </staff>
    <staff>
        <firstname>Mary</firstname>
        <lastName>Colson</lastName>
        <email>mary.colson@acme.com</email>
        <salary>200000</salary>
    </staff>
</company>
```

она выведет:

```
First Name : null
    Last Name : Smith
    email : paul.smith@acme.com
    Salary : 100000
First Name : null
    Last Name : Colson
    email : mary.colson@acme.com
    Salary : 200000
```

Конечно, о причине ошибки можно догадаться, заметив, что все элементы `First Name` содержат `null`. Однако было бы лучше, если бы вместо слова *null* программа выводила более точное описание проблемы, содержащее ошибочное имя элемента. Еще одна важная проблема: если забыть добавить в список имя какого-либо элемента, функция `String.format` возбудит исключение, попытавшись выполнить следующий код:

```
val parameters = format.second.map { element.getChildText(it) }
return String.format(formatString, *parameters.toArrayList().toArray())
```

В этом случае массив `parameters` будет содержать только три элемента вместо ожидаемых четырех. И выявить эту ошибку по трассировке исключения будет очень трудно. Фактически истинная причина проблемы

заключается в том, что мы удалили все конкретные данные из функции `readXmlFile`, такие как имя корневого элемента, путь к файлу и применяемый эффект, но функция `processElement` по-прежнему зависит от бизнес-сценария клиента. Функция `readXmlFile` позволяет читать только элементы, которые являются прямыми потомками корневого элемента, извлекая некоторые значения из их прямых потомков (чьи имена передаются вместе с форматом).

Третья проблема заключается в том, что функция `readXmlFile` принимает два аргумента одного типа. Если по забывчивости поменять аргументы местами, это приведет к ошибке, которую компилятор просто не заметит. Мы легко можем решить эту проблему, поэтому займемся ею далее, а потом устраним первые две проблемы.

14.4.4 Шаг 4: исправление проблемы с аргументами одного типа

Третья проблема легко решается использованием типов-значений, как описывалось в главе 3. Вместо аргументов `Result<String>` мы можем использовать аргументы `Result<FilePath>` и `Result<ElementName>`. `FilePath` и `ElementName` – это классы-значения для строк:

```
data class FilePath private constructor(val value: Result<String>) {

    companion object {

        operator fun invoke(value: String): FilePath =
            FilePath(Result.of({ isValidPath(it) }, value,
                               "Invalid file path: $value"))

        // Замените кодом, осуществляющим проверку действительности
        private fun isValidPath(path: String): Boolean = true
    }
}
```

Класс `ElementName` имеет похожую реализацию. Обратите внимание, что вы должны добавить код, осуществляющий проверку действительности значения. Самый простой способ проверки – использовать регулярное выражение. После определения этих новых типов можно изменить функцию `readXmlFile`:

```
fun readXmlFile(sPath: () -> FilePath,
                sRootName: () -> ElementName,
                format: Pair<String, List<String>>,
                effect: (List<String>) -> Unit): () -> Unit {
    val path = sPath().value
    val rDoc = path.flatMap(::readFile2String)
    val rRoot = sRootName().value
```

Как видите, изменений очень немного. Клиентский класс также нужно изменить:

```
fun getRootElementName(): ElementName =
    // Имитация вычислений, которые могут потерпеть неудачу.
```

```
    ElementName("staff")

fun getXmlFilePath(): FilePath =
    FilePath("/path/to/file.xml") // <- подставьте свой путь
```

Теперь пользователь нашей функции не сможет поменять аргументы местами и не получить предупреждение от компилятора.

14.4.5 Шаг 5: передача функции обработки элемента в параметре

Две оставшиеся проблемы можно решить, организовав передачу функции обработки элемента в метод `readXmlFile` в виде параметра. При таком подходе эта функция будет решать единственную задачу: читать список элементов первого уровня в файле, применять к этому списку указанную функцию и возвращать результат. Основное отличие в том, что функция больше не будет генерировать список строк и применять к ним эффект. Нам нужно сделать функцию универсальной, поэтому внесем следующие изменения:

```
fun <T> readXmlFile(sPath: () -> FilePath, ①
                    sRootName: () -> ElementName,
                    function: (Element) -> T, ②
                    effect: (List<T>) -> Unit): () -> Unit { ③
    val path = sPath().value
    val rDoc = path.flatMap(::readFile2String)
    val rRoot = sRootName().value
    val result = rDoc.flatMap { doc ->
        rRoot.flatMap { rootElementName ->
            readDocument(rootElementName, doc) }
                .map { list -> list.map(function) } ④
    }
    return {
        result.forEach(onSuccess = { effect(it) },
                       onFailure = { throw it })
    }
}
```

① **Функция сделана обобщенной**
② **Аргумент `Pair<String, List<String>> format` заменила новая функция, она будет использоваться для преобразования списка элементов в список значений `T`**
③ **Эффект теперь параметризован типом `List<T>`**
④ **Функции `toStringList` и `processElement` были удалены. Их заменит функция из клиентского приложения**

Теперь нужно привести в соответствие клиентскую программу. Реализованное решение избавляет нас от необходимости использовать трюк с типом `Pair` для передачи строки формата и списка с именами элементов:

```
const val format = "First Name : %s\n" + ①
                   "\tLast Name : %s\n" +
                   "\tEmail : %s\n" +
                   "\tSalary : %s"
```

```
private val elementNames = ②
    List("firstName", "lastName", "email", "salary")

private fun processElement(element: Element): String = ③
    String.format(format, *elementNames.map { element.getChildText(it) }
        .toArrayList()
        .toArray())

fun main(args: Array<String>) {
    val program = readXmlFile(::getXmlFilePath,
                              ::getRootElementName,
                              ::processElement, ④
                              ::processList)
    ...
```

① Теперь формат снова стал обычной строкой
② Список элементов тоже определяется отдельно
③ Функция `processElement` теперь реализуется клиентом
④ Функция `processElement` передается в виде аргумента

Эффект `processList` не изменился. Но теперь функцию преобразования одного элемента и эффект для применения к нему должен предоставить клиент.

14.4.6 Шаг 6: обработка ошибок в именах элементов

Теперь нам осталось избавиться от ошибок, возникающих при чтении элементов. Функция, переданная в `readXmlFile`, возвращает простой тип, т. е. она должна быть полной (не частичной) функцией, но это не так. Она была полной функцией в первоначальном примере, когда в случае ошибки возвращала `null`. Теперь, когда мы используем функцию `(Element) -> T`, мы могли бы использовать `Result<String>` в качестве реализации `T`, но это было бы непрактично, потому что получали бы результат `List<Result<T>>` и нам пришлось бы преобразовать его в `Result<List<T>>`. Здесь нет ничего сложного, но это решение обязательно нужно абстрагировать.

Для решения задачи используем функцию `(Element) -> Result<T>` и функцию `sequence`, чтобы преобразовать результат в `Result<List<T>>`. Вот новая версия функции:

```
fun <T> readXmlFile(sPath: () -> FilePath,
                    sRootName: () -> ElementName,
                    function: (Element) -> Result<T>, ①
                    effect: (List<T>) -> Unit): () -> Unit {
    val path = sPath().value
    val rDoc = path.flatMap(::readFile2String)
    val rRoot = sRootName().value
    val result = rDoc.flatMap { doc ->
        rRoot.flatMap { rootElementName ->
            readDocument(rootElementName, doc) }
                .flatMap { list ->
                    sequence(list.map(function)) } ②
    }
    return {
```

```
        result.forEach(onSuccess = { effect(it) },
                       onFailure = { throw it })
    }
}
...
```

① Теперь `readXmlFile` принимает функцию типа `(Element) -> Result<T>`
② Результат преобразуется в последовательность, и на выходе получается `Result<List<T>>`. Функцию `map` заменила функция `flatMap`.

Единственное, что нам осталось сделать, – добавить обработку ошибки, которая может возникнуть в функции `processElement`. Чтобы лучше понять, что нужно сделать, еще раз заглянем в код метода `getChildText` из JDOM:

```
/**
 * Возвращает текстовое содержимое указанного дочернего элемента или null,
 * если такого элемента нет. Это более удобный метод, потому что вызов
 * <code>getChild().getText()</code> может сгенерировать NullPointerException.
 *
 * @param cname имя дочернего элемента
 * @return текстовое содержимое элемента или null, если такого элемента нет
 */
public String getChildText(final String cname) {
    final Element child = getChild(cname);
    if (child == null) {
        return null;
    }
    return child.getText();
}
```

Как видите, этот метод не генерирует никаких исключений, но, если элемент не существует, он вернет `null`. Учитывая все это, мы можем изменить свою функцию `processElement`:

```
fun processElement(element: Element): Result<String> = ①
    try {
        Result(String.format(format, *elementNames.map {
                        getChildText(element, it) }
            .toArrayList()
            .toArray()))
    } catch (e: Exception) {
        Result.failure(
            "Exception while formatting element. " + ②
            "Probable cause is a missing element name in element " +
            "list $elementNames")
    }

fun getChildText(element: Element,
                 name: String): String = ③
    element.getChildText(name) ?:
        "Element $name is not a child of ${element.name}"
```

① Теперь функция возвращает `Result<String>`
② В случае исключения при форматировании конструируется конкретное сообщение об ошибке
③ Если получено значение `null`, оно заменяется сообщением об ошибке

На данный момент большинство потенциальных ошибок обрабатывается функциональным способом, но не все ошибки можно обработать функционально. Как я уже говорил, исключения, создаваемые эффектом, который передается методу `readXmlFile`, нельзя обработать таким способом. Эти исключения генерирует программа, возвращаемая функцией. Когда функция возвращает программу, она еще не была выполнена. То есть эти исключения должны перехватываться при выполнении получившейся программы, например:

```
fun main(args: Array<String>) {
    val program = readXmlFile(::getXmlFilePath,
                              ::getRootElementName,
                              ::processElement,
                              ::processList)
    try {
        program()
    } catch (e: Exception) {
        println("An exception occurred: ${e.message}")
    }
}
```

Полный код вы найдете в репозитории с примерами для книги http://github.com/pysaumont/fpinkotlin.

14.4.7 Шаг 7: дополнительные улучшения в прежде императивном коде

Кто-то может заметить, что функция `processElement` замкнута на ссылках `format` и `elementNames`, и этот факт может показаться противоречащим функциональному стилю. (Если вы не помните, что такое замыкание, вернитесь в главе 3.) В данном примере это не является проблемой, потому что данные ссылки являются константами. Но в реальной жизни они, скорее всего, не будут константами.

Решаются подобные проблемы просто. Такие замыкания являются дополнительными неявными аргументами функции `processElement`. Проблема в том, что в отличие от определения функции эти два значения должны быть частью клиентской программы, т. е. функции `main`.

Мы могли бы перенести функцию `processElement` в функцию `main` (объявив ее как локальную), но это исключит возможность многократного ее использования. Решение состоит в том, чтобы использовать явный аргумент с каррированной `val`-функцией, как описано в главе 3:

```
val processElement: (List<String>) -> (String) -> (Element) ->
        Result<String> = { elementNames ->
    { format ->
        { element ->
            try {
                Result(String.format(format,
                    *elementNames.map { getChildText(element, it) }
                        .toArrayList()
```

```
                        .toArray()))
                } catch (e: Exception) {
                    Result.failure("Exception while formatting element. " +
                        "Probable cause is a missing element name in" +
                        " element list $elementNames")
                }
            }
        }
    }
```

Теперь мы можем вызвать `readXmlFile` с частично примененной версией функции `processElement`. Значения `format` и `elementNames` все так же определяются клиентской программой, но функция `processElement` осталась обобщенной и больше не замкнута на этих значениях. Вы можете поместить обобщенные функции `processElement` и `getChildText` в файл `ReadXmlFile.kt` и определить `processList`, `getRootElementName` и `getXmlFilePath` как локальные функции внутри `main`:

```
fun main(args: Array<String>) {

    fun <A> processList(list: List<A>) = list.forEach(::println)

    // Имитация вычислений, которые могут потерпеть неудачу.
    fun getRootElementName(): ElementName = ElementName("staff")

    fun getXmlFilePath(): FilePath =
        FilePath("/path/to/file.xml") // <- подставьте свой путь

    val format = "First Name : %s\n" +
                 "\tLast Name : %s\n" +
                 "\tEmail : %s\n" +
                 "\tSalary : %s"

    val elementNames =
        List("firstName", "lastName", "email", "salary")

    val program = readXmlFile(::getXmlFilePath,
                              ::getRootElementName,
                              processElement(elementNames)(format),
                              ::processList)

    try {
        program()
    } catch (e: Exception) {
        println("An exception occurred: ${e.message}")
    }
}
```

Это универсальное решение. Абстрагируя каждую возможную подзадачу в функцию, вы сделаете свои программы более простыми для тестирования и, как следствие, более надежными. Вы также сможете повторно использовать эти функции в других программах без повторного их тестирования. В приложении B я расскажу, как применить абстракцию к тестированию.

Итоги

- Помещение значений в контекст `Result` является функциональным эквивалентом утверждений.
- Контекст `Result` позволяет реализовать безопасное чтение файлов свойств.
- Функциональный подход к задаче чтения свойств освобождает от необходимости обрабатывать ошибки преобразования.
- Свойства, подобно значениям любых типов, перечислений или коллекций, можно читать абстрактным способом.
- Автоматическое повторение операций можно абстрагировать в виде функций.
- Императивные библиотеки можно заключать в функциональные обертки.

Приложение А
Смешивание кода на Kotlin и Java

Kotlin создавался как язык, работающий под управлением виртуальной машины Java (JVM). Затем компания JetBrains представила Kotlin JS, версию языка, работающего под управлением виртуальной машины JavaScript, а также Kotlin/Native, версию, компилирующую программы в двоичный машинный код, выполняющийся без какой-либо виртуальной машины. Однако подавляющее большинство программ на Kotlin пишется для выполнения в среде JVM.

Такие программы могут пользоваться всеми преимуществами стандартной библиотеки Java, т. е. вызывать любые методы из этой библиотеки. Для этого нет никаких ограничений, хотя вам, вероятно, не следует вызывать эти методы, не убедившись, что Kotlin не предлагает лучшего решения. Программы на Java тоже могут использовать библиотеки, написанные на Kotlin, хотя и не так легко. Kotlin предлагает гораздо больше, чем Java, и эти возможности, отсутствующие в Java, должны использоваться с особой осторожностью. Не все библиотеки доступны для программ на Java.

В практике совместного использования языков Java и Kotlin наиболее часто применяются два подхода: использование скомпилированных библиотек на Java из программ на Kotlin и использование скомпилированных библиотек на Kotlin из программ на Java. Но есть еще и третий путь: смешивание исходного кода на Kotlin и Java в одном проекте, хотя и с некоторыми ограничениями. В этом приложении я покажу вам:

- как создавать смешанные проекты на Kotlin и управлять ими с помощью Gradle;

- как вызывать методы Java из Kotlin;
- как вызывать функции Kotlin из Java;
- характерные проблемы, возникающие в смешанных проектах на Kotlin/Java.

ПРИМЕЧАНИЕ Примеры кода для этого приложения доступны в репозитории https://github.com/pysaumont/fpinkotlin, в папке `examples`.

Создание и управление смешанными проектами

Наиболее эффективный способ создания проектов на Kotlin и управления ими – использовать Gradle – систему сборки, являющуюся фактическим стандартом для программ на Kotlin. Gradle также считается одним из лучших инструментов управления проектами на Java, поэтому неудивительно, что Gradle прекрасно подходит для создания смешанных проектов на Java/Kotlin. В смешанных проектах можно использовать лучшее из обоих миров, в том числе огромную массу кода на Java, доступного в виде исходных текстов.

Если вы уже использовали Gradle для создания своих программ на Java, вы сможете создавать и управлять проектами на Kotlin или смешанными проектами на Kotlin/Java, как уже делали это с проектами на Java. Если нет, тогда сейчас самый подходящий момент перейти на современный инструмент сборки. Но даже если вы уже имеете опыт использования Gradle, вам может пригодиться умение писать сценарии Gradle на языке Kotlin.

В мае 2016 года команда Gradle объявила, что Kotlin станет основным языком для написания сценариев Gradle. Прежде для этой цели использовался только язык Groovy. Это заявление не означает, что поддержка Groovy в Gradle будет прекращена. Вы по-прежнему можете писать сценарии для Gradle на Groovy, но Kotlin намного практичнее, и я рекомендую попробовать перейти на него. А если вы новичок в Gradle, я настоятельно советую сразу же начинать с Kotlin.

Kotlin предлагает массу преимуществ для сценариев Gradle, которые используются для сборки проектов на Kotlin или смешанных проектов на Java/Kotlin. Самое очевидное из них: вам не придется изучать еще один язык. Но самое большое преимущество – вы получите гораздо лучшую поддержку сценариев на Kotlin для Gradle в вашей интегрированной среде разработки (по крайней мере, в IntelliJ). IntelliJ предлагает тот же уровень поддержки Kotlin в сценариях для Gradle, что и в программах на Kotlin, включая проверку синтаксиса, автодополнение и рефакторинг.

Создание простого проекта в GRADLE

Создать простой проект Kotlin/Java с помощью Gradle проще простого. Взгляните:

```
plugins {
    application
```

```
    kotlin("jvm") version "1.3.21"
    // Измените версию в соответствии со своими требованиями
}

application {
    mainClassName = "com.mydomain.mysimpleproject.MainKt"
}

repositories {
    jcenter()
}

dependencies {
    compile(kotlin("stdlib"))
}
```

Сохраните этот сценарий в файле с именем `build.gradle.kts` в каталоге своего проекта, например в каталоге `MySimpleProject`. Добавьте в него структуру каталогов, как показано ниже:

```
MySimpleProject
    src
        main
            java
            kotlin
        test
            java
            kotlin
```

Вот и все! Теперь вы можете добавлять файлы на Kotlin и Java в свой проект и управлять им с помощью различных команд Gradle. Чтобы создать программу на Kotlin с зависимостью из Java, добавьте в проект следующие файлы:

```
MySimpleProject/src/main/java/com/mydomain/mysimpleproject/MyClass.java
MySimpleProject/src/main/kotlin/com/mydomain/mysimpleproject/Main.kt
```

Вот содержимое файла `MyClass.java`:

```
package com.mydomain.mysimpleproject;

public class MyClass {

    public static String getMessage(Lang language) {
        switch (language) {
            case ENGLISH:
                return "Hello";
            case FRENCH:
                return "Bonjour";
            case GERMAN:
                return "Hallo";
            case SPANISH:
                return "Hola";
            default:
                return "Saluton";
        }
    }
}
```

А это содержимое файла `Main.kt`:

```
package com.mydomain.mysimpleproject

fun main(args: Array<String>) {
    println(MyClass.getMessage(Lang.GERMAN))
}

enum class Lang { GERMAN, FRENCH, ENGLISH, SPANISH }
```

Теперь можно запустить проект, выполнив следующую команду Gradle в каталоге проекта (`MySimpleProject`):

```
<path_to_Gradle>/bin/gradle run
```

Скомпилировать и собрать проект в файл ZIP или TAR можно следующей командой Gradle, выполнив ее в каталоге проекта (`MySimpleProject`):

```
<path_to_Gradle>/bin/gradle assembleDist
```

Она должна создать следующие файлы:

```
MySimpleProject/build/distributions/MySimpleProject.tar
MySimpleProject/build/distributions/MySimpleProject.zip
```

Теперь можно извлечь содержимое любого из этих архивов в некоторую папку и выполнить находящийся в нем сценарий командной оболочки, чтобы запустить проект.

Импортирование Gradle-проекта в IntelliJ

Чтобы импортировать Gradle-проект в IntelliJ, достаточно:

- выбрать в меню пункт **File** > **New** > **Project from Existing Sources** (Файл > Новый > Проект из имеющегося исходного кода), выбрать каталог проекта и щелкнуть по кнопке **OK**;
- выбрать пункт **Import Project from External Model** (Импортировать проект из внешней модели), затем **Gradle** и щелкнуть по кнопке **Next** (Далее);
- выбрать **Use Default Gradle Wrapper** (Использовать обертку по умолчанию Gradle) и щелкнуть по кнопке **Finish** (Завершить).

Как вариант, можно выбрать локальный дистрибутив Gradle. Также можно выбрать минимальную версию JDK, например JDK 6 (используемую в Android). А чтобы получить пустой проект, для которого создан только сценарий сборки, можно выбрать вариант **Create Directories for Empty Content Roots Automatically** (Автоматически создать пустые корневые каталоги). В этом случае IntelliJ автоматически создаст структуру каталогов, включая подкаталоги для исходного кода на Java и Kotlin в каталогах `main` и `test`, а также каталог `resource`. Если какие-то из них уже существуют, они будут сохранены. Теперь можно продолжить разработку проекта и добавлять файлы на Kotlin и Java по своему желанию.

Добавление зависимостей в проект

Для добавления зависимостей в проект можно использовать стандартный синтаксис Gradle:

```
val kotlintestVersion = "3.1.10"
val logbackVersion = "1.2.3"
val slf4jVersion = "1.7.25"

plugins {
    application
    kotlin("jvm") version "1.3.21" // Измените версию при необходимости
}

application {
    mainClassName = "com.mydomain.mysimpleproject.MainKt"
}

repositories {
    jcenter()
}

dependencies {
    compile(kotlin("stdlib"))
    testCompile("io.kotlintest:kotlintest-runnerjunit5:$kotlintestVersion")
    testRuntime("org.slf4j:slf4j-nop:$slf4jVersion")
}
```

В блоке `plugins` нельзя использовать переменную для номера версии – допускаются только литералы, поэтому вам может потребоваться дважды указать номер версии в вашем файле, если этот номер будет нужен где-то еще. Это известное ограничение, которое будет снято в следующих версиях Gradle.

Создание проектов с несколькими модулями

Любой серьезный проект состоит из нескольких модулей. Чтобы создать такой проект, нужно добавить в репозиторий проекта два файла: `settings.gradle.kts` и `build.gradle.kts`. Файл `settings.gradle.kts` содержит список модулей, например:

```
include("server", "client", "common")
```

Файл `build.gradle.kts` содержит общую конфигурацию:

```
ext["kotlintestVersion"] = "3.1.10"
ext["logbackVersion"] = "1.2.3"
ext["slf4jVersion"] = "1.7.25"

plugins {
    base
    kotlin("jvm") version "1.3.21" // Измените версию при необходимости
}

allprojects {

    group = "com.mydomain.mymultipleproject"
```

```
    version = "1.0-SNAPSHOT"

    repositories {
        jcenter()
        mavenCentral()
    }
}
```

Для каждого модуля в каталоге проекта нужно создать отдельный подкаталог со своим файлом `build.gradle.kts`. Вот пример файла для модуля `server`:

```
plugins {
    application
    kotlin("jvm")
}

application {
    mainClassName = "com.mydomain.mymultipleproject.server.main.Server"
}

dependencies {
    compile(kotlin("stdlib"))
    compile(project(":common"))
}
```

ПРИМЕЧАНИЕ Здесь не требуется указывать версию плагина Kotlin. Gradle автоматически будет использовать версию, указанную в настройках родительского проекта.

Проект `server` зависит от проекта `common`, и эта зависимость определяется добавлением имени модуля с двоеточием перед ним. Конфигурация `application` определяет имя главного класса проекта, но не у всех подпроектов есть главный класс. Полная структура проекта с несколькими модулями выглядит следующим образом:

```
MyMultipleProject
    settings.gradle.kts
    build.gradle.kts
    client
        build.gradle.kts
    common
        build.gradle.kts
    server
        build.gradle.kts
```

Это все, что требуется. Теперь вы сможете создать новый проект в IntelliJ, следуя за описанием в предыдущем разделе. Все необходимые каталоги для исходных кодов на Java (`main` и `test`), Kotlin (`main` и `test`) и `resources` созданы.

Добавление зависимостей в проект с несколькими модулями

В каждом модуле можно определять свои зависимости, так же как в проекте с единственным модулем. Кроме того, чтобы упростить сопрово-

ждение, номера версий можно объявить в одном месте. Благодаря этому, когда понадобится обновить версию, сделать это можно будет только в одном месте – в родительском модуле. Для этого следует использовать специальный объект `ext`, который можно создать в родительском сценарии сборки и использовать в подпроектах. Например, добавьте в родительский сценарий сборки следующую строку:

```
ext["slf4jVersion"] = "1.7.25"
```

Теперь это значение можно использовать в подпроектах:

```
dependencies {
    compile(kotlin("stdlib"))
    testRuntime("org.slf4j:slf4j-nop:${project.rootProject.ext["slf4jVersion"]}")
}
```

Вызов Java-методов из Kotlin

Вызов методов в Java-библиотеках из кода на Kotlin – наиболее распространенный случай взаимодействий между двумя языками. Фактически ни одна программа на Kotlin, работающая под управлением JVM, не может избежать вызова Java-методов, явно или неявно, потому что большинство функций в стандартной библиотеке Kotlin вызывают методы из стандартной библиотеки Java.

Вызывать Java-методы из кода на Kotlin было бы просто, если бы типы в Kotlin и Java были одинаковыми, но это не так. К счастью, типы в Kotlin намного мощнее, чем типы в Java, но это означает, что нужно проявлять осторожность при преобразовании.

Использование примитивов Java

Самым заметным отличием, вероятно, является отсутствие примитивов в Kotlin. В Java тип `int` является примитивом, а `Integer` – объектом, и оба представляют целочисленные значения. В Kotlin присутствует только тип `Int`, который иногда называют *типом-значением*, что означает, что он может обрабатываться как объект, но работает как примитив. В ближайшее время в Java тоже могут появиться типы-значения[1].

Несмотря на то что `Int` в Kotlin – это объект, вычисления выполняются так, будто это примитив. То же верно и в отношении других примитивов Java, для которых в Kotlin имеются эквивалентные типы-значения (таких как `byte`, `short`, `long`, `float`, `double` и `boolean`; для всех этих примитивов в Kotlin имеются соответствующие типы-значения). Единственное отличие, кроме того, что с ними можно работать как с объектами, – имена типов в Kotlin начинаются с заглавной буквы.

Kotlin автоматически преобразует примитивы Java в типы-значения Kotlin. Вы все еще можете использовать Java-тип `Integer` в Kotlin, но ли-

[1] Описание типов-значений в проекте Valhalla можно найти по адресу http://cr.openjdk.java.net/~jrose/values/values-0.html.

шитесь преимуществ, которые дают типы-значения. В Kotlin также можно использовать другие числовые типы-объекты из Java, но, поскольку их имена совпадают, придется использовать полное имя, например `java.lang.Long`. Правда, в этом случае вы получите следующее предупреждение от компилятора:

```
Warning:(...) Kotlin: This class shouldn't be used in Kotlin. Use kotlin.Long
instead.
(Внимание:(...) Kotlin: Этот класс не следует использовать в Kotlin. Используйте
kotlin.Long.)
```

Практически всегда следует избегать использования числовых типов-объектов из Java, потому что разработчики Kotlin проделали большую работу для поддержки преобразований. Взгляните на следующий код:

```
val a: java.lang.Long = java.lang.Long.valueOf(3L)
```

Он не скомпилируется и, встретив его, компилятор сообщит об ошибке:

```
Error:(...) Kotlin: Type mismatch: inferred type is kotlin.Long! but java.lang.Long
was expected
(Ошибка:(...) Kotlin: Несоответствие типов: выражение имеет тип kotlin.Long! но
ожидается тип java.lang.Long)
```

Kotlin автоматически преобразовал результат вызова метода `java.lang.Long.valueOf()` в тип `kotlin.Long`, поэтому этот код не компилируется. Конечно, вы можете записать этот код чуть иначе:

```
val a: java.lang.Long = java.lang.Long(3)
```

Но зачем это нужно? Примитивы Java преобразуются в типы без поддержки `null` (например, Java-тип `int` преобразуется в тип `Int`), а объектные числовые типы преобразуются в типы Kotlin с поддержкой `null` (например, Java-тип `Integer` преобразуется в `Int?`).

Использование числовых типов-объектов Java

Java также предлагает числовые типы, не имеющие примитивных эквивалентов, такие как `BigInteger` и `BigDecimal`. Эти типы можно использовать в Kotlin, используя дополнительное преимущество упрощенной работы с ними. Например, можно написать такой код:

```
val a = BigInteger.valueOf(3)

val b = BigInteger.valueOf(5)

println(a + b == BigInteger.valueOf(8))
```

Он выведет:

```
true
```

С типом `BigInteger` можно не только использовать оператор `+`, но и определять равенство с помощью оператора `==`. Такая простота объясняется тем, что Kotlin преобразует оператор `+` в вызов метода `add`, а опе-

ратор == в вызов метода equals. (Проверка идентичности в Kotlin выполняется с помощью оператора ===.)

Быстрый отказ со значениями null

Все типы объектов в Java поддерживают значение null, а примитивы – нет. Kotlin различает типы с поддержкой и без поддержки null. Например, Kotlin определит тип выражения BigInteger.valueOf() как BigInteger?. Дополнительно Kotlin предлагает тип BigInteger, являющийся подтипом BigInteger?, но не поддерживающий null. Если бы требовалось указывать типы явно, вы должны были бы записать:

```
val a: BigInteger? = BigInteger.valueOf(3)
```

Но в этом случае вы не сможете использовать оператор + из-за вероятности получить исключение NullPointerException. Оператор разыменования . (точка) тоже нельзя использовать. Вместо этого вам придется использовать такой код:

```
val a: BigInteger? = BigInteger.valueOf(3)
val b: BigInteger? = BigInteger.valueOf(5)
println(a?.add(b) == BigInteger.valueOf(8))
```

Здесь используется оператор ?. – безопасный оператор разыменования, который возвращает null, если значение слева от него равно null. Если позволить компилятору Kotlin самому определять тип, он сделает что-то вроде следующего:

```
val a: BigInteger! = BigInteger.valueOf(3)
```

Этот код не компилируется, но показывает, как действует Kotlin. Тип BigInteger! называется *неденотируемым* типом, т. е. типом, который нельзя использовать в программе, но который используется компилятором Kotlin для своих нужд. (IntelliJ показывает эти типы там, где они используются.) На самом деле мы обычно пишем такой код:

```
val a: BigInteger = BigInteger.valueOf(3)
```

Kotlin сгенерирует исключение NullPointerException, если Java-метод вернет null. Конечно, этого никогда не произойдет с BigInteger.valueOf, но при использовании Java-методов, способных возвращать null (а их очень много в стандартной библиотеке Java), Kotlin гарантирует, что ваша программа максимально быстро потерпит неудачу, и тем самым предотвращает утечку null внутри вашего кода.

Если хотите, можете продолжать явно указывать типы, такие как поддерживающие null, например BigInteger?. Подробнее о поддержке null в типах Kotlin рассказывалось в главе 2.

Использование строковых типов Kotlin и Java

В Kotlin используется специальный строковый тип с именем kotlin.String, тогда как в Java используется тип java.lang.String. И снова тип в Kotlin

оказывается намного мощнее. Посмотрим для примера, как в Java можно удалить последний символ из строки:

```
String string = "abcde";
String string2 = string.substring(0, string.length() - 1);
```

А вот эквивалентный код на Kotlin:

```
val s: String = "abcde".dropLast(1)
```

Класс `kotlin.String` предлагает десятки подобных вспомогательных функций. Кроме того, преобразование между типами `java.lang.String` и `kotlin.String` выполняется полностью автоматически и прозрачно.

Преобразование других типов

Для коллекций и массивов предоставляются дополнительные средства преобразования типов, которые могут потребоваться при вызове методов Java. Массивы примитивов преобразуются в специальные типы: `ByteArray`, `IntArray`, `LongArray` и т. д. Массивы объектов типа `T` преобразуются в тип `Array<T>`, который является инвариантным (так же как массивы Java). Но при желании можно использовать ко- и контравариантные массивы, явно указав это; например, `Array<out T>`.

По умолчанию коллекции преобразуются в изменяемые типы коллекций, не поддерживающие `null`. Например, `java.util.List<String>` преобразуется в `kotlin.collections.MutableList<String>`.

Несмотря на то что Kotlin преобразует коллекции Java в типы коллекций, не поддерживающие `null`, он гарантирует только, что сами ссылки на коллекции не будут содержать `null`, но не проверяет наличие пустых элементов в коллекциях. Это может стать огромной проблемой, если предположить, что такая проверка будет произведена во время выполнения. Допустим, у нас есть следующий код на Java:

```
package test;

import java.util.Arrays;
import java.util.List;

public class Test {

    public static List<Integer> getIntegerList() {
        return Arrays.asList(1, 2, 3, null);
    }
}
```

И мы решили вызвать метод `getIntegerList` из Kotlin:

```
val list: MutableList<Int> = test.Test.getIntegerList()

println(list)
```

Он не вызовет ошибки и выведет:

```
[1, 2, 3, null]
```

Но, попробовав выполнить следующий код на Kotlin:

```
val list: MutableList<Int> = test.Test.getIntegerList()

list.forEach { println(it + 1) }
```

мы получим `NullPointerException`, когда Kotlin попытается применить функцию `add` к `null`.

Вызов Java-методов с переменным числом параметров

Для вызова Java-методов с переменным числом параметров Kotlin позволяет использовать массив с предшествующим ему оператором деструктуризации `*`. Взгляните на следующий пример Java-метода:

```
public static void show(String... strings) {
    for (String string : strings) {
        System.out.println(string);
    }
}
```

Этот метод можно вызвать из Kotlin, используя массив:

```
val stringArray = arrayOf("Mickey", "Donald", "Pluto")
MyClass.show(*stringArray)
```

Управление поддержкой null в Java

Как я уже говорил, все непримитивные типы в Java поддерживают значение `null`. Однако их можно аннотировать (и многие инструменты используют такие аннотации), чтобы явно указать, что ссылки этих типов никогда не должны быть пустыми. Для этого существует стандартная спецификация (JSR-305 `javax.annotation`), но многие инструменты предпочитают свой собственный набор аннотаций:

- в IntelliJ используются аннотации `@Nullable` и `@NotNull` из пакета `org.jetbrains.annotations`;
- в Eclipse применяется пакет `org.eclipse.jdt.annotation`;
- в Android имеются пакеты `com.android.annotations` и `android.support.annotations`;
- в FindBugs используется пакет `edu.umd.cs.findbugs.annotations`.

Kotlin понимает все эти (и многие другие) аннотации. Например, в Java свойство может быть аннотировано, как показано ниже:

```
@NotNull
public static List<@NotNull Integer> getIntegerList() {
    return null;
}
```

IntelliJ отметит элемент `null` сообщением «Passing 'null' argument to parameter annotated with @NotNull» (Передача аргумента null в параметре, аннотированном как @NotNull), но это не помешает компиляции кода на Java. Если вызвать этот метод из Kotlin, компилятор выведет его

тип как (Mutable)List<Int> вместо (Mutable)List<Int!>! и сгенерирует исключение:

```
java.lang.IllegalStateException: @NotNull method test/MyClass.getIntegerList must
not return null
(
java.lang.IllegalStateException: @NotNull-метод test/MyClass.getIntegerList не
должен возвращать null
)
```

К сожалению, это не относится к типам параметров. Взгляните на следующий Java-метод:

```
@NotNull
public static List<@NotNull Integer> getIntegerList() {
    return Arrays.asList(1, 2, 3, null);
}
```

В Java вы получите предупреждение о значении null, а Kotlin ничего не сообщит вам. Однако при попытке использовать значение null возникнет исключение NullPointerException. Применение аннотации @NotNull к типу параметра не отличается от объявления типа без поддержки null в Kotlin, потому что Kotlin определит тип как List<Int!> (неденотируемый тип), дающий точно такой же результат. Однако, если вы хотите использовать аннотации для типов параметров, вам придется использовать библиотеку org.jetbrains.annotation с версией не ниже 1.5.0 и установить в качестве целевого уровня компиляции Java 8 или выше.

ПРИМЕЧАНИЕ Kotlin поддерживает спецификацию JSR-305, поэтому, если вам понадобятся другие Java-аннотации, обращайтесь к описанию https://kotlinlang.org/docs/reference/java-interop.html.

Вызов методов свойств

Java-методы чтения и записи свойств можно вызывать в Kotlin как самые обычные методы. Однако в Kotlin поддерживается (и рекомендуется) также их вызов с использованием синтаксиса свойств. Допустим, у нас имеется следующий Java-класс:

```
public class MyClass {

    private int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Мы можем обратиться к свойству, используя синтаксис свойств или синтаксис методов:

```
val myClass = MyClass()
myClass.value = 1
println(myClass.value)

myClass.setValue(2)
println(myClass.getValue())
```

Я настоятельно рекомендую первый способ. Синтаксис свойств можно использовать для чтения Java-полей, не имеющих метода записи и инициализирующихся в конструкторе. Этот синтаксис можно использовать, даже когда поля нет вообще, например:


```
public class MyClass {

    public int getValue() {
        return 0;
    }
}
```

Но с помощью этого синтаксиса нельзя присвоить новое значение свойству, если оно не имеет метода записи. Для Java-свойств типа `Boolean`, метод чтения которых начинается с `is`, Kotlin использует имена, так же начинающиеся с `is`. Вот пример Java-класса:

```
public class MyClass {

    boolean started = true;
    boolean working = false;

    public boolean isStarted() {
        return started;
    }

    public boolean getWorking() {
        return working;
    }
}
```

Прочитать эти свойства из Kotlin можно так:

```
val myClass = MyClass()

myClass.started = true
myClass.working = false
println(myClass.started)
println(myClass.isStarted)
println(myClass.working)
```

Этот способ позволяет избежать конфликтов, которые могут возникнуть, если в Java-классе имеются методы `isSomething` и `getSomething`. Методы записи в Java на самом деле возвращают `void`. При вызове из Kotlin с использованием синтаксиса методов/свойств они будут возвращать тип `Unit` – объект-синглтон:

```
val result: Unit = myClass.setWorking(false)
```

Доступ к свойствам в JAVA с зарезервированными именами

Иногда может потребоваться обратиться из Kotlin к Java-свойству с именем, совпадающим с зарезервированным словом в Kotlin, например `in` и `is`, которые используется в качестве коротких имен для `input` или `inputStream`. Если вы собираетесь писать свои проекты сразу на двух языках, старайтесь избегать таких имен. Но если вы используете стороннюю библиотеку, можете использовать обратные апострофы, чтобы экранировать имя свойства. Например:

- класс на Java:

```
public class MyClass {

    private InputStream in;

    public void setIn(InputStream in) {
        this.in = in;
    }

    public InputStream getIn() {
        return in;
    }
}
```

- код на Kotlin:

```
val input = myClass.`in`
```

Вызов контролируемых исключений

В Kotlin все исключения – неконтролируемые. Как следствие, Kotlin позволяет вызывать Java-методы, возбуждающие контролируемые исключения, не заключая вызовы в конструкцию `try...catch`.

- Java:

```
try {
    Thread.sleep(10);
} catch (InterruptedException e) {
    // обработка исключения
}
```

- Kotlin:

```
Thread.sleep(10)
```

В отличие от того, что часто делается в Java, здесь не используются никакие обертки. Использовав блок `try...catch` и получив исключение, вы получите оригинальное исключение, а не неконтролируемое исключение-обертку.

SAM-интерфейсы

В отличие от Kotlin Java не поддерживает типы-функции; они имитируются преобразованием лямбда-выражений в нечто, эквивалентное реализации интерфейса с единственным абстрактным методом (Single Abstract Method, SAM). (Это не реализация интерфейса, но действует подобно такой реализации.)

В Kotlin, напротив, поддерживаются настоящие типы-функции, поэтому в таком преобразовании нет необходимости. Но, когда вызывается Java-метод, принимающий интерфейс SAM, функция на Kotlin автоматически преобразуется в его реализацию. Например, в Kotlin можно написать такой код:

```
val executor = Executors.newSingleThreadExecutor()
executor.submit { println("Hello, World!") }
```

Java-метод `submit` принимает параметр типа `Runnable`. В Java ту же программу можно записать так:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> System.out.println("Hello, World!"));
```

Обратите внимание, что в Kotlin можно явно создать `java.lang.Runnable`:

```
val runnable = Runnable { println("Hello, World!") }
```

Но обычно этого не делается. Лучше определить обычную функцию и позволить Kotlin автоматически преобразовать ее в `Runnable` при вызове Java-метода:

```
val executor = Executors.newSingleThreadExecutor()
val runnable: () -> Unit = { println("Hello, World!") }
executor.submit(runnable)
```

Вызов Kotlin-функций из Java

Вызов функций Kotlin из Java ничуть не сложнее, чем вызов методов Java из Kotlin. Но так как Kotlin обладает более богатыми возможностями, взаимодействие в эту сторону часто сопряжено с потерей некоторой функциональности.

Преобразование свойств Kotlin

Свойства в Java подчиняются определенным соглашениям, связанным с оформлением кода. В Kotlin, напротив, свойства являются частью синтаксиса языка. Даже притом что программы на Kotlin компилируются в байт-код Java, компилятор сам заботится о необходимых преобразованиях. Свойства Kotlin преобразуются в стандартное представление для Java – в поля и методы доступа:

- приватное поле с именем, совпадающим с именем свойства;
- метод чтения с именем, совпадающим с именем свойства, но начинающимся с заглавной буквы и префикса `get`;
- метод записи с именем, совпадающим с именем свойства, но начинающимся с заглавной буквы и префикса `set`.

Единственное исключение делается для свойств Kotlin с именами, начинающимися с префикса `is`. Это имя без изменений используется для метода чтения, а для метода записи выбирается имя, в котором префикс `is` меняется на `set`.

Использование общедоступных полей Kotlin

Общедоступные поля Kotlin доступны из Java как свойства (т. е. посредством методов чтения и записи). Чтобы получить возможность использовать их как поля Java, эти свойства нужно снабдить аннотацией:

```
@JvmField
val age = 25
```

В этом случае для свойств Kotlin не будут создаваться методы доступа. Также обратите внимание, что свойства Kotlin, объявленные с ключевым словом `lateinit`, доступны как поля и посредством методов чтения. Их нельзя снабдить аннотацией `@JvmField`.

Статические поля

Kotlin не имеет явных статических полей, но некоторые поля Kotlin можно рассматривать как статические. Это относится к полям, объявленным в объектах, включая объекты-компаньоны, а также к полям, объявленным на уровне пакета. Все эти поля доступны из Java как свойства, хотя и с некоторыми отличиями. К полям, объявленным в объектах-компаньонах, можно обращаться с помощью методов чтения/записи, используя следующий синтаксис:

```
int weight = MyClass.Companion.getWeight();
```

Но если поле объявлено с ключевым словом `const`, к нему следует обращаться как к статическому полю вмещающего класса:

```
int weight = MyClass.weight;
```

То же верно для полей с аннотацией `@JvmField`. Для доступа к полям в автономных объектах используется следующий синтаксис:

```
String firstName = MyObject.INSTANCE.getFirstName();
```

И снова, если поле объявлено с ключевым словом `const` или отмечено аннотацией `@JvmField`, к нему следует обращаться как к статическому полю:

```
String firstName = MyObject.firstName;
```

К полям, объявленным на уровне пакета, следует обращаться как к статическим полям класса с именем, совпадающим с именем файла без расширения .kt, но с окончанием Kt. Допустим, в MyFile.kt имеется следующий код:

```
const val length = 12

val width = 3
```

Эти поля будут доступны из Java как:


```
int length = MyFileKt.length;
int width = MyFileKt.getWidth();
```

Обратите внимание, что поля уровня пакета не требуется отмечать аннотацией @JvmField.

Вызов функций Kotlin из методов Java

К функциям, объявленным в классах с помощью ключевого слова fun, можно обращаться из Java как к методам. Функции, объявленные в объектах или на уровне пакета, доступны, как статические методы. Рассмотрим следующий файл MyFile.kt:


```
fun method1() = "method 1"

class MyClass {

    companion object {
        fun method2() = "method 2"

        @JvmStatic
        fun method3() = "method 3"
    }
}

object MyObject {
    fun method4() = "method 4"

    @JvmStatic
    fun method5() = "method 5"
}
```

Эти функции можно вызвать из Java несколькими способами:

```
String s1 = MyFileKt.method1();
String s2 = MyClass.Companion.method2();
String s3a = MyClass.method3();
String s3b = MyClass.Companion.method3();
String s4 = MyObject.INSTANCE.method4();
String s5a = MyObject.method5();
String s5b = MyObject.INSTANCE.method5();
```

Обратите внимание, что в ответ на попытку вызвать MyObject.INSTANCE.method5() компилятор сгенерирует предупреждение «Static member accessed via instance reference» (Попытка доступа к статическому члену через

ссылку на экземпляр), как если бы вы попытались вызвать статический Java-метод для экземпляра класса.

Вызов функций-расширений из Java

Функции-расширения в Kotlin компилируются в статические функции с дополнительным параметром-приемником. Их можно вызывать из Java в том виде, в котором они скомпилированы. Рассмотрим следующую функцию-расширение, объявленную в файле с именем `MyFile.kt`:

```
fun List<String>.concat(): String = this.fold("") { acc, s -> "$acc$s" }
```

В Kotlin эта функция используется, как если бы была функцией экземпляра списка строк, но в Java она должна вызываться как статический метод:

```
String s = MyFileKt.concat(Arrays.asList("a", "b", "c"));
```

Вызов функции под другим именем

Есть возможность изменить имя, под которым функция Kotlin будет вызываться из Java. Для этого нужно добавить к функции Kotlin аннотацию `@JvmName("newName")`. Есть несколько причин поступить так, а иногда даже бывает необходимо сделать это, даже если вы никогда не будете вызывать функцию из Java. Например, пусть есть две функции:

```
fun List<String>.concat(): String = this.fold("") { acc, s -> "$acc$s" }

fun List<Int>.concat(): String = this.fold("") { acc, i -> "$acc$i" }
```

Этот код не скомпилируется, потому что:

- функции Kotlin компилируются в байт-код Java;
- эффект стирания типов уничтожает типы параметров в процессе компиляции.

В результате эти функции эквивалентны следующим Java-методам:

```
MyFileKt.concat(List<String> list)
MyFileKt.concat(List<Integer> list)
```

Как вы понимаете, эти два метода не могут сосуществовать в одном классе (`MyFileKt`), потому что компилируются в:

```
MyFileKt.concat(List list)
MyFileKt.concat(List list)
```

Даже если вы никогда не будете вызывать эти функции из Java, вам все равно придется поместить их в отдельные файлы или изменить имя одного из них, использовав аннотацию:

```
fun List<String>.concat(): String = this.fold("") { acc, s -> "$acc$s" }

@JvmName("concatIntegers")
fun List<Int>.concat(): String = this.fold("") { acc, i -> "$acc$i" }
```

Аналогично можно изменять имена Java-методов чтения/записи для свойств Kotlin:

```
@get:JvmName("retrieveName")
@set:JvmName("storeName")
var name: String?
```

Значения по умолчанию и параметры

В Kotlin параметры функций могут иметь значения по умолчанию. Такие функции можно вызывать из Java, но при этом вам придется указать все параметры, даже те, которые имеют значения по умолчанию. Поддержка значений по умолчанию в Java реализована через перегрузку. Чтобы превратить функцию в набор перегруженных методов Java, используйте аннотацию `@JvmOverloads`:

```
@JvmOverloads
fun computePrice(price: Double,
                 tax: Double = 0.20) = price * (1.0 + tax)
```

Функция `computePrice` на Kotlin будет доступна в Java как:

```
Double computePrice(Double price,)
Double computePrice(Double price, Double tax)
```

Если имеется несколько параметров со значениями по умолчанию, вы не сможете получить все возможные варианты:

```
@JvmOverloads
fun computePrice(price: Double,
                 tax: Double = 0.20,
                 shipping: Double = 8.75) = price * (1.0 + tax) + shipping
```

В этой ситуации будут созданы следующие Java-методы:

```
Double computePrice(Double price,)
Double computePrice(Double price, Double tax)
Double computePrice(Double price, Double tax, Double shipping)
```

Как видите, в этой ситуации нельзя передать только параметры `price` и `shipping` и использовать значение по умолчанию для параметра `tax`.

Аннотацию `@JvmOverloads` можно применять также к конструкторам:

```
class MyClass @JvmOverloads constructor(name: String, age: Int = 18)
```

Вызов функций, возбуждающих исключения

Все исключения в Kotlin являются неконтролируемыми. Как следствие, при попытке перехватить исключение в Java-методе, вызывающем функцию Kotlin, которая генерирует исключения, компилятор сообщит об ошибке. Рассмотрим следующую функцию на Kotlin:

```
fun readFile(filename: String): String = File(filename).readText()
```

Вы можете попытаться вызвать эту функцию из Java, заключив вызов в блок `try...catch`:

```
try {
    System.out.println(MyFileKt.readFile("myFile"));
} catch (IOException e) {
    e.printStackTrace();
}
```

Однако этот фрагмент вызовет ошибку компиляции:

```
Error: exception java.io.IOException is never thrown in body of corresponding try
statement
(Ошибка: исключение java.io.IOException никогда не будет возбуждено в теле
инструкции try)
```

Чтобы решить эту проблему, нужно с помощью аннотации `@Throws` явно указать, какое исключение возбуждает функция:

```
@Throws(IOException::class)
fun readFile(filename: String): String = File(filename).readText()
```

Обратите внимание, что аннотация `@Throws` может быть указана для конкретной функции только один раз. Если функция возбуждает несколько исключений, которые считаются контролируемыми в Java, используйте следующий синтаксис:

```
@Throws(IOException::class, IndexOutOfBoundsException::class)
fun readFile(filename: String): String = File(filename).readText()
```

Неконтролируемые исключения можно не указывать, потому что Java позволяет их перехватывать без дополнительных объявлений.

Преобразование типов Kotlin в типы Java

Kotlin различает типы с поддержкой и без поддержки значения `null`, а Java – нет. При преобразовании такие различия теряются.

Числовые типы преобразуются в примитивы или в объекты в соответствии с объявлениями на стороне Java. Неизменяемые коллекции преобразуются в коллекции Java. Например, список, созданный с помощью `listOf` в Kotlin, преобразуется в Java в `Arrays.ArrayList`. Тип `Arrays.ArrayList` – это приватный класс в `Arrays`, который наследует `AbstractList` и не имеет реализации метода `add`. Как следствие, при попытке добавить элемент, вы получите исключение `UnsupportedOperationException`, из-за чего можно подумать, что список нельзя изменить. Вы не сможете добавлять или удалять элементы, но сможете изменять элементы вызовом метода `set`, передав ему индекс элемента для замены новым элементом. Неизменяемые списки в Kotlin становятся менее неизменными при преобразовании в Java.

В Kotlin также есть типы, отсутствующие в Java:

1. `Unit` преобразуется в `void`, хотя на самом деле это одно и то же. (В Kotlin тип `Unit` представлен объектом-синглтоном.) По своей природе он ближе к `Void` (возможно, вам доводилось сталкиваться с этим типом в Java), но своим поведением он больше напоминает `void`.
2. `Any` преобразуется в `Object`.
3. `Nothing` не преобразуется ни в какой тип, потому что в Java нет эквивалентного ему типа. Когда этот тип используется в параметре типа, например `Set<Nothing>`, в результате преобразования получается простой тип `Set`.

Функции, принимающие от 1 до 22 аргументов, преобразуются в Kotlin в специальные типы от `kotlin.jvm.functions.Function1` до `kotlin.jvm.functions.Function22`. Это SAM-интерфейсы, что имеет намного больше смысла, чем интерфейсы функций в Java.

Типы функций

Функции без аргументов преобразуются в интерфейс `kotlin.jvm.functions.Function0`, соответствующий интерфейсу `Supplier` в Java.

Функции `() -> Unit` в Kotlin преобразуются в экземпляры интерфейса `kotlin.jvm.functions.Function0<Unit>`, который соответствует интерфейсу `Runnable` в Java. `Function1<A, Unit>` соответствует Java-интерфейсу `Consumer<A>`, а `Function2<A, B, Unit>` – Java-интерфейсу `BiConsumer<A, B>`. Другие функции Kotlin, возвращающие `Unit`, не имеют аналогов в Java.

Все эти преобразованные функции можно использовать так же, как в Java, с той лишь разницей, что все они реализуют метод `invoke` вместо привычных для Java `apply`, `test`, `accept` или `get`. Если потребуется преобразовать одну из этих функций в Java-тип, вы легко сможете сделать это, если в Java имеется соответствующий тип. Например, рассмотрим следующую функцию Kotlin типа `Function1<Int, Boolean>`:

```
@JvmField
val isEven: (Int) -> Boolean = { it % 2 == 0}
```

Эту функцию можно использовать в Java непосредственно:

```
System.out.println(MyFileKt.isEven.invoke(2));
```

В Java она имеет тип `kotlin.jvm.functions.Function1<Integer, Boolean>`. Чтобы преобразовать ее в `IntPredicate`, нужно записать:

```
IntPredicate p = MyFileKt.isEven::invoke;
```

Характерные проблемы смешанных проектов на Kotlin/Java

Смешивать код на Kotlin и Java в одном проекте очень просто, но будьте осторожны, выполняя инкрементальную компиляцию. *Инкрементальная компиляция* – это компиляция только тех частей программы, которые необходимо скомпилировать. При ее использовании компилироваться могут только те классы, которые изменились с момента последней компиляции, и даже отдельные строки кода по мере их ввода.

Оба инструмента, Gradle и Intellij, поддерживают инкрементальную компиляцию, но недостаточно найти только классы, которые изменились с момента последней компиляции. Если класс не изменился, но зависит от другого, изменившегося класса, – оба должны быть скомпилированы.

Обычно инкрементальная компиляция не вызывает проблем в проектах только на Java или только на Kotlin, но в смешанных проектах ситу-

ация немного сложнее. Вы можете столкнуться с проблемой при работе над проектом, в котором несколько модулей зависят от модуля с именем `common`. При работе с модулем `A` вам, возможно, придется изменить модуль `common`. В этом случае при попытке запустить код во время разработки IntelliJ или Gradle скомпилируют только необходимые классы из модуля `A` и `common`. Если позднее вы измените модуль `B`, дополнительно добавив изменения в модуль `common`, при запуске скомпилируются только эти два модуля (`B` и `common`). Как следствие, модуль `A` может перестать работать из-за изменений в модуле `common`, но вы этого не увидите.

Лучший способ избежать этой проблемы – регулярно пересобирать проект целиком. В этом случае вы сразу увидите, если что-то сломается. В противном случае можете успеть внести много изменений, а потом надолго застрять, кропотливо выискивая и исправляя ошибки.

Другая проблема – получение разных результатов при запуске из IntelliJ и Gradle. В окружении разработки код обычно компилируется и запускается из IntelliJ, а впоследствии проект компилируется на сервере сборки с использованием Gradle. Прежде чем отправлять свои изменения в репозиторий, откуда сервер сборки будет извлекать исходный код, обязательно протестируйте сборку с Gradle на своей рабочей станции. Уверяю вас, вы немало удивитесь, увидев ошибки компиляции, которые не возникали в IntelliJ.

Причина этих ошибок проста, и их легко исправить. Но, не зная решения, можно потратить часы в попытках понять, почему при компиляции в Gradle компилятор не видит в вашем Java-коде новую функцию, добавленную в код на Kotlin, хотя в IntelliJ эта проблема отсутствует. Причина в том, что в отличие от IntelliJ Gradle может не удалять байт-код, сгенерированный компилятором Kotlin, и продолжать использовать старую скомпилированную версию (без вновь добавленной функции).

Если вы столкнулись с этой проблемой, добавьте в Gradle явное удаление байт-кода. Да, это увеличит время компиляции, зато избавит от ошибок времени выполнения. А кроме того, признайте, что это удачное везение: вы могли бы запустить свою программу, использующую старую версию функции и получающую ошибочный результат. В такой ситуации вы можете долго чесать голову, прежде чем найдете причину!

Итоги

В этом приложении вы узнали, как смешивать код на Java и Kotlin в своих проектах. Проект на Kotlin всегда неявно зависит от кода на Java (стандартной библиотеки Java), поэтому, даже если вы не будете создавать смешанные проекты явно, вам все равно придется освоить некоторые приемы, представленные здесь. Описанные приемы позволяют:

- настроить смешанный проект (с одним или несколькими модулями) и управлять им с помощью Gradle или IntelliJ, включая разработку сценариев сборки для Gradle на Kotlin. Если вы используете Eclipse, вам нужно создать проект Gradle и импортировать его. И хотя мож-

но собирать проект непосредственно в IntelliJ, гораздо удобнее и проще организовать сборку в Gradle и импортировать проект. Основное преимущество такого подхода – дескриптор проекта будет иметь удобочитаемый вид и его можно сохранить в репозиторий вместе с кодом;

- вызывать программы на Java из кода на Kotlin;
- вызывать программы на Kotlin из кода на Java.

Смешанные проекты позволяют использовать лучшее из обоих миров, в том числе огромную массу кода на Java с открытыми исходными текстами.

Приложение B
Тестирование на основе свойств

Тестирование является одной из самых противоречивых тем в программировании. Это противоречие касается всех аспектов тестирования: надо ли тестировать, когда тестировать, что, сколько, как часто, как оценить качество тестов, какой показатель охвата считается оптимальным и т. д. Но ни один из этих вопросов нельзя рассматривать в отрыве от других. И почти все они зависят от других вопросов, которые обсуждаются намного реже.

В этом приложении я расскажу, как писать эффективные тесты и как сделать программу более пригодной для тестирования, применив прием тестирования на основе свойств (property-based testing). Здесь вы узнаете, как:

- придумать набор свойств, которым должен соответствовать результат программы;
- писать интерфейсы, затем тесты и, наконец, реализации, чтобы тесты не зависели от реализации;
- использовать абстракции для упрощения тестирования, удаляя части, которым можно доверять;
- писать генераторы и получать с их помощью случайные значения для тестов вместе с сопутствующими данными, которые можно использовать для проверки свойств;
- настроить тесты, которые используют тысячи входных данных, и запускать их перед каждой сборкой.

ПРИМЕЧАНИЕ Примеры кода для этого приложения доступны в репозитории https://github.com/pysaumont/fpinkotlin, в папке `examples`.

Зачем нужно тестирование на основе свойств?

Почти каждый программист согласится, что модульное тестирование необходимо, даже притом что тестирование не является идеальным способом убедиться в правильности программ. Успешное выполнение тестов не доказывает правильности программы.

Неудачное завершение теста доказывает, что программа, вероятно, содержит ошибку (только «вероятно», потому что проблема может заключаться в самом тесте!). Но успешное выполнение теста не доказывает, что программа верна. Успех подтверждает только, что вам не удалось найти ошибки. Если к разработке тестов приложить столько же усилий, сколько было приложено к написанию программ, этого все равно будет недостаточно. И все же обычно вы вкладываете гораздо меньше усилий в тестирование, чем в написание программ.

Было бы здорово иметь возможность доказать, что наши программы верны. Именно эту цель преследует функциональное программирование, но на практике такое редко бывает возможно. Идеальной программу можно считать, только если она имеет единственную возможную реализацию. Это может показаться странным, но, если задуматься, вы поймете, что риск ошибок пропорционален количеству возможных реализаций программ. Поэтому вы должны попытаться свести к минимуму количество возможных реализаций.

Один из способов добиться этого – использовать абстракции. Возьмем для примера программу, которая находит все элементы в списке целых чисел, кратные заданному, и возвращает максимальный из них. В традиционном программировании мы могли бы сделать это с помощью индексированного цикла, как в следующем примере (ошибки в коде сделаны преднамеренно):

```
// example00
fun maxMultiple(multiple: Int, list: List<Int>): Int {
    var result = 0
    for (i in 1 until list.size) {
        if (list[i] / multiple * multiple == list[i] && list[i] > result) {
            result = list[i]
        }
    }
    return result
}
```

Как бы вы протестировали такую программу? Конечно, вы видите ошибки, поэтому перед написанием тестов сначала постараетесь исправить их. Но если бы эти ошибки допустили вы, то могли бы их не заметить. Вместо этого вы могли бы проверить реализацию на предельные значения. Функция имеет два параметра: целое число и список, и вы, вероятно, захотите проверить функцию, передав ей значение 0 в целочисленном параметре или пустой список – в списочном. В первом случае функция сгенерирует исключение `java.lang.ArithmeticException`: деление на ноль. Во втором случае функция вернет результат 0.

Обратите внимание, что, если передать 0 и пустой список, эта реализация не сгенерирует исключения, но вернет результат 0.

Другая ошибка в этом примере – функция игнорирует первый элемент. В большинстве случаев это не вызовет ошибки:

- если список пуст;
- если первый параметр 0 (потому что вы исправили ошибку деления на 0);
- если первый элемент списка не кратен первому параметру;
- если первый элемент списка кратен первому параметру, но не является наибольшим.

Теперь вы знаете, что делать: написать тест с ненулевым первым параметром и списком, в котором первый элемент является наибольшим кратным. Но… если вы настолько умны, чтобы понять это, вы бы не допустили эту ошибку, поэтому в этом случае тестирование бесполезно.

Некоторые программисты считают, что тесты должны быть написаны до реализации. Я полностью согласен с этим мнением, но как это поможет здесь? Вы могли бы написать тест для 0 и пустого списка, но как узнать заранее, что вы должны написать тест с ненулевым первым параметром и списком, в котором первый элемент является наибольшим кратным? Сделать это можно, только зная реализацию.

Тестирование известной реализации – не идеальный случай, потому что реализация написана вами, вы подойдете к тестированию предвзято. С другой стороны, если писать тесты для неизвестной пока реализации, будет интереснее попытаться заставить ее потерпеть неудачу.

Настоящая игра начинается, когда вы пытаетесь заставить программу потерпеть неудачу, не видя реализации. А поскольку вы пишете не только тест, но и реализацию, лучший момент для написания такого теста – до начала работы над реализацией. В этом случае процесс должен выглядеть так:

1. Написать интерфейс.
2. Написать тест.
3. Написать реализацию и протестировать ее.

Рассмотрим каждый из этих этапов поближе.

Интерфейс

Написать интерфейс просто. Для этого достаточно определить сигнатуру функции:

```
fun maxMultiple(multiple: Int, list: List<Int>): Int = TODO()
```

Тест

Теперь вы должны написать тесты. Используя традиционный подход, вы могли бы написать такой код:

```
// example00 test
import io.kotlintest.shouldBe
```

```
import io.kotlintest.specs.StringSpec

internal class MyKotlinLibraryKtTest: StringSpec() {

    init {
        "maxMultiple" {
            val multiple = 2
            val list = listOf(4, 11, 8, 2, 3, 1, 14, 9, 5, 17, 6, 7)
            maxMultiple(multiple, list).shouldBe(14)
        }
    }
}
```

Вы можете написать сколько угодно тестов, используя определенные значения. Конечно, нужно проверить все специальные значения, такие как 0 и пустой список, но что еще можно сделать?

Обычно программисты выбирают некоторые входные значения и проверяют правильность результатов. В предыдущем примере проверяется равенство между `14` и результатом, который возвращает функция, получившая `2` и `[4, 11, 8, 2, 3, 1, 14, 9, 5, 17, 6, 7]`.

Но как узнать, что результатом должно быть число `14`? Выполните вручную те же вычисления, которые выполняет функция. Вы может допустить ошибку, потому что люди не совершенны. Но чаще вас будет поджидать успех, потому что этот тест состоит из выполнения одного и того же действия дважды: один раз в вашей голове и один раз в компьютерной программе. Это мало отличается от подхода, когда сначала пишется реализация, затем она запускается с заданными параметрами, а далее пишется тест, чтобы убедиться, что реализация возвращает тот же результат.

ПРИМЕЧАНИЕ Вы проверяете, что написали правильную, по вашему мнению, реализацию. Вы не проверяете, что это правильное решение проблемы. Для более надежного тестирования вы должны проверить что-то еще, кроме равенства между результатом, вычисленным вашим кодом, и результатом, вычисленным в уме. Вот что такое тестирование на основе свойств.

Прежде чем рассматривать, как писать реализацию, давайте немного углубимся в тестирование на основе свойств.

Что такое тестирование на основе свойств?

Тестирование на основе свойств – это способ проверки некоторых свойств входных данных по результату. Например, если вы хотите написать программу объединения строк (с помощью оператора `+`), тогда можно проверить следующие свойства:

- `(string1 + string2).length == string1.length + string2.length;`
- `string + "" == string;`
- `"" + string == string;`
- `string1.reverse() + string2.reverse() == (string2 + string1).reverse()`.

Тестирования этих свойств достаточно, чтобы гарантировать правильность программы. (Первые три даже не нужны.) При этом можно проверить миллионы случайно сгенерированных строк, не заботясь о реальном результате. Единственное, что имеет значение, – это проверка свойств.

Первое, что вы должны сделать, прежде чем приступить к программированию (основной программы или теста), – подумать о задаче с точки зрения свойств входных и выходных данных функции, которые следует проверить. В этом случае вы сразу сможете увидеть, что с функцией без побочных эффектов гораздо проще работать.

Разрабатывая интерфейс для задачи поиска максимального кратного, вы тут же заметите, что не всегда легко найти такие свойства. Как вы помните, мы должны найти такой набор свойств, чтобы можно было проверить истинность результата для всех кортежей типа `((Int, List<Int>), Int)`, когда последний `Int` является правильным результатом применения функции к паре `Int, List<Int>`. Вспомним, как выглядит сигнатура функции:

```
fun maxMultiple(multiple: Int, list: List<Int>): Int = TODO()
```

Поиск некоторых свойств может показаться простым, но найти важные свойства сложнее. В идеале вы должны найти наименьший набор свойств, гарантирующий верность результата. Но вы должны сделать это без применения логики, которую будете использовать для реализации функции. Вот два способа поиска таких свойств:

- поиск условий, которые должны выполняться;
- поиск условий, которые не должны выполняться.

Например, перебирая элементы в списке, вы не должны искать элемент, кратный первому параметру и превышающий результат. Проблема в том, что при тестировании вы почти наверняка будете использовать тот же алгоритм, что и в реализации функции, поэтому такой подход ничуть не лучше двукратного вызова функции и проверки, что в обоих вызовах получен одинаковый результат! Решение проблемы – создание абстракции.

Абстракция и тестирование на основе свойств

В действительности вы должны найти способ абстрагировать отдельные части задачи, написать функции для реализации каждой части и протестировать их в отдельности. Вы уже знаете, как это сделать. Вспомнив, о чем рассказывалось в главе 5, вы легко заметите, что операция сводится к свертке. Данную задачу можно разделить на две функции – функцию свертки и функцию, используемую в качестве второго параметра функцией свертки:

```
fun maxMultiple(multiple: Int, list: List<Int>): Int =
    list.fold(initialValue) { acc, int -> ... }
```

Здесь я использовал стандартную функцию `fold`, принимающую список. Если вы решите использовать список, разработанный в главе 5, тогда реализация будет выглядеть так:

```
fun maxMultiple(multiple: Int, list: List<Int>): Int =
    list.foldLeft(initialValue) { acc -> { int -> ... } }
```

ПРИМЕЧАНИЕ В оставшейся части приложения я использую стандартные типы Kotlin.

Теперь нужно протестировать функцию, используемую для свертки. Как следствие, вы будете использовать не анонимную функцию, такую как лямбда-выражение в предыдущем примере, а, скорее, что-то вроде этого:

```
fun maxMultiple(multiple: Int, list: List<Int>): Int =
    list.fold(0, ::isMaxMultiple)

fun isMaxMultiple(acc: Int, value: Int) = ...
```

Абстрагировав итерационную часть, вы получаете реализацию, которую проще протестировать. При этом вам не требуется тестировать эту часть, потому что она уже была протестирована раньше, когда разрабатывалась. Используя стандартную функцию `fold`, вы просто доверяете разработчикам стандартной библиотеки Kotlin. А если вы выберете функцию `foldLeft` из главы 5, то мы ее уже протестировали и тоже можем доверять ей.

ВНИМАНИЕ Вы не должны тестировать функции из стандартной библиотеки языка или из какой-то внешней библиотеки. Если вы им не доверяете, то просто не используйте их.

Теперь вам нужно решить одну непростую проблему. Представьте, что параметр `multiple` имеет значение `2`. Реализация в этом случае выглядит просто:

```
fun isMaxMultiple(acc: Int, elem: Int): Int =
    if (elem / 2 * 2 == elem && elem > acc) elem else acc
```

Теперь вы должны заменить `2` значением параметра `multiple`. Это легко реализовать, если определить локальную функцию:

```
fun maxMultiple(multiple: Int, list: List<Int>): Int {
    fun isMaxMultiple(acc: Int, elem: Int): Int =
        if (elem / multiple * multiple == elem && elem > acc) elem else acc
    return list.fold(0, ::isMaxMultiple)
}
```

Беда в том, что такая реализация не решает проблему тестирования. И снова вам на выручку может прийти абстракция. Вы можете абстрагировать параметр `multiple`, как показано ниже:

```
// example01
fun isMaxMultiple(multiple: Int) =
```

```
    { max: Int, value: Int ->
        when {
            value / multiple * multiple == value && value > max -> value
            else -> max
        }
    }
```

Абстрагирование – это первое, что вы всегда должны делать. Например, здесь для любого программиста абсолютно естественно абстрагировать `value / multiple * multiple == value` в вызов функции `rem`, представленной инфиксным оператором `%`:

```
fun isMaxMultiple(multiple: Int) =
    { max: Int, value: Int ->
        when {
            value % multiple == 0 && value > max -> value
            else -> max
        }
    }
```

Теперь функцию `isMaxMultiple` легко протестировать, например:

```
fun test(value: Int, max:Int, multiple: Int): Boolean {
    val result = isMaxMultiple(multiple)(max, value)

    ... проверка свойств
}
```

Вы можете определить несколько свойств для проверки:

- `result >= max`;
- `result % multiple == 0 || result == max`;
- `(result % multiple == 0 && result >= value) || result % multiple != 0`.

Можно также определить другие свойства. В идеале вы должны получить минимальный набор свойств, гарантирующий верность результата. Но ничего страшного не случится, если вы определите несколько избыточных свойств. Избыточные свойства безвредны (если не требуют много времени для проверки). Недостаток свойств – это гораздо более серьезная проблема. Теперь вы получили огромное преимущество – можете проверить свой тест!

Зависимости для модульного тестирования на основе свойств

Для определения модульных тестов в Kotlin можно использовать разные фреймворки тестирования. Большинство из них так или иначе зависит от хорошо известных фреймворков тестирования для Java, таких как JUnit. Для тестирования на основе свойств обычно используется Kotlintest. Фреймворк Kotlintest основан на JUnit и добавляет предметно-ориентированные языки (Domain Specific Language, DSL), которые позволяют выбрать наиболее подходящий для вас стиль тестирования из множества,

включая тестирование на основе свойств. Чтобы использовать Kotlintest в своем проекте, добавьте следующую строку в блок `dependencies` в файле `build.gradle`:

```
dependencies {
    ...
    testCompile("io.kotlintest:kotlintest-runner-junit5:${project
.rootProject.ext["kotlintestVersion"]}")
    testRuntime("org.slf4j:slf4j-nop:${project
.rootProject.ext["slf4jVersion"]}")
}
```

Обратите внимание на переменные, используемые для представления номеров версий. Добавьте эти строки в сценарии сборки всех модулей. Чтобы избавиться от необходимости обновлять номера версий во всех модулях, добавьте следующие определения в сценарий сборки родительского проекта:

```
ext["kotlintestVersion"] = "3.1.10"
ext["slf4jVersion"] = "1.7.25"
```

Зависимость `testRuntime` от `Slf4j` не является обязательной. Опустив ее, вы получите предупреждение от компилятора, но тесты будут работать.

Фреймворк Kotlintest использует `Slf4j` для нужд журналирования и по умолчанию выводит подробную информацию. Если вам не требуется такое подробное журналирование тестов, используйте зависимость `slf4j-nop`, чтобы подавить журналирование и вывод ошибок, связанных с ним. Вы также можете выбрать любую другую реализацию журналирования и добавить соответствующую конфигурацию.

Разработка тестов на основе свойств

Тесты для Kotlintest размещаются там же, где и обычные тесты для JUnit: в каталоге `src/test/kotlin` каждого подпроекта. Если вы пользуетесь интегрированной средой IntelliJ, щелкните мышкой по функции, которую хотите протестировать, нажмите комбинацию **Alt+** и в контекстном меню выберите пункт **Create Test** (Создать тест). В диалоговом окне выберите версию JUnit. IntelliJ сообщит об отсутствии этой версии и предложит исправить. Игнорируйте эту ошибку и убедитесь, что целевой пакет находится в правильной ветке (`java` или `kotlin` в случае смешанного проекта). Если хотите, измените предложенное имя для класса, но не выбирайте никаких функций для тестирования, а затем щелкните по кнопке **OK**. IntelliJ создаст пустой файл в соответствующем пакете.

Каждый тестовый класс имеет следующую структуру:

```
// example01 test
import io.kotlintest.properties.forAll
import io.kotlintest.specs.StringSpec

class MyClassTest: StringSpec() {
```

```
    init {
        "test1" {
            forAll {
                // добавьте сюда проверку свойств
            }
        }
    }
}
```

При желании можно указать, сколько раз должны выполняться тесты (по умолчанию они выполняются 1000 раз):

```
import io.kotlintest.properties.forAll
import io.kotlintest.specs.StringSpec

class MyClassTest: StringSpec() {

    init {
        "test1" {
            forAll(3000) {
                // добавьте сюда проверку свойств
            }
        }
    }
}
```

Последний параметр функции `forAll` – это функция, которая должна возвращать `true`, если тестирование прошло успешно. Эта функция может принимать несколько аргументов, которые генерируются автоматически. Например:

```
// example01 test
class MyKotlinLibraryTest: StringSpec() {

    init {
        "isMaxMultiple" {
            forAll { multiple: Int, max:Int, value: Int ->
                isMaxMultiple(multiple)(max, value).let { result ->
                    result >= value
                        && result % multiple == 0 || result == max
                        && ((result % multiple == 0 && result >= value)
                                                || result % multiple != 0)
                }
            }
        }
    }
}
```

Тест в этом примере имеет имя `"isMaxMultiple"`, которое используется для вывода результата. В блок `init` можно поместить несколько таких блоков, но все они должны иметь разные имена. Имя тестируемой функции определяется с помощью механизма рефлексии. Отличие имен не проверяется во время компиляции, но, если во время выполнения будет найдено несколько тестов с одинаковыми именами, вы получите

исключение `IllegalArgumentException` с сообщением «Cannot add test with duplicate name isMaxMultiple» (Невозможно добавить тест с повторяющимся именем isMaxMultiple) и тесты выполняться не будут.

Все аргументы лямбда-выражения, переданные в параметрах функции `forAll`, генерируются стандартными генераторами, входящими в состав Kotlintest. Иногда это может быть нежелательно. Для генерации целых чисел по умолчанию используется генератор `Gen.int()`. В этом случае предыдущий тест можно было бы переписать так:

```
class MyKotlinLibraryTest: StringSpec() {

    init {
        "isMaxMultiple" {
            forAll(Gen.int(), Gen.int(), Gen.int())
                { multiple: Int, max:Int, value: Int ->
                    isMaxMultiple(multiple)(max, value).let
                        { result -> result >= value
                          && result % multiple == 0 || result == max
                          && ((result % multiple == 0 && result >= value)
                                || result % multiple != 0)
                        }
                }
        }
    }
}
```

Если вы решите указать хотя бы один генератор, придется указать все. Генератор `Gen.int()` по умолчанию генерирует последовательность `0`, `Int.MAX_VALUE`, `Int.MIN_VALUE` и затем множество случайных значений `Int`, чтобы создать 1000 тестов по умолчанию или указанное количество тестов. Каждый генератор пытается генерировать граничные значения плюс случайные значения. Например, генератор строк всегда генерирует пустую строку. Если вы запустите этот тест, произойдет сбой со следующим сообщением об ошибке:

```
Attempting to shrink failed arg -2147483648
Shrink #1: 0 fail
Shrink result => 0

...

java.lang.AssertionError: Property failed for
Arg 0: 0 (shrunk from -2147483648)
Arg 1: 0 (shrunk from -2147483648)
Arg 2: 0 (shrunk from 2147483647)
after 1 attempts
Caused by: expected: true but was: false
Expected :true
Actual   :false
```

Это означает, что тест потерпел неудачу на некотором значении. В таком случае Kotlintest пытается уменьшить значение, чтобы найти наименьшее значение, которое вынуждает тест потерпеть неудачу.

Это полезно для тестов, которые терпят неудачу, получив слишком большое значение. Было бы мало проку, если бы фреймворк просто сообщал об ошибке. Столкнувшись с ошибкой, вам нужно отыскать наименьшее допустимое значение, чтобы найти точку отказа. В примере выше можно видеть, что все сходится к аргументу 0. На самом деле тест всегда завершается неудачей, если первый аргумент равен 0, потому делить на 0 нельзя.

Здесь вам нужен генератор ненулевых целых чисел. Интерфейс `Gen` предлагает множество генераторов для всех случаев использования. Например, положительные целые числа можно сгенерировать с помощью `Gen.positiveIntegers()`:

```
// example02 test
class MyKotlinLibraryTest: StringSpec() {

    init {
        "isMaxMultiple" {
            forAll(Gen.positiveIntegers(), Gen.int(), Gen.int())
                { multiple: Int, max:Int, value: Int ->
                    isMaxMultiple(multiple)(max, value).let
                        { result -> result >= value
                          && result % multiple == 0 || result == max
                          && ((result % multiple == 0 && result >= value)
                                    || result % multiple != 0)
                        }
                }
        }
    }
}
```

И теперь тестирование выполняется благополучно.

Создание своих генераторов

Kotlintest предлагает множество генераторов для большинства стандартных типов (числа, логические значения и строки) и их коллекций. Но иногда бывает нужно определить свой генератор. Стандартный случай – создание экземпляров ваших собственных классов. Определить такой генератор просто – достаточно реализовать следующий интерфейс:

```
interface Gen<T> {
    fun constants(): Iterable<T>
    fun random(): Sequence<T>
}
```

Функция `constants` возвращает набор граничных значений (таких как `Int.MIN_VALUE`, `0` и `INT.MAX_VALUE` для целых чисел). Обычно вы будете возвращать из нее пустой список. Функция `random` возвращает последовательность случайно сгенерированных экземпляров.

Создание экземпляра – это рекурсивный процесс. Если конструктор класса, экземпляры которого требуется сгенерировать, принимает толь-

ко параметры, для которых уже имеются генераторы в Gen, просто используйте их. Для параметров других типов реализуйте и используйте свои генераторы.

Старайтесь избегать определения генератора с нуля. Все объекты данных являются комбинациями чисел, строк, логических значений и – рекурсивно – других объектов. В конце концов, вы можете создать любой объект, объединив существующие генераторы. Лучший способ объединения генераторов – использовать одну из их функций bind, такую как:

```
data class Person(val name: String, val age: Int)

val genPerson: Gen<Person> =
    Gen.bind(Gen.string(), Gen.choose(1, 50))
        { name, age -> Person(name, age) }
```

Использование своих генераторов

Свой генератор можно использовать в случаях, когда уже имеется подходящий стандартный генератор, но нужно исключить какие-то определенные значения или ограничить диапазон генерируемых значений некоторым диапазоном, т. е. когда требуется управлять сгенерированными значениями.

Например, если нужно написать программу, которая возвращает множество символов, используемых в строке, как убедиться, что она возвращает правильный результат для случайно сгенерированной строки? Вы можете разработать критерии для тестирования, такие как:

- все символы в результате должны присутствовать во входной строке;
- все символы во входной строке должны присутствовать в результате.

Выглядит довольно просто. Но представьте, что вместо множества функция должна создать ассоциативный массив, в котором символы из строки используются в роли ключей, а число вхождений этих символов – в роли значений. Например, представьте, что вам нужно написать программу, которая принимает список строк и группирует их по одинаковым множествам символов. Вы могли бы написать ее, как показано ниже:

```
// example03
fun main(args: Array<String>) {
    val words = listOf("the", "act", "cat", "is", "bats",
                       "tabs", "tac", "aabc", "abbc", "abca")

    val map = getCharUsed(words)

    println(map)
}

fun getCharUsed(words: List<String>) =
    words.groupBy(::getCharMap)
```

```
fun getCharMap(s: String): Map<Char, Int> {
    val result = mutableMapOf<Char, Int>()
    for (i in 0 until s.length) {
        val ch = s[i]
        if (result.containsKey(ch)) {
            result.replace(ch, result[ch]!! + 1)
        } else {
            result[ch] = 1
        }
    }
    return result
}
```

Эта программа вернет следующий результат (в одну строку):

```
{
    {t=1, h=1, e=1}=[the],
    {a=1, c=1, t=1}=[act, cat, tac],
    {i=1, s=1}=[is],
    {b=1, a=1, t=1, s=1}=[bats, tabs],
    {a=2, b=1, c=1}=[aabc, abca],
    {a=1, b=2, c=1}=[abbc]
}
```

Как бы вы протестировали эту программу? Можно попробовать передать ей несколько списков слов, но это не гарантирует, что она будет возвращать правильный результат для всех комбинаций. Лучше использовать тестирование на основе свойств, генерировать списки случайных строк, а затем проверять некоторые свойства. Но как придумать хороший набор свойств? Это будут сложные свойства.

Вместо разработки сложных свойств можно создать генератор строк, который будет добавлять случайные символы в сгенерированную строку при обновлении ассоциативного массива. Когда генерация завершится, генератор вернет пару `Pair<String, Map<Char, Int>>`. При наличии таких сгенерированных данных у вас останется только одно свойство для проверки: ассоциативный массив, созданный тестируемой программой, должен совпадать со сгенерированным ассоциативным массивом. Вот пример такого генератора:

```
// example03 test
val stringGenerator: Gen<List<Pair<String, Map<Char, Int>>>> =
    Gen.list(Gen.list(Gen.choose(97, 122)))
        .map { intListList ->
            intListList.asSequence().map { intList ->
                intList.map { n ->
                    n.toChar()
                }
            }.map { charList ->
                Pair(String(charList.toCharArray()),
                            makeMap(charList))
            }.toList()
        }
```

```
fun makeMap(charList: List<Char>): Map<Char, Int> =
    charList.fold(mapOf(), ::updateMap)

fun updateMap(map: Map<Char, Int>, char: Char) = when {
    map.containsKey(char) -> map + Pair(char, map[char]!! + 1)
    else -> map + Pair(char, 1)
}
```

Этот генератор сгенерирует последовательность из списка пар из одной случайной строки и одного ассоциативного массива, содержащего ключи с символами из строки и значения с количеством вхождений каждого символа. Обратите внимание, что ассоциативный массив создается не путем анализа строк. Строки строятся из тех же данных, что и ассоциативные массивы.

С помощью этого генератора легко протестировать программу `getCharUsed`, потому что проверить требуется только одно свойство. Это свойство все еще немного сложно читается, но, по крайней мере, вам не нужен огромный набор свойств, и вы уверены, что тест является исчерпывающим:


```
// example03 test
import io.kotlintest.properties.forAll
import io.kotlintest.specs.StringSpec

class SameLettersStringKtTest: StringSpec() {
    init {
        "getCharUsed" {
            forAll(stringGenerator) {
                list: List<Pair<String, Map<Char, Int>>> ->
                    getCharUsed(list.map { it.first }).keys.toSet() ==
                        list.asSequence().map { it.second }.toSet()
            }
        }
    }
}
```

Обратите внимание, что перед сравнением списки преобразуются в множества. Так учитывается то обстоятельство, что сгенерированный список строк может содержать несколько вхождений одной и той же строки. Генератор генерирует список до 100 строк длиной до 100 символов. Если вам понадобится параметризовать эти значения, можете написать новый генератор с нуля, например:

```
// example04 test
class StringGenerator(private val maxList: Int,
                      private val maxString: Int) :
        Gen<List<Pair<String, Map<Char, Int>>>> {

  override
  fun constants(): Iterable<List<Pair<String, Map<Char, Int>>>> =
    listOf(listOf(Pair("", mapOf())))

  override
  fun random(): Sequence<List<Pair<String, Map<Char, Int>>>> =
```

```
    Random().let { random ->
      generateSequence {
        (0 until random.nextInt(maxList)).map {
          (0 until random.nextInt(maxString))
            .fold(Pair("", mapOf<Char, Int>())) { pair, _ ->
              (random.nextInt(122 - 96) + 96).toChar().let
                { char ->
                  Pair("${pair.first}$char", updateMap(pair.second, char))
                }
            }
        }
      }
    }
}
```

Аналогично можно ограничить минимальное и максимальное значения символов в сгенерированных строках. Но лучше создать модифицированный генератор списков:

```
class ListGenerator<T>(private val gen: Gen<T>,
                       private val maxLength: Int) : Gen<List<T>> {

    private val random = Random()

    override fun constants(): Iterable<List<T>> =
        listOf(gen.constants().toList())

    override fun random(): Sequence<List<T>> = generateSequence {
        val size = random.nextInt(maxLength)
        gen.random().take(size).toList()
    }

    override fun shrinker() = ListShrinker<T>()
}
```

Этот генератор можно задействовать в генераторе строк:

```
fun stringGenerator(maxList: Int,
                    maxString: Int):
        Gen<List<Pair<String, Map<Char, Int>>>> =

    ListGenerator(ListGenerator(Gen.choose(32, 127), maxString), maxList)
        .map { intListList ->
            intListList.asSequence().map { intList ->
                intList.map { n ->
                    n.toChar()
                }
            }.map { charList ->
                Pair(String(charList.toCharArray()), makeMap(charList))
            }.toList()
        }
```

Длину сгенерированной строки можно ограничить добавлением фильтра к внешнему генератору списка (генерирующему список строк), но это очень неэффективно, потому что только 1/10 000 000 сгенерированных строк будет проходить через фильтр, поэтому генерирование

будет выполняться медленно. И кроме того, этот вид фильтрации позволит ограничить только длину сгенерированных строк, а не длину списка строк.

Вот как такой генератор можно использовать:

```
class SameLettersStringKtTest: StringSpec() {
    init {
        "getCharUsed" {
            forAll(stringGenerator(100, 100)) {
                list: List<Pair<String, Map<Char, Int>>> ->
                    getCharUsed(list.map { it.first }).keys.toSet() ==
                        list.asSequence().map { it.second }.toSet()
            }
        }
    }
}
```

Другой способ решить ту же проблему тестирования – продолжить абстрагирование, как было показано в начале этого приложения.

Упрощение кода дальнейшим абстрагированием

Функция `getCharUsed` уже использует две абстракции: функцию `groupBy` и функцию `getCharMap`. Однако эту функцию можно абстрагировать еще больше:

```
// example05
fun getCharUsed(words: List<String>): Map<Map<Char, Int>, List<String>> =
    words.groupBy(::getCharMap)

fun getCharMap(s: String): Map<Char, Int> = s.fold(mapOf(), ::updateMap)

fun updateMap(map: Map<Char, Int>, char: Char): Map<Char, Int> =
    when {
        map.containsKey(char) -> map + Pair(char, map[char]!! + 1)
        else -> map + Pair(char, 1)
    }
```

Функция `getCharMap` использует две абстракции. Одна из них – функция `fold`, которую не требуется тестировать, и другая – функция `updateMap`, единственная, нуждающаяся в тестировании. Найти проверяемые свойства в этом случае достаточно просто:

- для любых `char` и `map`, содержащих этот `char` в виде ключа, вызов `getCharMap(map, char)[char]` должен возвращать результат, эквивалентный `map[char] + 1`;
- для любых `char` и `map`, не содержащих этот `char` в виде ключа, вызов `getCharMap(map, char)[char]` должен возвращать 1;
- для любых `char` и `map` после удаления ключа `char` из результата вызова `getCharMap(map, char)[char]` и из `map` должны получаться одинаковые значения. (Это свойство проверяет отсутствие других модификаций в `map`.)

Для этого теста требуется сгенерировать случайные ассоциативные массивы, заполненные случайными данными. Вы можете написать следующий генератор `MapGenerator` (генерирующий символы в диапазоне `[a..z]`):

```
// example05 test
fun mapGenerator(min: Char = 'a', max: Char = 'z'): Gen<Map<Char, Int>> =
    Gen.list(Gen.choose(min.toInt(), max.toInt())
        .map(Int::toChar)).map(::makeMap)
```

и использовать его в тесте:

```
// example05 test
class UpdateMapTest: StringSpec() {

    private val random = Random()

    private val min = 'a'

    private val max = 'z'

    init {
        "getCharUsed" {
            forAll(mapGenerator()) { map: Map<Char, Int> ->
                (random.nextInt(max.toInt() - min.toInt())
                                            + min.toInt()).toChar().let
                    {
                        if (map.containsKey(it)) {
                            updateMap(map, it)[it] == map[it]!! + 1
                        } else {
                            updateMap(map, it)[it] == 1
                        } && updateMap(map, it) - it == map - it
                    }
            }
        }
    }
}
```

Обратите внимание, что принцип максимального абстрагирования также применим к тестам. Единственно, что можно абстрагировать, – это генерирование каждого символа. Интерфейс `Gen` не предлагает генератора символов, но его легко создать. И поскольку вам может потребоваться сгенерировать только некоторые конкретные символы, также можно абстрагировать выбор символов в отдельную функцию:

```
// example06 test
fun charGenerator(p: (Char) -> Boolean): Gen<Char> =
    Gen.choose(0, 255).map(Int::toChar).filter(p)
```

Теперь можно написать более ясный и чистый код теста:

```
class UpdateMapTest: StringSpec() {

    init {
        "getCharUsed" {
            forAll(MapGenerator,
```

```
                charGenerator(Char::isLetterOrDigit)) {
            map: Map<Char, Int>, char ->
                if (map.containsKey(char)) {
                    updateMap(map, char)[char] == map[char]!! + 1
                } else {
                    updateMap(map, char)[char] == 1
                }
                && updateMap(map, char) - char == map - char
        }
    }
  }
}
```

Но вы можете упростить код еще больше, использовав значение по умолчанию для ассоциативного массива. Когда запрашивается ассоциативный массив для отсутствующего символа, в ответ должно возвращаться значение `0`, потому что это число вхождений.

Вместо инициализации ассоциативного массива всеми возможными символами, связанным с `0` числом вхождений, нужно просто вызвать функцию `getOrDefault` или использовать синтаксис `[]` (эквивалентный функции `get`):

```
// example06 test
class UpdateMapTest: StringSpec() {

    init {
        "getCharUsed" {
            forAll(MapGenerator,
                   charGenerator(Char::isLetterOrDigit)) {
                map: Map<Char, Int>, char ->
                    updateMap(map, char)[char] == map.getOrDefault(char, 0) + 1
                              && updateMap(map, char) - char == map - char
            }
        }
    }
}
```

Мы сократили количество тестируемых элементов до минимума и создали мощный способ проверки результатов. Запустив тест, генерирующий тысячи значений для каждой, вы быстро протестируете свою программу на миллионах примеров данных. Поскольку основная программа и тест используют функцию `updateMap`, вам, вероятно, следует поместить ее в общий модуль.

Итоги

В этом приложении вы узнали, как писать эффективные тесты и как сделать свою программу более пригодной для тестирования, применив прием тестирования на основе свойств. Вы узнали, как:

- придумать набор свойств, которыми должен обладать результат работы программы;

- писать интерфейсы, затем тесты и, наконец, сами реализации, чтобы тесты не зависели от реализации;
- использовать абстрагирование для упрощения тестирования, применяя решения, которым можно доверять;
- писать генераторы для получения случайных значений в тестах вместе с сопутствующими данными, которые можно использовать для проверки свойств;
- подготавливать тесты, использующие тысячи образцов входных данных и выполняемые перед каждой сборкой.

Предметный указатель

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

К

Л

М

Н

О

П

Р

C

T

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу в книготорговой компании «Галактика»
(представляет интересы издательств «ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@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.a-planeta.ru**.

Пьер-Ив Сомон

Волшебство Kotlin

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 43,55. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**